the magazine of the accu

www.accu.org

{cvu}

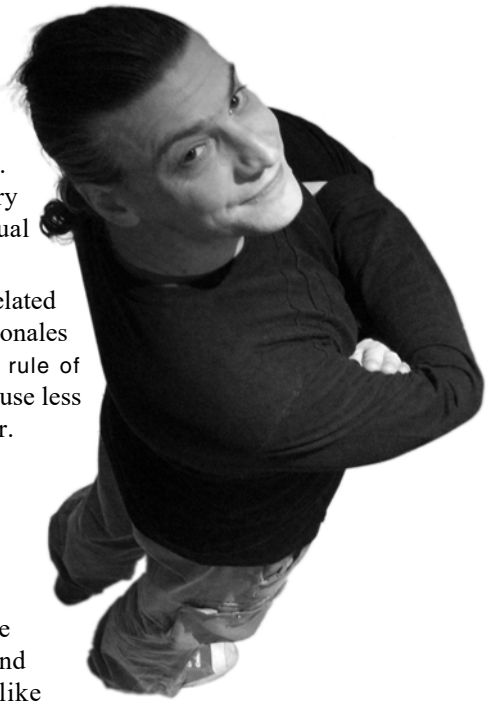
Volume 32 • Issue 4 • September 2020 • £4

Features

Regulars

{cvu}
Volume 32 Issue 4
September 2020
ISSN 1354-3164
www.accu.org

Optimal



One persistent theme in programming is that before we attempt any kind of optimisation, we should measure what needs to be optimised. This is certainly true when we're optimising for speed or memory consumption. It's rare, even for the most experienced programmers, to identify the real bottlenecks or memory hogs just by looking at source code. Performance analysis tools and run-time profilers very often find things we completely overlook as the actual culprits.

There are, however, other kinds of optimisation, not related to CPU cycles, or mega-bytes of memory. One of the rationales behind the advice often paraphrased as "the first rule of optimisation is: don't" is that altering code to be faster or use less memory very often makes the source code less clear. Source code is read more often read by humans than computers, and optimising for clarity is something we **can** do that doesn't usually require us to measure its effects.

Another target for optimisation is testability, at all levels, but especially the unit-test arena. Designing code so that it can be tested with automated tools quickly, and without needing access to environmental facilities like networks, databases and file systems, has consequences that go far beyond the tests. One side-effect is that, more often than not, the code is clearer for human readers, too.

Other targets for optimisation include ease of deployment, robustness under error conditions, supportability and uncomplicated maintenance. Not all of these things are the responsibility of the code, of course, but they all start there. Modular components make it easy to split applications into parts that can be deployed independently of each other. Supportability probably involves some kind of monitoring, with easy-to-understand logging messages. Maintenance often requires someone other than the author of some code to alter its behaviour.

There is one thing that unites them all. Before the first **rule** of optimisation is the first **target** of optimisation: clear code.

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 32.5: 1st October 2020
C Vu 32.6: 1st December 2020

Overload 159: 1st November 2020
Overload 160: 1st January 2021

WRITE FOR C VU

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

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

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Making a Linux Desktop

Alan Griffiths adds support for shell components to the desktop environment.

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

By providing the common features, and for the rest offering both sensible defaults and an easy way to customise them, Mir is designed to support a range of possible designs. This series of articles demonstrates how to make use of the facilities Mir provides.

So far, all the articles in the series show features of the egmde 'desktop environment' being implemented in the egmde program itself. But there are often reasons to add features in separate programs: these can be developed and updated separately, or even replaced to tailor the user experience.

Because such programs are part of the 'desktop environment' they often use features of the compositor that are not (or should not be) available to ordinary applications. In particular, they might need to request that they are docked to an edge of the display or appear behind (or in front) of applications. These are things a normal Wayland application isn't expected to do but are part of shell-specific Wayland extensions.

Mir provides implementations of several of these 'shell component' specific protocols, but leaves them disabled by default. In this article we're going to see how egmde enables these protocols just for shell components.

Configuring shell components

The first thing we need to do is identify the programs we want to run as shell components. In egmde, I recently added a configuration option 'shell-components' as follows:

```
CommandLineOption{
  run_shell_components,
  "shell-components",
  "Colon separated shell components to launch on
  startup", ""},
```

This, like the options we've seen in previous articles, is passed to the `run_with()` method of MirRunner. In common with other Mir configuration options, users of egmde can specify shell components on the command line, with an environment variable or in a config file. In this case, we'll just use a config file:

```
echo shell-components=waybar >>
  ~/.config/egmde.config
```

Oh, while we're at it, we ought to install 'waybar' – a simple, Wayland based, docking bar that can be used for this article (it doesn't have to be Waybar, for example, 'mate-panel' works too).

```
sudo apt install waybar
```

Identifying and running shell components

Now we need to launch the shell components on startup, and also ensure that we can identify them later so that we can allow them access to the special Wayland protocol extensions. We do that by keeping a note of their process identifiers (see Listing 1).

Listing 1

```
std::set<pid_t> shell_component_pids;
auto run_shell_components =
  [&](std::string const& apps)
{
  for (auto i = begin(apps); i != end(apps); )
  {
    auto const j = find(i, end(apps), ':');
    shell_component_pids.insert(
      launcher.run_app(
        std::string{i, j},
        egmde::Launcher::Mode::wayland));
    if ((i = j) != end(apps)) ++i;
  }
};
```

The `run_shell_components` lambda is passed to the `CommandLineOption` we saw above and handles the `shell-components` option configured. We'll use the list of `shell_component_pids` shortly.

Configuring the Wayland protocol extensions

The protocol extensions we're interested in for shell components are `wlr_layer_shell` and `xdg_output_manager`. We can, with the list of process identifiers we built earlier, enable these just for shell components as shown in Listing 2.

This configures `extensions` to enable these protocol extensions but also run all enabled protocol extensions through a filter that disables them for any processes not found in `shell_component_pids`.

The only thing left after this, is to pass `extensions` to the `run_with()` method of `MirRunner`.

Listing 2

```
// Protocols we're reserving for shell components
std::set<std::string> const shell_protocols{
  WaylandExtensions::zwlr_layer_shell_v1,
  WaylandExtensions::zxdg_output_manager_v1};
WaylandExtensions extensions;
for (auto const& protocol : shell_protocols)
{
  extensions.enable(protocol);
}
extensions.set_filter([&](Application const& app,
  char const* protocol)
{
  if (shell_protocols.find(protocol)
    == end(shell_protocols))
    return true;
  return shell_component_pids.find(pid_of(app))
    != end(shell_component_pids);
});
```


ALAN GRIFFITHS

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

Relish the Challenge

Pete Goodliffe challenges us to pick up the gauntlet.

Success is not final, failure is not fatal: it is the courage to continue that counts
~ Winston Churchill

We are 'knowledge workers'. We employ our skill and knowledge of technology to make good things happen. Or to fix it when they don't. This is our joy. It's what we live for. We revel in the chance to build things, to solve problems, to work on new technologies, and to assemble pieces that complete interesting puzzles.

We're wired that way. We relish the challenge.

The engaged, active programmer is constantly looking for a new, exciting challenge.

Take a look at yourself now. Do you actively seek out new challenges in your programming life? Do you hunt for the novel problems, or for the things that you're really interested in? Or are you just coasting from one assignment to the next without much of a thought for what would motivate you?

Can you do anything about it?

It's the motivation

Working on something stimulating, something challenging, on something that you enjoy getting stuck into helps keep you motivated.

If, instead, you get stuck in the coding 'sausage factory' – just churning out the same tired code on demand – you will stop paying attention. You will stop learning. You will stop caring and investing in crafting the best code you can. The quality of your work will suffer. And your passion will wane.

You will stop becoming better.

Conversely, actively working on coding problems that *challenge* you will encourage, excite you, and help you to learn and develop. It will stop you becoming staid and stale.

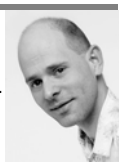
Nobody likes a stale programmer. Least of all, yourself.

What's the challenge?

So what is it that particularly interests you?

It might be that new language you've been reading about. Or it might be working on a different platform. It might just be trying out a new algorithm or library. Or to kick off that pet project you thought about a while ago. It might even be attempting an optimisation or refactor of your current system; just because it looks elegant, even if – shudder to think – it doesn't actually provide business value.

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

Making a Linux Desktop (continued)

Running the changes

You can install and run egmde as a snap:

```
snap install egmde --classic --edge
```

If you want to build it yourself, the source is on github, so you can download and build it (these instructions in Listing 3 are for Ubuntu, the details might vary for other distros).

Either way, you can then select ***egmde*** at the greeter.

Conclusion

The format of this article has changed from earlier ones in the series. There have been a lot of incremental improvements to egmde in since the last one and the changes discussed here were not simple to separate into a branch based on the previous article.

Listing 3

```
sudo apt-add-repository ppa:mir-team/release
sudo apt install libmiral-dev mir-graphics-
drivers-desktop libwayland-dev
sudo apt install libfreetype6-dev libxkbcommon-
dev libboost-filesystem-dev
sudo apt install qtwayland5
sudo apt install g++ cmake
git clone https://github.com/AlanGriffiths/
egmde.git
mkdir -p egmde/build
cd egmde/build
cmake ..
make
sudo make install
```

The current versions has, in addition to the 'shell component' support discussed here, workspaces, and some additional keyboard shortcuts. (And quite a few bugs fixed.) The program has grown, but is still a manageable size for a 'hobby project':

```
$ wc -l *.cpp *.h
  658 egfullscreenclient.cpp
  766 eglauncher.cpp
  115 egmde.cpp
  187 egshellcommands.cpp
  176 egwallpaper.cpp
  382 egwindowmanager.cpp
   53 egworker.cpp
  229 printer.cpp
  241 egfullscreenclient.h
   67 eglauncher.h
   80 egshellcommands.h
   58 egwallpaper.h
   83 egwindowmanager.h
   52 egworker.h
   54 printer.h
 3201 total
```

There's still some way to go before it is a full featured 'desktop environment', but I've found it usable for the majority of my computing needs. ■

References

The Mir homepage: https://mir-server.io/

The egmde git repo: https://github.com/AlanGriffiths/egmde

Often this kind of personal challenge can only be gained on a side-project; something you work on alongside the more mundane day-to-day tasks. And that's *perfect* – it's the antidote to dull 'professional' development. A programming panacea. The crap code cure.

What excites you about programming? Think about what you'd like to work on right now, and why:

- Are you happy to be paid for producing any old code, or do you want to be paid because you produce particularly exceptional work?
- Are you performing tasks for the kudos; do you seek the recognition of your peers or the plaudits of managers?
- Do you want to work on an open-source project; does sharing your code would give you a sense of satisfaction?
- Do you want to be the first person to provide a solution in a new niche, or to a tricky new problem?
- Do you solve problems for the joy of the intellectual exercise?
- Do you like working on a particular kind of project, or do certain technologies suit your strange peccadilloes?
- Do you want to work alongside and learn from certain types of developers?
- Do you look at projects with an entrepreneurial eye – seeking something you think will one day make you millions?

As I look back over my career, I can see that I've tried to work on things in many of those camps. But I've had the most fun, and produced the best software, when working on projects that I've been invested in; where I've *cared* about the project itself, as well as wanting to write exceptional code.

Don't do it!

Of course, there are a potential downsides to seeking out cool coding problems for 'the fun of it'. There are perfectly valid reasons not to:

- It's selfish to steer yourself towards exciting things all the time, leaving boring stuff for other programmers to pick up.
- It's dangerous to 'tinker' with a working system just for the sake of the tweak, if it's not introducing real business value. You're adding unnecessary change and risk. In a commercial environment, it's a waste of time that could be invested elsewhere more profitably.
- If you get side-tracked on pet projects or little 'science experiments', then you'll never get any 'real' work done.

  Remember: not every programming task *is* going to be glamorous or exciting. A lot of our day-to-day tasks are mundane plumbing. That's just the nature of programming in the real world.
- Life is too short. I don't want to waste my spare time working on code as well!
- Re-writing something that already exists is a gross waste of effort. You are not contributing to our profession's corpus of knowledge. You are likely to just recreate something that already exists, possibly not as good as the existing implementations, and full of new terrible bugs. What a waste of time!

Yada. Yada. Yada.

These positions do have some merit. But they should not become excuses that prevent from us becoming better programmers.

It is exactly *because* we have to preform dull tasks all day that we should also seek to balance them with the exciting challenges. We must be responsible in how we use out time do this, and whether we use the resulting code or throw it away.

Get challenged

So work out what you'd like to do. And then do it.

- Perform some code katas that will provide valuable deliberate practice. Throw the code away afterwards.
- Find a coding problem you'd like to solve, just for the fun of it.
- Kick off a personal project. Don't waste all your spare time on it, but find something you can invest effort in that will teach you something new.
- Maintain a broad field of personal interest, so you have good ideas of other things to investigate and learn from.
- Don't ignore other platforms and paradigms. Try re-writing something you know and love on another platform or in another kind of programming language. Compare and contrast the outcome. Which environment lent itself better to that kind of problem?
- Consider moving jobs if you're not being stretched and challenged where you're currently working. Don't blindly accept the status quo! Sometimes the boat needs to be rocked.
- Work with, or meet up with other motivated programmers. Try going to programming conferences, or join local user groups. Attendees come back with a head full of new ideas, and invigorated from the enthusiasm of their peers.
- Make sure you can see the progress you're making. Review source control logs to see what you've achieved. Keep a daily log, or a TODO list. Enjoy knocking off items as you make headway.
- Keep it fresh: take breaks so you don't get overwhelmed, stifled, or bored by bits of code.
- Don't be afraid of reinventing the wheel! Write something that has already been done before. There is no harm in trying to write your own linked-list, or standard GUI component. It's a really good exercise to see how yours compares to existing ones. (Just be careful how you employ them in practice.)

A bit on the side

I've already hinted above that some of these kinds of challenge can only be experienced on a side project; you can't blindly muck around in a production codebase. Does that mean every 'good' programmer must spend every scrap of personal time working on other coding side projects?

No! Please don't do that.

If you *want* to spend your personal time working on projects to practise your craft or to learn a new skill, do so. But never feel obliged to spend every working hour on coding projects at the expense of all else. Do not feel 'inferior' if you're not working on software development for as many hours as the hero programmer you're reading about on some blog.

The best software developers have rounded world views. They're great programmers, sure. But they also learn skills in other areas. They read. They practise other disciplines. They, heaven forfend, *have a life*. These good developers also relax, look after their body and mind, and get enough sleep.

Don't feel like you need to invest your entire life into a 'side project' at the expense of all else.

Conclusion

Yes, I've been hand waving. And I've slipped into motivational speaker territory. But this stuff is important. Do you have something you're engaged in and love to work on?

It's impractical and dangerous to just chase shiny new things all the time and not write practical, useful code. But it's also personally dangerous to get stuck in a coding rut, only ever working on meaningless, tedious software, without being challenged and having fun. ■

Questions

- Do you have projects that challenge you and stretch your skills?
- Are there some project ideas you've thought about for a while, but not started? Why not start a little side-project?
- Do you balance 'interesting' challenges with your 'day to day' work?
- Are you challenged by other motivated programmers around you?
- Do you have a broad field of interest that informs your work?

Piping Software for Less

Paul Grenyer continues his mission to build a DevOps pipeline on a budget.

Developing software is hard and all good developers are lazy. This is one of the reasons we have tools which automate practices like continuous integration, static analysis and measuring test coverage. The practices help us to measure quality and find problems with code early. When you measure something you can make it better. Automation makes it easy to perform the practices and means that lazy developers are likely to perform them more often, especially if they're automatically performed every time the developer checks code in.

This is old news. These practices have been around for more than twenty years. They have become industry standards and not using them is, quite rightly, frowned upon. What is relatively new is the introduction of cloud based services such as BitBucket Pipelines, CircleCI and SonarCloud, which allow you to set up these practices in minutes – however, this flexibility and efficiency comes with a cost.

Why

While BitBucket Pipelines, CircleCI and SonarCloud have free tiers, there are limits.

With BitBucket Pipelines, you only get 50 build minutes a month on the free tier. The next step up is $15/month and then you get 2500 build minutes.

On the free CircleCI tier, you get 2500 free credits per week [4] but you can only use public repositories, which means anyone and everyone can see your code. The use of private repositories starts at $15 per month.

With SonarCloud, you can analyse as many lines of code as you like, but again you have to have your code in a public repository or pay $10 per month for the first 100,000 lines of code.

If you want continuous integration *and* a static analysis repository that includes test coverage *and* you need to keep your source code private, you're looking at a minimum of $15 per month for these cloud based solutions and that's if you can manage with only 50 build minutes per month. If you can't, it's more likely to be $30 per month… that's $360 per year.

That's not a lot of money for a large software company or even for a well funded startup or SME, although as the number of users goes up so does that price. For a personal project, it's a lot of money.

Cost isn't the only drawback, with these approaches you can lose some flexibility as well.

The alternative is to build your own development pipelines.

I bet you're thinking that setting up these tools from scratch is a royal pain and will take hours, when the cloud solutions can be set up in minutes. Not to mention running and managing your own pipeline on your personal machine… and don't they suck resources when they're running in the background all the time? And shouldn't they be set up on isolated machines? What if I told you that you could set all of this up in about an hour and turn it all on and off as necessary with a single command? And that if you wanted to, you could run it all on a DigitalOcean Droplet for around $20 per month. Interested? Read on.

PAUL GRENYER

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

Definitions

Continuous Integration

Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day. Each integration can then be verified by an automated build and automated tests. While automated testing is not strictly part of CI it is typically implied. [1]

Static Analysis

Static (code) analysis is a method of debugging by examining source code before a program is run. It's done by analyzing a set of code against a set (or multiple sets) of coding rules. [2]

Measuring Code Coverage

Code coverage is a metric that can help you understand how much of your source is tested. It's a very useful metric that can help you assess the quality of your test suite. [3]

What

When you know how, setting up a continuous integration server such as Jenkins and a static analysis repository such as SonarQube in a Docker container is relatively straightforward. As is starting and stopping them altogether using Docker Compose. As I said, the key is knowing how, and what I explain in the rest of this article is the product of around twenty development hours, a lot of which was banging my head against a number of individual issues which turned out to have really simple solutions.

Docker

Docker is a way of encapsulating software in a container. Anything from an entire operating system such as Ubuntu to a simple tool such as the scanner for SonarQube. The configuration of the container is detailed in a Dockerfile and Docker uses Dockerfiles to build, start and stop containers. Jenkins and SonarQube each have publicly available Docker images, which we'll use with a few relatively minor modifications to build a development pipeline.

Docker Compose

Docker Compose is a tool which orchestrates Docker containers. Via a simple YML file, it is possible to start and stop multiple Docker containers with a single command. This means that – once configured – we can start and stop the entire development pipeline so that it is only running when we need it or, via a tool such as Terraform, construct and

What is Terraform?

Terraform [5] is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacentre. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

provision a DigitalOcean droplet (or AWS service, etc.) with a few simple commands, and tear it down again just as easily so that it only incurs cost when we're actually developing. Terraform and DigitalOcean are beyond the scope of this article, but I plan to cover them in the near future.

See the Docker and Docker Compose websites for instructions on how to install them for your operating system.

How

In order to focus on the development pipeline configuration, I'll describe how to create an extremely simple Dotnet Core class library with a very basic test and describe in more detail how to configure and run Jenkins and SonarQube Docker containers and setup simple projects in both to demonstrate the pipeline. I'll also describe how to orchestrate the containers with Docker Compose.

I'm using Dotnet Core because that's what I'm working with on a daily basis. The development pipeline can also be used with Java, Node, TypeScript or any other of the supported languages. Dotnet Core is also free to install and use on Windows, Linux and Mac which means that anyone can follow along.

A simple Dotnet Core class library project

I've chosen to use a class library project as an example for two reasons. It means that I can easily use a separate project for the tests, which allows me to describe the development pipeline more iteratively. It also means that I can use it as the groundwork for a future article which introduces the NuGet server Liget to the development pipeline.

Open a command prompt and start off by creating an empty directory and moving into it.

```
mkdir messagelib
cd messagelib
```

Then open the directory in your favourite IDE, I like VSCode for this sort of project. Add a Dotnet Core appropriate `.gitignore` file and then create a solution and a class library project and add it to the solution:

```
dotnet new sln
dotnet new classLib --name Messagelib
dotnet sln add Messagelib/Messagelib.csproj
```

Delete `MessageLib/class1.cs` and create a new class file and class called **`Message`**:

```
using System;
namespace Messagelib
{
  public class Message
  {
    public string Deliver()
    {
      return "Hello, World!";
    }
  }
}
```

Make sure it builds with:

```
dotnet build
```

Commit the solution to a public git repository, or you can use the existing one in my bitbucket account here: https://bitbucket.org/findmytea/messagelib.

A public repository keeps this example simple and, although I won't cover it here, it's quite straightforward to add a key to a BitBucket or GitHub private repository and to Jenkins so that it can access them.

Remember that one of the main driving forces for setting up the development pipeline is to allow the use of private repositories without having to incur unnecessary cost.

Remember that one of the main driving forces for setting up the development pipeline is to allow the use of private repositories without having to incur unnecessary cost.

Run Jenkins in Docker with Docker Compose

Why use Jenkins, I hear you ask. Well, for me the answers are simple: familiarity and the availability of an existing tested and officially supported Docker image. I have been using Jenkins for as long as I can remember.

The official image is here: https://hub.docker.com/r/jenkins/jenkins

After getting Jenkins up and running in the container we'll look at creating a 'Pipeline' with the Docker Pipeline plugin. Jenkins supports lots of different 'Items', which used to be called 'Jobs', but Docker can be used to encapsulate build and test environments as well. In fact this is what BitBucket Pipelines and CircleCi also do.

To run Jenkins Pipeline, we need a Jenkins installation with Docker installed. The easiest way to do this is to use the existing Jenkins Docker image from Docker Hub. Open a new command prompt and create a new directory for the development pipeline configuration and a sub directory called Jenkins with the Dockerfile in Listing 1 in it.

You can see that our `Dockerfile` imports the existing Jenkins Docker image and then installs Docker for Linux. The Jenkins image, like most Docker images, is based on a Linux base image.

To get Docker Compose to build and run the image, we need a simple `docker-compose.yml` file in the root of the development pipeline directory with the details of the Jenkins service (see Listing 2, overleaf).

Note the **`build`** parameter which references a sub directory where the Jenkins Dockerfile should be located. Also note the volumes. We want the builds to persist even if the container does not, so create a `.jenkins` directory in your home directory:

```
mkdir ~/.jenkins
```

Specifying it as a volume in `docker-compose.yml` tells the Docker image to write anything which Jenkins writes to `/var/jenkins_home`

Listing 1

```
FROM jenkins/jenkins:lts

USER root
RUN apt-get update
RUN apt-get -y install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common

RUN curl -fsSL https://download.docker.com/linux/
debian/gpg | apt-key add -
RUN add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/
linux/debian \
   $(lsb_release -cs) \
   stable"

RUN apt-get update
RUN apt-get install -y docker-ce docker-ce-cli
containerd.io
RUN service docker start

# drop back to the regular jenkins user - good
practice
USER jenkins
```

When I first started using Jenkins, there were plugins for lots of different static analysis tools ... Then SonarQube came along.

Listing 2

```
version: '3'
services:
  jenkins:
    container_name: jenkins
    build: ./jenkins/
    ports:
      - "8080:8080"
      - "5000:5000"
    volumes:
      - ~/.jenkins:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
```

in the container to `~/.jenkins` on the host – your local machine. If the development pipeline is running on a DigitalOcean droplet, spaces can be used to persist the volumes even after the droplet is torn down.

As well as running Jenkins in a Docker container we'll also be doing our build and running our tests in a Docker container. Docker doesn't generally like being run in a Docker container itself, so by specifying:

```
/var/run/docker.sock
```

as a volume, the Jenkins container and the test container can be run on the same Docker instance.

To run Jenkins, simply bring it up with Docker compose:

```
docker-compose up
```

(To stop it again just use CTRL+C.)

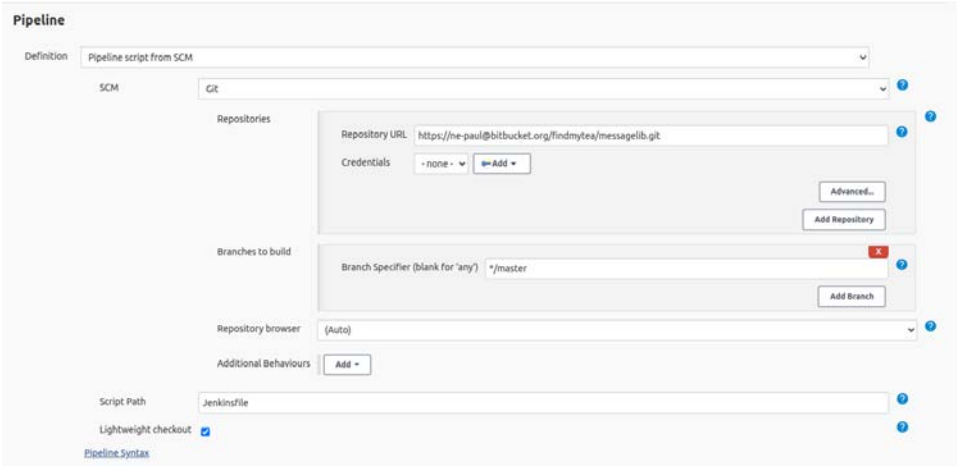
Make sure the first time that you note the default password. It will appear in the log like this:

```
Jenkins initial setup is required. An admin user
has been created and a password generated.

Please use the following password to proceed to
installation:

<password>

This may also be found at: /var/jenkins_home/
secrets/initialAdminPasswor
```

To configure Jenkins for the first time open a browser and navigate to:

```
http://localhost:8080
```

Figure 1



Listing 3

```
/* groovylint-disable CompileStatic,
GStringExpressionWithinString, LineLength */
pipeline
{
  agent
  {
    docker {image
    'pjgrenyer/dotnet-build-sonarscanner:latest'}
  }
  stages
  {
    stage('Build & Test')
    {
      steps
      {
        sh 'dotnet clean'
        sh 'dotnet restore'
        sh 'dotnet build'
      }
    }
  }
}
```

Then:

1. Paste in the default password and click continue.
2. Install the recommended plugins. This will take a few minutes. There is another plugin we need too which can be installed afterwards.
3. Create the first admin user and click *Save & Continue*.
4. Confirm the Jenkins url and click *Save & Finish*.
5. Click *Start Jenkins* to start Jenkins.

You now have Jenkins up and running locally in a Docker container!

To use Docker pipelines in Jenkins, we need to install the plugin. To do this:

1. Select ***Manage Jenkins*** from the left hand menu, followed by ***Manage Plugins***.
2. Select the ***Available*** tab, search for ***Docker Pipeline*** and select it.
3. Click ***Download now and install after restart***.
4. On the next page, put a tick in the ***restart after download*** check box and wait for the installation and for Jenkins to restart. Then log in again.

Next we need to create the Docker Pipeline for the Messagelib solution.

1. Select ***New Item*** from the left hand menu, enter **`Messagelib`** as the name, select ***Pipeline*** and click ***ok***.
2. Scroll to the ***Pipeline*** section and select ***Pipeline script from SCM*** from the ***Definition*** dropdown. This is because we're going to define our pipeline in a file in the Messagelib solution.
3. From the ***SCM*** dropdown, select ***Git*** and enter the repository URL of the Messagelib solution. (See Figure 1.)
4. Then click ***Save***.

Jenkins is now configured to run the Messagelib pipeline, but we need to tell it what to do by adding a text file called `Jenkinsfile` to the root of the Messagelib solution. (See Listing 3.)

This very simple Groovy script tells the Jenkins pipeline to get the latest 'dotnet-build-sonarscanner' Docker image and then use it to clean, restore and build the dotnet project. 'dotnet-build-sonarscanner' is a Docker image I built and pushed to Docker Hub using the `Dockerfile` in Listing 4.

Listing 4

```
FROM mcr.microsoft.com/dotnet/core/sdk:latest AS
build-env
WORKDIR /

RUN apt update
RUN apt install -y default-jre

ARG dotnet_cli_home_arg=/tmp
ENV DOTNET_CLI_HOME=$dotnet_cli_home_arg
ENV DOTNET_CLI_TELEMETRY_OPTOUT=1
ENV PATH="${DOTNET_CLI_HOME}/.dotnet/
tools:${PATH}"
ENV HOME=${DOTNET_CLI_HOME}

RUN dotnet tool install --global dotnet-
sonarscanner
RUN chmod 777 -R ${dotnet_cli_home_arg}
```

This creates and configures a development environment for Dotnet Core and Sonar Scanner, which requires Java. (There is a way to use the Dockerfile directly, rather than getting it from Docker Hub.[6])

Once the Jenkins file is added to the project and committed, set the build off by clicking ***Build now*** from the left hand menu of the MessageLib item. The first run will take a little while as the Docker image is pulled (or built). Future runs won't have to do that and will be quicker. You should find that once the image is downloaded, the project is built quickly and Jenkins shows success.

Run SonarQube in Docker with Docker Compose

When I first started using Jenkins, there were plugins for lots of different static analysis tools which would generate reports as part of the build. Then SonarQube came along as both a central repository for static analysis results and as a tool for grouping multiple static analysis tools together. [7]

SonarQube ships with a H2 in-memory database for evaluation purposes, but this isn't ideal for long term use. Fortunately it's very easy to run PostgreSQL in a Docker container and use that instead.

1. Create a `postgresql` directory at the root of your development pipeline project:

   ```
   mkdir postgresql
   ```

2. Move into the directory and create a new file called `init-sonar.sql` with following content:

   ```
   CREATE DATABASE sonar;
   CREATE USER sonar WITH PASSWORD 'sonar';
   GRANT ALL PRIVILEGES ON DATABASE sonar to sonar;
   ```

   This file is used by the PostgreSQL Docker container to create an empty Sonar database. It is created the first time the container starts. You may wish to use a more secure password, but as we'll see shortly, the database will only be accessible to other containers on the same Docker host.

3. Create a `Dockerfile` in the `postgresql` directory:

   ```
   FROM postgres:latest
   COPY init-sonar.sql /docker-entrypoint-initdb.d/
   init-sonar.sql
   ```

   The Dockerfile uses the official postgres image [8] and copies in the `init-sonar.sql` file.

4. Add the following to the `docker-compose.yml` file:

   ```
   db:
   build: ./postgresql/
   environment:
     POSTGRES_PASSWORD: 3PdrAz6xWe5yErnQ
   volumes:
     - ~/.postgresql/:/var/lib/postgresql/dat
   ```

Listing 5

```
sonar:
  container_name: sonar
  image: sonarqube:latest
  ports:
    - "9000:9000"
  environment:
    - SONARQUBE_JDBC_USERNAME=sonar
    - SONARQUBE_JDBC_PASSWORD=sonar
    - SONARQUBE_JDBC_URL=jdbc:postgresql:
      //db:5432/sonar
  volumes:
    - ~/.sonar/data:/opt/sonarqube/data
    - ~/.sonar/logs:/opt/sonarqube/logs
```

   To be able to build the postgresql image, you need to specify a root user password and configure a volume to write the data to your local machine. Note that there is no port (**`-p`**) argument. This is because, as we'll see, Docker containers running on the same host and started by Docker Compose are all added to the same network. So while the postgresql container is not accessible from the host or any remote machine, it is accessible to other Docker containers.

5. Create a directory on your local machine for the data:

   ```
   mkdir ~/.postgresql
   ```

The easiest way to add SonarQube is directly into the `docker-compose.yml` file and configure it there as the Docker image does not need to be modified.

1. Add the information in Listing 5 to the `docker-compose.yml` file.

   The SonarQube Docker image is used, the port, the database credentials and the volumes are specified. Note that the host section of the connection string is **`db`**, the name of the `docker-compose.yml` section which starts the postgresql database. This is how Docker images created on the same host by Docker Compose refer to each other.

2. Create directories on your local machine for the data and logs:

   ```
   mkdir ~/.sonar
   mkdir ~/.sonar/data
   mkdir ~/.sonar/logs
   ```

3. SonarQube uses an embedded Elasticsearch, so if you're using Linux you need to configure your local machine with the Elasticsearch production mode requirements and File Descriptors configuration. To do this execute the following at the command line:

   ```
   sysctl -w vm.max_map_count=262144
   sysctl -w fs.file-max=65536
   ulimit -n 65536
   ulimit -u 4096
   ```

   If you're running on Windows, you don't need to do this.

SonarQube is now ready to run, so run Docker Compose:

```
docker-compose up
```

The first time it runs, SonarQube will configure itself and create the database it needs. To see SonarQube running, open a browser and navigate to **`http://localhost:9000`**.

Static analysis with SonarQube

Now that SonarQube is up and running, we can create a project to hold the analysis results from Messagelib.

The default SonarQube username and password are both 'admin'. This can be easily changed. Let's create a project so we can analyse Messagelib:

1. Login to SonarQube (user: admin, password: admin).
2. Click ***Create new project***

Listing 6

```
pipeline
{
  agent
  {
    docker
    {
      image
      'pjgrenyer/dotnet-build-
sonarscanner:latest'
      args '--network host'
    }
  }
  stages
  {
    stage('Sonar setup')
    {
      steps
      {
        sh 'dotnet sonarscanner begin
k:messagelib'
      }
    }
    stage('Build & Test')
    {
      steps
      {
        sh 'dotnet clean'
        sh 'dotnet restore'
        sh 'dotnet build'
        sh 'dotnet test'
      }
    }
    stage('Sonar End')
    {
      steps
      {
        sh 'dotnet sonarscanner end'
      }
    }
  }
}
```

3. Enter **messagelib** as the Project Key and Display Name – this is how SonarQube knows which project it is being asked to analyse.
4. Click ***setup***.
5. Then ignore the rest of the process as we don't need it.

Next we need to return to the Messsagelib solution and update the `Jenkinsfile` so that it scans the project build output (see Listing 6). You can see the new sections of the file highlighted in darker text.

The Docker container which is used to run the build and tests isn't created by Docker Compose so we have to manually configure it to be on the host network so that it can talk to the SonarQube container by passing an extra argument:

```
args '--network host'
```

Before the build starts the SonarQube scanner needs to be started and configured with the project key for the SonarQube project:

```
stage('Sonar setup')
{
  steps
  {
    sh 'dotnet sonarscanner begin /k:messagelib'
  }
}
```

Docker containers running on the same host and started by Docker Compose are all added to the same network

Recall that the SonarQube scanner is part of the 'dotnet-build-sonarscanner' image and therefore available to all builds which use it in a Docker pipeline. After the build, the SonarQube scanner needs to be stopped and told to send the results to SonarQube:

```
stage('Sonar End')
{
  steps
  {
    sh 'dotnet sonarscanner end'
  }
}
```

Check the Jenkinsfile changes into your repository and kick off the Jenkins item again and once it's finished you should see something like Figure 2 in SonarQube after you login.

Then if you drill down (click on the project name) you get more detail (see Figure 3).

If you then click on the number of 'code smells' and then on one of the issues you'll get the details for the individual issues and their position in the code (see Figure 4).

I'll leave fixing the issues as an exercise for the reader.

Measuring test coverage with SonarQube

As well as static analysis, SonarQube also collates test code coverage. While SonarQube supports a handful of different code coverage tools, the important thing is to get the format of the coverage output file to match one of the supported tools.

SonarQube is showing 0% code coverage. While this is an accurate figure of the coverage, it's not actually a reflection of any measuring. In order to measure test coverage, we need to write and run some tests and tell SonarQube scanner where to find the output and what format it is in. Let's start with a test.

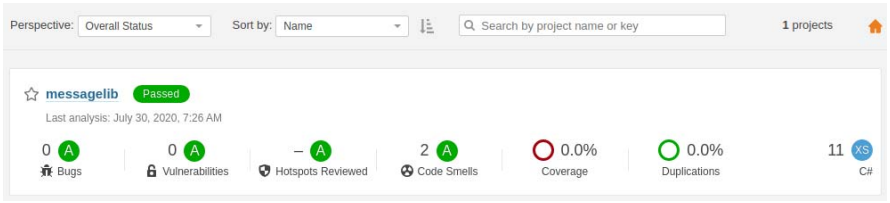


Figure 2

Go to your Messagelib solution top level directory, create an XUnit test project, add it to the solution and add the Messagelib class library project as a dependency:

```
dotnet new xunit --name MessagelibTest
dotnet sln add MessagelibTest/
MessagelibTest.csproj
dotnet add MessagelibTest/MessagelibTest.csproj
reference Messagelib/Messagelib.csproj
```

Dotnet Core XUnit projects come with OpenCover [9] by default, but to generate a coverage report an extra package is needed. Change to the MessagelibTest directory and add the package:

```
dotnet add package coverlet.msbuild --version 2.9.0
```

Then switch back to the root directory of the solution.

Delete `Messagelib/UnitTest.cs` and create a new class file and class called `MessageTest` (see Listing 7).

Check that the solution builds and the test passes by running:

```
dotnet test
```

at the command line. To generate the necessary output files and to tell SonarQube scanner where to find them we need to update the `Jenkinsfile` again (see Listing 8).

Adding the highlighted arguments tells SonarQube scanner to look for a file called `coverage.opencover.xml` in the `MessagelibTest` directory and not to measure the test coverage on the test classes themselves (see Listing 9, overleaf).

Adding the highlighted arguments tells dotnet to generate test coverage and output the results in OpenCover' format when it runs the tests.

Check the solution in, run the Jenkins item again and see the coverage reach 100% (Figure 5).

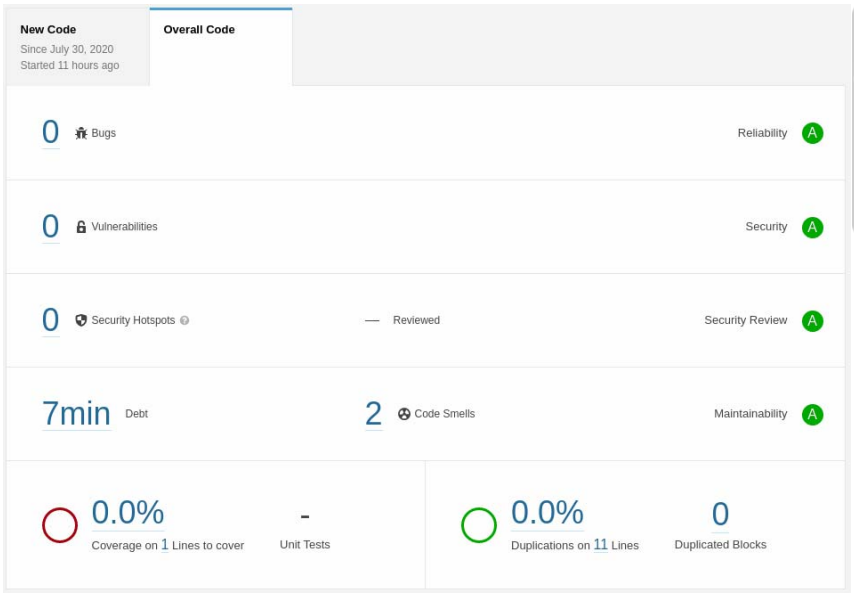


Figure 3

Finally

This brings us to the end. I've shown you:

1. How to build, configure and run a development pipeline with Docker, Docker Compose, Jenkins, SonarQube and Postgres.
2. How to set up a Docker pipeline on Jenkins for continuous integration.
3. How to perform static analysis and measure code coverage on a simple Dotnet Core project.

In many cases this will be all small project developers need. However, there's plenty more which can be done, including:

1. Building and running the development pipeline on DigitalOcean.
2. Adding a NuGet server to publish private packages to.

I hope this article has persuaded you that you can save money on your existing cloud based development pipelines and whetted your appetite for what can be achieved with Docker and Docker Compose.

Until next time... ■

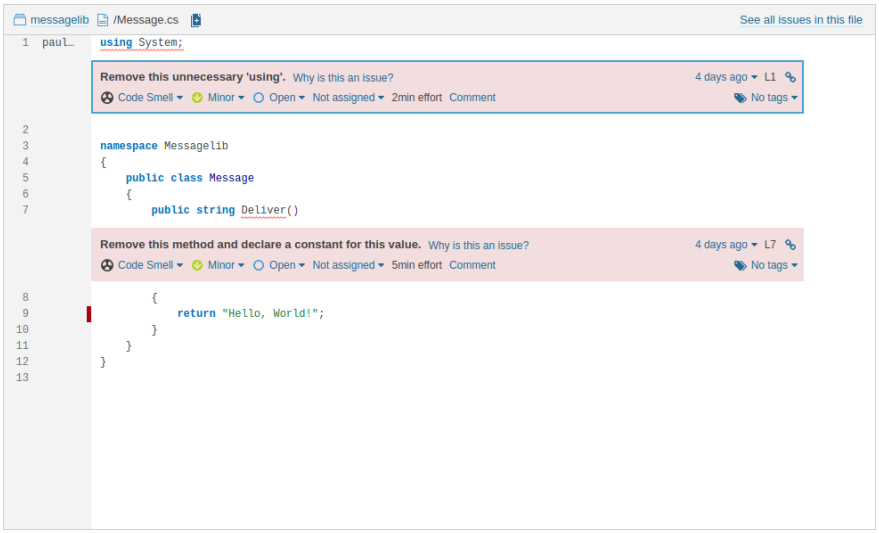


Figure 4

Acknowledgements

Thank you to Jez Higgins, Neil Carroll, Philip Watson, Alex Scotton and Lauren Gwynn for inspiration, ideas and reviews!

References

[1] Continuous Integration: https://codeship.com/continuous-integration-essentials
[2] Static Analysis: https://www.perforce.com/blog/sca/what-static-analysis
[3] Code Coverage: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage
[4] CircleCI Credits: Credits are used to pay for your team's usage based on machine type and size, and premium features like Docker layer caching. See https://circleci.com/pricing/
[5] Terraform: https://www.terraform.io/intro/
[6] Using Dockerfile directly: https://www.jenkins.io/doc/book/pipeline/docker/
[7] SonarQube official image: https://hub.docker.com/_/sonarqube/
[8] Postgres official image: https://hub.docker.com/_/postgres
[9] OpenCover is a code coverage tool for .NET 2 and above with support for 32 and 64 processes and

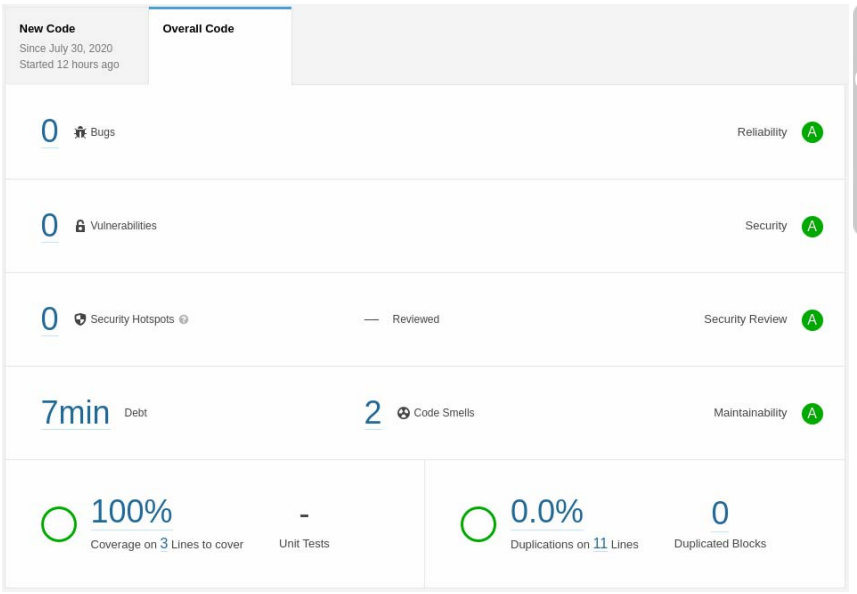


Figure 5

Jumping around in Emacs

Silas S. Brown shares a tip for navigating code.

When working with code I didn't write, I frequently find myself asking "can I check that function really does what its name suggests", i.e. I need to find the definition of a function (or class etc) that's mentioned, which might be in another file or directory. My habitual way of handling this has been to run the Unix **grep** tool from inside Emacs, look at (and, if necessary, search in) these search results, and figure it out. This is a habit that perhaps has some scope for optimisation.

The traditional solution for both Emacs and Vim involves Ctags, but this can be a hassle to set up for each project, especially when multiple programming languages are involved. So I recently started using an Emacs package called dumb-jump which is meant to take you there 'most of the time' with little setup. Dumb Jump can use normal **grep**, or (if installed) optimised versions of grep, such as **git-grep** which knows to check only tracked files in Git projects, or 'The Silver Searcher' maintained by Geoff Greer which has Boyer-Moore string search, **mmap()** etc and also tries to avoid binaries, editor-created backups etc. Dumb Jump's setup instructions are in its README [1] and then usage is normally a matter of holding down the ALT key while pressing dot (.) when the cursor is on the name of the thing whose definition you want to see, and then ALT-comma (,) to go back. It helps that dot and comma are next to each other on both the QWERTY and Dvorak keyboard layouts (if using Dvorak, you might have a right-hand ALT GR key you can use to make the combination easier).

I understand that JetBrains' products have similar functionality built in, although I haven't tried it as I've been using Emacs since last century (when it was jokingly called Eight Megabytes And Constantly Swapping: it's hard to believe it's now considered lightweight compared with modern IDEs, while all the while I was wondering if I need Vim skills to further reduce my memory requirements). ■

Reference

[1] Dumb Jump: https://github.com/jacktasia/dumb-jump

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

Piping Software for Less (continued)

covering both branch and sequence points: https://github.com/OpenCover/opencover'

Listing 7

```
using Messagelib;
using Xunit;

namespace MessagelibTest
{
  public class MessageTest
  {
    [Fact(DisplayName = "Check that the correct
delivery message is returned.")]
    public void DeliverMessage()
    {
      Assert.Equal("Hello, World!",
        new Message().Deliver());
    }
  }
}
```

Listing 8

```
stage('Sonar setup')
{
  steps
  {
    sh 'dotnet sonarscanner begin /k:messagelib
    /d:sonar.cs.opencover.reportsPaths=
    MessagelibTest/coverage.opencover.xml
    /d:sonar.coverage.exclusions="**Test*.cs"'
  }
}
```

Listing 9

```
stage('Build & Test')
{
  steps
  {
    sh 'dotnet clean'
    sh 'dotnet restore'
    sh 'dotnet build'
    sh 'dotnet test /p:CollectCoverage=true
       /p:CoverletOutputFormat=opencover'
  }
}
```

The Standards Report

Guy Davidson talks about executors and their journey towards acceptance into the standard.

About eight and a half years ago, A paper was published, N3378, by some folk from Google entitled 'A preliminary proposal for work executors'. In this paper, the authors outlined a future which contained "executors, objects that can execute units of work packaged as function objects". Little did they know where this would lead or how long it would take to get there.

The initial motivation was that well-written multithreaded programs contain discrete pieces of work which are executed asynchronously. Something needs to schedule the execution of these pieces, by launching a new thread or reusing an existing one, choosing the most appropriate thread.

Momentum gathered. Along the way, two additional models were proposed by Nvidia and by Chris Kohlhoff, which were aimed at different use cases: the Parallelism TS for Nvidia, and the Networking TS for Chris Kohlhoff.

the development of executors has taken such a twisty turny path that there are very few people who are aware of the full motivation for each part

Nearly thirty papers and many revisions later, we have arrived at Revision 13 of P0443. This paper hoped to unify Google's, Nvidia's and Chris Kohlhoff's models. Work started on this paper at the end of 2016; it was never going to make C++17 but it was hoped that it would make its way into the working draft in plenty of time for C++20. Once safely ensconced, this would leave time to unblock the Networking TS, allowing that to become a tentpole feature of C++20, alongside modules, coroutines, concepts and ranges.

Four years on and P0443 has yet to reach the Library Working Group (LWG), holding up networking library support. Frankly, C++20 contains an embarrassment of riches already and the fact that we're going to have to wait at least until C++23 for networking isn't so disappointing when we have all these new toys to learn to play with.

Agreement has been reached on the design, though, and in an attempt to move the paper through the pipeline, the chair of the Library Evolution Working Group (LEWG) convened six review groups of nine people. Each group would take a part of the paper and check API design and wording. I chaired one of the groups; we were asked to look at the new concepts proposed by the paper. I also participated in another group since there aren't 45 people available to LEWG!

This ended up being quite a tough gig. The hardest part is that the development of executors has taken such a twisty turny path that there are very few people who are aware of the full motivation for each part. Fortunately each group contained three of the authors (there are eleven authors and eight 'Other Contributors'). Over the course of five hours, we carefully parsed the paper and looked at the items presented, metaphorically kicking the tyres. It felt more like a tutorial than a review, though, with the authors explaining many details to us, but I hope the job is done and we provided useful feedback.

The weekly LEWG telecons have been reviewing the efforts of each group. When all six groups have reported back, which should be by the end of August, and the authors have responded to the feedback, there should be a further (and let's hope final) revision of the paper. LEWG can consider moving this forward to LWG, in the hope that it can make a speedy entry to the working draft of C++23 and allow the Networking TS to be merged.

Virtual plenary meetings

If you have been paying close attention to my remarks about ISO governance (thrilling stuff, I know), you will be aware that face-to-face meetings have been suspended, which means that we have not been able to have any plenary votes to amend the working draft of the next C++ standard. Recently however, regulations were changed In Light Of The Ongoing Pandemic. We will now be able to hold three virtual plenary meetings each year until further notice.

There are some complications to this. The first is timezones. One advantage of the face-to-face committee meetings is that everyone is in the same place, and therefore the same timezone. Delegates turn up to committee meetings from around the world, which means that any meeting is going to require some attendees to be awake in the middle of their night.

SG14 recently held what we termed an 'inverted' telecon. Rather than meet at 18:00 UTC we met at 06:00 UTC so that we could get attendees from Australia, New Zealand and the rest of the West Pacific region. This was remarkably well attended and highlights the thriving community in that part of the world, which should not be excluded from any plenary voting session. This makes choosing a timeslot even harder: the weekly LEWG meetings have been taking place at 15:00 UTC but that would rule out all but the most dedicated West Pacific folk. I imagine therefore that virtual plenaries might happen in the early evening UTC, perhaps 19:00, to give them a chance of attendance at the beginning of their day.

Conferences

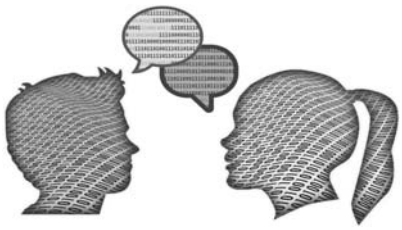
One more thing: you may have noticed that conferences have still been continuing online. I have spoken at a couple, and this does seem to be working out quite well. I do encourage you to consider the upcoming conferences if you want to get more C++ goodness in your life. Next time I'll be reviewing further developments in the executors story and looking at some of the highlights from the summer conferences.

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.

Code Critique Competition 125

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

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 1.

Listing 1

```
#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;
}
```


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.co.uk

Critiques

Dave Simmonds <daveme@ntlworld.com>

At first glance, it looks like the issue is **`t`** being an **`int`**. **`put_money`** is supposed to work with **`long double`** or **`basic_string`**.

But, the **`int`** will be promoted to a **`long double`**, and our string money variables will convert to **`int`** ok, so this isn't actually an issue.

As long as we don't overflow the **`int`**.

The issue is that we've imbued **`std::cout`** with the locale, but we haven't imbued the **`istringstream`** we use for **`get_money`**.

Another issue is that we are not checking whether we have valid input on the first **`get_money`** – if not we will throw an exception.

We should check it (and let the operator know if there is an issue).

So, we need to change the first **`get_money`** from:

```
std::istringstream(ln) >> std::get_money(m);
t = std::stoi(m);
```

to something like:

```
auto iss = std::istringstream(ln);
if (argv[1])
  iss.imbue(std::locale(argv[1]));
if (iss >> std::get_money(m))
  t = std::stoi(m);
else
{
  std::cout << "bad input" << std::endl;
  t = 0;
}
```

And the second one from:

```
if (std::istringstream(ln)
        >> std::get_money(m))
  t += std::stoi(m);
```

to:

```
iss = std::istringstream(ln);
if (argv[1])
  iss.imbue(std::locale(argv[1]));
  if (iss >> std::get_money(m))
    t += std::stoi(m);
  else
    std::cout << "bad input" << std::endl;
```

The program will then work as intended.

Other issues:

If we do not supply a locale, we will be working only with integers. Is that what is intended? Should we insist that a locale is supplied?

Is **`en_GB`** correct? Maybe we want **`en_GB.UTF-8`** – depending on how our terminal emulation works. **`en_GB`** will try to print a pound sign as char 163 – **`en_GB.UTF-8`** will print as 194, 163. If we are using putty in UTF-8 mode, which is common, we want **`en_GB.UTF-8`**.

This one is pretty obscure, but apparently it's possible for **`argc`** to be zero and **`argv`** to be **`NULL`**. It's allowed by the standard, although I've never seen it. So, just in case, we should define **`argc`** (change **`int main(int, char**argv)`** to **`int main(int argc, char**argv)`**) and change **`if (argv[1])`** (in a few places) to **`if (argc > 1)`**.

Hans Vredeveld <accu@closingbrace.nl>

The OP is right that she needs to use locales. She correctly set the locale on `std::cout` to format the total properly, but forgot to set the locale on the temporaries `std::istringstream(ln)` to read the input amounts properly. One way to do this is to set the locale on the temporary streams before an amount is read from it (this would imply that those temporaries have to be named). For an application like this, that has to use the same locale everywhere it uses a locale, it is worth considering to set the locale globally with a call to:

```
std::locale::global(std::locale("argv[1]"))
```

at the start of the program. Any stream that is now created is automatically imbued with our locale. We still have to imbue `std::cout` with the locale, as it was created before we set the global locale.

Now that we have a correctly working program, let's see what else we can improve.

The first thing is `if (argv[1])`. In all the cases that I know of, this happens to work, but that is not guaranteed by the standard. `argv` is an array of `char *` of length `argc+1`, where `argv[argc]` is a null pointer. If `argc` is at least `1`, `argv[0]` is a string representing the program's name and any following entries are the program's arguments. Although I have never seen it, `argc` can be `0` and `argv[1]` would reference memory past the end of the array. For the second and later program arguments, this way of testing whether the command line argument exists, references memory past the end of the array when the program is called without any arguments. A robust implementation will check whether `argc` has a large enough value before it references an entry in `argv`. In our case that would be: `if (argc > 0)`.

Next, the code reads only one amount per line, and only if that amount is the first thing on the line. This may or may not be what you want. If it should be possible to enter more than one amount per line, we could replace

```
std::getline(std::cin, ln);
std::istringstream(ln) >> std::get_money(m);
```

with

```
std::cin >> std::get_money(m);
```

(don't forget to imbue `std::cin` with the locale).

Now, let's assume that we want only one amount per line and that it is the first item on the line. Between the code for the first item and next items, there is some code duplication with a twist. If there is no first line, or if the line does not contain an amount, the program will fail with an exception. If, at one point, there is no next line to read, the program will stop parsing input and show the results. If a next line does not contain an amount, the program will silently skip the line. Another difference is that when reading the first item, we initialize the variable `t`, and when reading the next items, we add to it. We avoid some code duplication, and make the program more robust, when outputting the text "First item: " is immediately followed by the `while`-loop. Of course, then we also have to initialize `t` to 0 in its declaration.

Speaking of `t`, it is a non-descriptive name that needs a comment at its declaration. That comment gives us an excellent name for the variable: `total`.

The next variable that is declared is `ln`. Like `t`, this is rather non-descriptive and I like to replace it with the more descriptive name `line`. We also note that its type is `std::string`, but that the header `<string>` is not included directly, and that we rely on it being included via one of the other headers. This makes the code fragile, so let's `#include <string>` directly. Lastly, `line`, as it's now called, is only used in the while-loop, so it would be nice if we can restrict its scope to that loop. We can do that if we change the while-statement to a for-statement:

```
for (std::string line; std::getline(std::cin,
     line);)
```

The last variable is `m`. We rename it to `money` right away. Note that it is declared at the top of the function, but that it is only used in the `if`-statement inside the `for`-loop (former `while`-loop). Using C++17, we can use the init-statement for `if` to declare it right there. In the `if`-statement, we read an amount into a string and then convert that string to a number that is added to the total. We can also read an amount into a `long double` instead of an `std::string`. That would remove the need for the conversion from `std::string` to `int`. Putting it all together, the `if`-statement now becomes

```
if (long double money;
    std::istringstream(line)
    >> std::get_money(money))
  total += money;
```

To avoid implicit conversions, we also have to make `total` a `long double`. This also solves an issue with the use of the template `std::put_money`. When `total` was an `int`, the template argument was `int`, but the standard only allows for `long double` or a specialization of `std::basic_string`.

Finally some words on `std::get_money`. Playing with it for this code critique, I ran into some unexpected behaviour, at least for me.

First, the input stream must have the amount in the correct number of decimals. With the British locale (`en_GB`), "1200.00" and "1,200.00" will be processed correctly as £ 1,200.00. Unfortunately, "1200", "1,200" and "1,200.0" will all be consumed from the input stream, but won't result in an amount being put in the argument to `std::get_money`. On the other hand, if the input stream contains "1,200.004", the amount will be £ 1,200.00 and the stream still contains the character '4'.

Another surprise for me was the thousands separator. Being Dutch, I'm used to the thousands separator being a point (and the decimal separator a comma). With the locale set to `nl_NL`, I input "1.200,00" and got out € 0,00 instead of € 1.200,00. Instead, I had to use a space as thousands separator (which was introduced as thousands separator to solve the ambiguity with the point and comma) and input "1 200,00" (or "1200,00"). Going a little to the east, to Germany (locale `de_DE`), I still have to use a point as thousands separator and can't use the space. (If you think this isn't messy enough, go to Belgium. With the locale `nl_BE`, the thousands separator is a space, and with `fr_BE`, it is a point.) Of course, the results are dependent on the locale database on my computer. Results can differ with a different locale database.

In the end, I advise to not use `std::get_money` for anything but a throw-away program or for when you have full control over the input (although, in that case you can probably read the amount directly as an `int` or `double` without problem). In all cases, test it with the locales that you plan to use, and on the machines that you want to use it on.

On the other hand, `std::put_money` can be useful to quickly get the output formatted in the way that your user expects.

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

The first thing that struck me is the legality of the statement `if (argv[1])`. What is the value of `argv[1]` and is the expression always valid? The student does not make use of the first parameter of `main()`, usually called `argc`. The C++ standard states that "The value of `argv[argc]` shall be `0`". The standard does not say anything about `argv[x]` when `x` is greater than `argc`. If `argc` is ever `0` then accessing `argv[1]` would represent undefined behaviour. Can `argc` ever be zero? Running the following code on a Linux system, where `totals` is an executable (containing `main()`), results in `argc` of `totals` being zero.

```
char * arguments[] = {nullptr};
execv("./totals", arguments);
```

So, yes, it can be done. If there is a possibility of `argc` being zero it should be guarded against. I suggest making sure that `argc` is equal to 2 before imbuing `std:cout`.

```
if (argc == 2)
  std::cout.imbue(std::locale(argv[1]));
```

Now we come to the name of the locale as provided by the program argument. There is only one locale name that is standard, specifically, `C`. There is no guarantee that the one supplied will be recognised. On my

system **en_GB** is not recognised and results in **imbue()** throwing an exception of type **std::runtime_error**.

One way to discover the user's locale of a particular system is to execute the following statement.

```
std::locale("").name();
```

On my machine, this returns **"en_GB.UTF-8"**. Using this string as the program argument is accepted by **imbue()**.

My compiler failed to compile the statement

```
std::istringstream(ln) >> std::get_money(m);
```

although the same version of the compiler on Compiler Explorer (clang 9.0.0) compiled without error. I never discovered why. I had to separate the statement into two as shown below.

```
std::istringstream iss(ln);
iss >> std::get_money(m);
```

I had to give the **if** statement within the while loop similar treatment. The supplied code now compiles, runs and gives the problematic output described by the student.

It turns out that the wrong stream is being imbued with the locale. Imbuing **std::cout** has no effect because a complete line of text is being read from **std::cout** and placed in an **std::string**, no formatting is involved. The contents of the **std::string** is then transferred to an input string stream. This input string stream should be imbued with the locale. When this is done, the program works as desired.

Generally, it is better to leave the definition of variables as late as possible. The variable **t**, for example, can be defined and given a value in one statement, namely **int t = std::stoi(m);**. The definition of variables **ln** and **m** can also be moved down the code to just before they are used.

The readability of the supplied code would be improved considerably if more expressive variable names were used. Names such as **ln**, **m** and **t** convey very little information. I suggest names such as **line**, **money** and **total** would be more appropriate. This would remove the need for a comment in the supplied code explaining what **t** meant, for example.

We are encouraged not to use **std::endl** anymore. **std::endl** performs two functions; it writes **'\n'** to the stream and then flushes the stream. There are very few occasions where it is required to flush the stream explicitly. All the programmer needs to do in this case is write the new line character to the stream. This shows exactly what is being written to the stream and leaves the flushing to the compiler/operating system. This is a minor point, however, and will make little or no difference to the performance of the student's program.

The student's code can be simplified to some extent. Instead of transferring the contents of **std::cin** to an **std::istringstream** via an **std::string** and then imbuing the **std::istringstream**, **std::cin** can be imbued directly. This results in the simplified program shown below.

```
#include <iomanip>
#include <iostream>
#include <locale>
int main(int argc, char**argv)
{
  if (argc != 2)
  {
    std::cout << "Please provide a locale\n";
    return 0;
  }
  std::cin.imbue(std::locale(argv[1]));
  std::cout.imbue(std::locale(argv[1]));
  int total = 0;
  std::cout << "First item: ";
  for (std::string money; std::cin
    >> get_money(money);)
  {
    total += std::stoi(money);
    std::cout << "Next item: ";
  }
  std::cout << std::showbase << "Total: "
    << std::put_money(total) << '\n';
}
```

Incidentally, it is curious that the student's program is named `totals.c` rather than `totals.cpp`. The program is definitely written in C++.

Ovidiu Parvu <accu@ovidiuparvu.me>

We will start by describing why the total computed by the program is incorrect, and then we will continue to explore other improvements that can be made to avoid undefined behaviour, make the program more maintainable and improve the user experience.

Computing the total correctly

The reason why the total computed by the program is incorrect is that the locale provided as input to the program is not used when reading data from the standard input. Therefore, in most cases, the amounts used by the program to compute the total are different from the amounts input by the user.

For instance, the incorrect amounts used by the program when the user inputs the values from the problem description are given in the comments below:

```
std::istringstream("212.40")
  >> std::get_money(m);
std::cout << std::showbase
  << std::put_money(std::stoi(m)); // £2.12
std::istringstream("1,200.00")
  >> std::get_money(m);
std::cout << std::put_money(std::stoi(m));
  // £.01
```

The program can be fixed by using the locale when reading the amounts of money, specifically by calling the **imbue()** function on the **std::istringstreams** before they are used to read the values input by the user, as shown below.

```
std::istringstream firstItemStream("212.40");
firstItemStream.imbue(std::locale("en_GB"));
firstItemStream >> std::get_money(m);
std::cout << std::showbase
  << std::put_money(std::stoi(m)); // £212.40
std::istringstream secondItemStream(
  "1,200.00");
secondItemStream.imbue(std::locale("en_GB"));
secondItemStream >> std::get_money(m);
std::cout << std::put_money(std::stoi(m));
  // £1200.00
```

Improving code used to read amounts of money

There are several changes that can be made to the code used to read amounts of money in order to make it more maintainable and improve the user experience.

First of all the code for reading amounts of money is duplicated, namely it is defined once for reading the first amount and once for reading the next amounts. Instead the code could be extracted into a function and therefore be defined only once.

Secondly the **std::istringstream** objects that are used to read the amounts of money are constructed using a copy of the line of input read from the standard input. Since C++20 it is possible to move-construct **std::istringstream** objects using a **std::string** rvalue reference. However, an alternative approach, which we will use here, that is not restricted to C++20 is to stop using **std::istringstream** objects and read the input directly from **std::cin**. Note that this means that the **imbue()** function should be called on **std::cin** in order for the locale chosen by the user to be employed. Also the **<sstream>** header include can be removed.

Thirdly if the value input by the user is an invalid amount of money then the **std::stoi(m)** method call will throw an exception which is not caught, the program therefore crashes and the total so far is not printed to

the standard output. If I were a user of this program the last thing I would want is for the program to crash when I am entering the tenth amount incorrectly after inputting the previous nine amounts correctly, therefore losing all of the correctly input values. For a better user experience whenever an incorrect amount is entered by the user the program could output the total so far, print a useful error message and exit gracefully. The total so far could then be used as the first amount when re-running the program which means none of the amounts correctly entered previously would need to be re-entered.

Fourthly for each line of input, the amount of money is read from the beginning of the line and any remaining input is silently ignored. For transparency purposes the program should print a message whenever any user input is ignored.

Finally the names of variables **m** and **t** do not indicate what these variables represent. Providing details about the semantics of variable **t** in a comment next to the variable declaration does not address the naming issue because it requires future maintainers of the code to jump from the place in the code where **t** is used to the variable declaration in order to find the comment which describes what the variable represents. A better solution would be to use a more descriptive variable name, for both variables **m** and **t**. For instance, **m** and **t** could be renamed to **newAmount** and **total**, respectively.

All of the improvements described in this section have been implemented in the **tryGetNewAmount()** function which is called from the **main()** function as shown below.

```
std::optional<int> tryGetNewAmount(
  std::istream& in) {
  std::string newAmount;
  in >> std::get_money(newAmount);
  if (in.fail()) {
    std::cerr << "Ignoring invalid amount "
      "of money...\n";
    return {};
  }
  std::string extraInput;
  std::getline(std::cin, extraInput);
  if (!extraInput.empty()) {
    std::cerr << "Ignoring the extra input: "
      << extraInput << '\n';
  }
  return std::stoi(newAmount);
}
int main(int, char** argv) {
  ...
  if (argv[1]) {
    const std::locale locale(argv[1]);
    std::cout.imbue(locale);
    std::cin.imbue(locale);
  }
  std::cout << "First item: ";
  int total =
    tryGetNewAmount(std::cin).value_or(0);
  while (std::cin) {
    std::cout << "Next item: ";
    total +=
      tryGetNewAmount(std::cin).value_or(0);
  }
  std::cout << std::showbase << "Total: "
    << std::put_money(total) << std::endl;
}
```

Avoiding undefined behaviour

The approach used to verify if the locale was specified on the command line is to check if **argv[1]** is not null. Using this approach can lead to undefined behaviour in a certain scenario.

Specifically if the number of arguments **arg**` is zero then the value of **argv[1]** is undefined. As it stands, the program will access the value of **argv[1]** even when **argc** is zero. Note that the C++(17) standard specifies in Section 6.6.1, paragraph 2 that:

- The value of argc shall be non-negative

  (i.e. **argc** can be zero).
- If **argc** is nonzero, these arguments shall be supplied in **argv[0]** through **argv[argc-1]** as pointers to the initial characters of null-terminated multibyte strings

  where by "these arguments" the standard refers to the "arguments passed to the program from the environment in which the program is run".
- The value of argv[argc] shall be 0.

To the best of my knowledge, the C++ standard does not specify what the value of **argv[argc + 1]** should be.

An example of a scenario in which **argc** would be zero is if the program used to compute the total amount of money, called **totals**, would be executed using an **exec**-type function to which an array of **nullptr** command line arguments was provided as input, an example of which is given below.

```
#include <unistd.h>
int main() {
  char* args[] = {nullptr};
  execv("./totals", args);
}
```

In order to eliminate the opportunity for undefined behaviour the value of **argc** should be used to determine if the locale was provided as input on the command line.

In addition the program should check if the locale specified by the user is recognized by the operating system, and if it is not, then it should not crash due to an unhandled exception thrown by the **std::locale** constructor. Instead a user-friendly message should be printed and the program should exit gracefully.

The updated code used to handle the locale is given below.

```
std::optional<std::locale> tryGetLocale(
  char* cmdLineArg) {
  try {
    return std::locale(cmdLineArg);
  } catch (const std::runtime_error&) {
    return {};
  }
}
int main(int argc, char** argv) {
  if (argc != 2) {
    std::cerr << "The program should be "
      "executed using a single command-line "
      "argument representing the locale "
      " and it was not. "
      "Exiting...\n";
    return 1;
  }
  std::optional<std::locale> locale
    = tryGetLocale(argv[1]);
  if (!locale) {
    std::cerr << "The operating system does "
      "not have a locale named: "
      << argv[1] << ". Exiting...\n";
    return 2;
  }
  std::cout.imbue(locale.value());
  std::cin.imbue(locale.value());
  ...
}
```

Other minor improvements

Additional minor improvements that could be implemented are:

- The `totals.c` file contains C++ code, not C code. Therefore the file extension should be changed from `.c` to `.cpp`.

- `std::strings` are used but the `<string>` header is not explicitly included. It should be.
- There does not seem to be a reason for flushing when printing the total at the end of the `main()` function. Therefore `std::endl` could be replaced with `'\n'`.

Conclusions

In conclusion the recommended improvements are to:

1. Use the locale when reading the values input by the user;
2. Refactor the code used to read amounts of money;
3. Improve the usablity of the program by catching exceptions, printing useful error messages and exiting gracefully;
4. Eliminate undefined behaviour by using `argc` to check the number of arguments provided on the command line.

For reproducibility purposes, the updated source code containing all the recommended improvements is given below.

```
--- totals.cpp ---
#include <iomanip>
#include <iostream>
#include <locale>
#include <optional>
#include <string>
std::optional<std::locale> tryGetLocale(
  char* cmdLineArg) {
  try {
    return std::locale(cmdLineArg);
  } catch (const std::runtime_error&) {
    return {};
  }
}
std::optional<int> tryGetNewAmount(
  std::istream& in) {
  std::string newAmount;
  in >> std::get_money(newAmount);
  if (in.fail()) {
    std::cerr << "Ignoring invalid amount of"
      " money...\n";
    return {};
  }
  std::string extraInput;
  std::getline(std::cin, extraInput);
  if (!extraInput.empty()) {
    std::cerr << "Ignoring the extra input: "
      << extraInput << '\n';
  }
  return std::stoi(newAmount);
}
int main(int argc, char** argv) {
  if (argc != 2) {
    std::cerr << "The program should be "
      "executed using a single command-line "
      "argument representing the locale "
      "and it was not. "
      "Exiting...\n";
    return 1;
  }
  std::optional<std::locale> locale
    = tryGetLocale(argv[1]);
  if (!locale) {
    std::cerr << "The operating system does "
      "not have a locale named: "
      << argv[1] << ". Exiting...\n";
    return 2;
  }
  std::cout.imbue(locale.value());
  std::cin.imbue(locale.value());
  std::cout << "First item: ";
  int total
    = tryGetNewAmount(std::cin).value_or(0);
  while (std::cin) {
    std::cout << "Next item: ";
    total +=
      tryGetNewAmount(std::cin).value_or(0);
  }
  std::cout << std::showbase << "Total: "
    << std::put_money(total) << '\n';
}
```

Commentary

This critique is a simple attempt to use `get_money()` and `put_money()`. As Hans in particular points out, there are a number of issues with how the money handling in C++ works, especially when changing locale, and it does appear to hard to use the facility for the input of data whose formatting is not in your control.

It is also a little strange that `get_money()` specializations are defined only for specializations of `std::basic_string` and for `long double`, and it also seems unfortunate that there is very little flexibility in the format of the input string. I share the conclusion that `get_money()` should probably generally be avoided.

I think between them the entries covered pretty well everything I might have said, so I've got nothing else to add.

The winner of CC 124

All the critiques correctly identified that the root error was the failure to `imbue` the desired locale into the `std::istringstream` objects used for the `get_money()` calls. A couple of people suggested simplifying the program to read `std::cin` directly, rather than via an ancillary `std::string`. It would, however, be a good idea to check before changing the code as it's not clear which parts of the behaviour the current program should be retained.

Again, several critiques noticed the duplication of the code to read the amount of money from the input and suggested ways to avoid this. Creating a helper function, such as Ovidiu's `tryGetNewAmount()`, is a useful lesson in code design for the original author.

Hans realized that, since the numeric value of the input amount was required, using the overload of `get_money()` that takes a `long double` avoids the need in the original code to convert the string into a number (by calling `std::stoi()`.)

Locale handling is troublesome in C++ because there is little standardization in the set of supported locales. Hence it is usually a good idea important to anticipate problems and provide helpful diagnostics to assist the user of the program in diagnosis of failure.

As Dave points out, it is by no means clear the `en_GB` used in the problem description is the best choice, even for a British English user.

There were other pieces of advice given to the writer of the code that should prove useful – such as using good **names** for variables, and reducing the scope of variables when possible.

While there were several good entries this time round, I think that Hans' critique was (just!) the best one for I have awarded him the prize for this critique.

As always, thank you to all those who participated!

Code Critique 125

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

> I was reading about approximating π by working out the area under the quarter circle by dividing it into thinner and thinner slices, and treating each slice as a quadrilateral, and was curious about to know rapidly the approximation approached the actual value. So, I wrote a program where you provide the accuracy (or accuracies) you want and it tells you how many slices you need to get this close to the right value. However, when I run it, I noticed I always seemed to get a power of 2 for the number of slices – and I don't think this can be correct. What have I done wrong?

Further Comments on Code Critique 123

Additional information has been sent in response to an earlier Code Critique Competition.

Steven Singer <steven@pertinentdetail.org> writes in with some additional reflections on the last issue's code critique. For reference, the code is in Listing 1 (`pow.c`) and Listing 2 (`test.c`), both on the next page.

This is the original problem:

> 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…

Steven writes...

There is a significant optimisation that was missed in the answers to code critique #123, and I think it falls into the category of things *CVu* readers like to read about.

Marcel Marré mentioned something in passing at the end of his critique that could be interpreted as the same optimisation but it was bound up in comments about avoiding floating point. I think the optimisation is useful in its own right, and simple enough to explain if we stick to using floats.

But, before I get there, I learnt a couple of lessons whilst writing code to show the optimisation of interest, and I think those lessons are also worth passing on.

In these examples, I'm not going to worry about the return value for inputs less than one. That was covered adequately in the existing code critiques and I want to emphasise the core of the functions.

To test my code, I built a test framework to run the various code samples so I could benchmark them and check the results. I ran test cases for all the results that should produce a positive result and all the adjacent integers that should produce a negative result for a total of a little over 140,000 test cases.

I'm going to show some code samples. I'll use the typedef **`domain`** for the integer type used for the input and calculations to distinguish it from the integers used for other purposes (such as return values) and to allow me to try other types (unsigned and/or long integers).

The first thing I learnt was that the compiler I was using would quite happily hoist the **`sqrt(A)`** used as the loop limit out of the loop, but it wouldn't hoist the **`log(A)`**. That surprised me. I expected it to spot both or neither. I created a basic function with these optimisations that I could use as a timing baseline (see Listing 3, overleaf).

Note that the equality comparison between the result of the **`pow()`** and **`A`** is safe as the mantissa of the double has many more bits than the integer

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.co.uk

Code Critique Competition 125 (continued)

The code is in Listing 2, overleaf.

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

This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
#define _USE_MATH_DEFINES // for MSVC
#include <cmath>
using std::abs, std::pow, std::sqrt;
// Area of the quadrilateral inside this slice of
// the quadrant from [x, x+width)
double slice(double x, double width)
{
  double left, right;
  // By Pythagorus
  left = sqrt(1 - pow(x, 2));
  x += width;
  right = sqrt(1 - pow(x, 2));
  return (left + right) / 2 * width;
}
#include <iostream>
#include <string>
int main(int argc, char **argv)
{
  int slices;
  double accuracy, deltax, pi, quad, x;
```

Listing 2 (cont'd)

```
  if (argc < 2)
  {
    std::cout << "missing argument(s)";
    return 1;
  }
  for (int arg = 1; arg < argc; arg++)
  {
    accuracy = std::stod(argv[arg]);
    slices = 1;
    for (;;)
    {
      deltax = 1.0 / slices;
      quad = 0;
      for (x = 0.0; x < 1.0; x += deltax)
      {
        quad += slice(x, deltax);
      }
      pi = quad + 4;
      if (abs(M_PI - pi) < accuracy)
        break;
      slices++;
    }
    std::cout << slices << "=>" << pi << '\n';
  }
}
```

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;
}
```

being used to hold **A**, and we're giving **pow()** representable integer inputs. If the mantissa weren't big enough, then we'd need to either use a longer floating point type or a whole different strategy. Adding a tolerance is not good enough, if the answer is out by one, then we need to return zero even if our input is $2^{128}-1$.

On the random machine I was using (a 2.16 GHz Celeron N2840, so moderately modern but bottom of the range, with GCC 5.4.0 -O3), this took 243 seconds to run all my test cases. Code samples from the other answers using the same strategy had similar performance, varying slightly depending on whether the **log(A)** had been hoisted.

The interesting exception was the code from Marcel Marré, which was significantly faster. It took 76 seconds to run the tests. I could get this down to 56 seconds by rewriting the code to avoid long integers as follows:

```
int marre_no_long(domain A)
{
  if (A == 1) {
    return 1;
  } else {

    domain root_A = round(sqrt(A));
    domain a;

    for(a = A % 2 == 0 ? 2 : 3; a <= root_A;
        a++) {
      if (A % a == 0) {
        domain limit = A / a;
        domain p = a * a;
        while(p <= limit)
          p *= a;
        if (p == A)
          return 1;
      }
    }
    return 0;
  }
}
```

Listing 3

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

At first this result makes no sense. Marcel's code may be avoiding floating point operations in the loop but those are fast on modern computers (not as fast as integer, but fast nonetheless). Instead he has a loop which must be expensive.

The answer lies in the pre-test he does. Before going into the long loop, he checks whether the target is a multiple of base that's been chosen (the line **if (A % a == 0)**). Clearly if the target. which is greater than one, is not a multiple of the base, then it can't be a positive integer power of the base.

This modulo operation is fast. It's a single integer operation instead of (at least) two floating point operations. Nearly all choices of base fail this test (for example, when the loop index is seven, six out of seven target values will fail the test). That was the second lesson.

That suggests we can optimise the simple code to perform this integer test. It will cost us an extra integer operation in a few cases, but it will save us two floating point operations in most cases (**log** and **pow** and associated rounding and comparisons). This gives code like this:

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

This reduces the run time to 29 seconds, so eight times faster than the moderately optimised float version and not a significant change in strategy. Not bad for such a small change.

But, we can do a lot better than this, and that involves going back to the problem we're trying to solve and optimising our algorithm not our code.

The problem we're trying to solve is given a number, A, find out whether it can be written as one integer to the power of another, x^y with $y \geq 2$. It's not possible to solve this directly for both variables. The technique chosen in the solutions so far is to make an exhaustive search of x and then, for each x, look to see if there's a solution for y. This is fairly easy to do by taking the log to base x of both sides of the equation $x^y = A$ to give $y = \log_x A$ followed by a test to see whether y is an integer.

The trick is to spot that we can do this the other way, we can make a guess at y and find x. We can take the yth root of both sides to give $x = \sqrt[y]{A}$ and then check whether x is an integer.

As with the original approach, instead of directly testing whether we get an integer result, we calculate x^y and check we get the original result.

We need to find a limit for the search. In both cases, the limit for the value we're varying is found by looking at the minimum value of the other

variable. So, when we're varying x, we need to allow for y being two which means that we need to search up to $x = \sqrt{A}$. When we're varying y, we need to allow for x being 2 which means we need to search up to $y = \log_2 A$. That gives us the following code:

```
int root_not_base(domain A)
{
  if (A == 1)
    return 1;
  int limit = log(A)/log(2) + 0.0001;
  for(int root = 2; root <= limit; root++)
    if (pow(round(pow(A, 1.0/root)), root) == A)
      return 1;
  return 0;
}
```

This runs through the test cases in just 0.82 seconds which is about three hundred times faster than the original. This is not a bad solution and it's quite short and therefore readable.

The next obvious question is can we do better. Well, of course we can but we're going to have to sacrifice a little readability.

For large inputs, we end up searching a lot of roots which will have a result between two and three. That's because the largest power of two below `INT_MAX` is 2^{30} but the largest power of three is 3^{19}.

It turns out that, since contemporary computers hold numbers in base two, and can perform bit operations quickly, there's a neat trick to check if a number is a power of two. If we use that to catch the powers of two, then we can terminate our search at $\log_3 A$. We have to reject two explicitly as that's 2^1 and so doesn't satisfy our requirements but, on the other hand, the trick gives us the answer we want for one. This code uses a slightly different way of exiting at the limit:

```
int root_opt(domain A)
{
  if (A == 2)
    return 0;
  /* Power of 2 or zero */
  if ((A & (A - 1)) == 0)
    return 1;
  for(int root = 2; ;root++) {
    double base = pow(A, 1.0/root);
    if (pow(round(base), root) == A)
      return 1;
    if (base < 3)
      break;
  }
  return 0;
}
```

This has got our run time down to about 0.53 seconds.

Can we do better? Well yes. We've not learnt our lesson and once again we've fallen into the trap of micro-optimising too early. We need to back to the problem and attack it with some more maths. We can note that if we can find an integer x for some power y, then we can also find a solution for any factor of y. Reversing that, if any of the factors of y doesn't have an integral solution for x then there's no solution for y. As an example, one million is 10^6, but we don't need to find that out by taking the sixth root. Since six is two times three, there are also the solutions 100^3 and 1000^2. Since we care only whether an answer exists, we can just look at prime values of y:

```
#define ARRAY_SIZE(x) (sizeof(x)/sizeof(x[0]))

int root_primes(domain A)
{
  if (A == 2)
    return 0;
  /* Power of 2 or zero */
  if ((A & (A - 1)) == 0)
    return 1;

  static const double primes[] = {
    2, 3, 5, 7, 11, 13, 17, 19
  }; /* Enough for uint32 */

  for(size_t i = 0; i < ARRAY_SIZE(primes);
      i++) {
    double root = primes[i];
    double base = pow(A, 1.0/root);
    if (pow(round(base), root) == A)
      return 1;
    if (base < 3)
      break;
  }

  return 0;
}
```

Instead of using floating point operations to try at most twenty-nine roots, we're only trying at most eight.

This has now got our run time down to about 0.23 seconds, a thousand times faster than the original micro-optimised code (for the mix of values in the test cases I was using).

This is as far as I have got in the optimisation. Perhaps other people can take it further.

We can look at the overall limit on the performance for the different approaches, mostly because no discussion of algorithms is complete without using Big O notation. The original approach was $O(\sqrt{N})$. This approach is $O(\pi(\log N))$ where $\pi(x)$ is the number of primes less than or equal to x. This can be approximated to give an overall order of $O(\log N / \log \log N)$ but that limit becomes relevant only at extremely large N. Using `long int`s and increasing the maximum input value to sixty eight trillion needs only one more prime.

For comparison, a binary search of a table, as mentioned by Paul Floyd, is blindingly fast even just using *bsearch()*, ten milliseconds in my tests, which is probably below accuracy of my very dumb benchmarking. But as Paul correctly noted, it comes with the cost of a lot of memory. The size of the table scales roughly as the square root of the maximum value in the table. To extend it to sixty-eight trillion, increases the size of the table to over eight million entries.

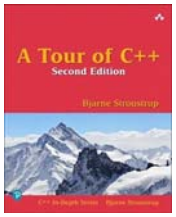
Reviews

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.

A Tour of C++ (Second Edition)



By Bjarne Stroustrup, published by Pearson Addison-Wesley, ISBN 0-13-499783-2, website: stroustrup.com

Reviewed by : Ian Bruntlett

Highly recommended

The Preface of this book states that it should be treated like a sightseeing tour of a city – but of C++ – after which the language can be explored in more detail.

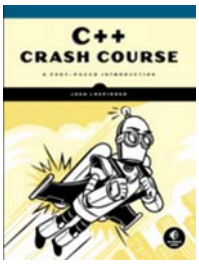
The first chapter deals with the basics – an overview of the fundamentals of C++ (the notation of C++, its model of memory and of computation) and the basic mechanisms for organising code into a program (procedural programming). The more advanced topics are dealt with in later chapters. There are a few example programs that the reader could type in. The rest are fragments. This is a pity but if you want examples to type in, there are other books that provide them. The next chapter deals with 'User-Defined Types' – structs/classes, unions (but prefer `std::variant`) and enum classes. The next chapter, 'Modularity', covers how C++ programs can be developed from individually developed parts – from functions through to class hierarchies and templates and logical modularity (namespaces). Exceptions and other error handling alternatives are covered, including contracts (sadly dropped from C++20) and static assertions.

A huge chunk of the C++ Language is covered by chapters 4 through to 7 and I will attempt to do them justice. Chapter 4, 'Classes', illustrates different uses of the class mechanism – concrete types, resource handles, abstract types, class hierarchies... with good examples. It left some details unanswered, though – I presume that the details are to be found in the author's other books (*The C++ Programming Language, 4th Edition*, published by Addison Wesley in 2013), the C++ standard or www.cppreference.com. The next chapter deals with 'Essential Operations', covering certain operations (construction, assignment, destruction) that the language makes certain assumptions about. It looks innocuous enough but it is full of essential information. It introduces the 'rule of zero' which states "either define all of the essential or none" – coupled with exceptions, this shows how to have memory and other resource handling performed automatically without the need for an explicit Garbage Collector. The next two chapters – 'Templates', 'Concepts and Generic Programming' are mind-bending and I will be supplementing the information given here with other C++ books.

The rest of the book (barring a 'History and Compatibility' final chapter which also details when certain features were introduced) is all about the standard library and would have been even better with more examples. It starts with an overview chapter and on to chapters on 'Strings and Regular Expressions', 'Input and Output', 'Containers', 'Algorithms', 'Utilities', 'Numerics', and 'Concurrency'. The book lacks a Glossary – however, the author provides one online at www.stroustrup.com/glossary.html.

This book is a useful, brief, introduction to C++17 (and hints of C++20 and later standards). I think its unique selling points are its brevity and the advice given by the author.

C++ Crash Course – A fast-paced introduction



By Josh Lospinoso, published by No Starch Press (2019), ISBN: 978-1-59327-888-5, website: ccc.codes

Reviewed by Ian Bruntlett

Highly recommended

This book consists of 'An Overture to C Programmers' followed by two main parts, 'The Core Language' and 'C++ Libraries and Frameworks'. At first glance, the order in which the language is introduced is somewhat unusual (but good) – it is intended for intermediate to seasoned programmers, so time will tell. Helpfully, most chapters end with a 'Further Reading' section. I used this book as a kind of C++17 Primer with examples. In addition, the text often explains example programs in great detail. I am not a C++ expert but I have done my best to verify my findings. No Starch Press kindly provided a physical and electronic copy of this work.

'An Overture to C Programmers' tries to sell this book – and C++ in general – to experienced C programmers as a kind of 'Super C'. It does give an assembler listing to show an optimisation of `constexpr` but only refers to an x86 book – there are plenty of x64 resources out there and most programmers using Intel/AMD hardware are most likely to be using x64 cores. I think that the examples should have been introduced with the proviso 'You are not intended to understand this… yet'.

Part 1 covers the core language in 9 chapters and just over 270 pages. It brings in the language concepts at the beginning. For simplicity and familiarity, it uses `printf` in its examples, rather than `iostreams`, which I feel is unfortunate.

The first chapter, 'Up and Running', explains an example C program, shows how to set up a C++ development environment (Visual Studio, Xcode, gdb). It can get quite involved and, ideally, you'll either know how to do it already or know someone else that does. It has a section on Bootstrapping C++ that introduces the basic concepts of C++. It finishes with a section on debugging with an example being debugged in the previously mentioned tools.

The next chapter, 'Types', is ambitious. It starts with fundamental types, and progresses through arrays and user defined types all the way to fully featured C++ classes – methods, access control, constructors and initialisation, and destructors. It is particularly detailed in its coverage of the different ways to initialise objects, concluding that in general always use `{}` except for when using certain **stdlib** classes. Strikingly, it overlooks the `protected` keyword.

The next chapter, 'Reference Types', concentrates on pointers and references and their differences. As is customary in some C++ texts, it recommends declaring a pointer variable as `int * my_ptr`. This can mislead the reader into thinking that you can declare two pointer

variables with `int * my_ptr, another_ptr`, which is wrong. It illustrates the hazards of sloppy pointer arithmetic. It does cover address space layout randomisation – but surprisingly it doesn't illustrate physical memory as a sequence of bytes, the machine stack or the free store. The `const` keyword (arguments, methods, member variables) are dealt with as is the `auto` keyword. The author recommends "As a general rule, use `auto` always" – which I believe is a bit extreme. Some examples are highly contrived and, at first glance, got me thinking 'What are they trying to achieve here?' but the abundant examples have greatly assisted my learning of C++. One thing bugged me – it kept on referring to C++ idioms as *Design Patterns*.

The 'Object Life Cycle' chapter discusses local, static, thread local and free store variables. It illustrates their behaviour. It gets one thing wrong – in its `thread_local` example, the relevant variable is never referenced so it got optimised out of existence. Some help from accu-general got it working properly. It also covers how to use and react to exceptions and provides a `SimpleString` class to show how constructors use exceptions and how destructors are used to release resources. It shows how naive `copy` semantics cause problems and how `move` semantics can add a welcome boost in performance. It does mention value categories, something I'll have to learn more about.

The 'Runtime Polymorphism' chapter washes its hands of implementation inheritance. It recommends the use of pure-virtual classes to implement interfaces. It discusses the use of constructor injection and property injection as well.

'Compile-time Polymorphism' is all about templates. It covers a lot of ground so some material is a bit thin because of that. I have some experience of putting templates into practice and even so, I found it necessary to re-read this chapter. It covers `const_cast<>` et al, a function template example, and a smart pointer template example. It also covers the Concepts TS and illustrates how to define your own concepts and the use of type traits as a compromise for situations where Concepts are not available. It recommends other books to build on this foundation.

'Expressions'. Everyone knows all about expressions, don't they? This chapter might change your mind. There is the usual operator precedence and associativity with a useful table. Unfortunately, when it comes to order of evaluation advice (on page 196) it is still based on C++11. See [1] for current wisdom. Because, to quote Bjarne Stroustrup's *A Tour of C++* 2nd Edition section 1.4, "The order of evaluation of expressions is left to right, except for assignments, which are right to left. The order of evaluation of function arguments is unfortunately unspecified". It shows how to overload new and delete and how to implement your own free store management. User defined literals are mentioned briefly. Type conversions are discussed in depth. There is an interesting `constexpr` example – `rgb_to_hsv`.

The 'Statements' chapter covers basics – the usual stuff plus namespaces, type aliasing, structured bindings, attributes, `constexpr if`. However, one example shows a range class for Fibonacci numbers, which I thought was particularly welcome.

The 'Functions' chapter finishes off Part 1. It seems fairly comprehensive, ranging from simple stuff through to mind-bending stuff (lambdas, templates). Again, the examples were really helpful.

Part 2 of this book (430+ pages) covers many things with chapters on testing, smart pointers, utilities, containers, iterators, strings, streams, filesystems, (`stdlib`) algorithms, concurrency and parallelism and Boost ASIO (network programming) and odds and ends for writing applications. For a lot of examples, the Catch2 TDD framework is used, to illustrate the behaviour being taught. A mixture of standard library and Boost's library facilities are covered here.

The 'Testing' chapter explores TDD using an extended example of a braking system for an autonomous vehicle. It starts off with simple assertions and then moves onto more ambitious facilities (Catch v2, Google Test, Boost Test). It covers Mocking Frameworks (Google Mock, Hippo Mock).

The 'Smart Pointers' chapter covers families of smart pointers (scoped, unique, shared, weak, intrusive). It finishes off with an example of a toy allocator.

The 'Utilties' chapter covers "a motley collection of tools". There are Data Structures (tribool, optional, pair, tuple, any, variant). There is quite a section on Dates and Times and finally Numerics (functions, complex numbers, constants, random numbers, numeric limits and compile time rational arithmetic).

The 'Containers' chapter covers a variety of containers from stdlib and Boost. It acts both as a tutorial and as an abbreviated reference. Some parts needed to cover broad topics, so further reading is required. Again, there are useful examples using Catch v2 to illustrate container behaviour.

The 'Iterators' chapter covers iterators with some very useful reference tables that lists the operations that different iterator types support.

The 'Strings' chapter is over 40 pages long, covering the varieties of `std::string`, a multitude of methods – including `std::string_view` and regular expressions from both stdlib and Boost.

The 'Streams' chapter is interesting but I feel that the later examples would be better understood after reading other C++ books.

The 'Filesystems' chapter covers stdlib and Boost facilities that provide platform agnostic facilities for interacting with the kinds of hierarchical provided by Linux and Windows and other operating systems. It covers a forest of facilities, not always supported by every platform – it can be difficult to separate the wood from the trees.

The 'Algorithms' chapter covers a big topic, is a large chapter with Catch v2 examples of algorithms in action, divided into the usual categories – non-modifying sequence operations etc.

The 'Concurrency and Parallelism' chapter is a necessarily short coverage of the topic. The examples require an up-to-date toolchain. There is quite a selection of examples to experiment with but I am not familiar enough with this topic to comment further.

The 'Networking with Boost ASIO library' chapter is present because the stdlib does not currently support networking. It is interesting but I am out of my depth with this chapter.

The final chapter, 'Writing Applications', is an assortment of useful things to know including `std::atexit`, `std::abort`, `std::system`, `std::getenv`, `std::signal`, and Boost ProgramOptions for dealing with command line parameters.

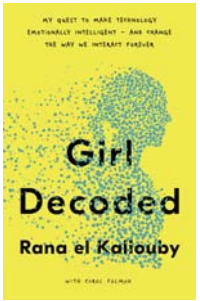
Conclusion: This is a comprehensive book, both in size (over 700 pages, taking 5 months to review) and breadth of coverage. It provides suggested reading at the end of most chapters and has a decent index. The examples are built using the CMake cross-platform system which helps a lot.

All in all, highly recommended.

References

[1] 'Refining Expression Evaluation Order for Idiomatic C++': http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0145r3.pdf

Girl Decoded: A Scientist's Quest to Reclaim our Humanity by Bringing Emotional Intelligence to Technology



By Rana el Kaliouby and Carol Colman, published by Penguin Random House, ISBN 9781984824769

Reviewed by Silas S. Brown

Recommended

I may be biased – Rana was my fellow PhD student (and she mentions me in this book) – but I must say it's a good read, although be warned Rana's story does have its dark times. Don't expect much technical content from this book: phrases like 'dynamic Bayesian networks' (in Rana's algorithm for interpreting facial expressions) are about as much detail as you'll get, although there's more of that in her thesis 'Mind-reading machines: automated inference of complex mental states', which is available

online. *Girl Decoded*" is Rana's memoir, showing how she became interested in the face's role in communication and wanted to unlock the medical potential of automated emotion detectors, especially for training people with autism (she collaborated with Simon Baron-Cohen, who had popularised the observation that autism is a spectrum not a single condition). Although her team later had to put this idea 'on hold' to focus on getting funding from market-research departments measuring test-audience reactions to advertising, they've opened their SDK for medical and safety benefits – autism training, suicide prevention support, measuring outcome of reconstructive plastic surgery, early diagnosis of Parkinson's disease (coded by a 15-year-old schoolgirl), healthcare-support robots, tools for self-training at speaking skills, and systems that remind inattentive drivers to stay safe. On the other hand, they refuse to let their SDK be used for surveillance purposes, turning down lucrative contracts even when the company was in trouble. The narrative is compelling and could be an inspiration to anyone wanting to get into coding, especially from a minority background.

One thing this book touches on only briefly is accuracy. No AI classifier will be right 100% of the time, and its limitations must be kept in mind to avoid 'automation bias' – the feeling that 'it must be true because a computer said it'. In this area, 90% is considered good, but that's still wrong one case out of every ten. But what surprised me in Rana's research is that humans can be even worse – in one situation involving subtle differences, they were answering wrongly as much as 30% of the time, a figure I like to quote whenever somebody thinks their colleague hates them because they can 'see it in their eyes' (there's a 30% chance you're misinterpreting) although it didn't end up in this book. If face-reading is to be used in job interviews, Rana would rather it be done by her algorithm than a human, and perhaps the most valuable piece of information in the book is the hint about what these algorithms will likely be set to look for.

While this is not the usual kind of technical book reviewed by ACCU, I think the ACCU readership would enjoy this and in some cases may be able to pass it on to inspire someone else to take up our craft.