the magazine of the accu

www.accu.org

{cvu}

Volume 27 • Issue 4 • September 2015 • £3

Features

The Very Model of a Model Modern Programmer
Pete Goodliffe

Anatomy of a CLI Program written in C++
Matthew Wilson

The Cat's Meow
Gail Ollis

Refactoring Guided by Duplo
Thaddaeus Frogley

Ode to the BBDB
Silas S. Brown

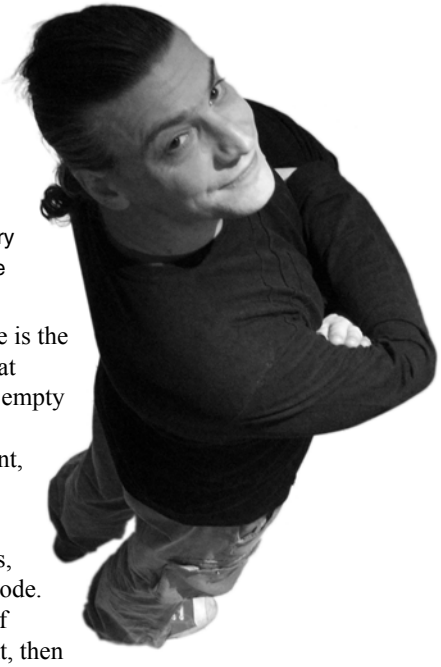
Raspberry Pi Linux User Mode GPIO in C++
Ralph McArdell

Regulars

C++ Standards Report
Code Critique
What Do People Do All Day?

{cvu}
Volume 27 Issue 4
September 2015
ISSN 1354-3164
www.accu.org

Developing programs



I started thinking about how the act of writing software became known as 'development'. It turns out that the etymology of the word *develop* comes from the old French *desveloper*, meaning to unwrap or reveal (etymonline.com is a good resource for this sort of thing). I like the connotations associated with 'reveal'. I've also heard people compare software development with the art of sculpture, with particular reference to the idea attributed to Michelangelo: "Every block of stone has a statue inside it and it is the task of the sculptor to discover it."

I'm fond of (sarcastically) remarking that writing code is the art of adding defects to an empty text file, but even that captures a similar concept of revelation. Of course an empty text file has no bugs in it – and it may even compile successfully! – but it's unlikely to satisfy a requirement, or pass a test.

It's this latter sentiment that really captures my imagination: that writing tests can reveal requirements, and these are then 'carved out' by the implementing code. This holds only if the code is being written test-first of course, and so if the real code is being written up front, then it too is being used as the tool to reveal the real requirements. The arguments used by the TDD community strikes a real chord here: driving the design with tests is more than just a way of having the tests in place; it also allows us to explore the design, and 'reveal' the underlying statue in the stone.

Perhaps in the end the TDD approach is more like the old-school development of photographs, which again is a process concerned with revealing the picture on the paper. The skill of the photographer in properly framing and exposing a picture might be likened to analysis of a given problem. The skill of the developer then is like that of the, er, developer in applying the right chemicals under very specific conditions to correctly reveal the photographer's original vision. Like the modern analyst and programmer, photographer and developer might very well be the same person. But I mustn't stretch this one too far, since these techniques are becoming out-dated by modern digital photography. Which leads finally to the question: will our skills as programmers be one day superseded in a similar way?

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

18 Standards Report
Jonathan Wakely reports the latest on C++17 and beyond.

20 Code Critique Competition
Competition 95 and the answer to 94.

21 Inspirational (P)articles: Use the DOM Inspector
Silas Brown shares a tip for debugging web pages.

22 What do people do all day?
Christopher Gilbert shares his routine in a software house.

REGULARS

24 ACCU Members Zone
Membership news.

FEATURES

3 The Very Model of a Model Modern Programmer
Pete Goodliffe asks what defines you as a programmer.

4 Refactoring Guided by Duplo
Thaddaeus Frogley gets to grips with duplicated code.

5 Ode to the BBDB
Silas S. Brown remembers different ways of managing email contacts.

6 Anatomy of a CLI Program written in C++
Matthew Wilson dissects a small program to examine its gory details.

11 The Cat's Meow
Gail Ollis reports from the App-a-thon World Record attempt.

13 WattOS R9 Worth Knowing About
Silas S. Brown recycles some old hardware with a new OS.

14 Raspberry Pi Linux User Mode GPIO in C++ (Part 2)
Ralph McArdell continues developing a C++ library for Raspberry Pi expansions.

SUBMISSION DATES

C Vu 27.5 1st October 2015
C Vu 27.6: 1st December 2015

Overload 130: 1st November 2015
Overload 131: 1st January 2016

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

The Very Model of a Model Modern Programmer

Pete Goodliffe asks what defines you as a programmer.

Programmers deal in the concrete, defining unambiguous definitions, interfaces, and recipes. Our stock and trade is to make the abstract explicit, to tease a functional specification out of ambiguous requirements, and to then implement it faithfully and transparently. Even when crafting abstract interfaces we're making that abstraction explicit by defining them.

Programmers define things. So then, how do programmers define *themselves*? There's got to be more to it than:

Programmer (n) /'prəʊgramə/ *one who writes programs*.

I was considering just this question when I read an article describing how your spending habits define who you are. You can look at how a person, or a family unit, spends money to see how they set their priorities, find fulfilment, the activities they enjoy and how they accomplish their day-to-day tasks.

Spend a thought

Someone who doesn't know you can look at your bank account, or credit card statement, and will very quickly build a picture of what motivates you, where your heart and interests lie.

With a little intuition it would become clear what quality of life you enjoy: whether you're comfortable and have resources to spare, or are struggling, only just getting by. They'd build a picture of your interests and hobbies, and learn what you value by the things purchased. Do you tend to buy for yourself, your family, or for others? Do you buy into certain styles, cliques, or feel the need to confirm to certain stereotypes? Are luxuries important to you? They'd clearly see how often you treat yourself, or whether you're frugal and don't splash out. They'd gain a picture of your patterns of giving, and whether this altruism is directed towards a close circle of people, or further afield.

All of these things reveal you as a person. In many ways, where you invest your money is how you define yourself. So, back to my question...

What defines *you* as a programmer? Why do you program? What is your primary motivator for doing it? Like looking at your spending habits, if someone looked at your coding habits, what picture would they build of you?

What motivates you to write code?

Is it money? You're working in a dull job, but it pays well.

Is it passion? You relish working on something you love, with money not a major motivator.

Is it the desire to do good work? To write the best software possible no matter what the problem area.

Is it the desire to give something back to humanity? To work on a project than benefits mankind using the skills you have.

Is it the desire to learn? You manoeuvre yourself into new, perhaps uncomfortable, situations to gain fresh experiences. Your work isn't necessarily perfect but you're constantly practising, and perfecting, skills that you'll find valuable later on.

Is it the desire to delight users? You enjoy the buzz of watching customers enjoy using your products.

Are you in the game to climb the corporate ladder? You're aiming for architect, manager, business owner, entrepreneur.

Is programming something you do now simply because it's the career you fell into? No better or worse than any other job, it's easier to carry on here rather than jump into a new career path.

Is your motivator something else entirely?

Or is it a combination of these things?

Looking from the other side

Can you say what drives you, what defines you as a programmer?

How does this affect the code you write and the way your work with others?

> The quality and style of your work will be determined directly by your motivation for working on the project.

It's not hard to see that many possible different code outcomes could result from programmers with different motivations.

For example, if you desire to delight users then you may care less about the internal structure of the code and eschew spending time cleaning and refactoring, instead striving to add delightful new features. You'd prioritise different activities than someone who cares about creating the neatest and most elegant codebase possible.

Look back at the list of potential programmer motivations above. How would each affect the way you might write code? Would they affect how you'd solve problems or prioritise your time?

Looking at it from the outside

Finally, consider this: could an outsider look at your code or product and work out what drives your work?

Does your code clearly reveal why you program? Does it show what you value in good code? Does it reflect your priorities?

Is it *really* possible for code to reveal this?

Which speaks louder about you: your money out your code?!

Conclusion

It's interesting to reflect on how the quality and style of your code will lead naturally from how you define yourself as a programmer, the things you value in good code, and from what motivates you to program.

It may be possible to tell a lot about you, as a programmer, by looking at the code you write. ■

Questions

1. What answers do you have to all the questions above?
2. Do you think that it is genuinely possible to tell the quality and properties of a programmer just by looking at their code?
3. How do different motivations affect the code you write, for good or ill?
4. What other things affect the qualities of the code you write?
5. Can you tell different programmers apart by looking at their code? How would someone recognise your work?
6. Do you need to review your motivation for writing code? Why? Is that even possible?

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

Refactoring Guided by Duplo

Thaddaeus Frogley gets to grips with duplicated code.

Reducing the amount of duplication within a code base can be a good proxy metric for improving the code base. Reducing duplication reduces total amount of code, in turn reducing executable size, and compile time, as well as making the code base easier to understand, and easier to modify. Smaller code bases have been shown to have less bugs than larger code bases. Defect counts go up in direct proportion to the number of lines of code.

Duplo is an open source implementation of the technique described in the paper 'A Language Independent Approach for Detecting Duplicated Code' [1]. It can be used to quickly identify code duplication, which can lead to refactoring opportunities that improve quality and reduce code size, resulting in an easier to maintain, and more efficient code base.

Getting Duplo

Duplo can be found on SourceForge:
http://duplo.sourceforge.net

And Daniel Lidstrom maintains a version on github:
https://github.com/dlidstrom/Duplo.git

For the purposes of this article, we will be using my fork:
https://github.com/codemonkey-uk/Duplo.git

To download & build (on a unix-like machine):

```
git clone https://github.com/codemonkey-uk/
Duplo.git
cd Duplo/
make
```

A project file for Microsoft Visual Studio is also included in the repository.

Generating a report

Duplo works from an explicit list of source files. For C++, on a unix-like system that could be generated like so:

```
find . | grep -e \.h$ -e \.cpp$ > filelist.txt
```

For C# you might do it like this:

```
find . -iname "*.cs" > filelist.txt
```

Or on a Windows based machine you could do it like so:

```
dir /s /b /a-d *.cpp *.h > files.lst
```

Unless you have a codebase measured in millions of lines, you probably want to start by analysing your whole codebase. The algorithm used by Duplo scales fairly well (see Table 1).

Table 1

Performance Measurements				
SYSTEM	FILES	LOCs	TIME	HARDWARE
3D Game Engine	275	12211	4sec	3.4GHZ P4
Quake2	266	102740	58sec	3.4GHZ P4
Computer Game	5639	754320	34min	3.4GHZ P4
Linux Kernel 2.6.11.10	17034	4184356	16h	3.4GHZ P4

THADDAEUS FROGLEY

Thaddaeus started programming on the ZX81 when he was 7 years old, and has been hooked ever since. He has been working in the games industry for over 20 years. On Twitter he is @codemonkey_uk or reach him by email: thad@bossalien.com

Once you have a list of source files, however you generate it, you can run Duplo from the command line. Duplo produces two sets of output. It writes a report to a file, containing all the duplicate blocks found. It also produces a summary list of files with a count of duplicate blocks in each. The duplicates report is written to a file, named via the command line. The summary is written to **stdout**. This can be seen by using the tool:

```
./duplo files.txt report.txt
```

Since files with no duplication are listed in the summary as 'nothing found', and files containing duplications are listed as having 'found *N* block(s), the summary can be easily filtered:

```
./duplo files.txt report.txt | grep "\\d*\\sblock"
```

And sorted:

```
./duplo files.txt report.txt | grep "\\d*\\sblock"
| sort -rnk3
```

But hold on! This sort falls over on file names with spaces. To get around that problem I introduced a colon following the 'found' in my fork of the project, so sorting the results of project that has spaces in its file names or paths still works:

```
./duplo files.txt report.txt | sort -t':' -rnk2
```

take care especially of any static or global state the code being refactored touches – either directly or indirectly

Examining the summary for files with a lot of duplication can provide some quick and easy wins. Files with high internal duplication can be less difficult to reason about refactoring, but be warned some algorithms, such as those found in lexer/parser code can contain high levels of local duplication that is actually very hard to remove. Feel free to remove files from the source list that produce false positives and re-run the tool to update the reports.

Another source of false positives (arguably) is the block of preprocessor include, or import directives found at the top of source files. Duplo supports filtering them out automatically with the **-ip** command line:

```
./duplo -ip files.txt report.txt
```

Duplo also supports reducing the amount of duplication reported by increasing the minimum number of lines of similarity (**-ml**), and minimum number of characters on a line (**-mc**) for a line to be counted. Both can be useful in helping target the worst offending sections of the codebase.

The default minimum lines of similarity is 4. This can be generate a lot of false positives on code bases where function arguments in both body and declaration are written out one per line.

```
./duplo -ml 8 files.txt report.txt
```

Refactoring

One of the simplest forms of duplication to eliminate is whole unit duplication: When there are multiple copies of the same method or type existing in the codebase under different names. These can usually be safely eliminated by simply removing all but one, and updating the references to the others in dependant code.

As with any code change, take care especially of any static or global state the code being refactored touches – either directly or indirectly. Having the codebase under test is good advice here, but should be considered as

Ode to the BBDB

Silas S. Brown remembers different ways of managing email contacts.

Chris Oldwood outlined one way of using mail folders in *CVu* 26.5 ('Taming the Inbox'). Generally I always archive rather than delete any email that might one day be needed again (however unlikely), and rely on search to find it on those rare occasions (if I ever find myself dragged before some kind of tribunal because somebody got upset about something I mistyped last year or whatever, then it might be useful to have my own record of the events in question), although I do sometimes delete things altogether especially if I believe they are already being archived elsewhere (list emails etc) and I don't usually keep large attachments especially if these are easily reproducible. Besides being useful on odd occasions, email archives can also be useful for training Bayesian spam filters such as SpamProbe, which generally need example collections of 'good' and 'bad' mail (I prefer to train my own filters instead of relying on an institution's, since the keywords in my email tend to be different especially now I have Chinese connections).

But I'd like to talk about something else (but related): the BBDB. BBDB stands for Big Brother DataBase, a reference to the 'Big Brother' persona adopted by the surveillance organisation in George Orwell's novel *Nineteen Eighty Four*. BBDB is a Lisp program that runs in Emacs and 'infiltrates' Emacs-based mail programs like VM, 'noticing' the names and addresses of people you correspond with and keeping dated records of these which can then be consulted when new messages arrive. You can add arbitrary notes to your contacts which will then pop up when their messages arrive, which might be useful for people with whom you don't correspond very often (as in, 'who WAS this person again?' infrequency); perhaps it's particularly useful for small businesses dealing with (potential) customers.

I no longer use BBDB itself, because I no longer use Emacs for email. This is firstly because I discovered that Mutt tends to be much faster at dealing with large mailboxes (especially if using maildir format), and Alpine tends to be better at minimising traffic when connecting over a slow mobile link (although it doesn't always handle mid-session disconnects very gracefully), and secondly because I wrote ImapFix [1], a script to do my email processing in-place using the university-provided IMAP server, which means I can connect to the fully-processed version from mobile devices without having to run my own server (previously my email processing happened only after the messages were fetched to my own machine, meaning I could not then access the post-processed mailboxes without logging in to that machine, which became impractical after I lost my always-on Internet connection; IMapFix on the other hand can run from just about any shell account without needing additional serving privileges).

Despite no longer using Emacs for email and therefore no longer using BBDB itself, I still support the concept of having personal folders that collect historical 'probably won't need this but keeping it just in case' notes and searching these when necessary, and for the sake of brevity I usually call these BBDB (after all B and D are right next to each other on my Dvorak keyboard). If I write myself a To-Do item that says 'bbdb the XYZ code', it means put the XYZ code into a BBDB folder and forget about it (but don't actually delete it, just in case). Of course if it's under version control then the version control system should take care of keeping track of the old version anyway and I can just go ahead and delete code. But it's not just code I'm talking about: it's also notes and other things. Having a 'BBDB' means I don't have to worry about totally losing something when I 'almost delete' it, but I can still clear it out of the way of my 'working set'. I feel that this, together with my 'postpone' system (a set of scripts that lets me dump a load of text into a file which then disappears from my view until a certain date I've set on it, at which point it pops up in a Web browser on my desktop, and I know I won't have to worry about it until then but can still find it via search if necessary), generally improves productivity, especially for a compulsive note-taker like me. I sometimes use 'dead time' while travelling etc to go through some old notes and figure out what needs BBDB'ing; this exercise usually also pulls up a few things I should have actioned ages ago but which somehow got lost in the great pile of notes. So yes, I recommend having a BBDB and from time to time going through your notes and other working-set with a view to deciding what to transfer to the BBDB. It might help productivity in the long run. ■

Reference

[1] ImapFix http://people.ds.cam.ac.uk/setup/imapfix.html

SILAS S. BROWN

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

Refactoring Guided by Duplo (continued)

an extra layer of security, not a replacement for actually thinking about the problem and fully understanding what the code does, and how it's used.

When duplication is found within the body of multiple methods of the same class: **`Extract Method`**, or **`Extract Class`** can be used. Where the duplicated functionality is pure (lacking in side effects, not modifying external state) this is trivial, but be careful of extracting methods that appear to do the same thing, are logically the same, but do have side effects, modifying different state.

When duplication like this is found across multiple classes, extracting the method to a common base class, or introducing a new class to the hierarchy, can be effective. But again, be wary of state modification.

Ultimately, however, each piece of duplication identified in a duplo report has to be considered on a case by case basis, and refactoring considered on their individual merits. Reducing duplication is a good rule of thumb, but is always a proxy for some other metric of improvement. Be it making the code easier to work with, or reducing the size of the resulting executable – always keep the end goal in mind: the creation of working software. ■

Reference

[1] http://www.iam.unibe.ch/~scg/Archive/Papers/Duca99bCodeDuplication.pdf

Anatomy of a CLI Program written in C++

Matthew Wilson dissects a small program to examine its gory details.

This article, the second in a series looking at software anatomy, examines the structure of a small C++ command-line interface (CLI) program in order to highlight what is boilerplate and what is application-specific logic. Based on that analysis, a physical and logical delineation of program contents will be suggested, which represent the basis of the design principles of a new library for assisting in the development of CLI programs in C and C++ to be discussed in detail in the next instalment.

In the first instalment of this series, 'Anatomy of a CLI Program written in C' [1], I considered in some depth the different aspects of structure and coupling of a simple but serious C program. The issues examined included: reading from/writing to input/output streams; failure handling; command-line arguments parsing (including standard flags `--help` and `--version` and application-specific flags); cross-platform compatibility.

The larger issues comprised:

- application of the EXECUTEAROUNDMETHOD pattern [2] to simplify and make more robust the initialisation of dependency libraries;
- specification of process identity according to DRY SPOT principles [3, 4];
- application of the principle of separation of concerns in the identification and classification of programmer-written CLI code into *decision logic*, *action logic*, and *support logic*. It is the simplification of the first two in, and the elimination of the third from, the task of the programmer that is the aim of this series; and
- decoupling of the action logic from the rest of the application code to facilitate the design philosophy of '***program design is library design***'.

In this second instalment I will consider further these issues, in the context of a small but serious C++ program, with the aim of defining a general CLI application structure that can be applied for all sizes of CLI programs.

Strictly speaking, some of the differences in sophistication and scope between the first instalment and this do not directly reflect the differences between the language C and C++. Rather, they reflect the different levels of complexity that it's worth considering when deciding in which language to implement a CLI application. I'll come back to this, and point out some rather important differences, in the third instalment.

DADS separation

Before we start working on the example program, I want to revisit the classification issue. In the first instalment I argued that CLI program code written (or wizard-generated) by the programmer is one of:

- **Decision logic** – the code that works out what needs to be done and which component(s) will do it;
- **Action logic** – the code that does the work deemed necessary by the decision-logic; and
- **Support logic** – all the other stuff, including command-line parsing, diagnostic logging, and so forth.

MATTHEW WILSON

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

To this list I now add a fourth:

- **Declarative logic** – declarations that influence the nature and behaviour of the program, including specifying its identity and the commands to which it responds.

In the examples of both instalments, the clearest example of declarative logic is the 'aliases' array that defines what command-line flags and options are understood by the program.

Example program: pown

To avoid destroying too many trees in the production of this month's issue, I'm going to try and keep the code listings as short as possible by focusing on a small program, albeit one with most real-world concerns; in the printed magazine, these are truncated dramatically, but they'll be available in full online [5]. For the purposes of pedagogy, I ask you to imagine that we need to write a program to show the owner of one or more files on Windows; in reality this is a feature (/Q) of the built-in dir command.

The features/behaviours of such a program include:

1. parse the command-line, either for the standard `--help` or `--version` flags, or for the path(s) of the file(s) whose owner(s) should be listed;
2. properly handle `--` special flag. It's very easy to simulate the problem with naïve command-line argument handling: just create a file called `--help` or `--version` (or the name of any other flags/options), and then run the program in that directory with `*` (or `*.*` on Windows);
3. expand wildcards on Windows, since its shell does not provide wildcard-expansion before program invocation;
4. for each value specified on the command-line, attempt to determine owner and write to standard output stream; if none specified, fail and prompt user;
5. provide contingent reports on all failures, including program identity as prefix (according to UNIX de facto standard);

Non-functional behaviour includes:

- use diagnostic logging;
- initialise diagnostic logging library before all other sub-systems (other than language runtime);
- initialise command-line parsing library before all other sub-systems (except diagnostic logging library and language runtime);
- include program identity and version information and include as required in output;
- do not violate DRY SPOT in program identity and version information.

pown.monolith

The first version of this program is done all in one file. Even such a simple program is remarkably large – over 230 non-blank lines – and a big part of its size is boilerplate. The program source has the following sections (parts of which are shown in much truncated form in Listing 1; the full version of this and all are available online):

- **includes** (18 lines): all required includes, including those required purely for boilerplate aspects, are present in their imposing glory;

Listing 1

```
// includes
#include <pantheios/pan.hpp>
#include <systemtools/clasp/main.hpp>
// . . . + Pantheios.Extras, STLSoft, etc.
#include <systemtools/clasp/implicit_link.h>
// . . . + 4 more impl-link
#include "MBldHdr.h"

// aliases
static clasp::alias_t const Aliases[] =
{
  . . . initialisers same as in Listing 4

// identity
#define TOOL_NAME  "pown"
int const toolVerMajor = __SYV_MAJOR;
// . . . + Minor, Revision, BuildNum
char const* const toolToolName = TOOL_NAME;
// . . . + summary, copyright, description
char const* const toolUsage = "USAGE: " TOOL_NAME
" { --help | --version | <path-1> [ ... <path-N>
] }";
extern "C" char const
PANTHEIOS_FE_PROCESS_IDENTITY[] = "pown";

// pown
int pown(char const* path)
{
  . . . 87 lines, retrieve owner (domain
  . . . & acct), result->stdout; diagnostic
  . . . logging to trace flow & log failures.

// main/program entry
int program_main(clasp::arguments_t const* args)
{
  // process flags and options
  if(clasp::flag_specified(args, "--help")){
    . . . 17 lines to initialise CLASP
    . . . usage structure, invoke usage
    . . . and return EXIT_SUCCESS
  if(clasp::flag_specified(args, "--version")){
    . . . 9 lines similar to "--help"
  clasp::verify_all_flags_and_options_used(args);

  // process values
  . . . rest of main() same as in Listing 4
}

// main/boilerplate
. . . 3 x ExecuteAround (see text)
```

- **aliases** (7 lines): as discussed in the previous instalment, the command-line parsing is handled by the CLASP library [6], and it uses a global alias array constant to specify declaratively which flags and options the program recognises. In the first case, this is **`--help`** and **`--version`**;
- **identity** (11 lines): this section includes pre-processor (**`TOOL_NAME`**) and C/C++ global constants (incl. **`toolVerMajor`**, **`toolVerMajor`**, …, **`toolToolName`**, …) specifying identity and version as used by the handlers of the **`--help`** and **`--version`** flags. It also includes a constant required by some of the simple stock front-ends provided with the **Pantheios** [7] diagnostic logging API library: **`PANTHEIOS_FE_PROCESS_IDENTITY`**;
- **pown** (94 lines): the file owner elicitation & printing logic, in the form of the **`pown()`** function;
- **main/program entry** (56 lines): this is the application-specific program main entry point, **`program_main()`**, including checks for the **`--help`** and **`--version`** flags and, if not, processing the given values or, if none specified, informing the user of his/her oversight;
- **main/boilerplate** (51 lines): truly the most boring of the lot, this is just the application of the EXECUTEAROUNDMETHOD pattern [2] (actually, it should be EXECUTEAROUND*FUNCTION*, since these are all free-functions) as follows:
  - **`main()`** executes **Pantheios.Extras.Main**'s **`invoke()`**, to initialise the **Pantheios** diagnostic logging library (and provide last-gasp outer-scope exception catching with contingent reporting and diagnostic logging [8]) around **`main_memory_leak_trace_()`**,
  - which executes **Pantheios.Extras.DiagUtil**'s **`invoke()`**, to trace memory leaks, around **`main_cmdline_()`**,
  - which executes **CLASP.Main**'s **`invoke()`** around **`program_main()`** to initialise the **CLASP** command-line parsing library.

It's pretty clear that boilerplate is eating space, not to mention effort. Furthermore, structuring source in such a manner is an imposition on programmer visibility (and, I would suggest, happiness).

Note the inclusion of the `MBldHdr.h` header and use of the symbols **`__SYV_MAJOR`**, **`__SYV_MINOR`**, etc. (whose names violate the standard's reservation of symbols, as they contain runs of two or more underscores). These are aspects of an *extremely* old, but still used, mechanism for controlling module version by an external tool, and I include them only to show how such schemes can be used with the proposed anatomical delineation discussed herein.

Separation of concerns – pown.alone

The first obvious thing is to partition the file. This can be done, at least in part, by identifying what parts confirm to the DADS classification. Let's tackle all the identified sections (apart from includes, which is a necessary evil of C and C++ programming):

- The aliases section is a *declarative*, and it is entirely about the behaviour of the (command-line) program; it has nothing (direct) to do with the owner-printing logic of **`pown()`**.
- The identity section is a *declarative*, and it is entirely about the identity of the (command-line) program; it has nothing to do with the owner-printing logic of **`pown()`**.
- The **pown** section is *action logic*, and is the entirety of the *program-as-library* part of the application, in the form of the function **`pown()`**.
- The **main/program entry** section is a mixture of *decision logic*, in the form of the tests of the presence of the flags and the presence of (one or more) values, and *action logic*, in the form of the loop over all present values and execution of **`pown()`** with each.
- The **main/boilerplate** section is *support logic*, pure and simple.

Given these designations, the parts may now be separated physically according to the scheme I have been evolving over the last few years, as follows:

- The files `pown.hpp` and `pown.cpp` contain, respectively, the declaration and definition of the **`pown()`** function. `pown.hpp` is a self-contained header file [9].
- The file `entry.cpp` contains the **aliases**, **main/program entry**, and **main/boilerplate** sections, and (only) their requisite includes.
- The files `identity.hpp` and `identity.cpp` contain, respectively, the declarations and definitions of the global constants identifying the program (and the anachronistic MBldHdr.h + symbols).
- The file `diagnostics.cpp` contains the definition of the global constant **`PANTHEIOS_FE_PROCESS_IDENTITY`** only; in more complex programs / program suites, additional diagnostic constructs would reside within such a file. In this way, the actual kinds of diagnostic logging (and other facilities) are separate from all code,

allowing for link-time decisions as to what kinds of facilities, and in what configurations, are employed. (Note that Pantheios is a diagnostic logging *API* library: its high-performance and 100% type-safe interface is designed and intended to be bolted atop the much richer logging libraries out there, which bolting in this compilation unit would be kept nicely separate from the rest of the program.)

- In the file `implicit_link.cpp` all the implicit-link includes are made. This keeps this useful but non-portable compiler-specific facility separate to every other part of the program.

Salient fragments of all the above are presented in Listing 2. Note that, for now:

Listing 2

```
// pown.hpp:
extern "C"
int pown(char const* path);

// pown.cpp:
. . .
#include "pown.hpp"
#include "identity.hpp"
. . .
int pown(char const* path)
{
  . . .

// entry.cpp:
. . .
#include "pown.hpp"
#include "identity.hpp"
. . .
static
clasp::alias_t const Aliases[] =
{
  . . .
int program_main(clasp::arguments_t const* args)
{
  . . .
. . . // other "main"s, including main()

// identity.hpp:
#define TOOL_NAME     "pown"

extern int const toolVerMajor;
extern int const toolVerMinor;
. . .

// identity.cpp:
#include "identity.hpp"
#include "MBldHdr.h"

int const toolVerMajor = __SYV_MAJOR;
int const toolVerMinor = __SYV_MINOR;
. . .
char const* const toolToolName = TOOL_NAME;
char const* const toolSummary  = "Example project
for Anatomies article series in CVu";
. . .

// diagnostics.cpp:
#include "identity.hpp"

extern "C" char const
PANTHEIOS_FE_PROCESS_IDENTITY[] = TOOL_NAME;

// implicit_link.cpp:
#include <systemtools/clasp/implicit_link.h>
// . . . + 4 more
```

- `pown.hpp` is included in `entry.cpp`, and `pown.cpp`; and
- `identity.hpp` is included in `diagnostics.cpp`, `entry.cpp`, `identity.cpp`, and `pown.cpp`.

'Program Design is Library Design' – pown.alone.decoupled

In the first instalment, I mentioned the importance I attach to being able, as much as is reasonable, to subject the guts of CLI programs to automated testing. As such, separating out the action logic into pown.[ch]pp is an important step. However, there's still a problem. Consider the current definition of **`pown()`** (which is that from Listing 1 transplanted into its own source file, with requisite includes): it has three areas of undue coupling:

- it writes its output to **`stdout`**;
- it issues its contingent reports to **`stderr`**;
- it **`#includes identity.hpp`** because it used the (preprocessor) symbol **`TOOL_NAME`** in its contingent reporting and diagnostic logging statements.

You may offer a fourth area of undue coupling – use of Pantheios C++ API diagnostic statements. The rejoinder I would offer to that is an article in itself, so in this context I will simply observe that diagnostic logging is important, it must reliably be available at all points during the lifetime of a program, it must be very efficient when enabled and have negligible runtime cost when not, it should be near impossible to write defective code using it, any non-standard (and they are all non-standard) diagnostic logging API will incur some coupling however far one might wish to abstract it, and that there is no (possibility of a) perfect solution. (Though I couldn't be more biased) I believe that **Pantheios** offers the best mix of features and, since it may be stubbed readily at both compile and link-time, I think it's as less bad as coupling can get.

To our three areas of undue coupling. The first two are basically the same thing: the output streams are hard-coded into the function, which restricts potential uses of the function. Even if we would always want those output streams in the wild, hard-coding makes automated testing more difficult. The answer is simple – to pass in the stream pointers as parameters to **`pown()`** – though the rationale may be less clear cut (see sidebar).

That just leaves coupling to identity. Fairly obviously, coupling to any preprocessor symbol is not a great idea. (The main reason why **`TOOL_NAME`** is even a preprocessor symbol is to facilitate widestring builds, which I'm not dealing with in this instalment; the other, minor one, is that it can be used in composing string fragments, as seen in the definition of the literal string initialiser for **`toolUsage`** in Listing 1.) The fix here is just as simple as with the streams: a parameter to the function, as shown in Listing 3.

Finally, though it's not shown in this example, I believe it's appropriate to place the action logic library components in a namespace, since it's conceivable that the names may clash with those in an automated framework (less likely) or with those of other components when used in other program contexts (more likely). I'll illustrate this clearly in the next instalment.

Summary

In these two articles I have considered some of the fundamental – important, but not very exiting – aspects of program structure in C and C++ CLI programs, and have outlined in this instalment a delineation scheme

Listing 3

```
#include <stdio.h>
int
pown(
  char const* path
, char const* program_name
, FILE*       stm_output
, FILE*       stm_cr
);
```

that is now sufficient for all CLI programs, even large ones for which multiple (implementation and/or header) files for action logic are required, and may be encapsulated into a framework and/or generated by a wizard. Program generating wizards can follow the separation defined previously, and can, in the same operation, generate automated test client programs that include the action logic header and implementation files.

There's nothing inherent in the scheme that requires use of CLASP for command-line parsing and Pantheios for diagnostic logging (and **Pantheios.Extras.Main** and **Pantheios.Extras.DiagUtil** for handling initialisation, outer-scope exception-handling, and memory-leak tracing); you may substitute your own preferences to suit, and a well-written wizard would be able to allow you to select whatever base libraries you require.

In the next instalment I will introduce a new library, **libCLImate**, which is a flexible mini-framework for assisting with the boilerplate of any command-line programs and which may be used alone or in concert with program suite-specific libraries to almost completely eliminate all the boring parts of CLI programming in C or C++. Listing 4 is a version of the exemplar pown project's `entry.cpp` using **libCLImate** alone; Listing 5 is the `entry.cpp` for the **pown** program in Synesis' system tools program suite: as you can see, almost every line pertains to the specific program, rather than any common boilerplate. (Having written this tool as

Listing 4

```
// includes
#include "pown.hpp"
#include "identity.hpp"
#include <libclimate/libclimate/main.hpp>
// . . . + 3 more

// aliases
extern "C"
clasp::alias_t const CLImate_Aliases[] =
{
  CLASP_FLAG(NULL, "--help",
    "shows this help and terminates"),
  CLASP_FLAG(NULL, "--version",
    "shows version information and terminates"),
  CLASP_ALIAS_ARRAY_TERMINATOR
};

// main / program entry
extern "C++"
int CLImate_program_main(clasp::arguments_t
  const* args)
{
  namespace sscli =
    ::SynesisSoftware::CommandLineInterface;
  if(clasp::flag_specified(args, "--help")) {
    return sscli::show_usage(args,
      CLImate_Aliases, stdout, toolVerMajor,
        //... + 9 params
  }
  if(clasp::flag_specified(args, "--version")) {
    return sscli::show_version(args,
      CLImate_Aliases, stdout, //. . . + 5 params
  }
  clasp::verify_all_flags_and_options_used(args);
  // process values
  if(0 == args->numValues)
  {
    fprintf(stderr
    , "%s: no paths specified; use --help for
      usage\n"
    , TOOL_NAME
    );
    return EXIT_FAILURE;
  }
  for(size_t i = 0; i != args->numValues; ++i) {
    pown(args->values[i].value.ptr, TOOL_NAME,
      stdout, stderr);
  }
  return EXIT_SUCCESS;
}
```

Listing 5

```
// includes
#include "pown.hpp"
#include "identity.h"
#include <SynesisSoftware/SystemTools/
program_identity_globals.h>
#include <SynesisSoftware/SystemTools/
standard_argument_helpers.h>
#include <stlsoft/util/bits/count_functions.h>

using namespace
  ::SynesisSoftware::SystemTools::tools::pown;

// aliases
extern "C"
clasp::alias_t const CLImate_Aliases[] =
{
  // stock
  SS_SYSTOOLS_STD_FLAG_help(),
  SS_SYSTOOLS_STD_FLAG_version(),
  // logic (HELP => elided help-string)
  CLASP_BIT_FLAG("-a", "--show-account",
    POWN_F_SHOW_ACCOUNT, . . . HELP),
  CLASP_BIT_FLAG("-d", "--show-domain",
    POWN_F_SHOW_DOMAIN, . . . HELP),
  CLASP_BIT_FLAG("-r", "--show-file-rel-path",
    POWN_F_SHOW_FILE_REL_PATH, . . . HELP),
  CLASP_BIT_FLAG("-s", "--show-file-stem",
   POWN_F_SHOW_FILE_STEM, ), . . . HELP),
  CLASP_BIT_FLAG("-p", "--show-file-path",
   POWN_F_SHOW_FILE_PATH, . . . HELP),
  CLASP_ALIAS_ARRAY_TERMINATOR
};

// main
int tool_main_inner(clasp::arguments_t
  const* args)
{
  // process flags & options
  int flags = 0;
  clasp_checkAllFlags(args, SSCLI_aliases,
    &flags);
  clasp::verify_all_options_used(args);
  // can specify at most one file-path flag
  if(stlsoft::count_bits(flags &
    POWN_F_SHOW_FILE_MASK_) > 1) {
    fprintf(stderr
    , "%s: cannot specify more than one file-path
      flag; use --help for usage\n"
    , systemtoolToolName
    );
    return EXIT_FAILURE;
  }
  // process values
  switch(args->numValues)
  {
    case 0:
      fprintf(stderr
      , "%s: no paths specified; use --help for
        usage\n"
      , systemtoolToolName
      );
      return EXIT_FAILURE;
    case 1:
      break;
```

Listing 5 (cont'd)

```
    default:
      if(0 == (POWN_F_SHOW_FILE_MASK_ & flags)) {
        flags |= POWN_F_SHOW_FILE_REL_PATH;
      }
      break;
  }
  for(size_t i = 0; i != args->numValues; ++i) {
    char const* const path =
      args->values[i].value.ptr;
    pown(path, flags, systemtoolToolName, stdout,
      stderr);
  }
  return EXIT_SUCCESS;
}
```

n exemplar for this article I realised a few enhancements – adding some behaviour options, splitting into functions eliciting ownership (as strings), and output to streams – would make **pown()**'s functionality a useful library in several tools, including a new, more powerful standalone **pown**.)

In the meantime, I plan to release wizards that generate CLI programs, starting with Visual Studio (2010–15), and possibly moving on to Xcode if I get time. Look out on the Synesis Software website in September for these resources, and feel free to make requests or lend a hand. ■

Acknowledgements

Many thanks to the members of accu-general who volunteered suggestions for the name of **libCLImate**, and to Jonathan Wakeley in particular, whose ghastly pun I will explain next time. Thanks too to the long-suffering editor whose patience with my lateness is never taken for granted.

a

Use of FILE* vs ...

Throughout the refactoring and repackaging work undertaken to a swathe of CLI programs, one issue stands out, and it's the one remaining substantive issue of debate/equivocation:

A: Should the program logic (as library) issue *contingent reports*? [8]

There are three corollary issues if so:

1. How should the program logic be provided the process identity?
2. To where should contingent reports be written?
3. In what form should contingent reports be written?

If the answer to **A** is 'no', then the library has to return to its caller sufficient information as is required to provide a suitable contingent report. If we consider the **pown()** function, there are four substantive actions (for a Windows implementation): get the file's security descriptor; get the security descriptor's owner SID; lookup the owner SID's information (specifically account-name and domain-name); and, finally, print out the results in the desired format. Any of these four operations can fail, raising the question of how the caller might wish to represent such failures, and with what detail. Given that we would rarely (if ever) be satisfied with a mere Boolean success/fail in such circumstances, is it useful, to an end-user (of the command-line tool), to know why and what/where the failure occurred, as well as simply that it did? Is this usefulness greater or less when such code is being used (in the nature of one library amongst many) in a larger program?

Assuming we want to know what/where, in addition to why, the means of communicating this to a caller has to be considered (briefly): Is it a more complex return code (perhaps a composite of why and where)? Is it an exception? All such approaches are fraught with leaked coupling and/or loss of information. I'm not going to explore further this aspect, because this is trespassing into the territory I was last exploring in Quality Matters some time ago (and intend to get back to very soon), and because in this context I'm interest in the affirmative option.

One further thing worth considering in the 'no' alternative is how to handling warnings, by which I mean contingent reports provided to a/ the user but not associated with failing conditions. One example might be in the case of a program's action logic library that acts on multiple targets (e.g. by specifying a directory and search patterns), and is unable to act on one (or several) matches while still being able to perform its work on the rest of the targets. In such a case, one might expect the program to continue to determine and output (to the standard output stream) the owners for other files, while emitting (to the standard error stream), but if the called library function does not have the ability to issue contingent reports, how is this to be expressed? Perhaps by populating a caller-supplied failed-target list? Whatever the case, such behaviour cannot be handled either by a return code or an exception: the former can't provide enough information; the latter will cause the callee to terminate with its work part-done. In many cases, therefore, I feel that allowing action logic to issue contingent reports can be the pragmatic choice, however much it may sniff of coupling (my personal least liked thing in programming, fwiw).

So, in the cases where the answer to **A** is 'yes', then we must consider the corollary questions outlined above. Again, coupling comes into it. The answer to question 1 is simple and, I think, uncontentious: simply pass in the process identity as a parameter (or as part of a chunk of information passed in a single parameter).

Question 2, concerning where contingent reports should be written, is a little more tricky. In the wild, the action logic is running inside CLI programs, which interact with their three streams – input, output, error –whether they be the streams of the console/terminal, or, via piping/ redirection, files and the inputs/outputs of other programs. Whatever the case, the program (and its action logic therein) works with what it believes to be its three standard streams, and I believe it is a valid choice to

What remains to decide is *Which?*, and *In what form?* Both decisions are informed by the '***program design is library design***' philosophy. The answer to *Which?* is simple: I believe that in order to support testability, the caller should supply separate output-stream (in the case where there is any output by the action logic) and contingent report-stream, as shown in Listing 3. Similarly, if there is a warning stream, that too should be separately specified. The answer to *In what form?* is a bit more involved.

In C, the three standard streams are represented and used most commonly in the form of the C Streams library globals **stdin**, **stdout**, and **stderr**, each of which is of type **FILE***. In C++, the received wisdom is that we should use the C++'s IOStreams library global instances **std::cin**, **std::cout**, and **std::cerr**, each of which is of type **std::ostream**, and should be used as **std::ostream&**.

Having documented previously [10, 11, 12] my legion reservations about the IOStreams, I won't bother to repeat its many flaws here. The main reason I choose to use **FILE*** forms of the streams in these circumstances is, again, in support of '***program design is library design***': I believe it's clear to use **NULL** (or **nullptr**, if you prefer) to specify no-stream as a caller, and equally easy to test against **NULL** in the callee. Secondarily, it results in lower coupling, and allows the callee to be implemented in C (or another language providing a C API) without changing the caller.

Finally, to question 3. This is really a horses-for-courses issue, for which I have no broad answer at this time. In the case of CLI programs that (follow established UNIX behaviour to) issue a contingent report in the fashion of ***<process-identity>: <problem-details>***, it is a simple matter, given that we've already accepted the caller-supply of process identity. In the next instalment I'll look at a more sophisticated means of handling all this, in the form of a program suite-specific contingent reporting mechanism, which simplifies and improves the programming of each program's action logic at the cost of coupling to the mechanism's constructs.

The Cat's Meow

Gail Ollis reports from the App-a-thon World Record attempt.

An attempt to set a new world record for the largest number of people learning to write an Android application at the same time.

"Buzzing! I have had an absolute blast today helping adults & children alike write their first apps." This was the Facebook status I posted one Saturday evening in June after having as much fun as I've ever had helping people to program.

The event

That day I was one of the team at Bournemouth University helping out with the BCS App-a-thon Guinness World Records Challenge [1], an event organised nationally by BCSWomen [2] with the voluntary support of organisations around the country. At 10:30 on Saturday, 13th June, I blew the timekeeper's whistle in a computer laboratory in Bournemouth. Something similar happened at 29 other locations (though I suspect mine was the only duck-design whistle) as 1093 interested adults and children in England, Scotland and Wales all started a world record attempt.

We had one hour to teach them all to write an Android app. This followed a common pattern across all the venues: people learning how to build a simple program that would play a meow sound when tapping on the image of a cat, and then extending it to 'purr' by vibrating. Because not all tablets can vibrate, smartphone owners tended to have more contented cats.

By the end of the hour, everyone had a working cat program on their device. Listen to the cat chorus from Aberystwyth University [3] and you'll have a good idea of how our lab sounded. Many people had moved on to adding extra animals, leading to some interesting bugs such as chickens responding with owl hoots. When time was called at 11:30 (the duck whistle again) the record attempt was complete, but the learners were clearly still having fun. Most took up the invitation to stay on and play some more; we provided printouts of some tutorials [6] to help give them ideas. Until mid afternoon, when we sent the last of them home, the lab continued in the same busy, productive and cheerful buzz as people tried out whichever idea appealed to them.

The tech

There were temporary accounts ready for all 41 of the Bournemouth learners on the lab computers and wifi network. This worked smoothly apart from the one phone that had evidently been deprived of a wifi connection for so long that for quite some time it was too busy getting its fix of updates to respond to anything else. Eventually it got its fill and was ready to download App Inventor Companion, an app which provides the easiest way to test the programs its owner would be writing.

AI Companion allows testing in real time and on the actual Android device, provided that the device is on the same network as the computer where the program is written. This is the recommended method, but does depend on every programmer having their own device. If they don't, another option is to use the emulator, but in preparing for the event we had not found this very easy or reliable. In any case it's just not the same as seeing, touching and feeling the fruits of your programming on an actual tablet or smartphone. The third testing option is to download to the device with a USB cable, which we didn't need because our network arrangements allowed us to stick with the preferred wifi method. Apart from a couple of minor hiccups, which went away upon restarting the AI Companion app, this worked very well.

The programs were written using a web-based tool, MIT App Inventor 2 [4]. For me, a programmer in text since 1982, trying to program with visual blocks was by far the hardest part of the whole thing. This may explain why my own very first app, a speech-to-text program written while preparing for the Appathon, said something that was Not Suitable For Work. Once I got used to filling in the blanks in the drag-and-drop blocks and clicking the blocks together it was easy to use but I'm not sure I will ever adjust to the fact that chunks of program logic lay scattered across a page. Even now I've found out how to collapse blocks I'd still be happier to have my code marshalled into neatly-labelled files; I've yet to find any way to impose meaningful order.

Several of the children had used Scratch [5] at school and found App Inventor similar and pretty easy to use. The adults got on fine with it too.

GAIL OLLIS

After years of professional programming, Gail is now a postgraduate student at Bournemouth University, researching ways to help software developers make life easier for each other. Contact her at gollis@bournemouth.ac.uk or @GailOllis

Anatomy of a CLI Program written in C++ (continued)

References

[1] 'Anatomy of a CLI Program written in C', Matthew Wilson, *CVu* September 2012.
[2] http://c2.com/cgi/wiki?ExecuteAroundMethod
[3] *The Pragmatic Programmer*, Dave Thomas and Andy Hunt, Addison-Wesley, 2000
[4] *Art of UNIX Programming*, Eric Raymond, Addison-Wesley, 2003
[5] http://synesis.com.au/publishing/anatomies
[6] An Introduction to CLASP, part 1: C, Matthew Wilson, *CVu* January 2012; also http://sourceforge.net/projects/systemtools
[7] http://pantheios.org/
[8] Quality Matters #6: Exceptions for Practically-Unrecoverable Conditions, Matthew Wilson, *Overload* 98, August 2010
[9] *C++ Coding Standards*, Herb Sutter and Andrei Alexandrescu, Addison-Wesley, 2004
[10] An Introduction to FastFormat, part 1: The State of the Art, Matthew Wilson, *Overload* #89, February 2009
[11] An Introduction to FastFormat, part 2: Custom Argument and Sink Types, Matthew Wilson, *Overload* #90, April 2009
[12] An Introduction to FastFormat, part 3: Solving Real Problems, Quickly, Matthew Wilson, *Overload* #91, June 2009

The most subtle problem was with blocks that looked connected but hadn't properly clicked together. There's an audible 'click', but it's not always audible in a busy lab and the visual cue is rather subtle too, so for some situations the tool could benefit from giving more noticeable feedback as components link up.

When they had created a program in App Inventor, learners were able to connect it to the AI Companion app installed on their phone or tablet just by scanning a QR code. Thereafter AI Companion (mostly) stayed in sync with any changes so that they could seamlessly test on their Android device as they went along. Once they had something they wanted to install permanently or share with others we showed them how to build it in App Inventor. With a couple of simple clicks the program is ready to install via another QR code or to save as a file. The disappointment that it would cost money to put an app in the Google Play store instantly evaporated when they learned that there were other ways to install and share it. The programming projects in App Inventor remain available for them to experiment with another day, linked to their Google account.

The outcome

The tutorials [6] helped people to learn to control the effect of a whole range of familiar actions: flicking a pirate ship towards gold coins to collect them; shaking a magic 8-ball to make it respond to questions with a random pick from answers you wrote yourself; hitting the creatures that pop up in a classic whack-a-mole style game where you decided to make it whack-a-wabbit instead. The joy of making things happen was evident even when people followed a tutorial verbatim, but still more so when they gained the confidence to explore, tweak and customise and unleashed their creativity with a twinkle in their eye.

Our learners in Bournemouth – children, adults and whole families – all got stuck in. Adults at other locations included a white haired grandma in Huddersfield, who was thrilled to get an owl to hoot on her phone. Children in our lab and no doubt elsewhere were showing and sharing with others with friendly ease. It was a memorable atmosphere that combined focus and excitement into a very productive buzz. The moments that moved me most, though, were the ones where parents of the current generation of schoolchildren discovered that they could do it too – the wonder in the voice of the mother who announced "I've just written my first program, in my forties!"

The point

So did we manage to set a record? We had the numbers – we needed over a thousand – so it's possible. At the time of writing Guinness World Records are in the process of checking the evidence: statements from all the independent witnesses and stewards about what they saw, and photos and videos taken on the day.

But even if we don't get the record, we achieved something more valuable. The sessions were led by women, with a team of women and men helping out. All those children and all those parents had the chance to see that women can and do 'do IT', and to discover that it can be fun, and that they could do it too.

That was the goal, and the inclusive approach appeals to me. According to the App-a-thon press release women represent just 16% of IT professionals. I have no figure for the percentage I'd like it to be, simply be the number who would choose to do it if every child were given sufficient opportunity to find out if it appeals to them and to consider it as a realistic career option. It would be rash to assume that now having computing on the school curriculum can achieve this; being a mainstream subject rather than just a club activity may help, but there are subjects of much longer standing that are still perceived as "boys' subjects" and "girls' subjects".

I can't help feeling that activities targeted specifically at girls risk reinforcing this stereotype; organising something for girls rather than kids in general flags girls as 'special' in computing. I worry that at the same time as telling them that girls CAN do computing, it carries the implicit message that nonetheless right now girls, on the whole, DON'T.

Singling out role models runs a similar risk; highlighting exceptional women doesn't necessarily lead others to think they could join them on the pedestal. Role projects, however, are great. I don't know the job title of anyone I saw on the broadcasts from the control room of the Philae lander, but my cheers at the news it had landed were even louder for seeing a woman there, front and centre just doing her job without individual fanfare on an inspirational technical project. Teaching people to make their phones meow doesn't compare! But similarly, we were just there, doing the stuff we know how to do. Normal. No big deal. But visible. ■

References

[1] http://www.bcs.org/content/ConWebDoc/54172
[2] http://www.bcs.org/category/8630
[3] https://www.youtube.com/watch?v=EdZPsRYHJ7o
[4] http://appinventor.mit.edu/
[5] https://scratch.mit.edu/
[6] http://appinventor.mit.edu/explore/ai2/tutorials.html

This is a personal account of the event. Opinions are my own and not the views of Bournemouth University or BCSWomen.

WattOS R9 Worth Knowing About

Silas S. Brown recycles some old hardware with a new OS.

If your life as a developer is anything like mine, from time to time you're called on by friends and friends of friends to sort out misbehaving computers, usually Windows systems on which someone has carelessly downloaded a bunch of unwanted software they don't know how to uninstall. As I speak Chinese, I'm usually asked to sort out laptops that are running the Chinese-localised version of Windows, which, unlike most other multilingual operating systems, has no way of temporarily switching the interface back to English for technical support, and it doesn't help that my ability to READ Chinese is less good than my ability to speak it: the user is typically unwilling to read the screen to me because it is 'too technical', and it's not usually possible to copy and paste the characters into dictionary software, so I sometimes have to make a few guesses. The strangest case I had recently was a laptop whose user was complaining about the Web browser not working despite her running a few anti-virus programs. "Just install Firefox" I thought, but it turned out most HTTPS certificates would fail to authenticate in any browser, although at least Firefox gave me a more verbose error message that led me to the root cause: the system clock was set to last year. The TLS library was refusing to accept certificates that seemed to be date-stamped to the future, and Windows' built-in NTP client refuses to automatically synchronize a clock if it's that far out.

for the first time in a long time I was actually asked by a family of computer novices to set up GNU/Linux for them

Most of these users are naturally extremely reluctant to have their operating system changed, but there comes a point when you can go no further. Recently I was faced with a Vista laptop that wouldn't boot, and after trying and failing to recover this I said "either let me put Linux on it or buy a new computer", adding "preferably a Mac" in selfish hope of having less support to do in future. They opted for Linux, and so for the first time in a long time I was actually asked by a family of computer novices to set up GNU/Linux for them.

When setting up GNU/Linux on an old piece of hardware, the first problem is 'which distro will install'. Most modern Linux distros require a DVD to install, but not every old laptop will boot from DVD. Some DVD distros can also install from a USB stick, but this is not always the case. For example, Mint 17, which is a version of Ubuntu's 2014 long-term support release which is supported until 2019 and adds some out-of-the-box features that might appeal to ex-Windows users, has an installer bug that causes it to fail if you install from USB. It is of course possible to patch around this bug, but as there are so many other distributions out there I'd rather just pick one that works cleanly to start with so that I have a simpler recommendation to make when asked (I'd rather be able to say 'install X' than 'install Y and do this bunch of patching'). I could have gone back to the last long-term support release of Mint: Mint 13, based on Ubuntu 2012 and supported until 2017, but I'd really rather give them the 2014 version because they wanted Chinese handwriting input and this has not developed much on the Linux platform until quite recently.

Then I found that the R9 version of WattOS was based on Ubuntu's 2014 long-term support release and still fits on a CD-ROM (no DVD required). It also installs successfully from USB stick if you don't have a CD, and the 'LXDE' version is quite novice-friendly (the 'Microwatt' edition perhaps less so, as it launches you into a window manager that will be most unfamiliar to the average Windows user, so I didn't give that version to this family). I had to download additional drivers for their Wi-Fi card, which I had to install via USB stick as we couldn't plug it in to Ethernet, but at least the provided tools told me which packages were required.

I also had to copy over their Windows files, including many unlabelled high-resolution photographs: it was necessary to go into the settings of the PCmanFM file manager and increase the maximum size of files that it will generate a preview for, as they were used to navigating their pictures by preview rather than by filename. Then it needed LibreOffice (for opening various .doc files they had), Chinese fonts ('apt-get install fonts-wqy-microhei' did the trick), WINE 1.7 (just in case) and some sort of Chinese handwriting input system. This particular family did not know how to use Pinyin or one of the other standard methods of inputting Chinese from the keyboard, and they had a proprietary trackpad-like device that plugged into the USB port and provided a handwriting input system, but of course there was no way I could find a Linux driver for this device so I just installed the package 'tegaki-recognize' (along with its data in 'tegaki-zinnia-simplified-chinese' etc) and set it up with an icon on the application launcher: it will let you write one character at a time with the mouse, and paste it into whichever window was foreground before it was launched; I explained this was not quite as usable as their previous tablet, but it's the best we can do and at least it's running on a nice fast operating system that works.

The other thing I did was to copy over their browser bookmarks. They had been using Chrome with automatic translation (I think it was me that suggested this circa 2009: automatic translation is not very good, but at least it gives them some vague idea of what they're looking at, and means they didn't have to call me every 5 minutes with 'what does this website say'), and it was no problem to save the Chrome bookmarks file and load it on another operating system, but it was more difficult to set the Linux version of Chrome to automatically translate into Chinese (for some unknown reason it kept on insisting that your first language must be English if you're on Linux) so I imported the bookmarks into Firefox and found an add-on to do the translation. I hadn't saved the passwords and cookies from Windows, which turned out to be a mistake because the lady didn't know her Hotmail password and had been relying on the computer to remember it for her for the last few years. All attempts at password retrieval failed, so she had to get a new email account. (Never ever assume a browser will always remember passwords for you: there are any number of events that could interfere with this. If it matters and you can't remember it yourself, keep a backup copy in a safe place. I'm glad my email is still provided by the local university: in the unlikely event that there's a problem with my account, I can walk into Reception with my passport or something and ask for help. You can't do that with the likes of Hotmail.)

But generally I was pleasantly surprised how smoothly it all went, and how readily they accepted the result. I was expecting to be screamed at because the desktop looked different or the icons were in all the wrong places or their favourite calendar widget was missing or something, but none of this happened: the only 'hiccup' was the missing Hotmail password. ■

SILAS S. BROWN

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

Raspberry Pi Linux User Mode GPIO in C++ (Part 2)

Ralph McArdell continues developing a C++ library for Raspberry Pi expansions.

Previously [1] I described the initial stage of developing a library called **rpi-peripherals** [2] to access general purpose input output (GPIO) on a Raspberry Pi running Raspbian Linux in C++ from user land – that is there are no kernel mode parts to the library. The library was built on memory mapping the physical memory locations of the Raspberry Pi's BCM2835 processor's peripherals' control registers using the **dev/mem** device accessed via an RAII (resource acquisition is initialisation [3]) resource managing class template called **phymem_ptr**.

Part 1 ended having described support for reading and writing single bit Boolean values representing the high/low voltage state of an associated GPIO pin in the forms of the **ipin** and **opin** types. Along the way we met various other entities such as the **pin_id** type representing the value of a BCM2835 GPIO pin, and the aforementioned **phymem_ptr** template.

This second instalment continues by describing adding support for some other IO functions and the challenges they presented.

But first...

One thing the **ipin** type does not support is waiting for a change of state to its associated GPIO pin before returning from a **get** operation. I really needed to address this as it was the very thing that started me on the road to writing my original Python Raspberry Pi GPIO library [4] [5].

In the Python GPIO library the need for a blocking read was specified in the call to an **open** function and a suitable object would be returned that had a **read** operation that would wait until the specified edge event (rising, falling or either) occurred.

As mentioned the **rpi-peripherals** library directly accesses peripherals' registers by memory mapping them curtesy of the **phymem_ptr** class template. The only readily available way to receive GPIO pin edge event notifications in user space is via the **/sys/classes/gpio** pseudo file system – as used by my Python library.

I did not like the idea of mixing GPIO access methods in **ipin** or some related class so I took a different approach. Instead a totally separate class called **pin_edge_event** encapsulates handling pin edge events via **/sys/classes/gpio**. In order to work with **/sys/classes/gpio** the GPIO pin's number is required and the pin should be exported [4] [6] and, of course, set up for input. As it happens an **ipin** instance, using the **pin_export_allocator** type to control access to GPIO pins between processes, just happens to fulfil all these criteria. Hence a **pin_edge_event** is constructed from an existing **ipin** instance together with an indication of which edge events are of interest:

```
ipin in_pin{pin_id{23}};
pin_edge_event pin_evt{in_pin,
  pin_edge_event::rising};
```

On construction the pin's associated **/sys/classes/gpio** edge event file is opened. The pin is marked as having an associated **pin_edge_event** object as I thought it too confusing to allow more than one per **ipin** at a time. On destruction the pin is effectively closed for edge events by passing the file descriptor obtained during the open process to the Linux **close** function and the pin marked as not having an associated **pin_edge_event** object.

The implementation of **pin_edge_event** revolves around a call to **pselect** [7] – chosen over **select** for the fairly flimsy reason that it allows timeout resolution in nanoseconds rather than microseconds. The **pin_edge_event** interface allows waiting in various ways for an event to be signalled, from waiting indefinitely for an event to just checking to see if an event has been signalled:

```
pin_evt.wait();
assert(pin_evt.signalled());
```

In between, in the style of certain C++11 library APIs, there are **wait_for** and **wait_until** member function templates to wait for a specific amount of time or wait until a specific absolute time for an event to be signalled. They are templates as they use **std::chrono::duration** and **std::chrono::time_point** specialisations for their time parameters:

```
auto wait_duration(std::chrono::milliseconds{
  100U});
bool edge_event_signalled{
  pin_evt.wait_for(wait_duration)};
...
auto now(std::chrono::system_clock::now());
auto wait_time(now+wait_duration);
edge_event_signalled =
  pin_evt.wait_until(wait_time);
```

As can be seen from the example usage the **signalled**, **wait_for** and **wait_until** operations return a **bool** value which is **true** if an event was signalled. The **wait** operation does not return a value as it waits indefinitely for an event: if it returns then there was an event.

The final operation supported by **pin_edge_event** is the **clear** operation which needs to be performed after an edge event has been signalled. This has to do with how **/sys/classes/gpio** edge event handling works in that the value of the input pin the event occurred on needs to be read from the relevant file before another event can be waited on. Another **/sys/classes/gpio** edge event handling quirk is that a **pin_edge_event** object is initially in the signalled state:

```
ipin in_pin{pin_id{23}};
pin_edge_event pin_evt{
  in_pin,pin_edge_event::rising};
assert(pin_evt.signalled());
pin_evt.clear();
assert(!pin_evt.signalled());
```

The time has come

Having sorted out single pin GPIO support the time had finally arrived to look at some of the other peripheral functions available. I thought that adding support for pulse width modulation (PWM) allowing me to make use of the motor controller on the Gertboard would be an interesting next step. But as 'pulse width modulation' hints at, regular pulses are required which implies the use of a clock. In the case of the BCM2835 the PWM

RALPH MCARDELL

Ralph McArdell has been programming for more than 30 years with around 20 spent as a freelance developer predominantly in C++. He does not ever want or expect to stop learning or improving his skills.

controller uses a separate but dedicated clock that has the same programming interface as the general purpose clocks that can be connected to GPIO pins. So in order to support PWM I would first have to provide support for clocks.

Like GPIO, clocks are controlled by a set of registers based at a specific physical address. Each clock is controlled by the clock manager peripheral and has its own set of two registers at an offset from this base address. Frustratingly the BCM2835 ARM Peripherals document [8] only gives the base address for the clock manager and the offsets from it for the three general purpose clocks' register sets, not for clocks associated with other peripherals such as the PWM clock – although mention is made that the PWM clock is designated **clk_pwm**. I had to refer back to the provided Gertboard C code to locate the required offset value.

As with GPIO I started with the clock registers layout. Each clock controlled by the clock manager has the same register structure so I split the implementation into two structures: one describing a single clock, which I called a **clock_record** and the main **clock_registers** structure which contained a **clock_record** member for each supported clock carefully placed so that it was at the correct offset from the start of a structure instance – and yes there is a test to check they are at the expected offsets.

Like **gpio_registers** I provided member functions to get or set the individual fields within a clock's registers. In the case of **clock_registers** which clock to operate on needs to be specified implying passing some sort of identifier. The easiest solution turned out to be to define the **clock_id** as a type alias for a pointer to **clock_record** member of **clock_registers**:

```
typedef clock_record clock_registers::* clock_id;
```

Each member function of **clock_registers** takes a **clock_id** as a parameter and uses it to pass on the call to the identified **clock_record** member:

```
clock_registers
{
  ...
  clock_src get_source(clock_id clk)
    volatile const
  {
    return (this->*clk).get_source();
  }
  ...
};
```

The specific clock ids were then defined as global **constexpr clock_id** instances initialised to the relevant **clock_record** member's 'address' value, for example:

```
constexpr clock_id pwm_clk_id
  {&clock_registers::pwm_clk};
```

Frequent division diversions

You would think the interface to a clock would be simple: specify the required frequency and provide operations to start, stop and possibly query the frequency and the running state. This is the sort of interface I wanted the public library clock support to provide.

However, at the lower levels it turns out to be not so simple. First you have to supply the clock with a source of oscillation at a fixed frequency – the Raspberry Pi has a 19.2 MHz oscillator that can be used as an external (to the BCM2835) clock source which seems the easiest to use. Next it needs to be divided down to the required frequency.

Dividing down the clock source is more complex than just supplying an integer divider. Most required frequencies will have no integer divisor that produces an exact match. For example if the clock source oscillates at 1 MHz and we require a 134 KHz clock then the best we can do is divide by 8, yielding a frequency of 125 KHz, or by 7, yielding a frequency of around 143 KHz. So in addition to integer division the clocks provide something called MASH filtering (MASH it appears stands for Multi-stAge noise Shaping) – about which I know very little other than the information provided in the BCM2835 clock peripheral documentation. When using one of the three MASH filtering modes a fractional division value is used in addition to the integer division value. The result is that the actual clock frequency varies slightly between a minimum and maximum value, but the average frequency should be very close to that requested. The down side is that the MASH filtering modes introduce a bunch of constraints on maximum frequency and minimum integer divider value.

I wanted to work in terms of frequency rather than modes and divisor values. Providing a frequency type would allow the use of frequency units such hertz, kilohertz and megahertz. Thinking about this I noted that the inverse of frequency – or cycles per second – is a duration value – seconds per cycle. The standard library has the **std::chrono::duration** class template along with type aliases for specialisations representing various common time units such as microseconds and hours. I felt there should be some way to use **std::chrono::duration** to represent frequency. However, a solution was not immediate forthcoming so to keep moving forward and not get distracted further I effectively copied the required parts of the **std::chrono::duration** class template as my library's **frequency** class template. The implementation was so similar that, in a somewhat hacked manner, when I produced a **frequency_cast** function template to cast between different frequency specialisation types, it was implemented in terms of **std::chrono::duration_cast** and **std::chrono::duration** – I had reached the end of my patience on these diversions! Completing the support for frequency I added a bunch of **frequency** specialisation type aliases for common frequency units: hertz, kilohertz and megahertz.

In addition to the frequency support I also added enumeration and simple class types and constant definitions to help with specifying a clock including a constant definition for the Raspberry Pi's 19.2 MHz oscillator. These all live in the `clockdefs.h` library public header.

I created the **clock_parameters** class to aid bridging between frequencies and clock modes and divisors. Instances are created from a clock source and frequency (external clock source at 19,200,000 Hz for the Raspberry Pi's 19.2 MHz external clock for example) along with the desired clock frequency specification combining the desired (average) frequency and an enumeration value specifying the level of MASH filtering required: maximum, medium, minimum or none – where none means use only an integer divisor.

During construction, after some basic parameter value checks, possibly repeated attempts are made to try to obtain a valid frequency value starting with the filter mode requested in the constructor parameters and falling back to lower levels if the maximum frequency produced is too high. If the frequency value exceeds even the substantially higher value allowed the finally attempted integer only division mode a **std::range_error** is thrown. A **std::range_error** will also be thrown if the integer divisor is too small for the selected filter mode.

If no exception is thrown during construction then the various parameters can be queried via non-modifying accessor member functions.

Now there are two

The purpose of the **clock_registers** class is for a volatile instance to be mapped to the clock peripherals' register block using a **phymem_ptr<volatile clock_registers>** instance. Some ability to detect trying to use the same clock peripheral multiple times would also be useful. So, as with the **ipin** and **opin** types and the **gpio_ctrl** singleton, a singleton type was created combining a **phymem_ptr<volatile clock_registers>** instance with simple in-process clock-in-use allocation provided by the **simple_allocator** class template, specialised on the number of things available to allocate (in this case the 4 clocks: pwm_clk and gpclk 0, 1 and 2) and based around a **std::bitset**. Only in-process allocation management was provided because I could not see any straight forward way to provide an open inter-process allocation management scheme for clocks or other peripherals.

Setting up a GPIO pin for use as one of the three general purpose clocks not only requires access to the **clock_ctrl** instance but also to the **gpio_ctrl** instance so as to allocate the pin and set the correct alternate

function for it. This would of course apply to any other peripherals supported by the library. When the only thing that needed to access the main GPIO registers were the **ipin** and **opin** types then **gpio_ctrl** could be left internal to the pin.cpp implementation file. But now there were two – the **ipin**/**opin** code in pin.cpp and the **clock_pin** code in clock_pin.cpp – some changes would be required.

As a bout of refactoring was inevitable it seemed prudent to decide on some conventions. First the library facilities were divided into the public API parts and library internal parts with the internal parts being placed in a nested **internal** namespace. Next, those headers required for using the public API were moved from the project src directory to the project include directory. This was always going to happen – it was just a matter of what and when. Finally, the **gpio_ctrl** code was moved out of pin.cpp and into its own library internal files gpio_ctrl.h and gpio_ctrl.cpp. A similarly named type and implementation file-pair were created for the clock peripherals: **clock_ctrl** in files clock_ctrl.h and clock_ctrl.cpp.

This leads to a basic pattern: for a peripheral ***p*** there would be a *p*_registers.h header file containing a ***p*_registers** class usually together with supporting entities that mapped ***p***'s register structure and associated values to C++ entities. This would be used, qualified **volatile**, to specialise a **phymem_ptr** mapped to the peripheral's registers' physical memory block start address along with some sort of in-use tracking in a ***p*_ctrl** singleton type implemented in *p*_ctrl.h and *p*_ctrl.cpp. The ***p*_registers** and ***p*_ctrl** types (and source files) are internal to the library. The public API would be presented by a type ***p*_pin** with *p*_pin.h being placed in the project include directory. Along the way there may be ancillary items which would often be internal to the library (such as **phymem_ptr** or the **/sys/classes/gpio** support in sysfs.h and sysfs.cpp) but sometimes – as with **pin_id** and those entities placed in the clockdefs.h header – would be part of the public API. Figure 1 shows the pattern as a UML class diagram; **ipin** and **opin** are included to show they only access **gpio_ctrl** while other peripherals additionally access their own ***p*_ctrl** type.

Can you do this?

The **clock_pin** class provides the library's public support for general purpose clock functions on GPIO pins and unsurprisingly requires a **pin_id** specifying which GPIO pin to use. Wherein lies a problem. Unlike general input and output which all GPIO pins can perform, alternative functions – such as a general purpose clock (gpclk) function – can only be performed by a few, sometimes only one, pin. Which alternate functions a pin can perform is given in a table in the BCM2835 ARM Peripherals document.

On the other hand no pin supports more than one clock peripheral so if a pin supports one of the general purpose clocks then the pin number uniquely defines which general purpose clock (0, 1 or 2).

To help check if a pin supports a given peripheral function and which of the six alternate pin functions it is supported by I created the **gpio_alt_fns** module that provides a set of overloaded **select** query

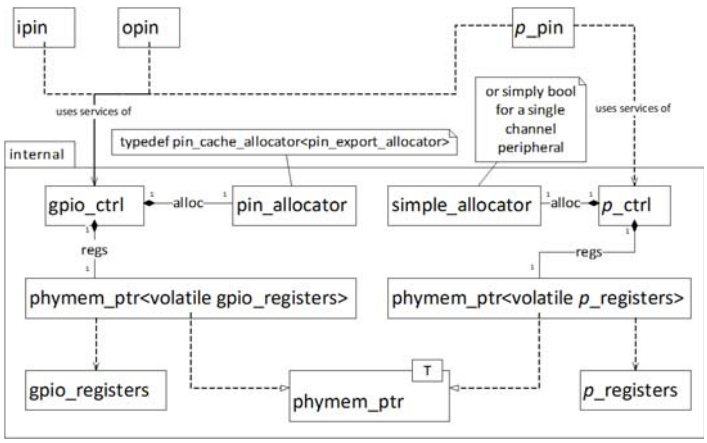


Figure 1

functions that select data from a statically initialised 2 dimensional array that defines the alternate functions each pin supports. The values are enumeration values taken from another table in the BCM2835 ARM Peripherals document that names the peripheral functions.

This allows questions such as which alternate function for pin *p* supports peripheral function *f* or which, if any, of a set of peripheral functions *fs* does pin *p* support? The **select** functions return a **result_set** object that has a partial STL container like interface allowing access via iterators, **operator[]** and **at** and can be queried for **size** and being **empty**. The items in the result set are of a simple descriptor type specifying the pin, the special peripheral function and the alternative pin function it is supported on.

During the development of the **gpio_alt_fns** module I found that I had prefixed almost all identifiers with **pin_alt_fn_**. This seemed silly so I gave in and placed the whole lot in its own **pin_alt_fn** nested namespace.

Easy time?

So how easy is it to use a GPIO pin as a general purpose clock?

Like **ipin** and **opin**, **clock_pin** uses RAII to manage the GPIO pin and general purpose clock resources. The most complicated operation is creating a **clock_pin** instance. Once successfully created the object can be used to easily start, stop and query whether the clock is running as well as obtain the values for the minimum, maximum and average frequencies the clock is using.

To create a clock the **clock_pin** constructor needs to be passed three things: the **pin_id** of the GPIO pin to use as a clock – which should support such a function of course, a clock source (passing **rpi_oscillator** defined in clockdefs.h is the easiest option), and finally a **clock_frequency** object specifying the desired clock frequency and the filter mode to apply. The **clock_frequency** type is defined in clockdefs.h.

For example we could create a 600 KHz clock with no MASH filtering (that is, using only integer division) like so:

```
clock_pin clk{
  gpio_gclk, rpi_oscillator,
  clock_frequency{kilohertz{600},
  clock_filter::none}
};
```

Note that **gpio_gclk** is defined in pin_id.h and yields a **pin_id** for GPIO pin 4, which supports gpclk0 as alternate function 0 and is available on pin 7 of the Raspberry Pi P1 connector. During construction all values are checked, with exceptions thrown in case of problems, and the GPIO pin and clock allocated and setup. The clock and pin are of course released during destruction, after ensuring the clock is stopped.

To check what frequencies are being used the **frequency_avg**, **frequency_min** and **frequency_max** member functions can be called. In this case we would expect a 600 KHz value for all three frequency values as only integer division of the clock source was applied and, as it happens, 600 KHz divides wholly into 19.2 MHz:

```
assert(clk.frequency_min()==hertz{600000U});
assert(clk.frequency_avg()==hertz{600000U});
assert(clk.frequency_max()==hertz{600000U});
```

We can check if the clock is running – which just after construction it should not be:

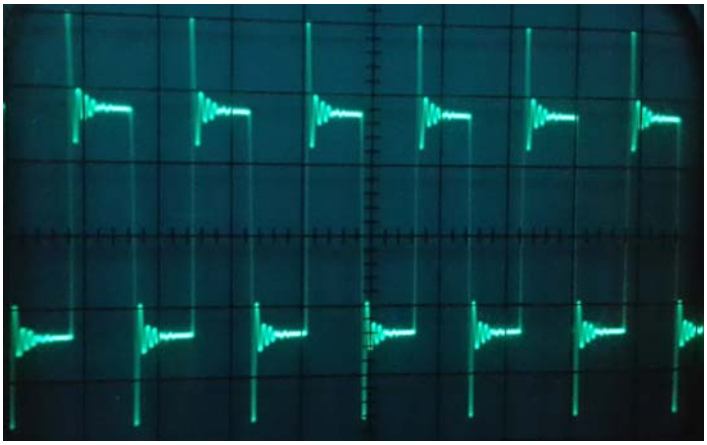
```
assert(!clk.is_running());
```

And of course we can start and stop the clock:

```
clk.start();
...
clk.stop();
```

The output of gpclk0 running at 600 KHz can be observed by connecting GPIO pin 4 to the input of an oscilloscope as shown in Figure 2 – in which the time-base used is 1 µS per division.

Figure 2



After clocking up all those distractions...

Having provided support for clk_pwm and the general purpose clocks and refactored the library, I could return to pulse width modulation. PWM [9] allows control of power to devices such as motors by varying the ratio of high to low time per clock cycle (the duty cycle). You will notice in the clock trace shown in Figure 2 that these are equal, a ratio of 0.5: during each cycle the clock pulse is high for half the time and low for the other half of the time. PWM allows this ratio to be varied dynamically.

The BCM2835 has a single PWM controller that supports two channels that are referred to as PWM0 and PWM1 (as denoted by pin alternate functions) or channels 1 and 2 (as denoted by the PWM controller's register descriptions), where channel 1 maps onto PWM0 and channel 2 to PWM1. Each channel can be used in either PWM mode or serialiser mode in which buffered data is written serially to the PWM channel's GPIO pin. I included support for serialiser mode in the `pwm_registers` class implementation for completeness but do not provide support in the `pwm_pin` class. There are two PWM channels associated with the PWM controller but only one clock – clk_pwm – thus the clock settings used for clk_pwm apply to both PWM channels.

As it happens both PWM channels are used by the Raspberry Pi for stereo audio – using one PWM channel per audio channel. PWM0 can also be accessed for other purposes on GPIO pin 18 via pin 12 of the P1 connector. Of course using PWM for other things will most likely mess up the Raspberry Pi's audio output.

The PWM peripheral has modes of usage I decided not to support: serialiser mode for a start. Other were DMA (direct memory access – an intriguing possibility – maybe one day) and FIFO buffering. I also decided to set certain other options to fixed values such as not to use inverted polarity and to only use the standard PWM sub-mode (the alternative so-called MS mode seemed like a half-way house between serialiser mode and standard PWM mode). Again, support was available in `pwm_registers` but not used by `pwm_pin` – other than to set the fixed feature values – generally to off (`false`).

Like the clock peripherals there are multiple (well, two) PWM channels so the member functions of `pwm_registers` mostly require a parameter to specify the channel (the exception being those functions relating shared resources such as DMA or the FIFO buffer). Unlike the clock peripherals there is no repeated register structure: some registers contain sections for each channel while others relate to one channel or the other. Hence a `pwm_registers` auxiliary enumerated type `pwm_channel` is used as a channel identifier and the enumerated values used to select either the required register or the required part of a register.

The two attributes that did need to be user-set for each channel were the range and the data. Together these are used to define the duty cycle ratio of the PWM output. The range value defines the number of bits over which the duty cycle high/low ratio waveform is spread and repeated. The data value defines how many of the bits of the range will be high and the algorithm used by the PWM controller will try to spread these out as evenly as possible. Taking the example from the BCM2835 ARM Peripherals document's PWM section, if 4 bits of a range of 8 (a ratio of 4/8 or 0.5) are to be high then the pattern would be:

```
1 0 1 0 1 0 1 0
```

Rather than:

```
1 1 1 1 0 0 0 0
```

Or:

```
1 1 0 0 1 1 0 0
```

Each bit of the range would be used to set the high/low state of the associated GPIO pin changing from one bit's state to the next on each clock 'tick', as provided by clk_pwm which should run at a reasonably high frequency – it is set to a default of 100 MHz by the hardware.

The handling of clk_pwm is split between `pwm_pin` and `pwm_ctrl`. The `pwm_pin` class provides static member functions to work with clk_pwm with the `pwm_pin::set_clock` member function performing a similar function to the constructor of `clock_pin`, but does not require a `pin_id` value as clk_pwm is never mapped to a GPIO pin. The other three functions provided are:

- `pwm_pin::clock_frequency_min`
- `pwm_pin::clock_frequency_avg`
- `pwm_pin::clock_frequency_max`

They return the values for the frequencies used by pwm_clk in the same fashion as:

- `clock_pin::frequency_min`
- `clock_pin::frequency_avg`
- `clock_pin::frequency_max`

The low level details of setting up the clock that require access to `clock_ctrl` are delegated to `pwm_ctrl::set_clock` – an additional piece of functionality for the `pwm_ctrl` singleton type in addition to the `phymem_ptr<volatile pwm_registers>` and `simple_allocator` specialisation for the two PWM channels.

Setting pwm_clk up in the same fashion as the `clock_pin` usage example would look like so:

```
pwm_pin::set_clock(rpi_oscillator,
  clock_frequency{kilohertz{600},
  clock_filter::none });
```

which has the same parameters as the `clock_pin` object construction example less the initial `pin_id` parameter.

Where a `pin_id` is required – unsurprisingly – is in the construction of `pwm_pin` instances to specify which pin we want PWM output on. Details of which PWM channel (if any) and which alternate function of the GPIO pin is used for the PWM function being asked of a `pin_alt_fn::select` function. The other `pwm_pin` constructor parameter is an unsigned integer range value, defaulting to a value of 2400 – a fairly long range value which is divisible by quite a few values. Hence the only value needed to construct a `pwm_pin` object is a `pin_id`:

```
pwm_pin pwm{gpio_gen1};
```

where `gpio_gen1` yields a `pin_id` value for GPIO pin 18 available on pin 12 of the Raspberry Pi P1 connector.

Now we have a `pwm_pin` object we can start and stop the PWM output, check whether it is running or not and set the ratio. The first three are simple to use and to implement:

```
assert(!pwm.is_running());
pwm.start();
...
assert(pwm.is_running());
...
pwm.stop();
...
assert(!pwm.is_running());
```

The `set_ratio` operation, although easy to use is more interesting in its implementation. There are two forms, one that takes a `double` floating point value and another that takes a `pwm_ratio` value.

Standards Report

Jonathan Wakely reports the latest on C++17 and beyond.

I'd like to start my first *CVu* standards report by thanking Mark Radford for writing the reports for many years, keeping them interesting and insightful. I hope I can do the same.

Just after Mark's last report in May's *CVu* the C++ committee met in Lenexa, Kansas for another six long days with several groups meeting every day, working on the main standard and a number of Technical Specifications (TS). With a new standard due in 2017 time is running out to decide on the features that will be included, as we will need some time to iron out the bugs and (we hope) get some early implementation experience before it goes to publication. We also need to send the final standard to ISO in plenty of time so they don't end up publishing it the following year, as nearly happened with the 2014 standard.

Bjarne Stroustrup gave a talk in Lenexa about what he hopes to see in the next standard, his main aims are to:

- Improve support for large-scale dependable software
- Provide support for higher-level concurrency models
- Simplify core language use, especially as it relates to the STL and concurrency, and address major sources of errors.

In his words, these are 'motherhood and apple pie', but there are specific proposals being discussed which work towards these goals, which I will cover below. Of course there are also many other proposals that work towards those goals less directly, or towards totally different goals, but I'm not going to focus on those for this report.

Large-scale dependable software

There are two groups working on proposals for C++ modules (N4465, N4466) and several discussions took place during the Lenexa meeting, which I hope will result in some common ground and a unified proposal in time for C++17. Modules will help replace the preprocessor and the simplistic header file model for using libraries, something which appeals to many people.

JONATHAN WAKELY

Jonathan's interest in C++ and free software began at university and led to working in the tools team at Red Hat, via the market research and financial sectors. He works on GCC's C++ Standard Library and participates in the C++ standards committee. He can be reached at accu@kayari.org

Raspberry Pi Linux User Mode GPIO in C++ (Part 2) (continued)

Before getting into **pwm_ratio** let's first look at the overload taking a **double**. The value should be in the range [0.0, 1.0], with values outside this range throwing a **std::out_of_range exception**. The value is used to calculate the proportion of the range value to set for the PWM channel's data register value with a value of 0.0 producing low values for the whole range, and a value of 1.0 all high values. For example the output could be set to be high for a quarter of the time like so:

```
pwm.set_ratio(0.25);
```

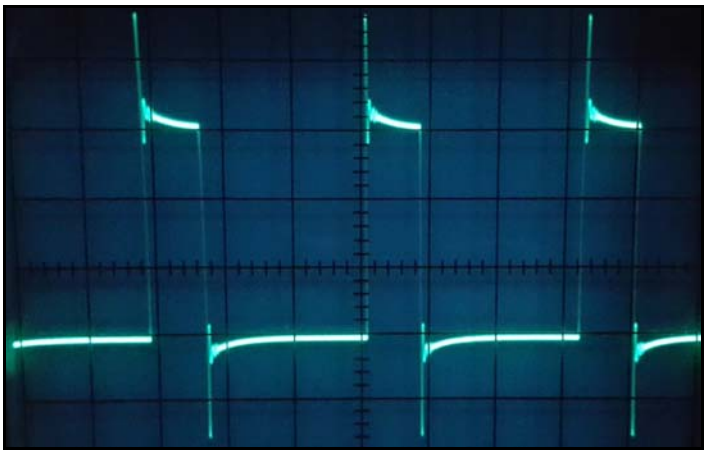
I thought it would be nice to be able to express setting the ratio as a ratio. So `pwm_pin.h` includes a **pwm_ratio** class template, specialised by an integer type and a **std::ratio** specialisation (or similar type). Instances of specialisations of **pwm_ratio** hold a count value of the template integer parameter type, and define **static constexpr** values **num** (numerator) and **den** (denominator) equal to those values of the **std::ratio** specialisation template parameter type. For example for 0.3 count could be 3 with **num** and **den** set from a **std::ratio** with **num** 1 and **den** 10 as per **std::deci**, or maybe count is 30 with **num** and **den** from a **std::ratio** with **num** 1 and **den** 100 as per **std::centi**. I also defined a set of type aliases for **pwm_ratio** specialisations for ratios as numbers of tenths (**pwm_tenths**), hundredths (**pwm_hundredths**), thousandths (**pwm_thousandths**) and millionths (**pwm_millionths**).

The other form of the **set_ratio** operation is a member function template that takes a **pwm_ratio** specialisation, and hence requires the same template parameters as **pwm_ratio**. Using a ratio to set the PWM output to be high 25% of the time would look like this:

```
pwm.set_ratio(pwm_hundredths(25));
```

Figure 3 shows the output of PWM0 using a 600 KHz clock, range of 2400 and a high ratio of 25% as observed by connecting GPIO pin 18 to the input of an oscilloscope this time using a time-base of is 2 μS per division. ■

Figure 3



References

[1] Raspberry Pi Linux User Mode GPIO in C++ – Part 1, *CVu*, Volume 27 Issue 2, May 2015

[2] dibase-rpi-peripherals library project: https://github.com/ralph-mcardell/dibase-rpi-peripherals

[3] See for example: http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

[4] Raspberry Pi Linux User Mode GPIO in Python, *CVu*, Volume 27 Issue 1, March 2015

[5] dibase-rpi-python library project: https://github.com/ralph-mcardell/dibase-rpi-python

[6] Documentation/gpio.txt in the Linux kernel tree, as located at: https://github.com/raspberrypi/linux/blob/rpi-3.2.27/Documentation/gpio.txt

[7] pselect man page, for example at: http://linux.die.net/man/2/pselect

[8] BCM2835 ARM Peripherals: http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

[9] Pulse width modulation: https://en.wikipedia.org/wiki/Pulse-width_modulation

There are also two competing proposals for contract-based programming (N4378, N4415). These contracts allow preconditions and postconditions of functions to be stated in code, so that they can be verified automatically by the compiler. There seems to be a lot of disagreement about the basic direction that contract checking in C++ should take.

The last active proposal in the large-scale dependable software category is a type-safe union, along the lines of `boost::variant`. There is only one proposal for a variant type, but the latest revision of it (N4542) is the fourth version, having gone through review sessions at previous meetings, and then changes in response to those previous discussions. There are a number of design decisions that need to be made for a variant type and there is no consensus on the right choices. There have been long discussions at the meetings and even longer ones afterwards by email. Anthony Williams wrote an excellent blog post [1] which covers the main controversial areas, and the comments on the post describe some of the opposing views.

Two of the hottest topics are what should happen if changing the type stored in a variant causes an exception to be thrown and what it should mean to declare a variant using the same type more than once, e.g. `variant<string, int, int>`. The discussions have gone round and round, apparently without changing anyone's views or getting any closer to consensus, as different people consider different aspects of the design to be higher priority and don't want to compromise on whichever part matters to them. Personally I've seen enough discussion and don't feel strongly enough to fight for any particular design. I'd rather have (almost) any kind of variant rather than debating it forever and not getting a variant in the standard at all. The discussions are expected to continue at the next meeting, probably in a larger group than previous discussions. Now that the proposal has graduated from Library Evolution Working Group (LEWG) sessions a wider audience from the full committee will be involved.

Higher-level concurrency models

C++11 gave us some low-level concurrency primitives such as mutexes, atomic operations and thread creation, but nothing higher-level than `std::future` and `std::async`. The low-level pieces are considered too difficult for many programmers to use correctly, or just the wrong level of abstraction. It's important to provide better abstractions that allow programmers to think about the work they want to perform, not about how many threads to create and how to do the error-prone synchronisation needed to communicate between threads.

The Parallelism and Concurrency study group (SG1) have been busy as ever, with the content of the Concurrency TS being completed and ready for a final draft to be considered by the national bodies. Mark previously discussed the Concurrency TS in this column, so I'll just say that it includes support for running continuations when the result of a future becomes ready, so that operations can be chained together. There are also competing proposals for some form of coroutines (N4402, N4403, N4398). Discussions are ongoing for some form of executor that would allow programmers to pass a unit of work to an executor. The executor would do the tricky job of scheduling work to run in some execution context without the programmer needing to create and manage a thread for every unit of work. Like variant, everyone wants executors, but don't agree what form they should take.

The Transactional Memory study group (SG5) have been working on the Transactional Memory TS, offering a simpler programming model than explicit atomic operations on individual atomic objects. That TS has now been sent to ISO for publication.

The Networking TS, based on Boost.Asio, also offers a higher-level concurrency model. Although the main purpose of the TS is to add support for working with network sockets and related things like IP addresses and name resolution, much of the Asio library's strength is the model used for doing I/O, name resolution etc. as asynchronous operations. That means the Networking TS will provide its own high-level concurrency model (there is clear overlap with the more general executor work being done by SG1, but that redundancy will have to be dealt with at a later date). There was an intensive week-long review session of the Networking proposal at the unofficial Cologne meeting in the spring, and LEWG had a couple of sessions in Lenexa to review Chris Kohlhoff's changes in response to the Cologne review. In most cases LEWG simply agreed with Chris's suggested changes. As the author of Asio and the proposal he knows the design, implementation and use cases better than anyone, but I think there were some very useful suggestions made in Lenexa which simplify some of the hairier parts of the specification. There will be more work on the Networking TS at the next meeting and I hope to produce an initial draft TS based on Chris's latest proposal.

Simplify core language use

The biggest news in this area is that reviewing the Concepts TS was finished in Lenexa, so following a final review done by teleconference shortly after the meeting, the TS has been sent to ISO for publication. More recently a complete implementation of the TS was merged into the GCC Subversion trunk. This means there is a working implementation of C++ Concepts available in a major compiler today (if you check out the development sources and build GCC yourself). It's too early to say whether Concepts will also be part of C++17, but even if it isn't we'll have the TS and support in at least one compiler.

Updating the standard library to use concepts is currently being done as part of Eric Niebler's work on Ranges. His proposal is being reviewed between meetings by teleconference, so there should be more news on that following the next meeting.

The Evolution Working Group are looking at several proposals for simplifying the use of advanced features, including uniform call syntax. This would add core language support for something that the library currently tries to do, but arguably doesn't do very consistently.

Finally, it would make sense to incorporate some of the utilities in the Library Fundamentals TS into C++17, particularly `string_view` and `optional`.

Other news from Lenexa

One of the other hot topics at the Lenexa meeting was a proposal to add several mathematical functions to the C++17 standard library. These functions were part of TR1 back in 2005, but unlike the rest of TR1 were not incorporated into the C++11 standard, due to the reported difficulty of writing good quality implementations. The functions were instead moved into a separate ISO standard, so a standard C++ implementation doesn't need to include them. That standard has now come up for a periodic review where we have to decide whether to renew it, re-confirm it or retire it. N4437 proposed to add these functions to C++17, as 'conditionally supported', so that implementations are still not obliged to provide them, but the committee would not need to maintain a separate ISO standard specifying them. In Lenexa, representatives of the target audiences for these functions argued that they need to be unconditionally supported, so the committee discussed that which proved even more controversial! This will be debated again in Kona.

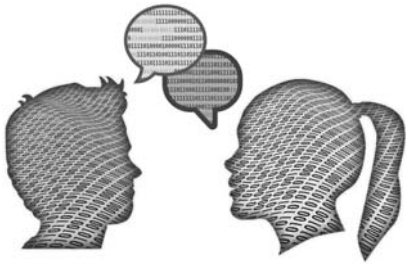
Finally, a new study group, SG14, covering 'Games and Low Latency' was started. They plan to look at subjects related to those topics and try to get members of the game development community to participate in C++ standards work, to ensure their needs are met. It's not entirely clear whether this is really just about game development, or also relevant to other low latency environments such as electronic trading and some embedded systems, time will tell.

Future meetings

The committee will be meeting in Kona, Hawaii, in October, and then in Jacksonville, Florida in February, and then Oulu, Finland in June. After the Kona meeting we might have a better idea of what will and won't be included in C++17.

Code Critique Competition 95

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



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. (We will remove email addresses!)

Last issue's code

> I had some code that used an anonymous structure within an anonymous union to easily refer to the high and low 32-bit parts of a 64-bit pointer. However, I get a warning that this is non-portable (I'm not quite sure why - MSVC and g++ both accept it) but after googling around for a solution I found one that uses #define. It all compiles without warnings now so I think it's fixed.

Can you give some advice to help this programmer?

The code is in Listing 1.

Listing 1

```
--- address.h ---
#pragma once
typedef struct {
  union {
    struct {
      int32_t offsetLo;
      int32_t offsetHi;
    } s;
    void *pointer;
  };
} address;
// simulate anonymous structs with #define
#define offsetLo s.offsetLo
#define offsetHi s.offsetHi

--- test program ---
#include <cstdint>
#include <iostream>
#include "address.h"
int main()
{
  int var = 12;
  address a;
  a.pointer = &var;
  std::cout << "Address = " << std::hex
    << a.offsetHi << "/" << a.offsetLo
    << std::endl;
}
```

Critiques

James Holland <James.Holland@babcockinternational.com>

The student says that warnings are issued but not when using MSVS or G++. It would appear that the student's compiler is trying to warn of something that the other compilers are keeping quiet about. It is a pity that the student does not tell us what the warnings are. In their absence, it is worth thinking about what the warnings could be. This may provide some clues as to why the code may not be portable.

It may be the case that the compiler is generating pointers of length other than 64 bits. Without instructing the compiler otherwise, it might well produce 32-bit pointers. This would have the effect that when the address of `var` is assigned to `a.pointer`, `offsetHi` would not get assigned a value. This is because a.pointer is only of sufficient size to cover `offsetLo`. The result is that `offsetHi` would be left with an undefined value. So, if the compiler is generating 32-bit pointers, it would be appropriate for a warning to be issued to the effect that `offsetHi` is never initialised. We need to find a way to guard against this scenario.

Probably the most direct way is to check at compile-time that the pointer is 64 bits in length. This can be achieved using the following `static_assert` statement.

```
static_assert(sizeof address::pointer == 8,
  "Pointers are not 64-bit!");
```

If the compiler is not generating 64-bit pointers, the error message will be issued when the program is compiled thus preventing executable code from being produced.

What else could go wrong? Well, for the output of the program to be interpreted correctly, it is essential that `offsetLo` is located on the least significant half of a.pointer and that `offsetHi` is located on the most significant half. There are two ways in which machines store multi-byte values; least significant bytes first followed by the more significant bytes or most significant bytes first followed by the lesser significant bytes. The former method is known as little endian, the latter is known as big endian. As an example, the table below shows the two types of endian for the value 0x12345678.

Endian	Big				Little			
Byte value	12	34	56	78	78	56	34	12
Byte offset	0	1	2	3	0	1	2	3

From the supplied code, it can be seen that the student is expecting the software to be run on a little endian machine as `offsetLo` has been declared before `offsetHo` and so `offsetLo` will be located at the lower memory address. The following code could be used to determine the endian of a particular machine.

```
a.address = reinterpret_cast<int *>(1);
if (a.offsetLo == 1)
{
  // Little endian
  ...
}
else
{
  // Big endian
  ...
}
```

I leave it to the student as to how the code is used in a practical program.

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

It is conceivable that the compiler will place some padding bytes between **offsetLo** and **offsetHi**. This will depend on the settings of the compiler and on machine for which the software is designed to run. For the program to work correctly there must be no padding. It is possible to test for the presence of padding by use of the following code.

```
const auto x =
  reinterpret_cast<size_t>(&a.offsetLo);
const auto y = sizeof address::offsetLo;
const auto z
  = reinterpret_cast<size_t>(&a.offsetHi);
if (x + y != z)
{
  std::cout <<
    "offsetLo and offsetHi not contiguous!" <<
    std::endl;
}
```

All of these problems occur because the program is relying on language features that are implementation dependant. It would be nice if the compiler would give warnings if such features are used but the standard does not mandate it and not many compilers do. If implementation dependant features must be used, it is up to the programmer to make checks to ensure the program is valid.

The student implies that when the inner structure is given a name and the **#defines** that simulate anonymous structures are used, the warnings disappear. This is very strange. **#defines** are little more than a text substitution mechanism and so I would not have thought they would have any effect on warning message issued. Also, I would not have thought giving the inner structure a name would have any affect either, so this remains a mystery. Irrespective of whether a warning is issued, any underlying problem would remain the same. There is still implementation dependant behaviour that must be considered.

As the student simply wishes to refer to the high and low order bytes of a 64-bit address, the supplied program could be simplified somewhat. The outer **struct** could be removed and the inner **struct** returned to being anonymous. There would then be no need for the **#defines**.

From these observations it can be seen that the student's code is not without its problems. Perhaps there is another method that can achieve the same result in a simpler way. It is always worth reflecting on the code one writes to see if there is a more elegant solution. I now describe one such method that is worth considering.

The method consists of two stages. Firstly, the high order bytes of the variables address are set to zero and the result stored for later display. Secondly, the high order bytes are shifted into the position occupied by the low order bytes. The result is also stored for later display. From a practical point of view, there is one slight complication. The address of the variable **var** has to be cast to an integer type before any zeroing or shifting can take place. This is no great hardship and has the advantage of making it clear to anyone reading the code that the address of **var** is being handled in somewhat unconventional way. Code using this method is shown below.

```
const auto address =
  reinterpret_cast<int64_t>(&var);
const auto offsetLo = address & 0xffffffff;
const auto offsetHi = address >> 32;
std::cout << "Address = " << std::hex <<
  offsetHi << "/" <<offsetLo << std::endl;
```

To conclude, it is preferable to write simple code that does not rely on features of the implementation. If this is not possible, include checks to ensure such features are consistent with the operation of the program. Ideally, such checks should take place at compile-time. If this is inconvenient or impossible, run-time checks should be performed.

Commentary

This problem revolves around trying to treat a pointer value as an integral value. This is inherently non-portable as C++ makes few guarantees about the sizes of fundamental types and the relationship between pointer values and integral ones. While this enables C++ to be efficient on a wide range of platforms with different address ranges and register sizes (since the compiler can use a 'natural' word size) it does make it harder to write code targeting specific platforms.

The types provided by the C header `<stdint.h>` (and available to C++ using the **<cstdint>** header) provide a way to access various sized integer types *if* such types are available on the target implementation. If the **intN_t** type is available it 'designates a signed integer type with width *N*, no padding bits, and a two's complement representation'.

I do not think however that a *signed* type is a good choice for breaking an address into two parts – it would be better to use the unsigned type **uint32_t**. The use of signed integers to represent address values is a common source of errors for 32 bit programs, typically when addresses are compared. It seems to be less of a problem for 64 bit programs since the top bit of the 64bit address is clear (at least for user-mode programs) on both Windows and Linux, so treating the address as a signed value does not change the meaning of any pointer comparisons. If provided, the type **uintptr_t** is a typedef for an integral type large enough to round-trip any pointer to void. (There is a signed equivalent too, but I personally don't recommend using that for the reasons above.)

The original code that used an anonymous structure and an anonymous union gets a warning when used in C++ programs since although anonymous structs are allowed by the ISO C standard (since C11) they are not part of ISO C++. (MSVC gives a warning in **C** mode that a non-standard extension is being used – but this refers to the superseded C standard.) I apologise for the lack of clarity in the code critique about the exact form of the code using the anonymous struct.

The technique of breaking a pointer into two parts by writing into one element of a union and reading back from another one normally works in practice, but care must be taken to avoid subtle bugs.

As James points out, the student's example assumes a little-endian machine so that the layout of the 32 bit integers results in the low 32 bits of the pointer value overlaying **offsetLo** and the high 32 bits overlaying **offsetHi**.

It would be well worth while adding some checks (some are possible at compile time using **static_assert**) to both verify that these assumptions hold now and to avoid unpleasant surprises when the code is recompiled with a different compiler or for a different target platform. It is unfortunately all too easy when writing these checks to accidentally invoke undefined behaviour.

The use of a **#define** to simulate the anonymous struct is worrying mostly because of the normal problems of preprocessor definitions – they are a textual substitution done without full integration with the syntax of the language.

In this example, suppose another piece of code uses the symbol **offsetLo** and includes the `address.h` header file?

Adding the following variable declaration to **main**:

```
int offsetLo = 0;
```

caused an error message, from one compiler, stating: **"'->': trailing return type is not allowed after a non-function declarator"**. It is not entirely clear to me why! gcc was more helpful and provided a note referring to the macro definition of **offsetLo**, but this 'poisoning' of the identifiers **offsetLo** and **offsetHi** is still a nuisance. One possible approach might be to provide inline member functions **offsetLo()** and **offsetHi()** and encapsulate the union 'hack' inside the implementation of the 'address' structure.

One last minor usability problem with the header file is that it uses data types defined in **<cstdint>** but relies on this header being *already* included. This makes code using the header fragile as an unrelated removal of a header file can cause compilation errors in this one.

The Winner of CC 94

There was only one entrant this time, but James covered most of the problems with the code and provided a helpful diagram of the difference

Inspirational (P)articles: Use the DOM Inspector

Silas S. Brown shares a tip for debugging web pages.

This may seem obvious to some, but not every developer knows about it so it might be worth a mention. A while ago I was trying to help a Web designer who was struggling with a complex layout problem, and I complained that her laptop's touchpad was difficult to use. She went off to get a real mouse, but as she walked in with the mouse I said, "Got it: it's a spurious right-margin on line 66 of main.css". But how did you...?

All of the major graphical browsers have some variation on the DOM (Document Object Model) inspector, usually accessed from the context menu (the right-click menu in the page). These allow you to inspect the document, not just by looking at the source code of the HTML, CSS and Javascript files, nor even a pretty-printed, formatted version of these: the usefulness of that is limited by the fact that modern websites can be changed on-the-fly by scripts, so the static source code before the scripts run might be a far cry from what actually ends up on the screen. Rather, the DOM inspector lets you inspect the current state of the document, after scripts have run (or while they are still running), and often helps to locate which stylesheet rules are doing what, and so on. In short, if you're doing anything that involves a complex Web front-end then you might want to be aware of this facility.

Have you experienced something which has changed your perspective, had a positive effect on you, or just given you a buzz? Let us know at cvu@accu.org.

SILAS S. BROWN

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

Code Critique Competition 95 (continued)

Listing 2

```
#include <stdio.h>
#define ARRAY_SZ(x) sizeof(x)/sizeof(x[0])
typedef struct _Score
{
  char *name;
  int score;
} Score;

void to_string(Score *scores, size_t n,
  char *buffer, size_t len)
{
  for (size_t i = 0; i < n; i++)
  {
    size_t printed = snprintf(buffer, len,
      "%s:\t%u\n",
      scores[i].name, scores[i].score);
    buffer += printed;
    len -= printed;
  }
}

void process(char buffer[])
{
  Score sc[] = {
    { "Roger", 10 },
    { "Bill", 5 },
    { "Wilbur", 12 },
  };
  to_string(sc, ARRAY_SZ(sc),
    buffer, ARRAY_SZ(buffer));
}

int main()
{
  char buffer[100];
  process(buffer);
  printf(buffer);
}
```

between little and big endian layout so I think he deserves the prize despite the lack of competition! Perhaps the next critique will attract a little more interest?

Code Critique 95

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

I have some C code that tries to build up a character array using `printf` calls but I'm not getting the output I expect. I've extracted a simpler program from my real program to show the problem.

With one compiler I get: `"Rog"` and with another I get `"lburp@"`.

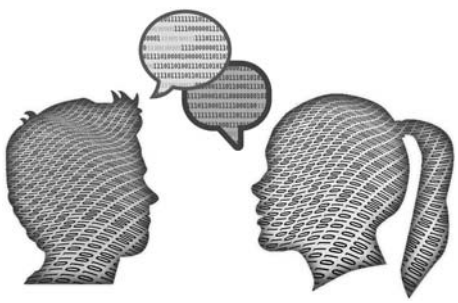
I'm expecting to see:

```
"Roger:  10
Bill:   5
Wilbur: 12"
```

What have I done wrong?

The code is in Listing 2.

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://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.



What do people do all day?

Christopher Gilbert shares his routine in a software house.

I work as a Senior Software Engineer for DataSift, the world's leading social data provider, inventor of the Twitter 'retweet' button, and fastest growing SaaS start-up in Europe. In my previous life I worked as a software engineer in the video games industry, working for award winning independent studios across the UK. These days I spend most of my time working on real problems in imaginary fields of buzzwords.

I recently finished working on one of the most exciting projects of my career, a project known as Facebook Topic Data. We (DataSift) are the first company to partner with Facebook to deliver real-time insights into brands, topics and audiences for Facebook customers. I am currently working on a top secret project, the details of which will be revealed in the not-too-distant future.

The team

DataSift currently employs over 140 people, distributed across San Francisco, New York, Reading and London. When I joined, the core engineering team consisted of just 10 developers, but has since swelled to over 50. Many of the developers are begrizzled veterans, with hard experience working at the coalface in related fields; ex-contractors and others of that ilk who are attracted to the particular brand of autonomy a startup company breeds.

The main office in Reading is single floor, mostly open plan, with a few meetings rooms which we have been spilling into as the team grows. There is a kitchen where lunch is served daily, and is otherwise fully stocked with free drinks and snacks, a chillout area with arcade machines, a table-tennis table and a pool table, and a quiet room for interruption free working, which makes for a pleasant retreat from the numerous – and increasingly elaborate – nerf wars. Lego and office toys are littered everywhere, sometimes bought from a store, sometimes homespun out of odd parts and a Raspberry Pi or Arduino by one of our hardware hackers.

Amongst other perks already mentioned, developers get to choose their own kit, attend conferences of their choosing – which I have been exploiting to attend ACCU, the best conference of the year – and work on their own projects in company sponsored hackathons and Innovation time (aka 20% time). I recently used my 20% time to research and develop a slew of algorithms that would accelerate our bespoke filtering and classification system, though other proposals have ranged from innovative new product features all the way through to obscure programming language concepts.

The software

The main product consists of around 400 or so components written in one of several languages including C++11, Python, Ruby, Node, PHP, Java, Scala and Go. Because of the range of languages in use, polyglot developers are highly sought after by our hiring team, though seemingly increasingly rare jewels to find. The freedom of language choice can be quite liberating, allowing us to draw upon the relative strengths of each language to solve any given use-case.

Our infrastructure consists of around 300 or so commodity servers running CentOS Linux and managed using Chef and colocated at our datacentre. Internally we have a self-serve provisioning system based on OpenStack and a cluster of dedicated servers, plus dedicated staging hardware. Metrics are provided via Graphite and Riemann, log aggregation via Logstash and Kibana. A custom written intranet portal provides a full range of dashboards for convenience.

Our main product can be thought of as a distributed information retrieval system built using a microservice style architecture, before microservice was a buzzword. In many ways the DataSift platform is like several products in one: generic data ingestion and normalisation, filtering and classification, generic data storage and delivery, historic ('big data') data access, aggregation and reporting, and more coming soon!

I primarily work on the real-time pipeline, which is mostly all C++11 code written within the last 5 years, and makes extended use of the Boost library. An in-house compiler and virtual machine runs filters that customers write in a DSL called CSDL, which is a bespoke filtering and classification language intended for our customers, and is also used internally by our data science team. Besides Boost we make use of third party open source software including Riemann, Kafka, Redis, MySQL, Zookeeper, ZeroMQ, memcached and many more. We package and maintain over 40 open-source projects besides the default packages provided by the OS, several of which were written in-house by our developers. Performance is always a primary concern, as C++ is chosen specifically as the implementation language for high-performance components.

The kit

Most Sifters are self-confessed hackers, so in DataSift, 'Windows' is a dirty word. Almost everyone works on custom desktops with an additional laptop running Ubuntu Linux or some other flavour of Linux and a minimalist tiling window manager like i3 or xmonad with emacs or Vim. Some developers prefer to develop on Mac, so MacBook Pro's and 4k monitors adorn the desks of the hipsters (myself included).

Development tools are not mandated, so there's complete freedom to choose what you want to work with. Some prefer Vim, others Sublime Text, some even use JetBrains IDE's. We license the best-in-class tools, and for everything else we draw upon FOSS to fill our needs. Although there are quite a few tools that just don't work well (if at all) under Linux, it still amazes me how large the open source community is, and the quality of the software available.

Every room has a monitor kitted with a Chromecast and Apple TV, and each meeting room is equipped with video conferencing gear. Many developers opted for standing desks that are height adjustable, and there are bean bags and comfy looking sofas around the place for anyone who wants to use them.

The process

We work in an agile way, but in the true spirit of agile we have customised our tools to fit the team, rather than fitting the team to the tools. It can take an enormous amount of effort to write custom tools, but in the case of ticket tracking we chose to instead spend a (relatively enormous) amount of money on Jira, which can be customised in every way imaginable. Tools are usually adopted by team consensus rather than mandate by management.

We implemented a continuous delivery pipeline using the recentlyopen-sourced Thoughtworks go.cd, and an elaborate system of levers and pulleys to automate testing and packaging of every component. The

CHRISTOPHER GILBERT

Chris is a C++ specialist, agile advocate, and open sourcerer who drinks from the firehose, wallows in data pools, and surfs waves of buzzwords. Follow him at @bigdatadev.

View from the Chair

Alan Lenton
chair@accu.org

Well it's mid-August as I write this, and I guess that, as is traditional, the summer rains are due to end immediately after the bank holiday weekend. The next Committee meeting is due in September – just in time for the autumn rains to have set in...

The traditional summer dead-news period has, at least on a tech level, been somewhat enlivened this year by the arrival of Windows 10. A number of people have asked me why I'm so ambivalent about it, given that I've been using and programming Windows since Windows 3.1 (the first usable version). There isn't the space to go into that issue in depth here, but here is the crux of my worries about the direction of Windows 10:

There are a number of issues, but for me the killer issue is the compulsory updates issue. Why? Obviously I suspect that sooner or later Microsoft will brick my computer, and everyone has concentrated on that in the tech press. But for me that is only a side issue, though an important one. It is the somewhat longer term implications of compulsory updates that worry me much more. You need to understand that although you may be able to accept, reluctantly perhaps, Windows 10 as it stands, you will have absolutely no choice about what it looks like, and what it does, in the future.

Even now, the more perceptive of the tech press are murmuring about this being 'the last' release of Windows. They are probably correct, but not one of them has understood the implications. Microsoft can use compulsory update for whatever it wants and you have no choice about. How about adverts for instance? Imagine having adverts displayed by the operating system? You think Ad-Blocker will be able to handle that? Impossible, do I hear you say? Sorry, but Microsoft has already done it – a few months back they used the older version of update to put an advert for Windows 10 into everyone's system tray!

Obviously, I could go on at length, but I won't. Do I think Microsoft are malicious? No, quite to the contrary. They are a business whose business model has been seriously undermined by technical innovations and societal developments outside the scope of their model. They are trying to develop a new business model which will restore their fortunes, and I would point out that they are doing so without resorting to the morally bankrupt methods of the big media companies.

Good luck to them, I say. I don't have to like it, and if they succeed I will probably take my day-to-day usage (i.e. my custom) over to Linux or BSD with an Xfce desktop. But that's my option as a customer, and if enough people do it, they'll probably have to re-think things.

And as a last thought on the subject, I'll just point out that if compulsory updates had been in Windows 7, we would *all* have ended up running Windows 8...

Moving on to a happier topic – ACCU Conference 2016 – I'd like to remind people that sometime between this issue of *CVu* being published, and the next one in November, the call for conference papers usually comes out. So, I thought I'd remind everyone to start thinking now about presenting at the conference. If you've never done it before, now is always a good time to start. Take it from me – even if you thought you knew the topic you wanted to present, building up the presentation will improve your understanding of it. And, if you want to be purely selfish about it, it definitely looks good on your CV!

If you've not done it before, help is available, and the local groups are always open to people trying out their putative conference presentations at a local meetings. Go on, give it a try.

I mentioned in the last edition the question of what we should set as the quorum for the on-line voting we now have. If anyone reading this knows of other organizations that have adopted this and what their decisions were, perhaps you could drop me a line with the details. For the record I'm moving towards the idea that the quorum for online should be twice that of the quorum for physical events, but I'm prepared to be persuaded otherwise, if the reasons are good.

I think that's about all for now, so, I'll sign off while keeping my fingers crossed that there won't be another power cut at the next committee meeting. There shouldn't be; the power people had a large hole dug in the middle of the street for several days after the last one!

Have fun programming.

Alan

What do people do all day? (continued)

cluster, which then use Docker to run integration tests using an open-source testing framework developed in-house.

We use git for source control. We used git flow for a time, but as we transitioned to ever more sophisticated automation it became increasingly difficult to handle multiple release branches, so we have been moving towards a simplified workflow. For practical purposes we still make use of feature branches, but we are tending towards a branch by ticket model, which allows us to track work all the way from jira through the entire delivery pipeline.

The future

Like any team we have challenges to overcome. Perhaps counter-intuitively, it's much harder to scale a team quickly than it is to scale software, and we're still learning the most effective ways of doing that. Other trends that seem to be emerging right now are increasing sophistication in how we trap and respond to errors, improving our agile process with new techniques, growing our team with other like-minded s/crazies/developers/, adding even more languages to our bulging repertoire (my money is on Rust being next), and building increasingly ambitious features to augment our core product offering.

However, if working for a start-up has taught me one thing, it's that what happens next is impossible to predict, but that's what excites me about this work. Working for a start-up is often chaotic, and sometimes a bit crazy, but always challenging and ultimately deeply rewarding. You are forced to innovate or die trying, and if you lose momentum you will be left in the ruins of wasted effort. For some this environment would be a terrible nightmare, but for those who thrive in the chaos, and maybe have just that tiny bit of crazy in them, it's the best job in the world.

If you'd like to find out more about DataSift please visit our website at http://www.datasift.com/. Check out my open source contributions on Github at http://www.github.com/bigdatadev or follow me on Twitter @bigdatadev