the magazine of the accu

www.accu.org

{cvu}

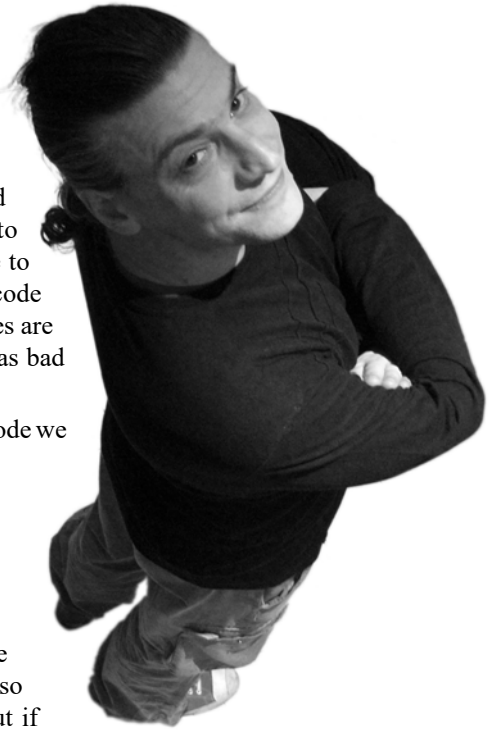
Volume 31 • Issue 5 • November 2019 • £4.50

Features

Regulars

{cvu}
Volume 31 Issue 5
November 2019
ISSN 1354-3164
www.accu.org

Sympathy for the Devil



The concept of 'Code Quality' has many facets. There is, right up front, the question of whether a piece of code is of good or bad quality. I think we all believe we recognise good or bad qualities in code. When care has been given to naming, and code is clearly laid out and simple to understand, we usually judge it as good. When code is hard to follow, too long or too wide, and names are meaningless tags for variables, we recognise it as bad quality, or at least, having less good quality.

We are usually making these judgements about code we read, which often means it was written by someone else. Are we always honest about making those same judgements of the code we've written ourselves? Pete Goodliffe discusses this in some detail in this edition in his latest column. One way of ensuring our own code lives up to the standards we cherish is to have it inspected by someone else. Your code may be so clear and simple that *you* can understand it, but if someone else believes it to be clear and simple too, then you can be really proud of it!

When we are deeply entrenched in solving some gnarly problem, it is all too easy to focus on how the code works and forget about how it looks. We can become blinkered about naming and blind to formatting. Code that provably works, runs quickly enough and passes the tests is all very well, but there are other qualities to which we need to remain vigilant. Having a fresh pair of eyes look critically over your work is a great way to avoid simple issues.

Even trivial suggestions can make large differences, such as "If this were named better, the intent would be clearer" or "Have you considered a unit test for this bit?"

The same applies, of course, when we are reviewing code for someone else. It's easy to be critical, but harder to be constructive. If we are sympathetic to each other, then everyone can enjoy good quality 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 31.6: 1st December 2019
C Vu 32.1: 1st February 2020

Overload 154: 1st January 2020
Overload 155: 1st March 2020

WRITE FOR C VU

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

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

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Coding Accountability

Pete Goodliffe looks at how we ensure we're living up to our own standards.

As iron sharpens iron,
so one person sharpens another.
Proverbs 27:17

I go to the gym. Frequently. I'm getting older, and my waistline is suffering. Perhaps it's guilt, but I feel I need to do something to keep it under control.

Now, let's be clear: I'm no masochist. Exercise is *not* my favourite thing in the world. Far from it. It ranks marginally above hot pokers being stuck in my eyes. There are plenty of things I'd rather do with my evenings. Many of them involve sitting down, preferably with a large glass of red wine and some code.

But I know that I *should* exercise. It's good for me. Is that fact alone enough to ensure I go regularly, every week?

It is not.

I dislike exercise and would gladly employ the weakest of excuses to get out of it.

What unseen force coaxes me to continue exercising regularly when guilt alone can't drag me out the door? What magical power leads me on where willpower fails?

Accountability.

I exercise with a friend. That person knows when I'm slacking, and encourages me out of the house even when I don't fancy it. That person turns up at the door, as arranged, before my lethargy sets in. I perform the same service back. I've lost count of the times that I wouldn't have headed into the gym, or would have given up halfway had I not had someone there, watching me and working alongside me.

And, as a by-product we enjoy the effort more for the company and shared experience.

Sometimes we *both* don't feel like it. Even if we admit it to each other, we won't let each other off the hook. We encourage ourselves to push through the pain. And, once we've exercised, we're always glad we did it, even if it didn't feel like a great idea at the time.

Stretching the metaphor

Some metaphors are tenuous literary devices, written to entertain, or for use as contrived segues. Some are so oblique as to be distracting, or form such a bad parallel as to be downright misleading.

However, I believe this picture of accountability is directly relevant to the quality of our code.

We hear our industry experts, speakers, writers, and code prophets talk about producing good, well-crafted code. They extol the virtues of 'clean' code and explain why we need well-factored code. But it matters not one jot if, in the heat of the workplace, we can't put that into practice. If the pressures of the codeface cause us to shed our development morals and resort to hacking like uninformed idiots, what use is their advice?

The spirit is willing, but when the deadline looms, all too often the flesh is weak. We can complain about the poor state of our codebases, but who do we look at to blame?

We need to bake into our development regimen ways to avoid the temptation for shortcuts, bodges, and quick fixes. We need something to lure us out of the trap of thoughtless design, sloppy, easy solutions, and half-baked practices. The kind of thing that costs us effort to do, but that in retrospect we're always glad we *have* done.

How do you think we'll achieve this?

Accountability counts

I know that in my career to date, the single most import thing that has encouraged me to work to the best of my abilities has been *accountability*, to a team of great programmers.

It's the other coders that make me look good. It's those other coders that have *made* me a better programmer.

> Being accountable to other programmers for the quality of your work will dramatically improve the quality of your coding.

That is a single simple, but powerful idea.

When I look at other teams, when I review other software companies, it is increasingly clear that the stellar coding outfits – the organisations that really pack and punch – are those that hold each other accountable for code design and quality.

Code++

To ensure you're crafting excellent code, you need people who are checking it at every step of the way. People who will make sure you're working to the best of your ability, and are keeping up to the quality standard of the project you're working on. (This is one of the reasons open source code is often of higher quality than proprietary code: you know that many other programmers will be looking at your work.)

This needn't be some bureaucratic big-brother process, or a regimented personal development plan that feeds back directly into your salary. In fact, it had better not be. A lightweight, low-ceremony system of accountability, involving no forms, no lengthy reviewing sessions or formal reviews is far superior, and will yield much better results.

Most important is to simply recognise the need for accountability; that being answerable to other people for the quality of your code encourages you to work your best. Realise that actively putting yourself into the vulnerable position of accountability is not a sign of weakness, but a valuable way to gain feedback and improve your skills.

How accountable do you currently feel you are for the quality of the code you produce? Is anyone challenging you to produce high-quality work, to prevent you from slipping into bad, lazy practices?

Accountability is worth pursuing not only in the quality of our code output, but also in the way we learn, and how we plan our personal development. It's even beneficial in matters of character and personal life (but that's a whole other topc).

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 it work

There are some simple ways to build accountability for the quality of code into your development process. In one development team we found it particularly useful when all the coders agreed on a simple rule: *all code passed two sets of eyes before entering source control*. With this as a peer-agreed rule, it was *our choice* to be accountable to one another, rather then some managerial diktat passed down from faceless suits on high. Grassroots buy-in was key to this success of the scheme.

To satisfy the rule, we employed pair programming and/or a low-ceremony one-on-one code review, keeping each checked-in change small to make the scheme manageable. Knowing another person was going to scrutinise your work was enough to foster a resistance to sloppy practice and to improve the general quality of our code.

> If you know that someone else *will* read and comment on your code, you're more likely to write good code.

This practice genuinely improved the quality of the team, too. We all learnt from one another, and shared our knowledge of the system. It encouraged a greater responsibility for and understanding of the system.

We also ended up with closer collaboration, enjoyed working with each other, and had more fun writing the code as a consequence of this scheme. The accountability led to a pleasant, more productive workflow.

Setting the standard

When building developer accountability into your daily routine, it is worth spending a while considering the benchmark that you're aiming for. Ask yourself the following questions:

How is the quality of your work judged? How do people *currently* rate your performance? What is the yardstick they use to gauge its quality? How do you think they *should* rate it?

- The software works, that's good enough.
- It was written fast, and released on schedule (internal quality is not paramount).
- It was well-written, and can be maintained easily in the future.
- Some combination of the above.

Which is seen as most important?

Who currently judges your work? Who is the audience for your work? Is it only seen by yourself? Your peers? Your superiors? Your manager? Your customer? How are they qualified to judge the quality of your handiwork?

Who *should* be the arbiter of your work quality? Who really knows how well you've performed? How can you get them involved? Is it as simple as asking them? Does their opinion have any bearing on the company's current view of your work's quality?

Which aspects of your work should be placed under accountability?

- The lines of code you produce?
- The design?
- The conduct and process you used to develop it?
- The way you worked with others?
- The clothes you wore when you did it?

Which aspect matters the most to you at the moment? Where do you need the most accountability and encouragement to keep improving?

The next steps

If you think that this is important, and something you should start adding to your work:

- Agree that accountability is a good thing. Commit to it.
- Find someone to become accountable to. Consider making it a reciprocal arrangement; perhaps involve the entire development team.
- Consider implementing a simple scheme like the one described above in your team, where every line of code changed, added, or removed must go past two sets of eyes.
- Agree on how you will work out the accountability—small meetings, end-of-week reviews, design meetings, pair programming, code reviews, etc.
- Commit to a certain quality of work, be prepared to be challenged on it. Don't be defensive.
- If this happens team-wide, or project-wide, then ensure you have everyone's buy-in. Draft a set of team standards or group code of conduct for quality of development.

Also, consider approaching this from the other side: can you help someone else out with feedback, encouragement, and accountability? Could you become another programmer's moral software compass?

Often this kind of accountability works better in pairs of peers, rather than in a subordinate relationship.

Conclusion

Accountability between programmers requires a degree of bravery; you have to be willing to accept criticism. And tactful enough to give it well. But the benefits can be marked and profound in the quality of code you create. ■

Questions

- How are you accountable to others for the quality of your work?
- What should you be held accountable for?
- How do you ensure the work you do today is as good as previous work?
- How is your current work teaching you and helping you to improve?
- When have you been glad you kept quality up, even when you didn't feel like it?
- Does accountability only work when you *choose* to enter into an accountability relationship, or can it effectively be something you are *required* to do?

Exodep : A Simple External Dependency Refresher

Pete Cordell introduces a library dependency tool for C++ projects.

A while back (May 2016), an email on the ACCU-General mailing list lamented the lack of third-party library download tools for C++. Recent tweets have echoed this deficiency. Languages such as Python and Ruby have tools like pip and rubygems, which are closely integrated with their respective ecosystems. A few attempts have been made to do the same for C++, such as biicode (now defunct) and conan.io [1], but so far it seems their adoption has been weak. Git submodules have also been used in this context, but often the reported experience is underwhelming.

For my own purposes, I work on a number of projects and have common bits of code used in each. Often these are small bits of code, such as string `ends_with()` level functionality or the legendary `left-pad` [2]. These often slowly collect functionality as subsequent projects need slightly different features. I also work on my desktop and laptop. So rather than have common libraries on each machine, I like to include the relevant library in each project's repo. This makes moving from machine to machine easier and also means I know exactly what my customer has working on their systems when it comes to dealing with bug reports.

For a long time I struggled with manually copying code from the library development area to the relevant project. This was tedious with .h and .cpp files being in different directories. There was also the nagging concern that during the manual copying process I might accidently copy code in the wrong direction, copying an old version of code from the project area into the library development area.

This and the ACCU-General mailing list discussion spurred me on to see if I could come up with a solution at least for my needs, but maybe also one that benefits others.

The result is something I have called 'exodep', and the code is on Github [3].

Design goals

Before I started developing a solution I identified a number a design goals the solution should attempt to satisfy. These included:

- **No Special Server Software** Working both on a desktop and a laptop meant that central repos such as GitHub, BitBucket and GitLab were ideal places for storing the library code I was interested in. I also wanted to make it easy for other people to make code available. Further, I recognised that some people may want to use the solution internally on their own private systems. That meant the server had to be simple, requiring no more than an HTTP server or just a file share.
- **'Democratic'** I wanted to avoid having a central registry where components are registered, such as with Ruby Gems. I didn't want to have the responsibility of running such a registry, nor did I want to run the risk of a central registry losing funding and going off air. I hope the latter will remove a barrier to people thinking of investing their time and code into the approach. I only have a partial solution to this, but what I have I will discuss later.
- **Exodep isn't all there is** When you dig into dependency management, you find it can get quite involved. For example, there is a vast range in the amount of code that may need to be downloaded. It could involve a single header file or something as big as Boost. Making exodep cater for all situations would complicate it. Therefore, I made the focus of exodep very much at the lower end, downloading low dozens of files. Equally, there is no revert or undo facility. The version control system can be used for that.
- **Minimal Magic** and **Dumb Software, Smart People** As mentioned, the complexities of dependency management can get involved. Solutions that automatically perform dependency management at run time or build time need to be smart and they potentially restrict how developers can structure their code. To allow a simpler tool I decided to use the smarts in developers and require dependency updates to be a human supervised activity. For example, typically this might involve performing a pull, refreshing dependencies, checking and fixing the build and pushing back. The `left-pad` saga [2] satisfied me that this was an acceptable situation. It also means you are able to dictate when updates are made to your code rather than the third-party software provider. My experience with Windows Updates has confirmed to me that that this approach was more than justifiable!
- **Multiple dependent versions are too hard** One of the harder problems in dependency management is two dependencies that each use incompatible versions of a third dependency. My clever solution to this problem was to decide that this was 'not my problem' and it is an issue for the third dependency to sort out. For example, where incompatibilities come into play, the third dependency should either be given a new name or have some mechanism to configure it in such as way that the two versions can co-exist, perhaps via clever namespace management.
- **Source Only** Some dependency tools are able to download binary versions of a library, after working out which particular build you need. My experience providing binaries for multiple platforms is that it's a fool's game. There are multiple versions of gcc, clang and Visual Studio. They can be built in debug and release modes. Visual Studio, at least, can have STL debugging enabled or not, use static or dynamic runtime and so on. You quickly end up with a combinatorial explosion. Supporting only source code download avoids this and also means a developer has full visibility of the code they are using (if they care to look). A side advantage is that exodep can be used on any source code, not just C++.
- **Visibility** For a number of tools, the 'recipes' are stored on a central server and only the name of the recipe is included in the local configuration. This makes it harder a-priori to know what will be downloaded. In line with the dumb software, smart people policy, I wanted the full recipe to be visible on the local machine. In addition to visibility, aside from allowing the developer to tweak and tune the download, it allows easier debugging of the update process. A side

PETE CORDELL

Pete Cordell started with V = IR many decades ago and has been slowly working up the stack ever since. Pete runs his own company, selling tools to make using XML in C++ easier. Pete can be reached at accu@codalogic.com.

issue of this is that, if the build is not working properly due to a dependency not downloading properly, the first action should be to check that the locally stored recipe is up to date.

A side benefit of this is that if you find a piece of code on the Internet that doesn't have an exodep recipe, you can create one yourself. In fact, you can create your own library of exodep files referring to other code out in the Internet.

- **Easy Setup** I wanted to make it easy to add a new dependency to a project, especially minimising the amount of typing of convoluted identifiers required. Making this aspect simpler has been a recurring theme as the software has evolved.
- **Readily Configurable** Even though a dependency should be easy to set up, that should ideally not be at the expense of the developer being able to structure the resulting code to their liking. This should be possible without having to directly modify the recipe file downloaded for the library.

Implementation Language

The principle action required for exodep is an HTTP download. I could have gone a long way with a solution written in Bash and using wget. But this gave me two problems: (1) the recipe and implementation mechanics would be highly intertwined if included solely in a Bash script, and (2) I work on Windows. I decided a scripting language (rather than a compiled language like C++) was the way to go as this would give me cross-platform support 'out-of-the-box' and performance was unlikely to be affected by being script based. I am a fan of Ruby, so that was an option, but in the end I opted for Python3 as I believe this is more widely deployed and developers have more experience with it.

Simple Beginnings

A Github URL for a specific raw file in a repository has the following structure:

```
https://raw.githubusercontent.com/{owner}/
{project}/{branch}/{path}{file}
```

Bitbucket and Gitlab have similar formats.

To avoid having to repeat the bulk of the URL, having a URL template and some variable substitution was an obvious way to go. This led to an exodep file format that looked something like:

```
uritemplate https://raw.githubusercontent.com/
${owner}/${project}/${strand}/${path}${file}
$owner codalogic
$project exodep
get exodep.py
```

The command `$owner codalogic` sets the variable `owner` to `codalogic`. When executing the exodep `get` command, the `${owner}` and `${project}` fields in the uri template are substituted with the contents of the respective `owner` and `project` variables. Requiring braces around the substitution variable name allows the variable to be immediately followed by an alphanumeric character. The `${path}` variable defaults to the empty string, but can be set by the user. The `${strand}` field has some magic to it, but defaults to `master`. The `${file}` field comes from the `get` command.

In practice the Github uri template is the default, so there's no need to specify it if that's what you want. The commands `hosting bitbucket` and `hosting gitlab` can quickly set up the uri template for those other hosting platforms.

This leads to simple recipes. For example, for Phil Nash's Catch2 unit test library, a basic exodep recipe might look like this:

```
$owner catchorg
$project Catch2
get single_include/catch2/catch.hpp
```

In practice you may not want your files being placed where the dependency dictates and I'll come back to that.

The next consideration is where to place the recipe, or, as it turns out, recipes, that exodep reads. My first pass at implementing exodep consisted of including a file called `mydeps.exodep` in a projects top-level directory. This could then contain exodep `include` commands that would cause library specific exodep files to be processed. This approach cluttered up the top-level directory, which I didn't like, so I opted to define a directory called `exodep-imports` and include the `mydeps.exodep` file therein. I quickly realised that rather than having to edit the `mydeps.exodep` file to add include statements for all the exodep files I wanted read, I could just do a file glob of all the `.exodep` files in the `exodep-imports` directory. Consequently, to add another dependency, all you have to do is download its exodep file in the `exodep-imports` directory.

A project that wishes to be used as an exodep source would list its `.exodep` files in the `exodep-exports` sub-directory within its online repo. So, to use a library, I would copy the `.exodep` files listed in the library's `exodep-exports` sub-directory into my project's `exodep-imports` sub-directory. The exodep file would be named after the project. For example, the exodep file for Catch2 might be called `Catch2.exodep` or `catchorg.Catch2.exodep`.

Software structure of library

The intention is that exodep doesn't restrict how a developer lays out their source code files. That said, some structure is useful, as much as anything to avoid similarly named files from different libraries overwriting each other.

Languages like C++ traditionally divide their code into include files and source files sub-directories. This leads to default file names such as `include/libname/file.h` and `src/libname/file.cpp`.

The significance of this is that the `#include` statements in the C++ files work best with exodep when they have a form similar to:

```
#include "libname/file.h"
```

The compiler can then have the set of include directories it uses configured so it can find the needed files. (Exodep has a `subst` command if this is considered too restrictive, but I'm hoping it won't be used often.)

Customising the behaviour

The `get single_include/catch2/catch.hpp` command in the example above stores the downloaded file in the same location as specified in the source, i.e. `single_include/catch2/catch.hpp`. This doesn't allow any input from the developer in terms of customisation. I wanted to have the option for the user to say where they want the file to be stored.

We could change the command to:

```
get single_include/catch2/catch.hpp ${inc_dst}
```

But the developer may not want all include files stored in the same location. It's better to have the command say something like `${Catch2_inc_dst}`. The user then needs a way to set this variable while also allowing a sensible default.

Bearing in mind we want to include the project name in the path of the target location, the catch exodep file can set a default value for the `${Catch2_inc_dst}` variable by doing:

```
default $Catch2_inc_dst include/Catch2/
```

The `default` command says, if the variable is not set already, then use this value.

Now I needed to allow the user to set the `$Catch2_inc_dst` variable to something different. This is supported by the magic file `exodep-import/__init.exodep`. This is run prior to globbing the files in the `exodep-import` directory and allows setting values like the above and overriding the defaults set by the recipes.

When it comes to the user setting their own values, they may like the option to work at different levels of granularity. For example, they may want to set variables for each file that is downloaded. Or at the other

extreme they may want to say 'all external files should go into the 'external' sub-directory'.

To accommodate this, the exodep file for a dependency would have to include something like the following:

```
default $extern_dst
default $inc_dst ${extern_dst}include/
default $Catch2_inc_dst ${inc_dst}catch/
```

This allows the user to modify any of **$extern_dst**, **$inc_dst** and **$Catch2_inc_dst** in the `__init.exodep` file. Modifying **$extern_dst** or **$inc_dst** would have an impact on all processed exodep files.

But this is tedious to set up and error prone. I only had about 3 or 4 exodep files to work with when I got fed up playing around with this sort of thing.

Instead I added the **autovars** command. When used after the **$project** variable is set, this creates a whole bunch of variables similar to that above. There are sets for include files and source files, plus sets for equivalent files used as part of testing.

With this the catch file would look something like:

```
$owner catchorg
$project Catch2
autovars
get single_include/catch2/catch.hpp
  ${Catch2_test_inc_dst}
```

Fixing build errors

To update dependencies, exodep.py runs through the exodep files that you have downloaded, vetted and stored locally. This can present a problem if the remote library has had new files added to it and the set of files that your local exodep file downloads is insufficient for the library to build. The **authority** command addresses this. It downloads the referenced remote exodep file and compares it to the local copy. If there are any differences, it reports an error prompting you to review and update the exodep files needed to build your project. The `authority` command uses the various variables identifying the project and the active uritemplate, so an example command might be:

```
authority exodep-exports/myproject.exodep
```

A library may also use other libraries. The exodep files for these dependencies would also conveniently be stored in the `exodep-exports` of the library that uses them. Additionally, a project's exodep file can contain **uses** commands that show the URL of other libraries that this library depends on. Exodep itself does not use the URLs in the **uses** commands. It is left to the developer to go into detective mode and track down the other exodep files needed.

Conditionals

Sometimes you may wish to modify the behaviour of a recipe depending on the environment in which it is being run.

For example, you may wish to do different things when used on Windows or Linux. Or you may wish to change the behaviour depending on which directories exist on the target system.

To support this, exodep allows conditional operation of commands. The format of a conditional command is **<condition> <command>**. If the condition is true, then the command is executed.

For an OS dependent download you may wish to download a `.sln` for Windows and a Makefile for Linux. This can be achieved using something like:

```
windows get my-project.sln ${build_dir}
linux get Makefile ${build_dir}
osx get Makefile ${build_dir}
```

For the PHP scripts I have, I use something like:

```
ondir htdocs default $php_dst htdocs/
ondir httpdocs default $php_dst httpdocs/
ondir wwwroot default $php_dst wwwroot/
default $php_dst ./
default $my_project_dst ${php_dst}
```

The 'Registry' – searching for code

Most dependency downloaders like pip and rubygems have a central repository that allows you to search for code that you want to use. After all, a dependency downloader that doesn't have anything to download isn't much use. But, as mentioned earlier, I want to avoid a central repository.

My solution to this is to use Google (other search engines are available). The idea is to include in a project's repo short description a special tag name that is a combination of the word 'exodep' and the language of the library. For example, for C++ it would be 'exodep_cpp'. By including this in the description of your Github repo, Google would be able to index it. You then prefix your Google search with the tag and hopefully you will find what you want. For example, you might search for "exodep_cpp unit test". As there are currently very few 'exodep_cpp' tagged projects I haven't yet been able to determine whether this will actually work!

Windows friendly(er)

The simplest way to update dependencies using exodep is to run the exodep.py program on the command-line in a shell opened at the relevant directory. This is great for Linux, but less so for Windows. To make things easier for Windows users I have created a simple installation file that adds a registry setting that adds a right-click option to Windows Explorer. After installing this, updating dependencies is as simple as right-clicking in the folder in Windows Explorer and selecting the 'Run Exodep here' option.

Summary

I recognise that exodep is at the toy end of the spectrum when it comes to the world of dependency tools (and open source projects in general). But it scratches an itch I've had for a number of years now and it's been fun to develop. For me, it represents quite a good return for a mere 800 or so lines of Python (some of which could probably do with being deleted by now!).

Being so small, it is easy to update and extend. If you think it might help you, but you'd like other features, feel free to fork it and have a play.

If you have any libraries that you think may be of use to others, please consider adding an `exodep-exports` directory with a suitable exodep file and tagging the project description with a relevant search tag (e.g. 'exodep_cpp'). Or build your own library of exodep files and share them. ■

References

[1] Conan, the C/C++ Package Manager for Developers: https://conan.io/

[2] Chris Williams (2016) 'How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript', at https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/ (published 23 Mar 2016)

[3] exodep on GitHub: https://github.com/codalogic/exodep

Making a Linux Desktop: Painting Some Wallpaper

Alan Griffiths continues his series on coding with the Mir library.

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 are common between the majority of designs. For example, it is common for applications to draw 'windows' and for these to be combined onto the screen.

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

Last issue we finished with a basic but functional desktop shell. This time we'll extend that with the capability to launch full screen 'applications' within the desktop and use that to paint some a simple wallpaper.

Preparation

The code in this article needs Mir 1.2 (or later). On Ubuntu 18.04 (and later), Mir 1.4 is available from the mir-team/release PPA. It is also useful to install the **weston** package as the example makes use of **weston-terminal** as a Wayland based terminal application and the Qt toolkit's Wayland support: **qtwayland5**. And finally, the **g++** compiler and **cmake**.

```
$ sudo apt-add-repository ppa:mir-team/release
$ sudo apt install libmiral-dev mir-graphics-drivers-desktop
$ sudo apt install weston qtwayland5
$ sudo apt install g++ cmake
```

All these we installed last time; this time we'll use a few more:

```
$ sudo apt install libwayland-dev libfreetype6-dev libxkbcommon-dev
```

We need **libwayland-dev** to write our full-screen client, to draw some text and to support the keyboard handling we'll need in the following articles.

Mir 1.4 is available on Fedora and can be built from source for many versions of Linux.

Building the example

The full code for this example is available on github:

```
$ git clone https://github.com/AlanGriffiths/egmde.git
$ git checkout Article-2
```

Naturally, the code is likely to evolve, so you will find other branches, but the **Article-2** branch goes with this article. Assuming that you've MirAL installed as described above you can now build egmde as follows:

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

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

Running the example

After this you can start egmde:

```
$ ./egmde
```

This time the Mir-on-X window should have a coloured gradient with the words "Ctrl-Alt-T = terminal | Ctrl-Alt-BkSp = quit" at the bottom. But there's more: the background can be configured by the user. If you type:

```
$ ./egmde -help
```

you'll get a list of options that includes the following:

```
--wallpaper-top arg (=0x000000)      Colour of wallpaper RGB
--wallpaper-bottom arg (=0x92006a)   Colour of wallpaper RGB
```

These are two options that allow you to experiment with different colours for the background gradient. For example:

```
$ ./egmde --wallpaper-top 0x171717 --wallpaper-bottom 0x7f7f7f
```

Once you have found some favoured colours you can save them in a config file:

```
$ cat ~/.config/egmde.config
wallpaper-top=0x170000
wallpaper-bottom=0x7f1f1f
```

A background that can be configured by the user makes egmde look a little more finished than it did last time (even if it doesn't do cat photos).

Installing the example

If you install egmde and it will be added to the greeter menu (typically a 'cog wheel' on the sign-on screen):

```
$ sudo make install
```

Depending upon the greeter used on your system, it may be necessary to reboot before egmde appears as a login shell.

To register applications with deskop environments we install a .desktop file in `/usr/share/applications` and to register Wayland based desktops with the greeter we install a `.desktop` file in `/usr/share/wayland-sessions/`. The content needed for both cases is similar, so for this example we use the same `.desktop` file and install it to both places.

The example code

There's a lot more code (about 20×) than last time, so I won't attempt to put it all in a listing with the article:

```
$ wc -l *.cpp *.h
  616 egfullscreenclient.cpp
   90 egmde.cpp
  176 egwallpaper.cpp
   46 egwindowmanager.cpp
  201 printer.cpp
  236 egfullscreenclient.h
   58 egwallpaper.h
   43 egwindowmanager.h
   54 printer.h
 1520 total
```

egmde.cpp

First let's look at the changes to `egmde.cpp`:

```
#include "egwallpaper.h"
#include "egwindowmanager.h"
#include <miral/command_line_option.h>
#include <miral/internal_client.h>
```

There are two headers from the example project and three from the MirAL library. We'll see the use of the MirAL library in the rest of this file and examine the example files after that.

This creates a **`Wallpaper`** object and ensures that the runner calls **`stop()`** on it just before closing down the server:

```
egmde::Wallpaper wallpaper;
runner.add_stop_callback([&] {
  wallpaper.stop(); });
```

Instead of the window manager provided by MirAL we use our own. There is only a small change and we'll see what that is later:

```
~~set_window_management_policy~~
  ~~<MinimalWindowManager>(),~~
set_window_management_policy
  <egmde::WindowManager>(wallpaper),
```

These three lines provide configuration options for the wallpaper and start an 'internal client' application that provides the wallpaper:

```
CommandLineOption{[&](auto& option) {
  wallpaper.top(option);}, "wallpaper-top",
  "Colour of wallpaper RGB", "0x000000"},
CommandLineOption{[&](auto& option) {
  wallpaper.bottom(option);}, "wallpaper-bottom",
  "Colour of wallpaper RGB",
  EGMDE_WALLPAPER_BOTTOM},
StartupInternalClient{std::ref(wallpaper)},
```

As we saw above, the configuration options are not just 'command-line': like other Mir configuration options they can be provided via environment variables and a config file.

egfullscreenclient.★

The largest chunk of new code is an abstract **`FullscreenClient`** class that provides a framework for managing full-screen windows. While useful, both for this article and for the next, it doesn't use the Mir APIs and won't be explored further here.

egwallpaper.★

The example **`Wallpaper`** class (Listing 1) provides the interface required to start an 'internal client' application: two function call operators, one to provide the Wayland connection to use, one for the server-side 'session' used by the window manager to identify the application.

It also provides two functions we need to retrieve the 'session' and to stop the wallpaper application running on shutdown (as seen in **`runner.add_stop_callback()`** above).

Finally, it provides two functions to allow the wallpaper colour gradient to be configured.

Internally, the **`Wallpaper::Self`** class derives from **`FullscreenClient`** and provides (Wayland based) code to paint the wallpaper on each screen. This is also where the `printer.*` files are used; they are a convenience wrapper around **`libfreetype`** for drawing text into a graphics buffer. Once again, a close examination of that code goes beyond this exploration of the Mir APIs.

egwindowmanager.★

The example **`WindowManager`** class (Listing 2) extends the **`MinimalWindowManager`** we saw in the last issue. The only change it makes is to extend the **`place_new_window()`** to check the session against the wallpaper and, for the wallpaper, ensure that the window can never get input focus (and, as a side effect, is never raised above other windows). There's currently no Wayland extension that allows a client to exploit the rich semantics of Mir's window type systems. But, as this is an internal client, we can work-around it by setting the type ourselves.

Listing 1

```
class Wallpaper
{
public:
  void operator()(wl_display* display);
  void operator()(std::weak_ptr
    <mir::scene::Session> const& session);
  auto session() const ->
    std::shared_ptr<mir::scene::Session>;
  void stop();
  // Used in initialization to set colour
  void bottom(std::string const& option);
  void top(std::string const& option);
private:
  std::mutex mutable mutex;
  std::weak_ptr<mir::scene::Session>
    weak_session;
  uint8_t bottom_colour[4] = { 0x0a, 0x24, 0x77,
    0xFF };
  uint8_t top_colour[4] = { 0x00, 0x00, 0x00,
    0xFF };
  struct Self;
  std::weak_ptr<Self> self;
};
```

Conclusion

This article shows how the Mir library can provide a stable starting point for adding functionality to a 'desktop environment'. The next article will continue from this point and add a 'launcher' for selecting and starting applications. ■

References

- The Mir homepage: https://mir-server.io/
- The egmde git repo: https://github.com/AlanGriffiths/egmde

Listing 2

```
class WindowManager : public MinimalWindowManager
{
public:
  WindowManager(WindowManagerTools const& tools,
    Wallpaper const& wallpaper);
  auto place_new_window(
    ApplicationInfo const& app_info,
    WindowSpecification const&
      requested_specification)
  -> miral::WindowSpecification override;
private:
  Wallpaper const* const wallpaper;
};
// Implementation
auto WindowManager::place_new_window(
  ApplicationInfo const& app_info,
  WindowSpecification const&
    requested_specification)
-> WindowSpecification
{
  auto result =
    MinimalWindowManager::place_new_window(
    app_info, requested_specification);
  if (app_info.application() ==
    wallpaper->session())
  {
    // Setting the window type.
    result.type() = mir_window_type_decoration;
  }
  return result;
}
```

The Standard Report

Guy Davidson reports from the C++ Standards Committee.

In this report I'm going to cover the new language proposals that were voted into the standard at the recent meeting in Cologne. Thirty-two motions came before the committee from the Library Working Group (LWG) for voting. Rather than iterate through them all, I shall present some highlights. Recall that any paper can be reviewed by visiting wg21.link and appending the proposal number, for example https://wg21.link/P1000.

P0645 was a particular favourite of mine. While iostreams is the recommended API for reasons of extensibility and type safety, **`printf`** offers advantages such as the separation of formatting arguments and more compact source and binary code. However, this paper proposes a new text formatting library that can be used as a safe and extensible alternative to the **`printf`** family of functions. It is intended to complement the existing C++ I/O streams library and reuse some of its infrastructure such as overloaded insertion operators for user-defined types. The example message looks like:

```
string message = format("The answer is {}.", 42);
```

As a mathematician, I was delighted by the passing of P0631, Math(s) constants. This proposal introduces a new namespace, **`std::numbers`**, and populates it with thirteen new constants. From the paper:

> The library-defined partial specializations of math constant variable templates are initialized with the nearest representable values of e, log2e, log10e, π, 1/π, 1/√π, ln2, ln10, √2, √3, 1/√3, Euler-Mascheroni ? constant, and golden ratio ? constant (1+√5)/2.

One paper that got in just under the wire was P1754, 'Rename concepts to standard_case for C++20, while we still can'. Concepts are one of the big ticket features for C++20, having nearly made it into C++11, and the habit has been to name concepts Using Leading Capitals, unlike the rest of the standard which uses_space_case for its identifiers. There was a session of LEWG to decide on the sixty-eight new names and to resolve the accompanying naming collisions caused by migrating a big pile of identifiers to standard format.

If you are familiar with three-way comparison through **`operator <=>`**, also known as the spaceship operator because of its resemblance to a piece of ASCII art videogame heritage, you will know that it has broad application throughout the standard library. P1614 is the paper which makes ALL the changes necessary to make this happen. It is the work of Barry Revzin and an excellent piece of stable-cleaning: a fairly dull and error prone task that has been well-executed.

Modules have precipitated a change to the way we think about headers. It is obvious that we need a natural and portable mechanism to import the standard library into a modules-enabled compilation. To do this, we want to provide a minimal, uninventive mechanism to expose the existing standard library organization to modular compilations, not get in the way of, or create any impediment to, a proper reorganization of the standard library into named modules, and reserve space for such a future reorganisation. This is met by P1502, which reserves module names beginning with **`std`**. While it is easy to assume that nobody would do a thing like that, the standard now prevents you from doing so; otherwise your program is ill-formed.

Two dramatic papers cross a large bridge towards constexpr compilation by making string and vector constexpr. This is the work of Louis Dionne and seems at first glance to be rather counterintuitive: how does one allocate at compile time? However, with the loosening of requirements on constexpr in P0748 and related papers we can make it so. There is good reason to do so: these containers will be very useful in the reflection efforts currently under way for standardisation. P0980 deals with **`std::string`**, while P1004 deals with **`std::vector`**. The wording is very simple: add constexpr to the front of each declaration.

Some interesting issues were encountered while reviewing the implementations in libc++. For **`std::string`**, small string optimisation might be thought to complicate matters but it is implemented using unions, rather than **`reinterpret_cast`** which would render it unavailable for constexpr. Also, **`char_traits`** needs to be constexpr for at least its **`char`** specialisation. Meanwhile, **`std::vector`** uses **`try catch`** blocks, which aren't permitted in constexpr expressions; there is a paper in flight, P1002, which might assist this but it has yet to make the standard and is unlikely to reach C++20. No doubt the implementers will come up with a solution.

One of my pet peeves with the standard library is the return type of **`size()`**. It's an unsigned integer, which is inconsistent with the advice on the use of unsigned. The design of span was such that its size would return a signed integer, which pleased me greatly, but this decision was overturned in Rapperswil 2018. Despite pleading and protestation, in the name of consistency, P1523 makes **`size()`** return an unsigned integer. At least we have **`ssize()`**.

A whole new header was introduced to the standard by P1208. The header is called **`source_location`** which introduces a struct by that name. One of the goals of the committee in recent years has been to reduce dependency on the pre-processor. The introduction of modules has gone a considerable way towards this by eliminating the need for **`#include`**. P1208 provides an alternative to **`__FILE__`** and **`__LINE__`**, as possibly suggested by this identifier. It also offers the function name and the column number: I look forward to seeing what that might enable.

Finally, there was the motion from WG21 itself, rather than from any of the working groups:

> Move to appoint an editing committee composed of Daniel Kruegler, Davis Herring, Nina Ranns, and Ville Voutilainen to approve the correctness of the C++ working paper as modified by the motions approved at this meeting, and to direct the Convener to transmit the approved updated working paper for CD ballot.

C++20 is 'complete' and the next stage is to send it to the National Bodies for comments. These NB comments will be discussed at the next meeting in Belfast.

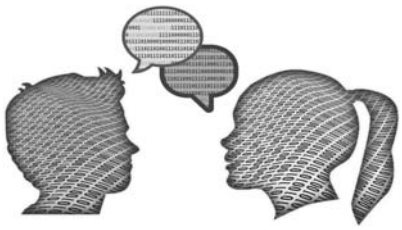
Normal business will still be carried out in Belfast and I'll report on that, but there will also be interesting procedural matters as NB comments are resolved. They will take priority over paper review since we must produce a draft for the National Bodies to agree to at the end of Belfast if we are to finalise the standard in Prague and send it to ISO for publication.

Stay tuned for more procedural detail.

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 120



Set and collated by Roger Orr. A book prize is awarded for the best entry.

Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Note: if you would rather not have your critique visible online please inform me. (Email addresses are not publicly visible.)

Last issue's code

I'm relatively new to C++ and I'm trying to build up a simple hierarchy of different entities, where their name and id and a few more characteristics are held in an **`Entity`** base class; and specific subclasses, such as **`Widget`** in the code below, hold the more specialised data – in this case simulated by the member name **`data`**. I've been having problems with the program I'm writing crashing and so I've simplified it down to a short example which usually produces output like this:

```
Test
none
Another widget
<core dump>
```

Please can you suggest what might be causing the core dump? I've had to use **`-fpermissive`** with gcc (but not msvc) – but surely that's not causing the problem?

Listing 1 contains `Entity.h`, Listing 2 is `Widget.h` and Listing 3 is `example.cpp`. As always, there are a number of other things you might want to draw to the author's attention as well as the presenting problem.

Listing 1

```
#pragma once
class Entity
{

private:
  char* pName{ NULL };
  int idNumber{ 0 };

public:
  void Setup(char const* name)
  {
    pName = strdup(name);
  }
  virtual ~Entity()
  {
    delete pName;
  }
  // Delete the current object using the dtor,
  // and re-create as a blank object
  void Reset()
  {
    this->~Entity();
    new(this) Entity();
  }
  char *Name()
  {
    return pName ? pName : "none";
  }
};
```

Listing 2

```
#pragma once
class Widget : public Entity
{
  int* data;
public:
  Widget() { data = new int[10]; }
  ~Widget() { delete data; }
};
```

Listing 3

```
#include <iostream>
#include <memory>
#include <string.h>

#include "Entity.h"
#include "Widget.h"

int main()
{
  Widget w;
  w.Setup("Test");
  std::cout << w.Name() << '\n';
  w.Reset();
  std::cout << w.Name() << '\n';
  w.Setup("Another widget");
  std::cout << w.Name() << '\n';
}
```

Critiques

Martin Janzen <martin.janzen@gmail.com>

Overly permissive?

The author is right about one thing: the error being masked by gcc's **`-fpermissive`** flag is not the cause of the core dump. The compiler is unhappy with the **`Entity::Name()`** method, which can return a non-const pointer to a string literal. Changing the return type to **`const char*`** (or **`char const*`**, to be trendy) removes the warning and makes the code safer, as it prevents the caller from attempting to overwrite **`pName`**'s memory – or the literal! – through the returned **`char*`**.

The **`Name()`** method itself ought to be const too, as it's clearly not intended to allow modifications to the Entity.

Placement new?!

Instead, the core dump is caused by the bizarre behaviour of **`Entity::Reset()`**, which invokes the object's dtor manually, then uses placement **`new`** to initialize a new **`Entity`**, for reasons known only to the author.

The problem is, **`Entity`** is meant to be used as a base class. Since its dtor is virtual, the call to **`this->~Entity()`** correctly causes the derived

ROGER ORR

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

class's dtor to be called. But `Reset()` then has no way to re-create the derived class; it can initialize only the `Entity` slice of the object.

In this example, `Widget` has allocated an array pointed to by `data`, which is destroyed by the explicit dtor call. But only the `Entity` ctor is called, not the `Widget` ctor; so the value of `data` is not reset. Therefore, when `Widget` goes out of scope in `main()`, `data` is deleted a second time, resulting in the core dump.

(This is easy to diagnose by running the code under valgrind or, in gcc or clang, by enabling ASAN with the `-fsanitize=address` flag. Either tool will show the stack trace at the time of each delete call, making it clear what happened.)

To fix this, `Entity::Reset()` shouldn't try to get so fancy: it should just clear its data members. Also, it should be made `virtual` so that when the object is to be reset to its initial state, derived classes get a chance to reset their own data members too.

Or, we could follow Herb Sutter's 'Non-Virtual Interface' idiom, keeping the public `Reset()` method non-virtual in the base class, but calling a private `virtual ResetImpl()` which the derived classes can override.

(Some programmers will try to invent other clever alternatives, using CRTP and the like. That seems like overkill, and an invitation for trouble, especially given that the author is "relatively new to C++".)

C vs. modern C++ style

The real question, though, is why all of this allocation and deletion is happening in the first place. As we've seen, it's error-prone in even a simple example, and completely unnecessary in this enlightened age.

Most importantly, the `Entity::pName` pointer should be replaced with a `std::string`. This single change:

- manages the memory for the name string safely and automatically;
- eliminates the memory leak which now occurs if `Setup()` is called twice;
- permits the dtor to be replaced by a simple `=default`; and,
- if the name is short enough for the small string optimization to be used, may even make the code run faster, with no allocations at all.

(There's a whole discussion that could be had about how best to handle sink arguments, as for `Setup()`. If this method were modified to take a `std::string` argument rather than a `char const*`, the argument could be moved into `Entity::name` rather than copied, possibly allowing the compiler to do even more optimization. But for now, let's assume that the interface is to be changed as little as possible.)

In the derived class, the `Widget::data` array has a fixed size, so it can be replaced by a C-style array or `std::array` member rather than allocating it on the heap. This allows both the ctor and dtor to be replaced with `=default`. Also, removing the unnecessary indirection makes the code just a bit more cache-friendly.

Basic hygiene

The `example.cpp` test program shouldn't have to know that it needs to include `<memory>` and `<string.h>` in order to use `Entity` and `Widget`. Each class's header should `#include` all of its dependencies, so that a would-be user doesn't have to worry about the finer points of the implementation (which, as we've seen, can and should change).

Since `Entity` appears to be intended as an abstract base class but has no pure virtual methods, its ctor could be given protected access so that it can be called only from derived classes; no standalone `Entity` instances can be created.

Speaking of access, `Entity::idNumber` is initialized to 0, but it has private access and is never touched in the `Entity` base class. It needs to be generated somehow by `Entity`, or given protected access, or simply removed; it's of no use to anyone right now. Similarly, `Widget::data` is also private but not used. But presumably this code is simply an early experiment, with more methods to be added later.

I won't waste any time on coding style except to note that the capitalized method names are a bit unusual and archaic-looking. As we're trying to minimize changes to the interface, they can be left as is.

Learning the craft

Finally, I think it'd be worthwhile to offer a couple of bits of general advice to the author.

- The use of `strdup()`, etc. suggests a C programming background, and/or some 1990s C++. It would be worth spending some time with one or two more recent books, such as *A Tour of C++* (the 2nd edition) and *Effective Modern C++*; also, at the *C++ Core Guidelines*, and at Bjarne's homepage at www.stroustrup.com, which has a couple of papers on the subject of how to write good modern C++.
- Most C++ compilers today are very, very good. As a rule, it's best to never ignore their warnings unless you are absolutely sure you understand the reasons both for the warning and for why it should be ignored in a specific situation. There is hardly ever a good excuse for this; it's good practice to strive for zero warnings.
- The `-fpermissive` flag, in particular, is a flagrant admission that your code is broken. It should never be needed when writing new code.
- Learn to use tools like valgrind and ASAN: they can save you from hours or days of frustration, or much worse.
- Finally, given how rapidly and substantially the C++ language is changing, do make sure to remain current by taking advantage of the endless resources available today: conferences, books, videos, web sites, courses, podcasts – and of course ACCU publications!

[ED: Full source code was supplied, but I elided it to save space.]

Hans Vredeveld <accu@closingbrace.nl>

Before we look into the core dump, let's look into the need for `-fpermissive`. With `-fpermissive`, you tell the compiler to ignore a group of errors and just compile it. Hopefully it will not bite you at runtime. In this case, `-fpermissive` is needed because the function `Entity::Name` can return the constant `none` (type `char const*`), while it advertises to return a (modifiable) `char*`. Just make it return a (non-modifiable) `char const*` and this problem is solved. Also, the function itself can be made `const`, giving the signature:

```
char const* Name() const
```

On to the core dump. Running the program on my Linux system with the environment variable `MALLOC_CHECK` set to 3 gives some information about what is going wrong:

```
*** Error in `./cc119': free(): invalid pointer:
0x00000000014e5c20 ***
Aborted (core dumped)
```

Apparently, we try to `free` some memory that can't be `free`d. For further investigation, let's build the program with address sanitizer (`-fsanitizer=address` in gcc/clang) and run it. The first thing address sanitizer complains about is a mismatch between `operator new[]` and `operator delete`. Indeed, in `Widget`'s constructor, we `new[]` an array, and in the destructor we `delete` a single object. Changing `delete` to `delete[]` in the destructor solves this issue.

Building the program again with address sanitizer and running it, shows that address sanitizer is not done with us. Now it complains that we used `malloc` to allocate memory and `delete` to deallocate it. The `malloc` is hidden in `strdup` in `Entity::Setup`. We can fix this by changing `delete pName;` in `Entity`'s destructor to `free pName;`. (Even better: change the type of `pName` from `char*` to `std::string`. Also rename it to `name` in that case.)

Still, we haven't found the reason for the core dump. Build it a third time with address sanitizer and run it. Now address sanitizer complains that we are attempting a double-free. In other words, we try to `free` something that we `free`d before. To understand why, we have to look at what happens when we call `w.Reset()` on a `Widget w`. Before we call

`w.Reset()` and after we constructed `w`, `w.data` is pointing to some allocated memory. Now, when we call `w.Reset()`, we first destruct `w` (via a call to the virtual destructor of `Entity`). The memory pointed to by `w.data` is deallocated, but the pointer itself is not changed. Next in `Reset()`, a new `Entity` object is constructed by the placement `new`. This placement `new` knows nothing about inheritance and does not construct a `Widget`. The memory previously occupied by `w.data` is left untouched and when interpreting the new object as a `Widget`, it contains the same `w.data` that points at unallocated memory. When `w` finally goes out of scope, we are going to destruct it as if it is a `Widget`, resulting in trying to delete the unallocated memory that `w.data` is pointing to.

We could solve this problem by making `Reset` a virtual function in `Entity` and implementing it in `Widget`, but it will be hard to create something that is robust and easy to understand and maintain. Before we continue, let's have a look at what the code really means.

In `main()` we begin with declaring and default constructing a `Widget` `w`. This `w` does not have a name, and conveys the meaning of no widget. It is just a placeholder for a widget. Next we call `w.Setup("Test")`, which converts `w` into 'Test' widget. This is the moment that we actually construct a `Widget`. Following this, we call `w.Reset()`. This destructs the widget `w` and changes it again into an empty placeholder. With the call to `w.Setup("Another widget")` we construct 'Another' widget at `w`. Finally, when `w` goes out of scope, we destroy the widget for the last time.

This analysis of the code makes clear that `Widget` has two functions:

1. It is a container of size at most one for `Widget`s.
2. It is a widget, but not a specific one. It represents widgets of different types.

Note that I talk about widgets here, but that everything I talk about is not implemented by the class `Widget`, but by `Entity`.

We can improve the code by separating the two functions. The container function can be implemented easily with an `std::unique_ptr<Entity>`. Now `Entity` no longer needs the functions `Setup` and `Reset`, but it will need a constructor:

```
class Entity
{
private:
  std::string name;
  int idNumber{ 0 };
public:
  explicit Entity(std::string name_) :
    name{name_}
  { }
  virtual ~Entity() = default;
  std::string const& Name() const
  {
    return name;
  }
};
```

Instead of a single class `Widget` that has to represent both a test widget and another widget, I would create two classes `TestWidget` and `AnotherWidget`. In this test program they will have similar implementations. For `TestWidget` this is (also changing its data member, to simplify its use):

```
class TestWidget : public Entity
{
  int data[10];
public:
  TestWidget() : Entity("Test")
  { }
};
```

Some attention must be given to copying and moving `Entity`s and `Widget`s. This is left as an exercise for the reader.

We can now rewrite `main()` as:

```
int main()
{
  std::unique_ptr<Entity> w =
    std::make_unique<TestWidget>();
  std::cout << w->Name() << '\n';
  w.reset();
  //Error: std::cout << w->Name() << '\n';
  w = std::make_unique<AnotherWidget>();
  std::cout << w->Name() << '\n';
}
```

Finally, we have to clean up the includes. `Widget.h` has to include `"Entity.h"`, `Entity.h` has to include `<string>`, and `example.cpp` has to include `"Entity.h"`, `"Widget.h"`, `<memory>` (which was not needed in the original code) and `<iostream>`. The include for `<string.h>` is not needed any more (originally, it was needed in `Entity.h`, not in `example.cpp`).

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

While the reason for gcc's complaint that `-fpermissive` must be used is indeed not the cause for the core dump, any programmer is generally ill-advised to ignore warnings. In this case, `Entity::Name()` returns a non-const `char*`, but in case of a `nullptr` in `pname`, a pointer to the literal `"none"` is returned, which is by definition `const`. It is also good style to use `const` for getters, so the user knows an object's state is not changed by calling it. There is no need to change the body of the method, but the signature is best changed to `char const *Name() const`.

Let us now look at the reason for the core dump. With `Widget w;` in `main()`, an object of class `Widget` is created and the compiler knows that it needs to call its destructor `~Widget` when it goes out of scope. In `Entity::Reset()` the widget destructor is called for this object, because we have a virtual destructor and invoke it through a pointer. However, only an `Entity` object is created in the `Widget`'s place. When `w` goes out of scope at the end of `main()`, the compiler does not use the virtual destructor, however, because (it thinks) it knows the precise type of `w`, since it was created on the stack. (Polymorphism in C++ works only with references and pointers, not with concrete objects.) Hence, `~Widget` is used on an object whose type is `Entity`, and in particular `Widget::data` is deleted twice, likely causing the core dump. (Incidentally, the code did not core dump on our machine, showing how insidious bugs due to undefined behaviour can be.)

There is a multitude of ways to fix this problem. With the code given it is obviously difficult to see why the author felt the need to re-use objects. If the main fear is that heap allocations and deallocations are slow, from C++17 onwards other memory allocators exist that can be used. Another possibility that eschews heap allocations is `virtual Reset()` and every class that introduces additional members overriding it with its own and calling their ancestor's `Reset()`. This can be error prone (both due to forgetting to implement `Reset()` for a child class requiring it and due to forgetting to call the immediate base class's `Reset()` and the author should make sure the effort is worth the payoff. Depending on the use case, there are other approaches that can be used, such as an Entity-Component-System (see https://en.wikipedia.org/wiki/Entity_component_system), that do not use inheritance and thus do not present the same problem.

There are a few further points on the code.

Both `Entity` and `Widget` have compiler-generated copy constructor and copy assignment, but when one makes a copy, a shallow copy is created, meaning that only the pointers are copied, but not the content pointed to (which would be a deep copy). Someone expecting to be able to make a copy of an object and then be able to change one without affecting the other will find that this works for `idNumber`, but not for `pName` or `data`. Additionally, the latter members' destructors would be called twice. Note also that copying can be dangerous in the case of a hierarchy, because assigning, e.g., a `Widget` object to an `Entity` object will lead to object slicing (see https://en.wikipedia.org/wiki/Object_slicing).

It is good style to avoid raw pointers to benefit from automatically working copy- and move-members. Hence, use `std::string` instead of `char*`

for **pName** and **std::array<int, 10>** for **data**. See *CppCoreGuideline* R.1 (Manage resources automatically using resource handles and RAII), R.5 (Prefer scoped objects; don't heap-allocate unnecessarily) and R.11 (Avoid calling new and delete explicitly).

It is a good idea for headers to include standard and other own headers they require; in the author's code neither does `Widget.h` include the base class header, nor does `Entity.h` include the string header it needs, although the main code file does not itself require it directly.

As a matter of self-documenting code, change the signature of the subclass destructor to **~Widget() override**.

Another minor problem is that the string literal **"none"** is not technically an invalid name for an **Entity**. With **std::string**, an empty string could indicate an unnamed **Entity**. Another method is to make the possibility of a missing value explicit by using **std::optional** (since C++17: **boost::optional** was available before).

James Holland <James.Holland@babcockinternational.com>

My first impressions are that the code seems to be a mixture of C and C++. Perhaps the student comes from a C background and has not yet learnt the C++ way of doing things. I shall say more about this later but for now let's persevere with the code presented.

When compiling the code on my system I get one error, namely that an object of type **const char *** cannot be used to initialise a return object of type **char *** in function **Name()**. This is because a string literal is considered **const** and allowing the conversion to non-const would enable the string literal to be modified which would not make sense; hence the error message. Perhaps the simplest way to overcome this problem is to make the return type of **Name()** into **const char ***. The code now compiles without error. There are still problems, however.

I notice the student is using a function named **strdup()**. After a little research I discover this function has recently been added to the C language. According to cppreference, the function duplicates the string pointed to by its parameter and returns a pointer that must be **free**d using the function **free()**. An inspection of the student's code reveals that **delete** is being used. This will result in undefined behaviour.

There is a similar problem with the destructor of **Widget**. **Widget**'s constructor creates an array of 10 **int**s on the heap using **new[]**. The destructor, therefore, needs to call **delete[]**, not **delete**. It should also be noted that the variable **idNumber** is not used in the sample code.

Running the revised code using Clang Address Sanitiser results in a message stating that data is being deleted twice. This would result in undefined behaviour. A quick and simple way of resolving this problem is to assign data the value **nullptr** after deleting **data**. The program now produces the required output. Unfortunately, the design of the program leaves a lot to be desired.

Some minor improvements could be made to the software. There is a superfluous set of braces placed on the end of the placement **new** statement. It is better to assign **pName** the value **nullptr** rather than **NULL**.

More seriously, the **Reset()** function creates a new version of **Entity** in the same memory location as the previous incarnation of **Entity** but not before calling **Entity**'s destructor. This will result in undefined behaviour as the new object resides in unallocated memory and may be overwritten at any moment by some other part of the system. Perhaps the direct call to the destructor of **Entity** could be removed but this results in memory leaks. To cure the memory leak problem **pName** needs to be freed. This may well result in code that works perfectly but things are getting very complicated and difficult to understand.

The student states that he or she is relatively new to C++ and this shows in the construction of the code. In many ways, C++ is quite a simple language once certain rules and idioms have been studied and understood. The best way to develop a C++ program is to take advantage of the facilities the programming language provides. Advantage can be taken of constructors, for example, to that objects can be created and initialised in a single statement. This will make functions such as **Setup()** unnecessary. Text strings should employ **std::string** instead of creating and using strings based on pointers to chars. The main feature of **std::string** is that they look after themselves. They will allocate storage on the heap, perform any initialisation and relinquish heap memory when they go out of scope. They have many more features that make them simple and safe to use. Try to use a higher level of abstraction.

The student appears to be attempting to reuse memory when a new **Widget** objects are required. In such cases a new object should be created. The old object should be left to be destroyed when it goes out of scope. There is rarely a need to destroy objects explicitly. It is important not to be too clever when writing software. The simplest approach is usually the best.

Finally, I provide a simple program that operates in a similar way to the student's but uses higher level features that are provided by the C++ standard.

```
#include <array>
#include <iostream>
#include <string>
class Entity
{
  public:
  Entity(const char * new_name)
  : entity_name(new_name){}
  std::string entity_name;
  int ID_number{0};
};
class Widget : public Entity
{
  public:
  Widget(const char * widget_name)
  : Entity(widget_name){}
  void reset()
  {
    entity_name.clear();
    ID_number = 0;
  }
  void setup(const char * new_name)
  {
    entity_name = new_name;
  }
  std::array<int, 10> data;
  std::string name() const
  {
    return entity_name.empty() ? "none" :
      entity_name;
  }
};
int main()
{
  Widget w("Test");
  std::cout << w.name() << '\n';
  w.reset();
  std::cout << w.name() << '\n';
  w.setup("Another widget");
  std::cout << w.name() << '\n';
}
```

Commentary

The fundamental issue with this code, as all the critiques explained, is the explicit invocation of the destructor in **this->~Entity();**. The comment for the **Reset()** method explains that the intent is to replace the current object with a blank one.

The trouble is that the static types of the object deleted and the new object created do not match. The destructor is **virtual** and so the call **this->~Entity();** actually invokes **Widget::~Widget()** and so destroys the **Widget** object. When the variable **w** goes out of scope at the end of

`main`, the `Widget` destructor is called but the actual type of the object being destroyed is `Entity` and so the behaviour is undefined.

One 'solution' would be to just destroy the `Entity` part of the object, but while this can be done (`this->Entity::~Entity();`) it doesn't help achieve the goal of the `Reset()` method. The question is whether `Reset()` is intended to be used to create *another* object or simply to re-initialise the *existing* one. At this point you'd probably need to talk to the original author to find out what their intended use cases are.

As James noted, this 'two-phase' construction via a constructor and a `Setup()` method not idiomatic C++. It would usually be better to provide the name at construction and create a complete object.

As Hans pointed out, you can argue that we appear to have a single class (`Entity`) with two different responsibilities – it is both a container for an optional object (hence the `Name` is `"none"` when the entity is not yet populated by a call to `Setup()`) *and* a base class for a class hierarchy

The winner of CC 119

All the critiques identified the root cause of the presenting problem – the core dump – and also gave an explanation of the compiler warning.

There were various suggestions for how to improve the design (as well as more detailed suggestions on places where the code was more C than C++)

Hand and James both pointed out the memory allocation/deallocation mismatches (`malloc` + `new[]` being incorrectly paired with `delete`) and every critique included suggestions to more use `std::string` and an array or `std::array` to remove the explicit memory management completely.

Several submissions also recommended using an address sanitizer or a memory checker as ways to help the person to solve their own problem. There are a wide variety of tools now available for this sort of assistance, and I suspect many newer programmers will need some guidance on how to pick the correct one for a given problem. Martin's recommendations under 'Learning the craft' would be a good place to start!

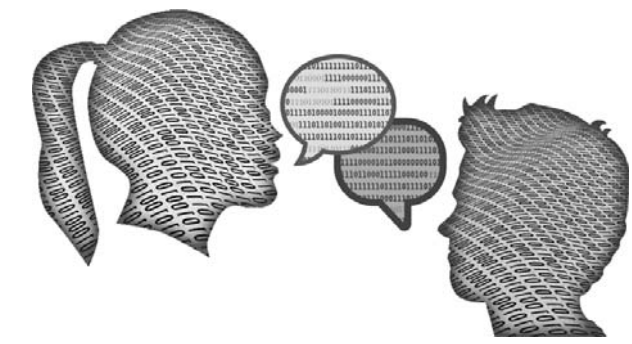
I found this a hard critique to judge since there were so many strong entries – thanks to all for your work in providing our readers with food for thought. However, in my opinion Martin provided the best entry for this critique by a short head.

Code Critique 120

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

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

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

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

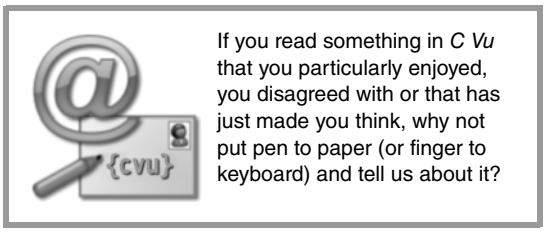


Listing 4

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

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

  string s = ss.str();

  smatch m;

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

Listing 4 contains the code. As always, there are a number of other things you might want to draw to the author's attention as well as the presenting problem.

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.



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

View from the Chair

Bob Schmidt
chair@accu.org

In Memoriam

Hubert Matthews
1961 – 2019

On Friday, 20 September 2019 Kevlin Henney posted the following on accu-general [1]:

> It is with great sadness that I pass on the news that Hubert Matthews passed away this week.
>
> Whether you knew him as a voice on this list or in person at conferences, on courses or socially, you will know him to have been someone of exceptional insight and enthusiasm, generous with his time, whether ready with an answer or ready to explore and find an answer.
>
> ACCU and, indeed, the world will be a poorer place without him. I hope we are able to kindle that spirit of generosity and enthusiasm; it is now ours to pass on.

In a subsequent post [2] Kevlin shared

> […] I have learned from a relative that it seems Hubert passed away in his sleep.
>
> This makes his loss no less shocking, but I am thankful that his passing was peaceful.

Hubert was found peacefully in bed at home. An inquest was opened but a cause of death could not be determined, and has been recorded as being by natural causes. His funeral service is scheduled for 14 October 2019 at the Oxford Crematorium in Headington.

Hubert served as Chair of ACCU for two years, from April 2010 to April 2012. He reluctantly took over as Chair during a period of instability in the organization; in his two years he reversed the trend of financial losses. Hubert was also responsible for negotiating our current relationship with Archer-Yates Associates, the company that runs our conferences.

Hubert was a familiar face at ACCU conferences and at local group meetings in Oxford and London, where he spoke on a wide range of C++ issues: *Optimising a Small Real-World C++ Application*; *The C++ Type System is Your Friend*; *C++ Concepts for Developers*; among others.

Hubert received a D. Phil. in Engineering Science from the University of Oxford in 1987. He served as Technical Director for several companies after finishing his studies. Hubert also was the Director at Oxyware, a small company he founded that provided consultancy and training services to software development companies. He was associated with NDC Conferences and ProgramUtvikling, where his biography [3] includes:

> Hubert lives in Oxford and in his abundant spare time he likes to pretend that he coaches rowing, dances salsa, dabbles with martial arts and drives too fast.

Unfortunately, I didn't know Hubert as well as some of you – separated as we were by large swathes of geography, my opportunities to interact with him were limited. I first met Hubert at my first ACCU conference, in Oxford in 2011. He was one of the first to welcome me to the organization. Since that first conference I have always looked forward to seeing him and catching up.

I attended his class at the April 2019 conference, and after the class I talked with Hubert and told him I was envious of his stage presence and ease at public speaking. Hubert told me that it was due in part to his being trained as an opera singer. In the encomiums on ACCU General, several members reported being regaled with samples of his singing.

Hubert was friendly and funny; kind and generous; talented and wise.

Hubert is survived by his mother, Joanna; brothers Charles and Roddy; nephews George and Olley; and nieces Heley and Hattie.

Rest in peace, Hubert. You will be missed.

The members of ACCU extend our deepest condolences to Hubert's family.

My thanks to Hubert's nephew, George Matthews, for providing details on Hubert's family.

My thanks also to Henriette Motzfeldt and NDC Conferences AS for permission to use the photo.

Links

[1] Kevlin Henney email and thread (requires login): https://lists.accu.org/mailman/private/accu-general/2019-September/054891.html

[2] Kevlin Henney email (requires login): https://lists.accu.org/mailman/private/accu-general/2019-September/054932.html

[3] ProgramUtvikling: https://programutvikling.no/en/instructor/hubert-matthews/

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

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

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

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

qbs SOFTWARE