the magazine of the accu

www.accu.org

{cvu}

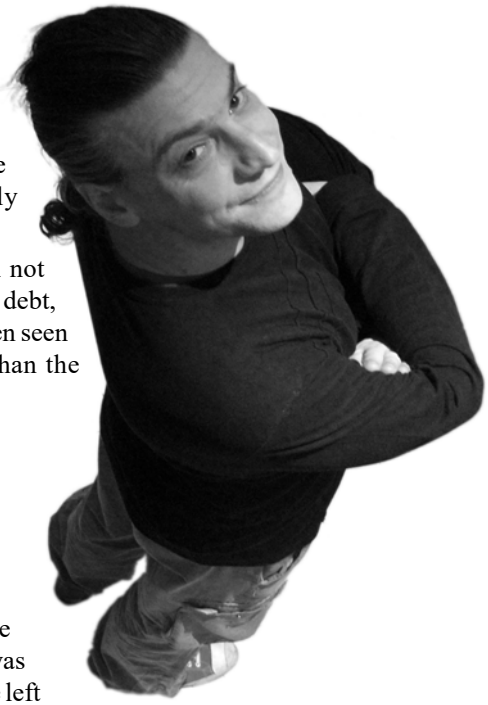
Volume 32 • Issue 3 • July 2020 • £4.50

Features

Regulars

{cvu}
Volume 32 Issue 3
July 2020
ISSN 1354-3164
www.accu.org

More than a label



Sometimes as programmers we get accused of being perfectionists. I say accused, because it's usually *not* a term of endearment. It's often used in a context such as "we don't have time for perfectionists who want to spend all their time noodling with the code until it's *perfect*, and so never actually *ship* anything." We're more likely to hear it in a sentence containing the words 'over-engineered' than the words 'beautifully crafted'.

We're also more likely to hear it in conjunction with not having time to refactor the code, address the technical debt, automate the build or write tests. These things are so often seen as obstacles to actually *delivering* software, rather than the things that *enable* it to be shipped in working order.

This problem, like many others, is not limited to software development. We have to face up to it sometime – software isn't special! I recently read an article in the music press about a rock band complaining that the label had ruined their record by insisting on releasing it, even though the musicians weren't happy with the final mix. They wanted a little more time to get it right, but were refused. The record was released to widespread criticism, and the musicians were left having to defend a recording of which they were less than proud.

I have met programmers who are quite happy to 'noodle' with code, and never finish it. I'd generally associate them more with the term 'hacker' than with 'perfectionist', but either way, they're vastly out-numbered by the number of developers I know who ship working software, sometimes under very challenging circumstances. They usually want the tests to pass, the build to be reliable, the deployment repeatable, and the release itself undramatic. Being accused of being a 'perfectionist' for wanting those things isn't helpful.

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 32.4: 1st August 2020
C Vu 32.5: 1st October 2020

Overload 158: 1st September 2020
Overload 159: 1st November 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

Expect the Unexpected (Part 2)

Pete Goodliffe continues to deal with the inevitable.

In the previous instalment of this mini-series, we looked at the landscape of 'error conditions' in our code. We investigated what errors are, what causes them, how we detect, and how we report error situations.

Now, let's look at the best strategies to *handle* error conditions in our code, to ensure our application logic recovers well. Or as well as can be expected. This is where we get practical.

Handling errors

Love truth, and pardon error.
~ Voltaire

Errors happen. We've seen how to discover them and when to do so. The question now is: What do you do about them? This is the hard part. The answer largely depends on circumstance and the gravity of an error – whether it's possible to rectify the problem and retry the operation or to carry on regardless. Often there is no such luxury; the error may even herald the beginning of the end. The best you can do is clean up and exit sharply, before anything else goes wrong.

To make this kind of decision, you must be informed. You need to know a few key pieces of information about the error:

- Where it came from

  This is quite distinct from where it's going to be handled. Is the source a core system component or a peripheral module? This information may be encoded in the error report; if not, you can figure it out manually.

- What you were trying to do

  What provoked the error? This may give a clue toward any remedial action. Error reporting seldom contains this kind of information, but you can figure out which function was called from the context.

- Why it went wrong

  What is the nature of the problem? You need to know exactly what happened, not just a general *class* of error. How much of the erroneous operation completed? *All* or *none* are nice answers, but generally, the program will be in some indeterminate state between the two.

- When it happened

  This is the locality of the error in time. Has the system only just failed, or is a problem two hours old finally being felt?

- The severity of the error

  Some problems are more serious than others, but when detected, one error is equivalent to another – you can't continue without understanding and managing the problem. Error severity is usually determined by the caller, based on how easy it will be to recover or work around the error.

- How to fix it

  This may be obvious (e.g., insert a floppy disk and retry) or not (e.g., you need to modify the function parameters so they are consistent). More often than not, you have to infer this knowledge from the other information you have.

Given this depth of information, you can formulate a strategy to handle each error. Forgetting to insert a handler for any potential error will lead to a bug, and it might turn out to be a bug that is hard to exercise and hard to track down – so think about every error condition carefully.

When to deal with errors

When should you handle each error? This can be separate from when it's detected. There are two schools of thought.

- As soon as possible

  Handle each error *as* you detect it. Since the error is handled near to its cause, you retain important contextual information, making the error-handling code clearer. This is a well known self-documenting code technique. Managing each error near its source means that control passes through less code in an invalid state.

  This is usually the best option for functions that return error codes.

- As late as possible

  Alternatively, you could defer error handling for as long as possible. This recognizes that code detecting an error rarely knows what to do about it. It often depends on the context in which it is used: A missing file error may be reported to the user when loading a document but silently swallowed when hunting for a preferences file.

  Exceptions are ideal for this; you can pass an exception through each level until you know how to deal with the error. This separation of detection and handling may be clearer, but it can make code more complex. It's not obvious that you are deliberately deferring error handling, and it's not clear where an error came from when you do finally handle it.

  In theory, it's nice to separate 'business logic' from error handling. But often you can't, as clean-up is necessarily entwined with that business logic, and it can be more tortuous to write the two separately. However, centralized error-handling code has advantages: You know where to look for it, and you can put the abort/continue policy in one place rather than scatter it through many functions.

Thomas Jefferson once declared, "Delay is preferable to error." There is truth there; the actual *existence* of error handling is far more important than *when* an error is handled. Nevertheless, choose a compromise that's close enough to prevent obscure and out-of-context error handling, while being far enough away to not cloud normal code with roundabout paths and error-handling dead ends.

> Handle each error in the most appropriate context, as soon as you know enough about it to deal with it correctly.

Possible reactions

You've caught an error. You're poised to handle it. What are you going to do now? Hopefully, whatever is required for correct program operation. While we can't list every recovery technique under the sun, here are the common reactions to consider.

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

- **Logging**

  Any reasonably large project should already be employing a logging facility. It allows you to collect important trace information, and is an entry point for the investigation of nasty problems.

  The log exists to record interesting events in the life of the program, to allow you to delve into its inner workings and reconstruct paths of execution. For this reason, all errors you encounter should be detailed in the program log; they are some of the most interesting and telling events of all. Aim to capture all pertinent information – as much of the previous list as you can.

  For really obscure errors that predict catastrophic disaster, it may be a good idea to get the program to 'phone home' – to transmit either a snapshot of itself or a copy of the error log to the developers for further investigation.

  What you do *after* logging is another matter.

- **Reporting**

  A program should only report an error to the user when there's nothing left to do. The user does not need to be bombarded by a thousand small nuggets of useless information or badgered by a raft of pointless questions. Save the interaction for when it's really vital. Don't report when you encounter a recoverable situation. By all means, log the event, but keep quiet about it. Provide a mechanism that enables users to read the event log if you think one day they might care.

  There *are* some problems that only the user can fix. For these, it is good practice to report the problem immediately, in order to allow the user the best chance to resolve the situation or else decide how to continue.

  Of course, this kind of reporting depends on whether or not the program is interactive. Deeply embedded systems are expected to cope on their own; it's hard to pop up a dialog box on a washing machine.

- **Recovery**

  Sometimes your only course of action is to stop immediately. But not all errors spell doom. If your program saves a file, one day the disk will fill up and the save operation will fail. The user expects your program to continue happily, so be prepared.

  If your code encounters an error and doesn't know what to do about it, pass the error upwards. It's more than likely your caller will have the ability to recover.

- **Ignore**

  I only include this for completeness. Hopefully by now you've learned to scorn the very suggestion of ignoring an error. If you choose to forget all about handling it and to just continue with your fingers crossed, *good luck*. This is where most of the bugs in any software package will come from. Ignoring an error whose occurrence may cause the system to misbehave inevitably leads to hours of debugging.

  > Ignoring errors does not save time. You'll spend far longer working out the cause of bad program behaviour than you ever would have spent writing the error handler.

  You can, however, write code that allows you to *do nothing* when an error crops up. Is that a blatant contradiction? No. It is possible to write code that copes with an inconsistent world, that can carry on correctly in the face of an error – but it often gets quite convoluted. If you adopt this approach, you must make it obvious in the code. Don't risk having it misinterpreted as ignorant and incorrect.

- **Propagate**

  When a subordinate function call fails, you probably can't carry on, but you might not know what else to do. The only option is to clean up and propagate the error report upwards. You have options. There are two ways to propagate an error:

  - Export the same error information you were fed (return the same reason code or propagate exceptions)
  - Reinterpret the information, sending a more meaningful message to the next level up (return a different reason code or catch and wrap up exceptions)

Ask yourself this question: Does the error relate to a concept exposed through the module interface? If so, it's okay to propagate that same error. Otherwise, recast it in the appropriate light, choosing an error report that makes sense in the context of your module's interface. This is a good self-documenting code technique.

Crafting error messages

Inevitably, your code will encounter errors that the user must sort out. Human intervention may be the only option; your code can't insert a floppy disk or switch on the printer by itself. (If it can, you'll make a fortune!)

If you're going to whine at the user, there are a few general points to bear in mind:

- Users don't think like programmers, so present information the way they'd expect. When displaying the free space on a disk, you might report Disk space: 10K. But if there's no space left, a zero could be misread as OK – and the user will not be able to fathom why he can't save a file when the program says everything's fine.
- Make sure your messages aren't too cryptic. You might understand them, but can your computer-illiterate granny? (It doesn't matter if your granny won't use this program – someone with a lower intellect almost certainly will.)
- Don't present meaningless error codes. No user knows what to do when faced with an Error code 707E. It is, however, valuable to provide such codes as "additional info" – they can be quoted to tech-support or looked up more easily on a web search.
- Distinguish dire errors from mere warnings. Incorporate this information in the message text (perhaps with an Error: prefix), and emphasize it in message boxes with an accompanying icon.
- Only ask a question (even a simple one like Continue: Yes/No?) if the user fully understands the ramifications of each choice. Explain it if necessary, and make it clear what the consequence of each answer is.

What you present to the user will be determined by interface constraints and application or OS style guides. If your company has user interface engineers, then it's their job to make these decisions. Work with them.

Code implications

Show me the code! Let's spend some time investigating the implications of error handling in our code. As we'll see, it is not easy to write good error handling that doesn't twist and warp the underlying program logic.

The first piece of code we'll look at is a common error-handling structure. Yet it isn't a particularly intelligent approach for writing error-tolerant code. The aim is to call three functions sequentially – each of which may fail – and perform some intermediate calculations along the way. Spot the problems with Listing 1.

Listing 1

```
void nastyErrorHandling()
{
  if (operationOne())
  {
    ... do something ...
    if (operationTwo())
    {
      ... do something else ...
      if (operationThree())
      {
        ... do more ...
      }
    }
  }
}
```

Syntactically it's fine; the code will work. Practically, it's an unpleasant style to maintain. The more operations you need to perform, the more deeply nested the code gets and the harder it is to read. This kind of error handling quickly leads to a rat's nest of conditional statements. It doesn't reflect the actions of the code very well; each intermediate calculation could be considered the same level of importance, yet they are nested at different levels.

Can we avoid these problems? Yes – there are a few alternatives. The first variant (see Listing 2) flattens the nesting. It is semantically equivalent, but it introduces *some* new complexity, since flow control is now dependent on the value of a new status variable, **ok**.

We've also added an opportunity to clean up after any errors. Is that sufficient to mop up all failures? Probably not; the necessary clean-up may depend on how far we got through the function before lightening struck. There are two clean-up approaches:

- Perform a little clean-up after each operation that may fail, then return early. This inevitably leads to duplication of clean-up code. The more work you've done, the more you have to clean up, so each exit point will need to do gradually more unpicking.

  If each operation in our example allocates some memory, each early exit point will have to release all allocations made to date. The further in, the more releases. That will lead to some quite dense and repetitive error-handling code, which makes the function far larger and far harder to understand.
- Write the clean-up code once, at the end of the function, but write it in such a way as to only clean up what's dirty. This is neater, but if you inadvertently insert an early return in the middle of the function, the clean-up code will be bypassed.

If you're not overly concerned about writing *Single Entry, Single Exit (SESE)* functions, the next example removes the reliance on a separate control flow variable. (Although this clearly isn't SESE, I contend that the previous example isn't, either. There *is* only one exit point, at the end, but the contrived control flow is simulating early exit – it *might as well* have multiple exits. This is a good example of how being bound by a rule like SESE can lead to bad code, unless you think carefully about what you're doing.) We do lose the clean-up code again, though. Simplicity renders Listing 3 a better description of the actual intent.

A combination of this short circuit exit with the requirement for clean-up leads to the approach in Listing 4, especially seen in low-level systems code. Some people advocate it as the *only* valid use for the maligned **goto**. I'm still not convinced.

You can avoid such monstrous code in C++ using *Resource Acquisition Is Initialization (RAII)*) techniques like smart pointers [1]. This has the bonus of providing exception safety – when an exception terminates your function prematurely, resources are automatically deallocated. These techniques avoid a lot of the problems we've seen above, moving complexity to a separate flow of control.

The same example using exceptions would look like this (in C++, Java, and C#), presuming that all subordinate functions do not return error codes but instead throw exceptions (see Listing 5).

This is only a basic exception example, but it shows just how neat exceptions can be. A sound code design might not need the **try**/**catch** block at all if it ensures that no resource is leaked and leaves error handling to a higher level. But alas, writing good code in the face of exceptions requires an understanding of principles beyond the scope of this chapter.

Raising hell

We've put up with other people's errors for long enough. It's time to turn the tables and play the bad guy: Let's raise some errors. When writing a function, erroneous things will happen that you'll need to signal to your caller. Make sure you do – don't silently swallow any failure. Even if you're sure that the caller won't know what to do in the face of the problem, it *must* remain informed. Don't write code that lies and pretends to be doing something it's not.

Which reporting mechanism should you use? It's largely an architectural choice; obey the project conventions and the common language idioms. In languages with the facility, it is common to favour exceptions, but only

Listing 2

```
void flattenedErrorHandling()
{
  bool ok = operationOne();
  if (ok)
  {
    ... do something ...
    ok = operationTwo();
  }
  if (ok)
  {
    ... do something else ...
    ok = operationThree();
  }
  if (ok)
  {
    ... do more ...
  }
  if (!ok)
  {
    ... clean up after errors ...
  }
}
```

Listing 3

```
void shortCircuitErrorHandling()
{
  if (!operationOne()) return;
  ... do something ...
  if (!operationTwo()) return;
  ... do something else ...
  if (!operationThree()) return;
  ... do more ...
}
```

Listing 4

```
void gotoHell()
{
  if (!operationOne()) goto error;
  ... do something ...
  if (!operationTwo()) goto error;
  ... do something else ...
  if (!operationThree()) goto error;
  ... do more ...
  return;
error:
  ... clean up after errors ...
}
```

Listing 5

```
void exceptionalHandling()
{
  try
  {
    operationOne();
    ... do something ...
    operationTwo();
    ... do something else ...
    operationThree();
    ... do more ...
  }
  catch (...)
  {
    ... clean up after errors ...
  }
}
```

use them if the rest of the project does. Java and C# really leave you with no choice; exceptions are buried deep in their execution run times. A C++ architecture may choose to forego this facility to achieve portability with platforms that have no exception support or to interface with older C code.

We've already seen strategies for propagating errors from subordinate function calls. Our main concern here is reporting fresh problems encountered during execution. How you determine these errors is your own business, but when reporting them, consider the following:

- Have you cleaned up appropriately first? Reliable code doesn't leak resources or leave the world in an inconsistent state, even when an error occurs, unless it's *really* unavoidable. If you do either of these things, it must be documented carefully. Consider what will happen the next time your code is called if this error has manifested. Ensure it will still work.
- Don't leak inappropriate information to the outside world in your error reports. Only return useful information that the caller understands and can act on.
- Use exceptions correctly. Don't throw an exception for unusual return values – the rare but not erroneous cases. Only use exceptions to signal circumstances where a function is not able to meet its contract. Don't use them non-idiomatically (i.e., for flow control).
- Consider using assertions if you're trapping an error that should never happen in the normal course of program execution, a genuine programming error. Exceptions are a valid choice for this too – some assertion mechanisms can be configured to throw exceptions when they trigger.
- If you can pull forward any tests to compile time, then do so. The sooner you detect and rectify an error, the less hassle it can cause.
- Make it hard for people to ignore your errors. Given half a chance, someone *will* use your code badly. Exceptions are good for this – you have to act deliberately to hide an exception.

What kind of errors should you be looking out for? This obviously depends on what the function is doing. Here's a checklist of the general kinds of error checks you should make in each function:

- Check all function parameters. Ensure you have been given correct and consistent input. Consider using assertions for this, depending on how strictly your contract was written. (Is it an offence to supply bad parameters?)
- Check that invariants are satisfied at interesting points in execution.
- Check all values from external sources for validity before you use them. File contents and interactive input must be sensible, with no missing pieces.
- Check the return status of all system calls and other subordinate function calls.

Exceptions are a powerful error reporting mechanism. Used well, they can simplify your code greatly while helping you to write robust software. In the wrong hands, though, they are a deadly weapon.

I once worked on a project where it was routine for programmers to break a `while` loop or end recursion by throwing an exception, using it as a non-local `goto`. It's an interesting idea, and kind of cute when you first see it. But this behaviour is nothing more than an abuse of exceptions: It isn't what exceptions are idiomatically used for. More than one critical bug was caused by a maintenance programmer not understanding the flow of control through a complex, magically terminated loop.

Follow the idioms of your language, and don't write cute code for the sake of it.

Managing errors

The common principle uniting the raising and handling of errors is to have a consistent strategy for dealing with failure, wherever it manifests. These are general considerations for managing the occurrence, detection, and handling of program errors:

- Avoid things that *could* cause errors. Can you do something that is guaranteed to work, instead? For example, avoid allocation errors by reserving enough resource beforehand. With an assured pool of memory, your routine cannot suffer memory restrictions. Naturally, this will only work when you know how much resource you need up front, but you often do.
- Define the program or routine's expected behavior under abnormal circumstances. This determines how robust the code needs to be and therefore how thorough your error handling should be. Can a function silently generate bad output, subscribing to the historic *GIGO* principle (that is, *Garbage in, garbage out* – feed it trash, and it will happily spit out trash).
- Clearly define which components are responsible for handling which errors. Make it explicit in the module's interface. Ensure that your client knows what will always work and what may one day fail.
- Check your programming practice: *When* do you write error-handling code? Don't put it off until later; you'll forget to handle something. Don't wait until your development testing highlights problems before writing handlers – that's not an engineering approach.

> Write all error detection and handling *now*, as you write the code that may fail. Don't put it off until later. If you must be evil and defer handling, at least write the detection scaffolding now.

- When trapping an error, have you found a symptom or a cause? Consider whether you've discovered the source of a problem that needs to be rectified here or if you've discovered a symptom of an earlier problem. If it's the latter, then don't write reams of handling code here, put that in a more appropriate (earlier) error handler.

Conclusion

To err is human; to repent, divine; to persist, devilish.
~ Benjamin Franklin

To err *is* human (but computers seem quite good at it, too). To handle these errors is divine.

Every line of code you write must be balanced by appropriate and thorough error checking and handling. A program without rigorous error handling will not be stable. One day an obscure error may occur, and the program will fall over as a result.

Handling errors and failure cases is hard work. It bogs programming down in the mundane details of the Real World. However, it's absolutely essential. As much as 90 percent of the code you write handles exceptional circumstances [2]. That's a surprising statistic, so write code *expecting* to put far more effort into the things that can go wrong than the things that will go right. ■

Questions

- Are *return values* and *exceptions* equivalent error reporting mechanisms?
- How should you handle the occurrence of errors in your error-handling code?
- How thorough is the error handling in your current codebase? How does this contribute to the stability of the program?
- Do you naturally consider error handling as you write code, or do you find it a distraction, preferring to come back to it later?

References

[1] Stroustrup (1997) *Resource Acquisition Is Initialization (RAII)*) techniques like smart pointers

[2] Bentley, Jon Louis (1982) *Writing Efficient Programs*. Prentice Hall Professional, ISBN-10: 013970244X

Greenback Backup

Paul Grenyer demonstrates a DevOps pipeline.

Why

When Naked Element was still a thing, we used DigitalOcean almost exclusively for our client's hosting. For the sorts of projects we were doing, it was the most straightforward and cost effective solution. DigitalOcean [1] provided managed databases, but there was no facility to automatically back them up. This led us to develop a Python-based program which was triggered once a day to perform the backup, push it to AWS S3 and send a confirmation or failure email.

We used Python due to familiarity, ease of use and low installation dependencies. I'll demonstrate this later on in the Dockerfile. S3 was used for storage as DigitalOcean did not have their equivalent, 'Spaces' [2], available in their UK data centre. The closest is in Amsterdam, but our clients preferred to have their data in the UK.

Fast forward to May 2020 and I'm working on a personal project which uses a PostgreSQL database. I tried to use a combination of AWS [3] and Terraform [4] for the project's infrastructure (as this is what I am using for my day job) but it just became too much effort to bend AWS to my will and it's also quite expensive. I decided to move back to Digital Ocean and got the equivalent setup sorted in a day. I could have taken advantage of AWS' free tier for the database for 12 months, but AWS backup storage is not free and I wanted as much as possible with one provider and within the same virtual private network (VPC).

I was back to needing my own backup solution. The new project I am working on uses Docker [5] to run the main service. My Droplet (that's what Digital Ocean calls its Linux server instances) setup up is minimal: non-root user setup, firewall configuration and Docker install. The DigitalOcean Market Place [6] includes a Docker image so most of that is done for me with a few clicks. I could have also installed Python and configured a backup program to run each evening. I'd also have to install the right version of the PostgreSQL client, which isn't currently in the default Ubuntu repositories, so is a little involved. As I was already using Docker it made sense to create a new Docker image to install everything and run a Python programme to schedule and perform the backups. Of course some might argue that a whole Ubuntu install and configure in a Docker image is a bit much for one backup scheduler, but once it's done it's done and can easily be installed and run elsewhere as many times as is needed.

There are two more decisions to note. My new backup solution will use DigitalOcean spaces, as I'm not bothered about my data being in Amsterdam and I haven't implemented an email server yet so there are no notification emails. This resulted in me jumping out of bed as soon as I woke each morning to check Spaces to see if the backup had worked, rather than just checking for an email. It took two days to get it all working correctly!

What

I reached for Naked Element's trusty Python backup program affectionately named Greenback after the arch enemy of Danger Mouse (Green-back up, get it? No, me neither…) but discovered it was too specific and would need some work, but would serve as a great template to start with.

It's worth nothing that I am a long way from a Python expert. I'm in the 'reasonable working knowledge with lots of help from Google' category. The first thing I needed the program to do was create the backup. At this point I was working locally where I had the correct PostgreSQL client installed, `db_backup.py` (see Listing 1).

Listing 1

```
db_connection_string=os.environ['DATABASE_URL']
class GreenBack:
  def backup(self):
    datestr = datetime.now()
      .strftime("%d_%m_%Y_%H_%M_%S")
    backup_suffix = ".sql"
    backup_prefix = "backup_"

    destination = backup_prefix + datestr
      + backup_suffix
    backup_command = 'sh backup_command.sh '
      + db_connection_string + ' ' + destination
    subprocess.check_output
      (backup_command.split(' '))
    return destination
```

I want to keep anything sensitive out of the code and out of source control, so I've brought in the connection string from an environment variable. The method constructs a filename based on the current date and time, calls an external bash script to perform the backup:

```
# connection string
# destination
pg_dump $1 > $2
```

and returns the backup file name. Of course, for Ubuntu I had to make the bash script executable. Next I needed to push the backup file to Spaces, which means more environment variables:

```
region=''
access_key=os.environ['SPACES_KEY']
secret_access_key=os.environ['SPACES_SECRET']
bucket_url=os.environ['SPACES_URL']
backup_folder='dbbackups'
bucket_name='findmytea'
```

So that the program can access Spaces and another method:

```
class GreenBack:
  ...
  def archive(self, destination):
    session = boto3.session.Session()
    client = session.client('s3',
    region_name = region, endpoint_url=bucket_url,
    aws_access_key_id = access_key,
      aws_secret_access_key=secret_access_key)
      client.upload_file(destination, bucket_name,
        backup_folder + '/' + destination)
      os.remove(destination)
```

It's worth noting that DigitalOcean implemented the Spaces API to match the AWS S3 API so that the same tools can be used. The `archive` method creates a session and pushes the backup file to Spaces and then deletes it from the local file system. This is for reasons of disk space and security. A future enhancement to Greenback would be to automatically remove old backups from Spaces after a period of time.

PAUL GRENYER

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

Listing 2

```
class GreenBack:
  last_backup_date = ""

  def callback(self, n, loop):
    today = datetime.now().strftime("%Y-%m-%d")
    if self.last_backup_date != today:
      logging.info('Backup started')
      destination = self.backup()
      self.archive(destination)

      self.last_backup_date = today
      logging.info('Backup finished')
    loop.call_at(loop.time() + n, self.callback,
      n, loop)
...

event_loop = asyncio.get_event_loop()
try:
  bk = GreenBack()
  bk.callback(60, event_loop)
  event_loop.run_forever()
finally:
  logging.info('closing event loop')
  event_loop.close()
```

The last thing the Python program needs to do is schedule the backups. A bit of Googling revealed an event loop which can be used to do this (see Listing 2).

On startup, **`callback`** is executed. It checks the **`last_back_date`** against the current date and if they don't match it runs the backup and updates the **`last_backup_date`**. If the dates do match, and after running the backup, the **`callback`** method is added to the event loop with a one minute delay. Calling **`event_loop.run_forever`** after the initial **`callback`** call means the program will wait forever and the process continues.

Now that I had a Python backup program I needed to create a Dockerfile that would be used to create a Docker image to setup the environment and start the program (Listing 3).

The Dockerfile starts with an Ubuntu image. This is a bare bones, but fully functioning, Ubuntu operating system. The Dockerfile then installs Python, its dependencies and the Greenback dependencies. Then it

Listing 3

```
FROM ubuntu:xenial as ubuntu-env
WORKDIR /greenback

RUN apt update
RUN apt -y install python3 wget gnupg sysstat
python3-pip

RUN pip3 install --upgrade pip
RUN pip3 install boto3 --upgrade
RUN pip3 install asyncio --upgrade

RUN echo 'deb http://apt.postgresql.org/pub/
repos/apt/ xenial-pgdg main' > /etc/apt/
sources.list.d/pgdg.list
RUN wget https://www.postgresql.org/media/keys/
ACCC4CF8.asc
RUN apt-key add ACCC4CF8.asc

RUN apt update
RUN apt -y install postgresql-client-12

COPY db_backup.py ./
COPY backup_command.sh ./

ENTRYPOINT ["python3", "db_backup.py"]
```

Listing 4

```
image: python:3.7.3

pipelines:
  default:
    - step:
  services:
    - docker
  script:
    - IMAGE="findmytea/greenback"
    - TAG=latest
    - docker login --username $DOCKER_USERNAME
      --password $DOCKER_PASSWORD
    - docker build -t $IMAGE:$TAG .
    - docker push $IMAGE:$TAG
```

installs the PostgreSQL client, including adding the necessary repositories. Following that, it copies the required Greenback files into the image and tells it how to run Greenback.

I like to automate as much as possible so while I did plenty of manual Docker image building, tagging and pushing to the repository during development, I also created a BitBucket Pipeline [7], which would do the same on every check in (see Listing 4).

Pipelines, BitBucket's cloud based Continuous Integration and Continuous Deployment feature, is familiar with Python and Docker so it was quite simple to make it log in to Docker Hub [8], build, tag and push the image. To enable the pipeline all I had to do was add the `bitbucket-pipelines.yml` file to the root of the repository, checkin, follow the BitBucket pipeline process in the UI to enable it and add then add the build environment variables so the pipeline could log into Docker Hub. I'd already created the image repository in Docker Hub.

The Greenback image shouldn't change very often and there isn't a straightforward way of automating the updating of Docker images from Docker Hub, so I wrote a bash script to do it, **`deploy_greenback`** (Listing 5).

Now, with a single command I can fetch the latest Greenback image, stop and remove the currently running image instance, install the new image, list the running images to reassure myself the new instance is running and follow the Greenback logs. When the latest image is run, it is named for easy identification, configured to restart when the Docker service is restarted and told where to read the environment variables from. The environment variables are in a local file called `.env`:

```
DATABASE_URL=...
SPACES_KEY=...
SPACES_SECRET=...
SPACES_URL=https://ams3.digitaloceanspaces.com
```

And that's it! Greenback is now running in a Docker image instance on the application server and backs up the database to Spaces just after midnight every night.

Finally

While Greenback isn't a perfect solution, it works, is configurable, a good platform for future enhancements and should require minimal configuration to be used with other projects in the future.

Greenback is checked into a public BitBucket repository and the full code can be found here: https://bitbucket.org/findmytea/greenback/.

Listing 5

```
sudo docker pull findmytea/greenback
sudo docker kill greenback
sudo docker rm greenback
sudo docker run -d --name greenback --restart
always --env-file=.env findmytea/
greenback:latest
sudo docker ps
sudo docker logs -f greenback
```

When Will Python 2 End: An Update

Silas S. Brown warns of some of the risks with unsupported software.

In *CVu* 30.6 (January 2019) I said that, although Python 2 upstream support (i.e. security patching) was due to stop in January 2020, Ubuntu had indirectly promised to support their version until 2028.

A few things have happened since then, so my article needs an update.

Firstly, Ubuntu have clarified that their 10-year support plan for 18.04 will be only for paying customers. Support for the free version ends in 2023. It's not yet clear how Ubuntu will implement pushing out patches to paying customers only; the Python license certainly allows for this, although as 'security through obscurity' has been shown to be inadvisable, they might choose a more open approach, perhaps publishing source packages for all but binary packages for paying customers only. We don't know yet.

Unexpectedly, though, Ubuntu also included Python 2 in their April 2020 release of Ubuntu 20.04 LTS, meaning they undertook to support the free version until 2025 or paid until 2030. (Equally unexpectedly, upstream issued a 'final' release of Python 2.7 in April 2020 although they said they'd stop in January.)

Meanwhile, Red Hat Enterprise Linux 8 was released in May 2019 based on Fedora 28, which still had Python 2, and is expected to receive security maintenance until 2029 (which also benefits its Oracle and CentOS derivatives).

So the answer to the question 'when will Python 2 end' is now: 2025 on Ubuntu, 2029 on CentOS.

The Fedora distribution (which acts as Red Hat's 'cutting edge' guinea pig) has already dropped Python 2, and frankly I'm not very impressed at how the transition was handled. Fedora 30 pushed all packages to use Python 3 instead of Python 2 or be deleted. (Fedora 31 followed up by making the 'python' command refer to Python 3 by default, and Fedora 32 dropped Python 2.) That push in Fedora 30, for packages to switch to Python 3 or be deleted, unfortunately seems to have made some package maintainers do a 'rush job', marking Python 2 scripts as being 'Python 3' when they weren't. I know this happened in at least two (unrelated) packages, one of which involved my own script. To their credit, the maintainers were able to update the scripts when I raised bug reports, but I hope there weren't others I missed.

It's funny that, of all the programs I've carefully crafted, the one that actually ended up in every Linux distribution was the humble little script that performs a quick calculation of LaTeX paper-size settings for arbitrary zooming, and outputs the result using `print`. This script had been accepted by CTAN and from there it was noticed by the maintainers of TexLive, and that's how it got into every distro's texlive-extras package. But it wasn't Python 3 compatible, because I hadn't used parentheses on my Python-2 style `print` statement. The Fedora package maintainer must have been under a lot of pressure to label that script 'Python 3' when any attempt to run it under Python 3 would crash immediately with a syntax error. I didn't receive any requests for a Python-3 compatible version (although when I realised what happened, I updated the version on my home page and sent a fix to CTAN, and asked the Fedora maintainers to update to that). The moral of this one is probably don't use Fedora unless you like being at the 'bleeding edge' which sometimes hurts a bit.

It's still possible to install Python 2 on Fedora 32 if you compile it by hand: you can get both Python 2 and Pip 2 into /usr/local by compiling with:

```
./configure \
  --with-ensurepip=install \
  --enable-optimizations
make
make install
```

but you will then be on your own as far as security maintenance is concerned (perhaps look out for advisories from Ubuntu and Red Hat until 2025 and 2029, and take appropriate precautions depending on what they found).

The Python package repository 'Pip' have announced they will drop support for Python 2.7 by January 2021 (Pip 21). The announcement did not say how long the older versions of Pip will continue to be supported by their package servers after that. So if you do want to continue with Python 2 into 2021, you'd better have already done any `pip2 install` commands you need by then.

Ideally you should try to add Python 3 compatibility to your legacy Python 2 code (I wrote about this in C Vu 32.1, March 2020) but that can be a big 'ask' for a large legacy codebase that gets only occasional use. I have added Python 3 compatibility to about 80% of my code, but I still have two major scripts that require Python 2, one of which is difficult to convert because it relies on specific behaviour of Python 2's email libraries for handling mislabelled character sets, and the other hardly ever gets used anymore (and when it is, it's given 100% trusted data that I produced myself, so security is not a concern); it doesn't seem justified to take out a couple of working weeks to add Python 3 compatibility to these. But I wouldn't encourage the continued use of Python 2 unless you know what you're doing. ■

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

Greenback Backup (continued)

The Greenback Docker image is in a public repository on Docker Hub and can be pulled with Docker: `docker pull findmytea/greenback` ■

References

[1] Digital Ocean: https://www.digitalocean.com/
[2] Digital Ocean Spaces: https://www.digitalocean.com/products/spaces/
[3] AWS: https://aws.amazon.com/
[4] Terraform: https://www.terraform.io/
[5] Docker: https://www.docker.com/
[6] DigitalOcean Marketplace: https://marketplace.digitalocean.com/
[7] BitBucket Pipelines: https://bitbucket.org/product/features/pipelines
[8] Docker Hub: https://hub.docker.com/

The Trouble with GitHub Forks

Silas S. Brown describes a problem with stale copies.

Originally, 'forking' a project meant taking a copy of that project for independent development (think OpenOffice versus LibreOffice). Many developers saw it as a last resort, since, although it is possible for new features and fixes to be applied to both versions, this takes extra effort (especially as they diverge), and frequently one version ends up lagging behind. Nevertheless, the ability to fork is an important test of a project's freedom.

GitHub provides a one-click 'fork' option on their Web interface: you can be looking at somebody else's project, and click to 'fork' it, copying it into your own GitHub account in its current state. If you want to propose a change to a public project for which you don't have commit rights, the standard method is to create a 'fork', change your fork, and then create a 'pull request' which is delivered to the upstream maintainers for their consideration. Like the creation of forks, the creation and review of pull requests can be done via the Web interface; so far so good.

The problem is what happens to a fork after the pull request is done, or if someone decides they don't want to make changes after all. I'll write this from the viewpoint of the upstream maintainer. I publish a project on GitHub, and somebody forks it. Fine. Then I find and fix a bug. Now, their fork still has the bug which I have fixed upstream. They don't bother to update, and their version is visible to the public. Now, I don't mind my embarrassing mistakes being preserved for historians, but it should be made clear that this is an old version and has been fixed. Although the 'desktop' version of GitHub's Web interface does say that their project is forked from mine, and is N commits behind, the mobile version does not make this very clear at all, and anyone landing on their account instead of mine might think their outdated version is the latest when it's not.

GitHub makes it easy to click to create a fork, but there's nowhere to click to update that fork to upstream when there are no conflicts. You must do it from the command-line (Listing 1), which most people who keep forks in their accounts don't seem to know.

```
git clone git@github.com:ME/REPO.git
  git remote add upstream https://github.com/
  AUTHOR/REPO.git
  ...
  git fetch upstream
  git rebase upstream/master
  git push
```

Listing 1

Moreover, I as the upstream maintainer have absolutely no way of sending a courtesy message to the people who've forked me to let them know that a fix is available, unless I sleuth out their email and hope to make it past their spam filters. I cannot raise an issue against their fork (since most forks have issues disabled), and so far there is no such thing as a 'reverse pull request' from the upstream project to the fork. When they copy my code into their own (non-fork) projects, it might be harder for me to find them, but when I do, at least I stand a chance of getting a message through like 'please update to latest upstream version, here, have a pull request'. But when they fork me, I can do nothing except sit here and hope they notice the tiny message (only on the desktop version of GitHub) telling them how many commits their fork has fallen behind.

when they fork me, I can do nothing except sit here and hope they notice the tiny message telling them how many commits their fork has fallen behind

I myself have forked other people's projects, but I generally delete the fork from my account when it is no longer necessary (e.g. my pull request has been accepted and I'm not planning on making another soon). I don't use forking as a protection against the rare event of an original repository being deleted by its maintainer, since an easier way to do that is simply to clone it on your local machine and type 'git pull' every so often to keep up-to-date (and then consider republishing only if they actually do delete it; no need to clutter up your own account with 'just in case' copies). But I do think the whole idea of GitHub forking would work much better if they made it easier to keep forks up-to-date, perhaps even providing an option to do so automatically (which people can disable if they know what they're doing). ■

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

ACCU York – May 2020

Frances Buontempo reports from a local ACCU group.

ACCU York held a virtual workshop on May 28th. Gareth LLoyd led the session. We were given a coding challenge, which has been used has an interview question:

> Given a string, like "Can you reverse these words?", write code to reverse the words in the string to get "words? these reverse you Can" without extra memory allocation.

That caused a degree of discussion, and may have been off-putting for people more used to other languages. With hindsight, I wonder if we could have first reversed the words in a string by any means, then maybe if people had used a vector to copy the words into, then upped our game to avoid the extra copies.

We discussed possible approaches and somebody waved a copy of K&R around (https://en.wikipedia.org/wiki/The_C_Programming_Language: *The C Programming Language* by Brian Kernighan and Dennis Ritchie). I love this book, but it set me thinking about C-style pointers to solve problems.

We were encouraged to try out the problem in Godbolt: https://godbolt.org/z/HpVcXw This contained a main and a function to fill in.

Had I followed instructions and done what I was told, passing standard strings around may have stopped me making a mess on paper with a pen, attempting to come out with neat K&R style `char *s` and `char * d` algorithms. Furthermore, the assert would have a provided a way to check what I was doing. Oh well.

Most of us came out with something similar – using an index which we had to increment, or a pointer, to walk the string. Without spoiling the workshop for people who haven't done this yet, the point emerged that C++ has many algorithms that make your life much easier.

Listing 1

```
#include <string>
#include <cassert>
void reverseWords(std::string &) noexcept
{
  // code here
}
int main()
{
  using namespace std::string_literals;
  auto str = "Can you reverse these words?"s;
  reverseWords(str);
    // Use no extra allocated memory
  assert(str == "words? these reverse you Can");
}
```

We had a long discussion afterwards. In particular, using the 's' suffix, from the `string_literals` namespace led to a detailed talk about argument dependent lookup – ADL. Again, this may have put off some newer C++ programmers. However, the flow of the examples and chance to talk was very well balanced. If you've not had the chance to attend ACCU York yet, the next meetup will probably be online too – join in. They're a friendly bunch.

FRANCES BUONTEMPO

Frances has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com

Static Analysis in GCC and Clang

Silas S. Brown shares some experiences of analysing code.

Did you know that GCC version 10 (available in Debian 10, Fedora 32, etc) has a flag called `-fanalyzer` that turns on static analysis?

Clang has had a 'Clang Static Analyzer' since at least 2011, although this is not always included in the default Clang distribution (in GNU/Linux it's usually packaged separately under 'clang-tools' or 'clang-analyzer'). It can be run with the command `scan-build make` as long as your Makefile uses `$(CC)` and `$(CXX)` to compile C and C++ with the default compiler.

Static analysis can be likened to having extra-paranoid warnings. It can 'false positive' and warn about things that are not in fact problems. I was incorrectly warned by GCC 10.1.1 that Listing 1 leaks memory (it didn't seem to realise the static global was kept for later), and Clang gave me a 'use after free' warning on a branch that gives a pointer to `realloc()` but uses the original pointer if `realloc()` returned `NULL` (Clang's analyser seemed to think `realloc()` could both return `NULL` and also free the pointer, which is not according to the standard). Clang also warned me about null pointer dereferencing because it hadn't realised a function I'd called would, in the event of null pointer, return a false value that would stop the code from going down the branch that uses the pointer. (That function was in another module, so the analyser couldn't check it and had to assume it might return true when given null.) But the output of a static analyser can still be worth checking and thinking about: it found a couple of unused assignments in my code for example, and one place where there could be a division by zero given invalid user input.

```
#include <stdlib.h>
typedef struct { int a; } S;
static S *sList;
static int sLen;
int alloc_sList(int n) {
  sList = calloc(n,sizeof(S));
  if(!sList) n = 0;
  sLen = n;
  return n;
}
```

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 Standard Report

Guy Davidson reports on the latest developments.

Since my last report, several C++ committee groups have been meeting for regular telecons. SG14 has held a monthly telecon for over five years now. Now the Library Evolution Working Group (LEWG) and the Evolution Working Group (EWG) are meeting weekly. As I mentioned last month:

> Library Evolution meetings are scheduled for alternating Mondays at 15:00 UTC and Tuesdays at 17:00 UTC, while Evolution meetings are scheduled for alternating Thursdays at 17:00 UTC and Wednesdays at 15:00 UTC. This works out at early morning for the US West Coast, lunchtime for the US East Coast, teatime for the UK and dinnertime for Europe (or thereabouts).

This pattern has held and is working well. There is one problem for your author: as I also suggested last month, making time in the working day for two 90-minute meetings a week is rather tough, and I have had to set EWG participation aside in favour of attending the LEWG telecons. I have library papers in flight and I should keep my hand in the process.

There are some large issues being discussed in LEWG at the moment. Let's start with the modularisation of the standard library, https://wg21.link/P0581. Unless you are new to C++ or you have been living somewhere with no internet, you are probably aware that C++20 saw the introduction of modules to the language. This feature brings more semantic heft to the matter of repeated declarations, hitherto supported only by the preprocessed inclusion of header files. The good folk of cppreference.com are gamely assembling their C++20 documentation, and you can check out their current efforts on describing modules at https://en.cppreference.com/w/cpp/language/modules. Modules introduce a way of creating components, and there is hope that it may improve compilation times as well by mimicking the behaviour of precompiled headers.

now that we have modules, we should put them to work in the standard library and practice what we preach

Of course, now that we have modules, we should put them to work in the standard library and practice what we preach. The question that needs answering is "how do we divide up the standard library into modules?" Do we have one big module for the entire standard library, or a few modules for IO, STL and so on, or lots of small modules at the per-container level?

After 90 minutes of discussion, voting revealed that we don't like intermediate sized modules, but we like the idea of both fine-grained modules and one module for the entire library. We will have to decide which later. There was also an interest in having a single module for the freestanding implementation. Of course, the biggest problem is that there is no field experience of modules, and making decisions about modularising the standard library is possibly unwise before we accumulate that.

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.

Let's stay with freestanding and consider the developments there, which you can review at https://wg21.link/P1641 and https://wg21.link/P1642. Freestanding implementations don't have an OS, which can make things like allocation and exception handling difficult and require the creation of a separate dialect of C++. This is not a great idea, as this bifurcates the language. There is a simple solution: mark parts of the standard with // freestanding to highlight that it should be available to freestanding implementations as well as to hosted implementations. So far, the [utilities], [ranges] and [iterators] clauses have been reviewed for qualification as freestanding components.

Another big-ticket item that needs library support is coroutines. Experimental support for coroutines has been available in the Microsoft implementation for a while now, and one thing that has emerged is that it would be useful to be able to constrain a generic function to only accept parameters to which you can apply the `co_await` operator. https://wg21.link/P1288 seeks to provide a way to offer that to the standard library. The use of the word constrain should highlight that this is achieved through the definition of appropriate concepts.

Concepts both satisfy and constrain, so the idea of introducing one to the standard is slightly unnerving: once a concept symbol is in the library, it can't be relaxed or tightened. Given that, as with modules and coroutines, we don't have much field experience of concepts, the idea of proposing a feature which makes use of two new features seems slightly hazardous.

This proposal contains some concepts, type traits and helper functions. The review focused on the concepts. `co_await` cannot be used in an unevaluated context like decltype because it is context dependent: the result of a `co_await` expression depends on things like the promise type. This makes it devilishly hard to write a concept that accurately answers the question "Can I `co_await` on this thing?" Ultimately, the paper was welcomed but needs another revision to more clearly address these matters.

Finally, a matter close to my heart: linear algebra. There are two papers in flight on this subject, https://wg21.link/P1385 and https://wg21.link/P1673. The first of these has some involvement from me and has yet to receive LEWG review. The second is a proposal to insert a BLAS interface into the standard library.

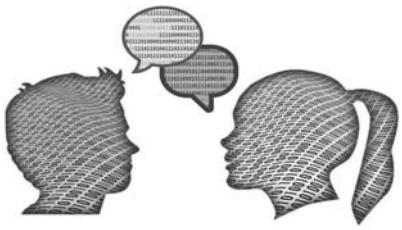
The problem it is trying to solve is that the BLAS is supremely battle hardened, but it has achieved this through age and has an inappropriate API for C++. The interface has already been standardised with bindings for C and FORTRAN, but that has baked the types on which it operates into the API. The function signatures are unintuitive and unhelpful, with names like DGEMV, taking eleven parameters, for multiplying a matrix by a vector composed of elements of type double.

The paper proposes function templates with more meaningful names, such as `triangular_matrix_vector_product` instead of DTRMV. It is well on its way to completion: it has 13 authors and is fully worded.

Next time I'll cover more of the LEWG reviews, and also cover the work EWG if there are sufficient papers of general interest. There may also be developments on my own paper.

Code Critique Competition124

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.

Last issue's code

A rather shorter critique than usual this time, and in C, to give some variety.

> Please find a bug in my code based on finding whether a positive integer can be expressed as pow(a, b) where a and b are non negative integers and b>1. My code outputs for 1000 (10^3), 0 where it should be 1. Please find the bug…

Thanks to Francis Glassborow for passing this example on to me, originally from alt.comp.lang.learn.c-c++. He also points out that this competition is now in its 21st year (he originated it). That means it has been running for longer than most software/developer magazines.

The code is in Listing 1 (`pow.c`) and Listing 2 (`test.c`). Can you help?

Listing 1

```
/**
  * @input A : Integer
  *  * @Output Integer
  */
int isPower(int A) {
  if(A==1) return 1;
  else
  {
    int i;
    if(A%2==0)
    {
      for(i=2; i<=sqrt(A); i=i+2)
      if(pow(i, (int)(log(A)/log(i)))==A)
      return 1;
    }
    else
    {
      for(i=3; i<=sqrt(A); i=i+2)
      if(pow(i, (int)(log(A)/log(i)))==A)
      return 1;
    }
    return 0;
  }
}
```

Listing 2

```
#include <stdio.h>
int main(void)
{
  int A = 1000;
  printf("isPower(1000) => %i\n", isPower(A));
  return 0;
}
```


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

Dave Simmonds <daveme@ntlworld.com>

First of all lets tidy up a bit.

These issues would be pointed out by any decent compiler – we need to declare the functions we are using.

Listing 2 (Listing 1 in this edition of *CVu*) needs:

```
#include <math.h>
```

Listing 3 (now Listing 2) needs:

```
extern int isPower(int);
```

The bug is that when we divide the **log**s, we get a **double** which we convert to an **int**. Rounding here gives us too low a number.

So, if we add 0.5 before rounding down to an **int**, we should be good:

```
if (pow(i, (int)(log(A)/log(i) + 0.5)) == A)
```

This solves the particular issue as reported.

Two other issue with **int**/**double** conversions:

- **pow** returns a **double** and we truncate to **int** – that could give us another rounding issue. The easiest solution would be to write a version of **pow** that only deals with **int**s.
- **sqrt** returns a **double**, we then compare with **int** – would be reasonable to add 0.5 here too.

Other criticisms:

The tabbing – **for**, **if**, **return** all at same tabbing level.

i=i+2 would be better as **i+=2**.

We repeat the same loop with a different starting point – let's just set the starting point appropriately and use the same code.

```
#include <math.h>
static int intpow(int a, int b)
{
  int r = 1;
  while(b--)
    r *= a;
  return r;
}

int isPower(int A)
{
  if (A == 1) return 1;
  else
  {
    int i = 2;
    if (A%2)
      i = 3;
    for (; i <= (int)(sqrt(A) + 0.5); i+=2)
    {
      if (intpow(i, (int)(log(A)/log(i) +
          0.5)) == A)
        return 1;
    }
  }
  return 0;
}
```

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

Computers can only approximate real numbers. Not all numbers can be calculated or represented exactly. This can cause problems when performing mathematical calculations. For example, performing the calculation 1.0 / (0.1 / 0.3) on my machine resulted in 2.9999999999999995559. The result should be 3 exactly.

A similar problem occurs when the supplied program is executed. Part of the program calculates `log(A) / log(i)` where `A` is 1000 and `i` is 10. The result is 2.9999999999999995559 (on my machine) and is not the mathematically correct value of exactly 3. After performing this calculation, the program takes the integer part of the result (which is 2) and raises `i` to this value and compares the result with `A`. In other words, 10^2 is not equal to 1000 and so the program erroneously concludes that 1000 is not equal to 10^3.

One possible solution to this problem is to add a very small amount (such as 0.00000000000001) to `log(A) / log(i)` to ensure the integer part of the number is correct. The question now is can adding this small amount cause any other problems. This is difficult to answer and so I leave it to the reader to ponder.

There is quite a bit of duplicated code in the supplied program that can be reduced by writing `isPower()` as shown below.

```
int isPower(int A)
{
  if (A == 1)
    return 1;
  else
  {
    int i;
    for (i = A % 2 == 0 ? 2 : 3; i <= sqrt(A);
         i = i + 2)
      if (pow(i, (int)(log(A) / log(i) +
          0.00000000000001)) == A)
        return 1;
  }
  return 0;
}
```

Also, for clarity, statements in the original code should be indented to show that the `return` statement is part of the `if` statement and that the `if` statement is part of the `for` statement, for example.

Hans Vredeveld <accu@closingbrace.nl>

By just compiling the code with gcc, the compiler will already inform you of the first problem: all the mathematical functions in pow.c and the function `isPower` in test.c are implicitly declared. According to the C90 standard, the compiler should assume a declaration of the form `int f();` that is, a function that takes any number of arguments, also of any type, and that returns an `int`. When it comes to the mathematical functions, gcc is a bit smarter and knows that, for example, the signature of `sqrt` should be `double sqrt(double)` and generates object code accordingly. C99 made implicitly declared functions an error, but gcc still treats them as a warning.

To improve the code and to make it C99 compliant, we have to add an `#include <math.h>` to `pow.c` and a declaration `int isPower(int)` to `test.c` (or better, put that declaration in a file `pow.h` and `#include` `pow.h` to both `test.c` and `pow.c`, so the compiler can see that the definition matches the declaration).

One thing to keep in mind when working with floating point numbers is that many numbers cannot be represented exactly, but only approximated. This also goes for the result of calculations. We should therefore consider two floating point numbers equal when their difference is small. What is small depends on the magnitude of the numbers.

Also note that casting a `double` to an `int` truncates. So (int)2.99999999999999955591 results in 2, not 3.

Given the definition

```
const double margin = 10 * DBL_EPSILON;
```

(where `DBL_EPSILON` is defined in `float.h`) and introducing the functions `toInt` and `isEqual` as

```
static int toInt(double d)
{
  return d * (1.0 + margin);
}
static int isEqual(double a, double b)
{
  return fabs(a - b) <
    ((a + b) * (1.0 + margin));
}
```

this means the following for the code in `isPower`:

- `i<=sqrt(A)` has to be replaced by `i <= toInt(sqrt(A));`
- `(int)(log(A)/log(i))` has to be replaced by `toInt(log(A) / log(i));`
- `pow(…)==A` has to be replaced by `isEqual(pow(…), A)`.

These replacements have to be made in both branches of the `if(A%2==0)` statement. Looking closer at these branches, we see that the only difference is that in the `for`-loop `i` is initialised to 2 when `A` is even and to 3 when `A` is odd. We can remove the `if`-statement and one of the branches when we change the initialisation in the remaining `for`-statement to `i = (A%2 == 0) ? 2 : 3`.

Some other improvements to `isPower` are:

in the first `if`-statement the `else` keyword is not needed; with or without it, the code is only executed when `A != 1`;

`i` is only used in the `for`-statement, and can be declared in the initialisation of the `for`-statement instead of before it;

instead of `i = i + 2`, write `i += 2`, to better signal to a reader that we are only incrementing `i` and don't do something more fancy with it;

properly indent the body of the `for`-statement, so it is immediately clear what is executed as part of what;

as an optimisation, calculate `toInt(sqrt(A))` once before the loop, instead of in each iteration;

as an optimisation, calculate `log(A)` once before the loop, instead of in each iteration.

With all these changes, the complete function now looks as follows:

```
int isPower(int A) {
  if (A == 1) return 1;
  int root_A = toInt(sqrt(A));
  double log_A = log(A);
  for (int i = (A % 2 == 0) ? 2 : 3;
       i <= root_A;
       i += 2)
    if (isEqual(pow(i, toInt(log_A / log(i))),
        A))
      return 1;
  return 0;
}
```

Finally a note on the doxygen-like documentation. It only says that `isPower` takes an integer input named `A` and outputs an integer. This can already be inferred from the function signature. A more useful documentation would state what `isPower` does, what the restrictions are for the input parameter and what it represents, and what possible values the function can return and what those values mean. A possible improvement of the documentation is

```
/**
 * Find whether a positive integer @c A can be
 * expressed as pow(a, b), where a and b are
 * non-negative integers and b > 1.
 * The function returns 1 when such numbers a
 * and b exist, and 0 otherwise.
 */
```

Marcel Marré <marcel.marre@gerbil-enterprises.de> and Jan Eisenhauer <mail@jan-ubben.de>

The main issue with the code is that it mixes the use of floating point numbers with that of integers. For computers, these are two very different things. Often, both need roughly the same memory, but have very different requirements, leading to very different and at times surprising behaviour.

Both in mathematics and computer floating point numbers, there may be more than one way to write a number. For example, 9.99 recurring is actually the same as 10, being 9 + 9/9. If you naively took the integer part of 9.99 recurring, you might say this is 9. And that is what happens in this case.

Generally, to find a bug it is a good idea to narrow down where the error lies either in an interactive debugger or by adding debug output. We added debug outputs displaying `log(A)/log(i)` versus `int(log(A)/log(i))`. And for the case of `A` = 1000 and `i` = 10 the float value is displayed as 3.000000, but the integer as 2, showing the exact problem described above.

Also note that the given code, irrespective of rounding errors, yields `isPower(2) == 1`, which is incorrect.

One could work with proper rounding or checking whether either `ceil(log(A)/log(i))` or `floor(log(A)/log(i))` yield the desired result, but while this will work for `A` = 1000, this could still go awry for larger numbers. We decided to take an approach using integers exclusively. Rather than using `log` this means trying to find a suitable base and exponent. Note that we use a `long int` where we expect a possible overflow.

```
#include <limits.h>
#include <math.h>
#include <stdio.h>

int isPower(int A) {
  if (A == 1) {
    return 1;
  } else {
    long int a;
    for (a = 2; a * a <= A; a = a + 1) {
      if (A % a == 0) {
        int e = 2;
        long int p = a * a;
        while (p < A) {
          e = e + 1;
          p = p * a;
        }
        if (p == A) {
          // printf("%i = %i^%i\n", A, a, e);
          return 1;
        }
      }
    }
    return 0;
  }
}
```

We also developed a version that uses binary search to find the right base within a linear search to find the right exponent, which, for the numbers 1 to 1,000,000 was 24x faster than the `float` version in addition to being correct. However, we feel this is beyond the scope of this code critique.

Ovidiu Parvu <accu@ovidiuparvu.me>

Before making any (potentially breaking) changes to the `isPower()` function unit tests were implemented. The unit tests have been implemented using the [Check](https://libcheck.github.io/check/) unit testing framework for C and are given in the `unit_tests.c` file included at the end of this submission.

Some of the implemented unit tests failed to run successfully (as expected). Any issues highlighted by the failing unit tests and issues found during code inspection, together with the suggested changes, are described in the Subsections below.

Issue 1: Numerical approximation errors

The main issue with the implementation of the `isPower()` function is that numerical approximation errors that can be introduced when representing real numbers using fixed-width floating point variables are not taken into consideration. Specifically when calling `isPower(1000)` the following numerical approximation errors occur:

```
log(1000) = 6.907755278982137
log(10) = 2.302585092994046
log(1000)/log(10) = 2.9999999999999996
 // Should be 3.0
(int) log(1000)/log(10) = 2
 // Should be 3
pow(10, (int) log(1000)/log(10)) = 100
 // Should be 1000
```

Such numerical approximation errors can be avoided in the `isPower()` function by rounding the result of the logarithm values division, namely replacing `log(A)/log(i)` expressions with `round(log(A)/log(i))`.

Issue 2: Not handling non-positive input values correctly

A second issue is that the code does not explicitly handle non-positive input values correctly. If the `isPower()` function is called with a value of zero then it returns an incorrect result of 0 instead of 1 (given that '$0^2 = 0$'). Also if the `isPower()` function is called with a negative value then that negative value is passed into the `sqrt()` function, which would result in a domain error (as per https://www.cplusplus.com/reference/cmath/sqrt/).

To avoid this issue, two conditional statements can be used at the beginning of the `isPower()` function to handle any input values less than or equal to 1:

```
int isPower(int A) {
  if (A < 0) return 0;
  if (A <= 1) return 1;
  ...
}
```

Issue 3: Code duplication

The `isPower()` function contains two loops which are identical except for the value with which the variable `i` is initialized. By extracting the code used to initialize `i` outside of the two loops, the two loops can be replaced by a single loop as follows:

```
int isPower(int A) {
  ...
  i = (A % 2 == 0) ? 2 : 3;

  for (; i <= sqrt(A); i = i + 2) {
    if (pow(i, (int)(round(log(A) / log(i))))
      == A) return 1;
  }
  ...
}
```

Other minor issues

Other minor issues that can be addressed are:

- Including the missing `<math.h>` header in `pow.c`.
- Declaring the `isPower()` function in a `pow.h` header file in order for the function declaration to be visible in `test.c` (after including `pow.h` in `test.c`).
- Rewriting the comment corresponding to the `isPower()` function declaration such that it explains what the function does rather than state what type the result and input parameter have, which is already recorded by their corresponding types.

Code listings

The code that includes all the improvements presented above and the corresponding unit tests are given below.

```
--- pow.h ---
  /**
    * Returns:
    * - 1, if A is a positive number that can be
    * expressed as A = a^b,
    * where a > 0 and b > 1.
    * - 0, otherwise.
    */
  int isPower(int A);
--- pow.c ---
  #include <math.h>
  #include "pow.h"
  int isPower(int A) {
    int i;
    if (A < 0) return 0;
    if (A <= 1) return 1;
      i = (A % 2 == 0) ? 2 : 3;
    for (; i <= sqrt(A); i += 2) {
      if (pow(i, (int)(round(log(A) / log(i))))
          == A) return 1;
    }
    return 0;
  }
--- test.c ---
  #include <stdio.h>
  #include "pow.h"
  int main()
  {
    const int A = 1000;
    printf("isPower(%i) => %i\n",
      A, isPower(A));
  }
--- unit_tests.c ---
  #include <limits.h>
  #include <math.h>
  #include <stdlib.h>
  #include <string.h>
  #include <check.h>
  #include "pow.h"
  int logarithm(int base, int value) {
    return log(value) / log(base);
  }
  START_TEST(test_is_power_negative) {
    ck_assert_int_eq(isPower(INT_MIN), 0);
    ck_assert_int_eq(isPower(-4), 0);
    ck_assert_int_eq(isPower(-1), 0);
  }
  END_TEST
  // [Editor] additional tests for
  //   test_is_power_zero,
  //   test_is_power_one,
  //   test_is_power_positive,
  //   test_is_power_one_thousand, and
  //   test_is_power_non_negative_numbers
  // elided, to save space
  Suite *power_suite(void) {
    Suite *s;
    TCase *tc_core;
    s       = suite_create("isPower");
    tc_core = tcase_create("TestCase");
    tcase_add_test(tc_core,
      test_is_power_negative);
    // [Editor] likewise for the elided tests
    suite_add_tcase(s, tc_core);
    return s;
  }
  int main(void) {
    int      number_failed;
    Suite *  s;
    SRunner *sr;
    s  = power_suite();
    sr = srunner_create(s);
    srunner_run_all(sr, CK_VERBOSE);
    number_failed = srunner_ntests_failed(sr);
    srunner_free(sr);
     return (number_failed == 0) ? 0 : 1;
  }
```

Paul Floyd < paulf@free.fr>

First, a superficial answer. The reason that this code does not work is floating point rounding. Both **pow** and **log** take **double**s and thus have limited precision. In this case a cast is used

```
(int)(log(A)/log(i))
```

which means that the rounding used is truncation. If, due to the limited precision, the result of the log division is a tiny bit less than a full integral value (say 4.99999) then it gets truncated to 4 rather than the 'correct' 5 that might be hoped for.

The quick fix would be to use some tolerance. So the pseudo-code would be something like

```
if (abs(round(log division) - (log division))
    < magic tol) {
  // use round to nearest and test like before
} else {
  // not really a candidate anyway
}
```

Problem solved?

Not really, in my opinion. What should be used for **magic tol**? If your name is William Kahan you might be able to work out exactly what the upper bound is for the error in the log division expression. If I were to attempt this then I'd probably guess something around 10 units in the last place (ULP) and then try to test this assumption. My feeling is that it would work reasonably well for **int** but would have problems if ever a **long int** version were required. The danger is that if you choose a **magic tol** that is too lax then close misses might become false positives.

To avoid any issue of floating point precision, I would probably solve this in one of two ways:

- use repeated integer division
- use a reverse lookup table. If you can afford 384 kbytes of memory then you can construct a table of all 48,036 possible candidates for a 32bit int, and then use a binary search. That's assuming you allow inputs up to **INT_MAX**, but again doesn't extend well to **long int**.

Usual nits: missing <math.h> header.

Commentary

This critique is quite short, but does demonstrate several problems.

The first problem is the behaviour of a C program when there is no function prototype provided, which occurs when:

- **isPower()** uses **log()**, **pow()**, or **sqrt()**
- **main()** uses **isPower()**

The C standard explicitly encourages a warning, but warnings are easy to ignore... and the behaviour of *this* program is undefined. (C++ solves this problem by not allowing undeclared functions to be called.)

For the case of the functions **log()**, **pow()**, and **sqrt()** that are defined in <math.h>, some compilers (eg. gcc) try to be helpful and implicitly provide the function prototypes while other compilers (eg. MSVC) do not – and so they follows the C rule of assuming the functions return an **int**. They also does not know what the argument types are and so perform default argument promotions. The result is almost meaningless as the binary values passed will be misinterpreted by the called function, and the return value will be misinterpreted by the caller!

In the case of **isPower()**, since the actual arguments and the implied return type *do* match the actual function definition, this code is valid – but potentially risky, as if the function signature changes the program could misbehave.

The second problem is the mix of floating point and integer types resulting in a loss of precision. While many integer values will convert to an exact floating point value, calculations done in floating point will often end up very close to the exact value, but not exactly equal.

As the critiques observed, the troubling calculation here is `(log(A)/log(i))` as we need the exact **integer** result from the `(int)` cast to give the correct answer, and this is not in general obtained. Fortunately C provides functions that will round a floating point value to the nearest integer, which is exactly what we need here, so a solution using `round()` rather than the cast would resolve this.

The third problem is the failure of the algorithm to deal correctly with values less than one. This may not be a problem that needs fixing, depending on the values expected to be encountered in practice, but if *not* fixed then the preconditions on the values to provide are neither stated nor checked. Defensive programming practices would encourage a check of some kind.

The fourth item I was expecting someone to comment on was the optimisation of only checking even candidates for even numbers and odd candidates for odd numbers. While this is a sensible micro-optimisation I would have liked to see the addition of an explanation – or a meaningful variable name – to help with understanding this code.

The Winner of CC 123

All the critiques identified the problem with the floating point value being truncated to an integer value different from the mathematically correct value.

James, Hans, and Ovidiu all show the numeric approximation (2.990...) that causes the problem in the case presented, which I think is useful as it helps understanding the issue when you see what's actually going on. (Of course, the decimal expansion they each show is in turn only an approximation to the exact value of the floating point number.)

Dave lists the missing `<math.h>` as the first thing to fix, which is good, although he didn't explain *why* it should be fixed. While Hans did recognise that gcc was being helpful in providing an implicit prototype, no-one drew attention to the danger of the missing `<math.h>` if the code is compiled with a less 'helpful' compiler.

Several people eliminated the code duplication of the repeated loops, and either implicitly or explicitly tidied up the poor formatting of the original code.

Hans and Ovidiu both suggest giving some meaning to the 'doxygen-like' comment at the heading of the function. This would allow documentation of the preconditions and semantics of the function.

Marcel and Jan helpfully describe the process of narrowing down the error by, in this case, adding debug print statements. This may well be something the writer of the code would find useful both for this problem but also for their *future* debugging.

I like Ovidiu's approach of starting by producing some unit tests – this helps to think about a wider range of use cases before prematurely *just* fixing the presenting case!

I couldn't reproduce Marcel and Jan's remark that the original code produced 1 for `isPower(2)`; but since the original program contains undefined behaviour this is a possible outcome!

I thought the mention of alternatives such as doing the whole calculation with integers, using a combination of binary and linear search, or using a lookup table, was good; for cases like this where the inputs and the outputs are all integers avoiding floating point may spare a great deal of head scratching. Additionally, Hans provided a couple of caching opportunities that would reduce the cost of the algorithm.

The critiques all provided a lot of illumination of the code and how to improve it. Overall I decided that Ovidiu's contribution wins the prize, by a short head.

Listing 3

```
#include <iomanip>
#include <iostream>
#include <locale>
#include <sstream>

int main(int, char**argv)
{
  int t; // total
  std::string ln;
  std::string m;
  if (argv[1])
    std::cout.imbue(std::locale(argv[1]));
  std::cout << "First item: ";
  std::getline(std::cin, ln);
  std::istringstream(ln) >> std::get_money(m);
  t = std::stoi(m);
  std::cout << "Next item: ";
  while (std::getline(std::cin, ln))
  {
    if (std::istringstream(ln) >>
      std::get_money(m))
      t += std::stoi(m);
    std::cout << "Next item: ";
  }
  std::cout << std::showbase <<
    "Total: " << std::put_money(t) << std::endl;
}
```

Code Critique 124

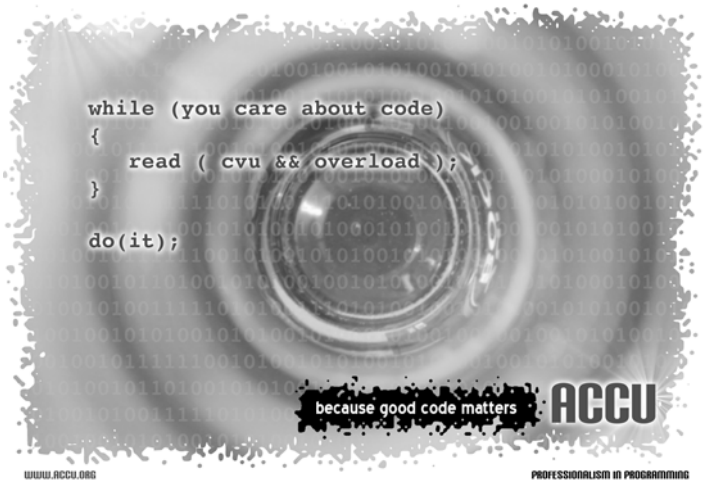
(Submissions to scc@accu.org by Aug 1st)

> I am trying to write a little program to add up amounts of money and print the total, but it isn't working as I expected. I realise I need to use a locale but I think there's something I don't understand.
>
> I run the program like this:
>
> ```
> $ totals en_GB
> First item: 212.40
> Next item: 1,200.00
> Next item: ^D
> ```
>
> And I get this result:
>
> ```
> Total: £2.13
> ```
>
> I wanted to get:
>
> ```
> Total: £1,412.40
> ```
>
> Can you help? "

The code (`totals.c`) is in Listing 3.

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.



Not quite the reaction you were expecting to the latest release?

Clearly stated...

It may not be the software. How clear are the release notes? What about the product manual, online help, training materials, ...?

Changes may meet a business need, but if what worked yesterday doesn't work today, people may resent them. And when today's way involves extra steps, people work around them. After all, their priority is getting the job done.

Result: those shiny new features remain unused, and your application appears not to live up to its promise.

If you would like some help in turning nervous cats into contented ones, get in touch.

T 0115 8492271

E info@clearly-stated.co.uk

W www.clearly-stated.co.uk

We share your belief in professionalism and are members of the Institute of Scientific and Technical Communicators, the UK professional body for technical authors and related professions (visit www.istc.org.uk)

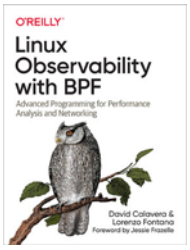
Reviews

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.

Linux Observability with BPF: Advanced Programming for Performance and Networking



By David Calavera & Lorenzo Frazelle, published by O'Reilly, website: github.com/bpftools/linux-observability-with-bpf, ISBN: 978-1-492-05020-9

Reviewed by Ian Bruntlett

Verdict: I liked it. Mind-bending but good.

BPF is a technology that emerged out of bpf and eBPF. Whilst not a kernel expert, I am a programmer and I bought this book with a view to learning what goes on inside the kernel and to help when testing pre-release versions of Ubuntu/Lubuntu Linux. This slim book (162 pages) that brings many things together in one place. It has its source code on the book's GitHub page. Warning: this is bleeding edge stuff.

The 'Introduction' chapter covers the history and architecture of bpf. It turns out that bpf programs have certain limitations which in turn makes certain guarantees possible.

The 'Running your first BPF program' chapter has a short example that has to be compiled using LLVM's C compiler. It then goes on to catalogue the different BPF program types and gives info on the BPF verifier and other details.

The 'BPF Maps' chapter is highly technical and fiddly – it describes an illustrates the BPF Map API where a Map can be some arbitrary key-value pairs – other data structures – confusingly still called Maps – have been added to BPF over time. Finally, it explains the BPF Virtual Filesystem.

The 'Tracing with BPF' chapter is all about efficiently collecting data from the Kernel and User-Space programs for debugging and profiling. It introduces the BPF Compiler Collection (BCC) and mostly uses Python programs to provide hosts for BPF programs. It goes into detail about tracing probes which it defines as 'exploratory programs designed to transmit information about the environment in which they are executed'. This sounds dangerous but the BPF verifier validates code before it is accepted for execution. It finishes with two ways to visualise accumulated data – flame graphs and histograms.

The 'BPF Utilities' chapter covers some very interesting tools. BPFTool is an awesome tool for working with BPF programs and the data they generate. In Ubuntu, searching the package repositories for BPFTool found nothing but later on I found it is provided by the linux-tools-common package. BPFTrace implements a concise, domain-specific-language for writing BPF programs – it provides more support for the programmer than BCC does – however, it is less suited for advanced programs. It covers kubectl-trace, a kubernetes tool which I cannot comment upon. Finally, it covers eBPF Exporter, a tool to export data to Prometheus, a monitoring tool alerting system.

The 'Linux Networking and BPF' chapter is interesting but a bit beyond my experience. It covers packet filtering with BPF and illustrates how tcpdump in turn writes bpf programs. It disassembles a BPF program's byte code and explains what it does. I did notice that on page 96 there is a small problem with the Ethernet frame header offsets. It goes on to cover the Traffic Control subsystem with examples.

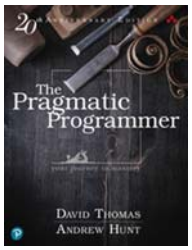
The 'Express Data Path (XDP)' chapter is more networking. It is all about running BPF programs whenever a network interface receives a packet. It is intricate, complete with useful diagrams. I was impressed by its flexibility to handle denial of service attacks. It also covers the writing and testing of XDP programs. It finishes with XDP Use Cases – monitoring, DDOS mitigation, Load Balancing and Firewalling.

The 'Linux Kernel Security, Capabilities, and Seccomp' chapter explains Secure Computing (abbreviated to Seccomp), a security layer of the kernel for filtering syscalls and provides a C program example.

The final chapter, 'Real-World Use-Cases', is a couple of anecdotes on real-world uses of BPF – Sysdig and Flowmill.

To conclude: this is a slim book that covers a highly technical subject. It has a reasonable index but no bibliography. Should you read this book? Maybe. There is another BPF book available (BPF Performance Tools by Brendan Gregg) which is considerably larger. This book has an improved awareness of BPF in the Kernel and was sufficiently brief to make reading it a manageable project.

The Pragmatic Programmer (20th Anniversary Edition)



By David Thomas & Andrew Hunt, published by Addison-Wesley, website: pragprog.com/titles/tpp20, ISBN: 0-13-595705-2

Reviewed by Ian Bruntlett

Verdict: A classic.

Physically this is a beautiful book that will endure much use. Split into 9 chapters, 53 topics and 100 tips (with a tear-out tips reference card) it is an impressive piece of work. A short Bibliography is provided. Also, some chapters have exercises and challenges for the reader – with possible answers to the exercises at the back of the book.

According to the Preface, one third of topics in this second edition are new and the majority of the rest have been re-written – either partially or totally. Also, 'Kaizen', the Japanese term that captures the concept of continually making many small improvements is introduced.

The 'Pragmatic Philosophy' chapter presents the authors philosophy regarding software development – taking responsibility for your career and the consequences of your actions; technical debt; being a catalyst for change; trade-offs in quality; and managing your knowledge portfolio; and finishes off covering different types of communication and, how to do it well.

The 'Pragmatic Approach' chapter builds on the previous chapter, but deals with issues involved in coding, backed up with example Ruby code. It boldly proclaims 'There are certain tips and tricks that apply at all levels of software

development, processes that are virtually universal, and ideas that are almost axiomatic' and that this chapter presents them to you. It covers design principles, strategies for dealing with change, the start of projects, domain languages and estimation.

The 'Basic Tools' chapter builds on the programmer-as-artisan metaphor where a maker starts with a basic set of good quality tools. It is heavily biased towards the Unix way of doing things – the importance of plain text, command shells, power editing (text), version control and text manipulation languages. More universal is its recommendation of maintaining Engineering Daybooks. The Unix command line tools have been around for a long time – and, thanks to the FSF, will hopefully will be freely available for a long time.

The 'Pragmatic Paranoia' chapter is all about dealing with everyone's imperfections. The first three topics (Design by Contract, Dead programs tell no lies, assertive programming) all help with the building of correct software. The 'How to Balance Resources' topic is all about resource management and how it interacts with scope and exceptions. The final topic 'Don't Outrun Your Headlights' advocates taking small steps – to avoid taking on tasks that are impossible to estimate.

The 'Bend or Break' chapter is all about aiming for flexible code and avoiding the creation of fragile code. In particular it discusses decoupling, building upon this book's Easier to Change principle. It has a tip, Avoid Global Data and I really think that the related principle Parameterise from Above should have been discussed. In the Juggling the Real World topic the ability to be responsive to external change / events covers finite state machines, the Observer pattern, Publish/Subscribe and, Reactive Programming and Streams. So there are plenty of things to learn from this chapter. The Transforming Programming topic raises command-line pipelines and the importance of approaching some problems as a case of taking input data, transforming it, and then outputting it – either at the Unix command-line or in an Elixir program using the `|>` operator. The Inheritance Tax topic is no fan of inheritance and discusses alternatives. I was surprised that the SOLID acronym wasn't discussed. Finally, it looks at the role of configuration for flexibility.

The 'Concurrency' chapter discusses temporal coupling and the role of time in software architectures – 'Concurrent and Parallel code used to be exotic. Now it is required'. It discusses Actors and Processes and Blackboards as potential solutions.

The 'While You Are Coding' chapter is a selection of 8 topics and 38 tips on coding and is packed full of useful advice. Amongst other things it covers refactoring, testing, and security.

The 'Before the Project' chapter discusses requirements and how developers can glean requirements for a project from the users and then build on and refine that information. It also gives advice on how to solve seemingly impossible puzzles. One topic, 'Working Together' advocates Pair Programming for the production of higher quality software. The final topic covers the essence of Agile software development.

The final chapter, 'Pragmatic Projects', is about the bigger picture and how tips from this book can be applied to a project. In its 'Pragmatic Starter Kit' topic it covers three critical things – Version Control, Regression Testing and automation of Building and Testing. The 'Delight Your Users' topic reinforces the need for developers to be exposed to many aspects of the customer's organisation.

The Postface of this book reflects on the responsibility of software developers to have a moral compass and ask "Have I protected the user?" and "Would I use this myself?". It then concludes with : "It's Your Life. Share It. Celebrate it. Build it. AND HAVE FUN!"

To conclude, this is an interesting compilation of short essays on Software Development. Well worth reading.

Letters to the Editor

Two people have written to share their thoughts.

Dear Editor,

In my article 'Thoughts on 'Computational Thinking'' (*CVu* 32(2):8-10, May 2020), I mistakenly said that Cambridge University Faculty of Education's research project on Computational Thinking is a collaboration with Cambridge Assessment. This is incorrect: Cambridge Assessment are not involved (I misunderstood). The project does hope to develop an assessment tool, but only to inform the improvement of teaching methods [1] and not for grading or segregating children as I feared. While I'd still be concerned about the dangers if such an assessment is misused, I'm pleased to report that's not their aim.

Sílas S. Brown

Reference

[1] Rina P.Y. Lai, 'The Design, Development, and Evaluation of a Novel Computer-based Competency Assessment of Computational Thinking', *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* pp.573–574, https://doi.org/10.1145/3341525.3394002

Dear Editor,

I very much enjoyed Simon Sebright's article on names and their complexity from a modelling perspective, and wanted to bring other readers attention to another classic article about names and their complexity: https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/

All the best,

Burkhard Kloss

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

JOIN THE ACCU!

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

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

With full ACCU membership you get:

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

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

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

How to join

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

Also available

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

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG