the magazine of the accu

www.accu.org

{cvu}

Volume 27 • Issue 2 • May 2015 • £3

{cvu}

Volume 27 Issue 2
May 2015
ISSN 1354-3164
www.accu.org

Editor
Steve Love
cvu@accu.org

Contributors
Tom Björkholm, Pete Goodliffe, Vassili Kaplan, Ralph McArdell, Roger Orr, Mark Radford, Adam Tornhill

ACCU Chair
chair@accu.org

ACCU Secretary
secretary@accu.org

ACCU Membership
Matthew Jones
accumembership@accu.org

ACCU Treasurer
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

In Between

The congruence of a variety of events means that I am writing this whilst attending the ACCU 2015 conference. This is simultaneously a good thing and a bad thing: I am surrounded by interesting people, and immersed in an atmosphere that is intensely geeky (and not *just* about programming, either!). There are almost 400 people here in Bristol, representing 24 countries from Sweden to Brazil and Russia to Saudi Arabia. This is my 14th conference, and I keep coming because I know the talks will be entertaining and educational, the people will be fascinating, and I will learn plenty of new things *every single day*. And therein lies the problem...there is so much to do, see and learn, there's precious little time in which to concentrate on writing the 300 or so words for the magazine!

The truth is that the conference schedule is punishing. It's rare to find a session slot having only one talk I wish to attend, but the difficulty of choosing doesn't end there. There are always interesting conversations happening in the corridors, bar corners, over breakfast and lunch, excursions in the evenings to find dinner and perhaps a drink or two (ahem!) and then even after all that, the hotel bar is frequently teeming with delegates until the early hours. It's not even limited to actual conversation, either. Twitter and other similar virtual cafes are alive with chatter and anecdotes – which sometimes mean you can get a gist of some of the things you've missed, and sometimes mean you feel you've missed out on something important!

After all that, it's still necessary to catch at least a few hours' sleep 'in between'. By the end of the week, I know I shall be a near-wreck: the effects of a head full of exciting ideas, inspired by whole talks or just snippets of conversation, combined with too much alcohol and insufficient sleep will leave me exhausted. But happy. Because whilst it's inevitable that I missed some things (you can't do **everything**!), I will have done some of it – and profited hugely by it. If you were at the conference, I hope you enjoyed it at least as much as I always do. If you've never been, I cannot recommend it enough.

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 27.3	1st June 2015
C Vu 27.4:	1st August 2015
Overload 128:	1st July 2015
Overload 129:	1st September 2015

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

Wallowing in Filth

Pete Goodliffe wades into the dreaded cesspit of ‘low-quality code’.

As a dog returns to its vomit, so fools repeat their folly.
Psalms 26:11

We've all encountered it: *quicksand code*. You wade into it unawares, and pretty soon you get that sinking feeling. The code is dense, not malleable, and resists any effort made to move it. The more effort you put in, the deeper you get sucked in. It’s the man-trap of the digital age.

How does the effective programmer approach code that is, to be polite, *not so great*? What are our strategies for *coping with crap*?

Don’t panic, don your sand-proof trousers, and we’ll wade in...

Smell the signs

Some code is great, like fine art, or well-crafted poetry. It has discernible structure, recognisable cadences, well-paced meter, and a coherence and beauty that make it enjoyable to read and a pleasure to work with.

But, sadly, that is not always the case.

Some code is messy and unstructured: a slalom of `goto`s that hide any semblance of algorithm. Some is hard to read: with poor layout and shabby naming. Some code is cursed with an unnecessarily rigid structure: nasty coupling and poor cohesion. Some code has poor factoring: entwining UI code with low-level logic. Some code is riddled with duplication: making the project larger and more complex than it need be, whilst harbouring the exact same bug many times over. Some code commits ‘OO abuse’: inheriting, for all the wrong reasons, tightly associating parts of code that have no real need to be bound. Some code sits like a pernicious cuckoo in the nest: C# written in the style of JavaScript.

Some code has even more insidious badness: brittle behaviour where a change in one place causes a seemingly unconnected module to fail – the very definition of *code chaos theory*. Some code suffers from poor threading behaviour: employing inappropriate thread primitives or exercising a total lack of understanding of the safe concurrent use of resources. This problem can be very hard to spot, reproduce, and diagnose, as it manifests so intermittently.

(I know I shouldn’t moan, but sometimes I swear that programmers shouldn’t be allowed to type the word *thread* without first obtaining a licence to wield such a dangerous weapon.)

Be prepared to encounter bad code. Fill your toolbox with sharp tools to deal with it.

To work effectively with alien code, you need to able to quickly spot these kinds of problems, and understand how to respond.

Wading into the cesspit

The first step is to take a realistic survey of the coding crime scene. You arrive at the shores of new code. What *are* you wading into?

The code may have been given to you with a pre-attached stigma. No one wants to touch it because they know it’s foul. Some quicksand code you discover yourself when you feel yourself sinking.

It’s all too easy to pick up new code and dismiss it because it’s not written in the style you’d prefer. Is it really dire work? Is it truly *quicksand code*, or is it merely unfamiliar? Don’t make snap judgments about the code, or the authors who produced it, until you’ve spent some time investigating.

Take care not to make this personal.

Understand that few people set out to write shoddy code. Some filthy code was simply written by a less capable programmer. Or by a capable programmer on a bad day. Once you learn a new technique or pick up a team’s preferred idiom, code that seemed perfectly fine a month ago is an embarrassing mess now and requires refactoring.

You can’t expect any code, even your own, to be perfect.

Silence the feeling of revulsion when you encounter ‘bad’ code. Instead, look for ways to practically improve it.

The survey says...

There are many techniques you might employ to navigate a path into an unfamiliar codebase. Consuming the documentation (or noting the lack thereof), understanding the build/test/deploy process, spelunking the version control information, and code visualisation tools are all useful headstarts. But the only true way to start learning a piece of code is to start physically *working* with it.

As you do so, you build a mental model of the new piece of code. And then you can truly begin to gauge its quality using benchmarks like:

- Are the external APIs clean and sensible?
- Are the types used well chosen, and well named?
- Is the code layout neat and consistent? (Whilst this is certainly not a guarantee of underlying code quality, I do find that inconsistent, messy code tends also to be poorly structured and hard to work with. Programmers who aim for high-quality, malleable code also tend to care about clean, clear presentation. But don’t base your judgment on presentation alone.)
- Is the structure of cooperating objects simple and clear to see? Or does control flow unpredictably around the codebase?
- Can you easily determine where to find the code that produces a certain effect?

It can be hard to perform this initial survey. Maybe you don’t know the technology involved, or the problem domain. You may not be familiar with coding style.

Consider employing *software archaeology* in your survey: mine your revision control system logs for hints about the quality. Determine: how old is this code? How old is it in relation to the entire project? How many people have worked on it over time? When was it last changed? Are any recent contributors still working on the project? Can you ask them for information about the code? How many bugs have been found and fixed in this area? Many bugfixes centred here indicates that the code is poor.

Working in the sandpit

You’ve identified quicksand code, and you are now on the alert. You need a sound strategy to work with it.

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn’t wear shoes. Pete can be contacted at pete@goodliffe.net or @petegoodliffe

What is the appropriate plan of attack?

- Should you repair the bad code?
- Should you perform the minimal adjustment necessary to solve your current problem, and then run away?
- Should you cut out the necrotic code and replace it with new, better work?

Gaze into your crystal ball. Often the right answer will be informed by your future plans. How long will you be working with this section of code? Knowing that you will be pitching camp and working here for a while influences the amount of investment you'll put in. Don't attempt a sweeping rewrite if you haven't the time.

Also, consider how frequently this code has been modified up to now. Financial advisors will tell you that *past performance is not an indicator of future results*. But often it is. Invest your time wisely. This code might be unpleasant, but if it has been working adequately for years without tinkering, it is probably inappropriate to 'tidy it up' now, especially if you're unlikely to need to make many more changes in the future.

Pick your battles. Consider carefully whether you should invest time and effort in 'tidying up' bad code. It may be pragmatic to leave it alone right now.

If you determine that it is not appropriate to embark on a massive code rework right now, that doesn't mean you are necessarily left to drift in a sea of sewage. You can wrestle back some control of the code by cleaning progressively.

Cleaning up messes

Whether you're digging in for the long haul, or just making a simple fix-and-run, heed Robert Martin's advice and follow 'the Boy Scout Rule': *Always leave the campground cleaner than you found it.* It might not be appropriate to make a sweeping improvement today, but that doesn't mean you can't make the world a slightly less awful place.

Follow the Boy Scout Rule. Whenever you touch some code leave it *better* than you found it.

This can be a simple change: address inconstant layout, correct a misleading variable name, simplify a complex conditional clause, or split a long method into smaller, well-named sub-functions.

If you regularly visit a section of code, and each time leave it slightly better than it was, then before long you'll wind up with something that might be classified as *good*.

Making adjustments

The single most important advice when working with messy code is this:

Make code changes slowly, and carefully. Make one change at a time.

This is so important that I'd like you to stop, go back, and read it again.

There are many practical ways to follow this advice. Specifically:

- Do not change code layout whilst adjusting functionality. Tidy up the layout, if you must. Then commit your code. Only *then* make functional changes. (However, it's preferable to preserve the existing layout unless it's so bad that it gets in the way.)
- Do everything you can to ensure that your 'tidying' preserves existing behaviour. Use trusted automated tools, or (if they are not available) review and inspect your changes carefully; get extra sets of eyeballs on it. This is the prime directive of *refactoring*: the well-known set of techniques for improving code structure.

  This goal can only be reached effectively if the code is wrapped in a sound set of unit tests. It is likely that messy code will not have any tests in place, so consider whether you should first write some tests to capture important code behaviour.
- Adjust the APIs that wrap the code without directly modifying the internal logic. Correct naming, parameter types, and ordering; generally introduce consistency. Perhaps introduce a new outer interface – the interface you wish that code had. Implement it in terms of the existing API. Then at a later date you can rework the code behind that interface.

Have courage in your ability to change the code. You have a safety net: source control. If you make a mistake, you can always go back in time and try again. It's probably not wasted effort, as you will have learnt about the code and its adaptability in doing so.

Sometimes it is worth boldly ripping out code in order to replace it. Badly maintained code that has seen no tidying or refactoring can be too painful and hard to correct piecemeal. There is an inherent danger in replacing code wholesale, though: the unreadable mess of special cases *might* be like that for a reason. Each bodge and code hack encodes an important piece of functionality that has been uncovered through bitter experience. Ignore these subtle behaviours at your peril.

An excellent book that deals with making appropriate changes in quicksand code is Michael Feathers' *Working Effectively with Legacy Code*. [1] In it, he describes sound techniques to introduce seams into the code – places where you can introduce test points and most safely introduce sanity.

The Curious Case of the Container Code

There was a container class. It was central to our project. Internally, it was foul. The API stank, too. The original coder had worked hard to wreak code mischief. The bugs in it were hidden by the already confusing behaviour. Indeed, the confusing behaviour was a bug itself.

One of our programmers, a highly skilled developer, tried to refactor and repair this container. He kept the external interface intact, and improved many internal qualities: the correctness of the methods, the buggy object lifetime behaviour, performance, and code elegance.

He took out nasty, ugly, simplistic, stupid code and replaced it with the polar opposite. But in his effort to maintain the old API, this new version was internally far too contrived, more like a science project than useful code. It was hard to work with. Although it succinctly expressed the old (bizarre) behaviour, there was no room for extension.

We struggled to work with this new version, too. It had been a wasted effort.

Later on, another developer simplified the way we used the container: removing the weirder requirements, therefore simplifying the API. This was a relatively simple adjustment to the project. Inside the container, we removed swaths of code. The class was simpler, smaller, and easier to verify.

Sometimes you have to think laterally to see the right improvement.

Bad code? Bad programmers?

Yes, it's frustrating to be slowed down by bad code. The effective programmer does not only deal well with the bad code, but also with the people that wrote it. It is not helpful to apportion blame for code problems. People don't tend to purposefully write drivel.

There is no need to apportion blame for 'bad' code.

Perhaps the original author didn't understand the utility of code refactoring, or see a way to express the logic neatly. It's just as likely there

Writing Good C++ APIs

Tom Björkholm examines some common pitfalls that make code hard to use.

In my view (although every programmer may have a different view on this subject) a good C++ API is an API that guides the user into writing good application code. In particular, it should be:

- easy to understand and use correctly,
- impossible, or at least hard, to misunderstand,
- impossible, or at least hard, to use incorrectly,
- const correct,
- context neutral.

Over the years a fair number of programmers have appreciated my APIs, so I guess there is some merit to my view of what constitutes a good API.

One question I have asked myself is if the Standard C++ library serves as a good example for API design. Admittedly the Standard C++ library APIs are clever and offer a very effective programming interface. However, it is often designed in a way that is possible to misunderstand. That is OK for the Standard C++ library as there are lots of books out there to explain the correct usage of the API. Also, as it is the Standard C++ library every serious programmer will spend time learning the correct way to use it. That is a luxury that we mortal API developers do not have. Nobody is going to write a book about how to use our APIs, and even if someone wrote such a book, no programmer would find time to read it. We need to define our APIs with more focus on ease of use and making it 'impossible' to use incorrectly.

Naturally, the Standard C++ library has introduced a number of concepts, that are now familiar to programmers. Concepts like iterators going from `begin()` to `end()`, where `end()` is past the last valid element, are familiar concepts. Building our APIs on familiar concepts like that makes it easier for programmers to learn our APIs.

Stating what constitutes a good API like this is the easy part. Writing code to create a good API is a lot harder. Naturally, we can keep the view of what constitutes a good API in our mind when creating APIs, but this is still kind of abstract. For this article I will take another route and have a look at some common pitfalls, and try to suggest ways to improve the APIs.

SQL API example and RAII

As the first example we will take a look at a piece of simple code that uses the Oracle C++ database access library OCCI. Such code might look like Listing 1.

I guess that you immediately spotted the problems with this code: If `doSomeThing()` throws an exception, this code will leak a `ResultSet`, a `Statement`, a `Connection` and an `Environment`. The functions `executeQuery()`, `createStatement()` and `createConnection()` calls may also throw exceptions leading to resource leaks. You could argue that the programmer who wrote this code was criminally incompetent to ignore RAII (Resource Acquisition Is Initialization), but I claim that it is the design of the API that has tricked the programmer into ignoring RAII. The code is written in the style that naturally fits the design of the API. Unfortunately many other database APIs are just as bad as the Oracle OCCI API. I ended up writing a wrapper library for OCCI just to make it 'RAII compatible'. Using this wrapper library the code for the same operation would look like Listing 2.

TOM BJÖRKHOLM

Tom Björkholm has been using C++ professionally since 1994, programming control systems for industrial robots and telecom systems. Currently the most familiar compilers and operating systems are gcc 4.9.2 on Linux and Solaris, and clang on Mac OS X.

Wallowing in Filth (continued)

are other similar things you do not yet understand. Perhaps they felt under pressure to work fast and had to cut corners (believing the lie that it helps you get there faster; it rarely does).

But of course, you know better.

If you can, *enjoy* the chance to tidy. It can be very rewarding to bring structure and sanity to a mess. Rather than see it as a tedious exercise, look at it as a chance to introduce higher quality.

Treat it as a lesson. Learn. How will you avoid repeating these same coding mistakes yourself?

Check your attitude as you make improvements. You might think that you know better than the original author. But do you always know better?

I've seen this story play out many times: a junior programmer 'fixed' a more experienced programmer's work, with the commit message 'refactored the code to look neater'. The code indeed looked neater. But he had removed important functionality. The original author later reverted the change with the commit message: 'refactored code back to working'.

Conclusion

You can do nothing but *expect* to be presented with 'bad' code at some point. You have to learn good, mature, ways to deal with it. That is the lot of the professional programmer.

Ultimately, we must heed the valuable advice mentioned earlier: *employ the Boy Scout Rule*. Leave every piece of code you touch better, if even only fractionally. ■

Questions

- Why does code frequently get so messy?
- How can we prevent this from happening in the first place? Can we?
- What are the advantages of making layout changes separately from code changes?
- How many times have you been confronted with distasteful code? How often was this code *really* dire, rather than 'not to your taste'?

References

[1] Michael Feathers, *Working Effectively with Legacy Code*, Upper Saddle River, NJ: Prentice Hall, 2004

Listing 1

```
void occiUse(const LoginInfo & loginInfo)
{
  oracle::occi::Environment * envP =
    ::oracle::occi::Environment
    ::createEnvironment();
  oracle::occi::Connection * conP =
    envP->createConnection(loginInfo.username,
    loginInfo.password(), loginInfo.alias);
  const std::string query =
    "select count(*) from user_tables";
  oracle::occi::Statement * stmtP =
    conP->createStatement(query);
  oracle::occi::ResultSet * resP =
    stmtP->executeQuery();
  if (resP->next()) {
    doSomeThing("occiUse", resP->getInt(1)); }
  stmtP->closeResultSet(resP);
  conP->terminateStatement(stmtP);
  envP->terminateConnection(conP);
::oracle::occi::Environment
  ::terminateEnvironment(envP);
}
```

Please ignore **`log`** for a moment, I will get back to it. Here you can see that the API clearly guides the application programmer into writing code that uses RAII. **`Connection`**, **`Statement`** and **`ResultSet`** are all created using the constructor and destroyed using the destructor. (If you wonder what happened to the **`Environment`**, it has now become a member of the **`Connection`**.)

This API is still not perfect. I am not happy with the fact that the newly created **`ResultSet`** is positioned before the first row in the result. This seems a bit odd in C++. On the other hand this is the way most SQL APIs distinguish between an empty **`ResultSet`** and one with rows in it, so if the application programmer is used to other SQL APIs this seems natural. I am playing with the idea of modelling the result set as an input iterator, but that idea has not yet moulded into a final design. Also I am not happy with the way **`getSomething(1)`** is used to get the first column. C++ uses the convention to let index 0 denote the first element, whereas SQL uses index 1 for the first element. Either scheme of indexing is likely to fool some users, because we are mixing C++ and SQL. I am playing with the idea of using the input **`operator >>`** to get the next column of the **`ResultSet`**, which would circumvent any confusion about indexing altogether.

Context neutral

A good API should be context neutral. We want to be able to use the same API in many different contexts. The context dependent parts should be factored out. I will use the previous database API example to illustrate this.

If we write an API for database access we should not limit its usage to only a specific context, like a daemon (server/service) program, a command line utility, a graphical user interface application, or a library that will be wrapped in JNI, or...whatever. We want our API to be usable in all foreseeable contexts. For some of these contexts we might want to do some things in different ways.

One thing that is present in almost all bigger software products is logging. This is sometimes called trace logging, sometimes debug logging, and on some UNIX daemons it might be referred to as 'syslogging'. The idea is to output information about the internal state and the decisions taken in the program flow, to enable the support technician to understand what went wrong when the customer complaints arrive. It is easy to see the generalized functionality: the programmer prints information and when printing it mentions what type of information it is. The log class (or framework) will then look at runtime configuration to determine if the log printout shall be carried out or if the output shall be suppressed. Logging an error shall not be confused with handling the error. The logging is only done to help technicians locate the fault.

Listing 2

```
void myDbUse(const LoginInfo & loginInfo,
             std::shared_ptr<MyLib::LogBase> log)
{
  MyDb::Connection con(log, loginInfo);
  const std::string query =
    "select count(*) from user_tables";
  MyDb::Statement stmt(con, query);
  MyDb::ResultSet res(stmt);
  if (res.next()) { doSomeThing("myDbUse",
    res.getInt(1)); }
}
```

Logging is something that is typically handled differently in different contexts. For instance a database problem might cause an error message to be written to the standard error stream if it is a command line utility, but it should be logged to a log file when the API is used in a daemon and result in a status pane update if it is used in GUI. By defining a log base class that has pure virtual methods for logging, and by accepting a reference to such a log base class the API has been decoupled from the decision of where the log ends up. The user of the API might choose one of a number of predefined derived log classes, or the user of the API might derive her own log class. This is the log argument in the database API example. In that example there is an additional twist to it. The problem that should be reported might actually happen in the destructor. To have a derived log class object accessible in the destructor, it is actually passed in to the constructor as a **`std::shared_ptr`** and stored in the object for later use.

Finding context dependent parts to be factored out is not limited to logging. The challenge here is to imagine all the possible contexts where the API might be used, and to recognize and factor out the parts that are specific to some contexts. Still we need to keep the API small and tidy, so that it is easy to understand.

Ways to pass arguments and return values

Let's look at a function declaration as it might appear in some API.

```
XY_Receipt * XY_sendData(XY_Data * datap,
  ? XY_Flags * flagsp);
```

Although APIs like this are quite common, I think that it is a scary example of a bad API. The problem is that the usage remains unclear to the application programmer:

- A pointer to a receipt is returned. Shall the caller delete the receipt, or is it a pointer to some statically allocated data?
- Does this function modify the pointed to data and flags? (For a function called **`send`**, it appears strange to modify passed data and flags. However, passing the arguments as non-**`const`** pointers indicate that they will be modified – or is missing **`const`** just caused by sloppy programming?)
- Does the function take ownership of the pointed to data and flags?
- Must data and flags be allocated with **`new`** to allow the function to deallocate them with **`delete`**?

With a function declared like this, the user has to rely on documentation and comments to try to understand how to use it. Relying on something that is not in the code itself is not good. Too often the code deviates from the comments and documentation. They might match when the code is initially written, but then there will be a number of bug fixes and change requests in hectic 'survival of the company' projects and the code just starts to deviate from the comments.

It is much better if as much as possible can be communicated from the API programmer to the application programmer in the code itself. Let's fix the obvious flaws in the above function API. We will then get:

```
XY_Receipt XY_sendData(const XY_Data & data,
                       const XY_Flags & flags);?
```

This function declaration makes it much clearer how to use it:

- The receipt is returned by value. Now there is no need to worry about if or how it should be deallocated.
- The arguments are passed by `const` reference. As they are `const` it is clear that data and flags will not be modified by the function.
- As the arguments are passed by `const` reference instead of pointers, there is no longer any issue about ownership, allocation and deallocation of the data and flags.

I prefer to use the return values to pass values from a function to the caller. This is much cleaner and easier to communicate than to modify non-`const` reference arguments. In C++11 (and C++14) we have `move` constructors, and decent compilers use RVO (return value optimization). The result is that it is usually cheap to return even large values (like conglomerates of `std::map`, `std::vector` and `std::string`). In my view there is no longer any good case to use non-`const` reference (or pointer) arguments for passing out values from a function.

When it comes to arguments I favour `const` references. Admittedly some objects (like `int`s) might be faster to copy than to pass by `const` reference. Still I feel that consistent use of `const` reference communicates with the user of the API (the application programmer) in an easy to understand way. (Especially as the compiler does not differentiate between `const A a` and `A a` in function declarations. I found that many users of my APIs get confused if they see `func(const A a)` in the header file, but get a compiler error message about `func(A a)`.)

When modifying data we have the option of modifying the argument, or keeping the argument unchanged and returning the changed data. Let's take a function that capitalizes all words in a string as an example

```
void capitalizeAllWords(std::string & s);
std::string capitalizeAllWords
  (const std::string & s);
```

Here you can argue that there is a performance cost of using a return value instead of modifying the argument. Still in most cases I prefer the version with return value. The code using it becomes much easier to read. (And please, never provide both the above versions as overloads. That would definitely be confusing for the user.)

I guess the readers of this magazine already know everything about `const` correctness. I will not dwell on it now. If there is enough interest that might be the scope of another article.

Unintended API consequences

As my next example I will use a class that stores a parsed C++ source file. (It might be part of a C++ IDE or it might be part of a tool like Doxygen.) With some designers this class might have an interface that includes:

```
class CppFile {
public:
  // ...
  unsigned int & numberOfFunctions();
};
```

The interface actually allows the class to be used like this:

```
void g(CppFile & cpp) {
  cpp.numberOfFunctions() = 0;
  // remove all functions

  cpp.numberOfFunctions() = 5000;
  // create some other functions
  //...
}
```

Unless you consider this to be legitimate usage, the member function should not return a non-`const` reference. Here I cannot see how the class implementation could know what code to create as the source code of these created functions, so I think the interface should not return a non-`const` reference. Having access methods that return non-`const` references is useful for container classes where we have well-defined semantics for accessing and changing the contained elements. If the class is a wrapper around another class (like `MyDb::Connection wrapping oracle::occi::Connection`) there is also a legitimate case for having a member function that returns a non-`const` reference to the wrapped object. In most other cases I feel that returning a non-`const` reference just creates trouble.

Call-backs – when action is needed from the application

Sometimes an interface wants to allow the calling code to react to some events. I prefer to use call-backs for this. Call-backs are also useful if the interface implementation would like to let the calling code make decisions about how to handle some situations.

Call-backs can be of three different styles:

- C-style function pointers
- C++ standard library style callable objects (like the predicate in `std::find_if` or the callable object used in `std::for_each`)
- Abstract base classes with pure virtual member functions.

Often I prefer to use the abstract base class style call-backs. I find that these allow the API designer to impose structure and communicate to the application designer what events the application code is supposed to (or allowed to) handle. To make this more concrete and easier to understand, I will demonstrate it using a small example (see Listing 3) with a simplified SMS sending API. (SMS is another name the short text messages sent over the mobile telephony network. SMSC is the telephony network server you

Listing 3

```
class SMS_Data { /* ... */ };
class SMS_Address { /* ... */ };
class SMS_Channel {
public:
  class Cb;
  class SendCb;
  SMS_Channel() = delete;
  explicit SMS_Channel(Cb & cb);
  enum class SendError {
    OK,
    NoSmscConnection,
    sizeLimitExceeded
  };
  struct SendResult {
    unsigned long long id;
    SendError sendError;
  };
  SendResult send(const SMS_Data & data,
                  const SMS_Address & address,
                  std::shared_ptr<SendCb> cb);
  // ...
};
class SMS_Channel::Cb {
public:
  virtual void lostSmscConnection() = 0;
  virtual void dataReceived(
     const SMS_Data & data,
     const SMS_Address & to,
     const SMS_Address & from) = 0;
  // ...
};
class SMS_Channel::SendCb {
public:
  virtual void okPoR(?unsigned long long id,
     const SMS_Data & sentData,
     const SMS_Address & wasTo) = 0;
  virtual void errorPoR(?unsigned long long id,
     const SMS_Data & sentData,
     const SMS_Address & wasTo,
     const SMS_Data & responseData) = 0;
  // ...
};
```

Writing a Technical Book

Adam Tornhill discusses motivation, publishing and how to stay focused without ruining your life.

Do you dream of writing your own technical book? I hope you do – our programming profession needs more high-quality books. In our fast evolving field there's an endless amount of new topics to cover and timeless ideas to rediscover. This article is here to help you get started. I'll make sure to give you a high-level overview on everything from the initial planning to tips on different publishing models. You'll also learn about the motivational hacks I used to stay on track and make a steady progress on my own books.

Why care?

Well, there are certainly people with deeper insights than me. But what I can share is advice from a perspective that might be close to where you are. I'm not a professional author in the sense that I make a living from it. Instead I balance a full-time job, a family and something resembling a social life with my writing activities. That means I had to learn to use my time efficiently.

My writing career has also been non-traditional in the sense that I started-out with self-publishing. I wrote two books on my own before I decided to get a publisher. That means I can compare these competing models. As you'll see, there isn't a clear-cut between them; Both models have their advantages and drawbacks. I tell you more about it soon. I'll also share my experience of working with a publisher, what they can do for you and what you can expect when it comes to royalties.

All my advice is subjective. This is what worked for me. I view this article as the kind of guide I'd like to send back in time to my younger self to save both time and effort. Since I cannot do that (yet), I decided to share it with you. I hope you'll find it just as useful as my younger self would.

Start small

How often have you heard that writing a book is hard work? A work that will consume all your waking time, ruin your social life and leave you with less monetary return than what you'd earn from refunding deposit bottles. I won't dispute it – there's a lot of truth to it. That's why you need to approach your book like you would take on any other large project that affects your life.

ADAM TORNHILL

With degrees in engineering and psychology, Adam tries to unite these two worlds by making his technical solutions fit the human element. While he gets paid to code in C++, C#, Java and Python, he's more likely to hack Lisp or Erlang in his spare time.

Writing Good C++ APIs (continued)

connect to in order to send a message. PoR stands for 'Proof of Receipt' and is either a positive or negative acknowledgement to a sent message.)

The abstraction captured in this API is about sending data, receiving data and reacting to events.

When we have a connection to an SMSC we might at any time receive a message, or lose the connection. Things like these, which are not related to any specific message we send, are handled in the **`SMS_Channel::Cb`** call-back that is passed to the constructor. The **`SMS_Channel`** class might cooperate with some event-dispatcher framework, or it might have an internal thread that listens to input from the network. In either case the call-back is called when there is an event that the application code needs to handle. With these call-back base classes it is easy for the API designer to tell the application programmer what events the application needs to handle.

In a similar way the **`send`** member function takes a call-back object, that is called to handle events that are caused specifically by sending the message. This can be the reception of a positive or negative acknowledgement 'PoR'. As the application code has probably exited the local function scope where the **`send`** member function was called long before the call-back is called, the call-back is this time passed as a **`std::shared_ptr`**. Without a call-back class like this, it is quite hard to communicate to the application programmer what events the application code needs to handle.

The **`SMS_Channel::Cb`** is passed as non-**`const`** reference. This is not an obvious choice. The non-**`const`** reference prevents the user from creating a temporary object in the call to the **`SMS_Channel`** constructor (which can be seen as a very desirable thing to do). On the other hand, a **`const`** reference would require all the call-back member functions to be **`const`**, and thus prevent the call-back object from changing state. I made the judgement that the possibility to change state and keep state between call-back invocations is more valuable.

Allowed order to call function in an interface

If an interface has several functions, the application programmer should be allowed to call them in any order that is allowed by the syntax. Remember, it should be 'impossible' to use the interface incorrectly. This can be a real challenge for the API designer.

One way to solve this is to use the C++ syntax rules to limit the possible order that functions may be called in. If we look back at the database API example the **`Statement`** constructor needs a **`Connection`** as an argument. This is an effective way of specifying the order. It does also specify the order of destructions (and the API user will do it correctly without any focus on order requirements).

Conclusions

It is hard to write a good C++ API. There are many pitfalls. By keeping an open eye for the pitfalls and by thinking about the API from the viewpoint of how we would like to use it as application programmers, I think that we can improve our API designs. Whenever an application programmer asks questions about an API that I wrote, I think that it is important to not only answer the question, but also to analyse if the question is a sign that the API design should be improved. I hope that some of the readers have found some useful hints in this article, and that maybe some other readers have silently nodded and recognized pitfalls and good practices from their own experience. ■

Returning these bottles (Figure 1) earns me more money than an hour of writing does.

Figure 1

While I wanted to write a book for a long time, that's not how I started. Instead I tried to build my experience and develop my writing style in smaller steps. The first pieces I wrote were articles. At that time, back in 2004, I'd come back to code in C after some intense years in object-oriented land. That perspective made me approach the code in a different way. More specifically, I realized that some principles I've learned could benefit C programmers as well. I decided to do a write-up of my experience.

These articles grew into a series I called 'Patterns in C' [1]. I published it on my blog [2] and in *C Vu*. By starting small I got to develop my own voice. In fact, as I re-read the articles now, I note the gradual change in style over the course of the series.

I recommend a similar approach: get started by writing smaller pieces on a regular basis. You'll find that you learn a lot that translates to larger writing projects. It's also a way to save time once you embark on your full book. Writing is just like any other activity: the more you practice, the more fluent and easy it comes. This is particularly important if you, like me, aren't a native English speaker (I grew up bilingual with Swedish and Czech and had to practice a lot to get a decent command of the English language).

You'll notice that all the practice you put in lets you write faster. That's a good thing since it allows you to spend more time on background research, structure and re-drafting your writing. Remember, what makes good writing is constant re-writing. Often, writing the initial version of a chapter or article takes-up less than half of the total time spent on it. Practice gives you the margin you need.

Expand your horizon

Once you've created a collection of blog posts or articles you might want to take the next step. You're fortunate; today, it's easier than ever to grow your collection into a book.

My first book was more or less an experiment to gain experience with a larger format. I had read about Leanpub, the new self-publisher start-up, and wanted to give it a try. I decided to package my article series on 'Patterns in C' [1] into a book. I'm glad I did since it taught me a lot. It's also a motivational boost once the first royalties from your own writings start to come in. It probably won't be much money, but trust me on this one; You can't put a price on motivation. The knowledge that someone *pays* real money to read your work will help you keep going.

I'll discuss self-publishing in a minute. But first, let's consider the reasons why you should write a book.

Money and motivation

Ever heard that 95 percent of all programmers never read a technical book? Unfortunately that's a pretty precise estimate (perhaps the only precise one within software). For you, as an aspiring author, it means your potential market is limited from the start. Technical books just don't sell well. And the rare exceptions, classics like *Design Patterns* or *Refactoring*, are just that: exceptions. We'll have a discussion about royalties soon, but my advice is that money shouldn't be your primary motivation. Fortunately there are many other rewards for you.

One of the biggest wins of writing comes when you view it as part of a learning process. When you try to explain a concept to someone else, something amazing happens. The process of explaining gives you a new perspective. You'll find that you learn just as much about the subject as your readers do. The knowledge you gain is the most valuable part. It will transfer to your day job and make you a better programmer.

Another big reward is the feeling of achievement. Through your writing you're able to affect people and change the course of their technical lives. Over the years I've received emails from readers all over the world. There's something deeply fascinating about that and it gives you something that's worth more than the next royalty cheque.

Finally, there's the very act of creating something. If you look into motivational psychology you'll find that what makes us happy isn't necessarily fame or commercial success. Happiness is grown when you work towards challenging, yet achievable, goals. Writing about a complex topic like programming will put you right in that spot.

Let me share a story. My second book, *Lisp for the Web* [3], wasn't something I planned. Years earlier I had published an article with the same name. Around that time the Lisp language experienced a small Renaissance. Paul Graham had written about how Lisp allowed his start-up to develop at a higher pace and outperform their competitors when developing a web application.

So Lisp was cool again. But there was virtually no material available on how you actually approach web development in Lisp. I decided to find out and share what I learned.

The 'Lisp for the Web' [3] article was as close to a hit as I probably ever get. My mailbox flooded with comments and nice feedback. If I was smart I should have evolved it into a book right away. Since I wasn't, I just let the article stick on my homepage and get outdated. Never let an opportunity slip like that, please.

Six years later I started to feel sad about the state of my article. The libraries I used for the code had evolved, some of the open-source packages were abandoned and my sample application didn't even work anymore. I didn't want people interested in Common Lisp to come to my article, become frustrated when it didn't work and were thereby discouraged from learning Lisp. I decided to give the article a face lift and get rid of the bit rot in the process.

Since I was already familiar with Leanpub I evolved the article into a short book. A model like Leanpub fits this material well. Because they only do e-books, it's easy to keep the material up to date as the technologies advance. Another advantage is that self-publishing allows you to control the price. Let's see what that means to you.

Experiment with your book price

Most self-publishing services let you specify the sales price yourself. How much of that do you get? Well, that varies a lot. At Leanpub you get a royalty rate of 90 percent. As you'll see when we discuss publishers, 90 percent is an astronomical rate. Compare this to Amazon where you get either 35 or 70 percent. Of course, the royalty rate alone isn't the whole story. Amazon probably lets you reach more people. The good thing is that you don't have to choose as long as you retain the copyright yourself.

So what's an appropriate price for a book? It depends on your goals. With *Lisp for the Web* [3] my main motivation was to give something back to the community. Since I wrote the book to be read, I used the option to make it available for free. I also specified a suggested price of 4.99$. To date, more than 1300 people have downloaded *Lisp for the Web*. Of those, approximately 20 percent chose to pay for it. A pleasant surprise is that some readers pay *more* than the suggested price. Programmers are good people.

Making your book available for free could be a good business model. It helps you build a personal brand and may help you boost your main business. If you're a consultant or want to use the book to build buzz around a certain product you've invested in, the free as in beer model may be just right.

Of course, free books is just one possible segment. You've invested a ton of time and expertise in your book. Charging 10–20$ for all that is still a bargain to any reader. When we buy a technical book, our primary investment isn't the money we pay for it but the time we spend reading it. So my advice is to look at the price of similar books and charge *at least* the same amount yourself; If someone's interested in your book, a few dollars extra won't stop them from buying it.

Decide your publishing model

So far I've only talked about self-publishing. While I like that model, a good publishing company has a lot to offer. It's also a radically different way to work that will affect your writing experience.

After self-publishing two books I got a contract with the The Pragmatic Bookshelf [4] for my new book, *Your Code as a Crime Scene* [5]. My experience has been almost exclusively good and, thanks to the Pragmatics, I managed to write a much better book than I could have done on my own. You see, a good publisher will do a lot for you:

- Contract equals motivation: Your book is only in an early stage, yet someone already believes in it and is prepared to invest time and real money in making your vision come true. Getting a publisher is a motivational boost that helps you get started.
- You get a project editor: Your editor is a skilled author that will coach you, provide continuous feedback and review your material as you write. You'll spend a lot of time with your editor so make sure it feels right from the very start.
- Focus the message: Let's admit it – we programmers love to add cool stuff to our creations. Writing a book is no different. You'll probably find that you try to cover too much ground. A good publisher helps you focus on the topics that really matter to your readers.
- A step up in quality: Publishers do a lot of hard work for you. They provide a copy editor, design your cover and make sure to index your book. If you self-publish, you want to hire someone to do that for you. It makes a marked difference.
- Gain status: While anyone can self-publish a book, getting your work out under the brand of a respected publishing house means something in our industry. It gives you credibility and makes people pay attention to your work.

With the benefit of hindsight, would I go down the publishing route again? Yes, definitely. The Pragmatics are amazing and I'm happy that I worked with them. I learned a lot and I'd definitely recommend them to you. When I write my next book, I'll consider the Pragmatics again.

That said, you may well find that self-publishing fits you better. Self-publishing has some real benefits:

- You're in control: When you self-publish you can be as provocative as you want, you get to use your own style and have full control over every word, image and marketing step. After all, it's your vision and here you own it.
- The topic is yours: Perhaps you like to explore non-mainstream techniques and esoteric programming languages? That's all good and it's an important way to grow as a developer. However, the market for embedded *Idris* programming or recursive decent parsers in *colorForth* is unfortunately quite limited. As a consequence, you may be unlikely to interest a publisher in the stuff you want to write about. Self-publishing lets you ignore the market and write the kind of book you care about. Remember, the main value of writing a book is the learning experience you'll go through.
- Write in an agile way: An interesting trend is that several authors now release their work in a very early stage. Some books meet their audience with little more than a planned table of contents. This lets you publish your work early, get feedback and improve as you go along.
- Retain the copyright: Since you own your own work, you're free to re-package it and publish it any way you want. Perhaps you want to re-work parts of it into another book? Or you want to publish individual chapters as articles on your blog? That's fine – you own it.

From my experience, giving up full control over my work was the hardest step. A good publisher, like the Pragmatics, will of course still listen to you. But you will have to give up on some of your own ideas. When you self-publish, you own every step of the process. That's important to remember as you chose your publishing model.

Find a publisher

While my personal experience with a publisher has been great, everyone hasn't had the same fortune. Publishers differ a lot. You can see that in the books you buy yourself. Some publishers serve as a quality mark while others vary much more in what they allow to pass through.

You can build on that when searching for a publisher. Consider your favorite technical books – who published them? Since you want nothing but the best for your own book, that's the route you want to take. Sure, you as the author are the most important ingredient to your book, but a publisher will affect your work at a profound level. Make sure you know why you choose the publisher you do.

Once you've narrowed the field of potential publishers you need to consider what they offer you. Will they do both print and e-books? What distribution channels do they use? How can they help market your book? And what about the royalty rate?

When I started to consider a publisher I had narrowed my choice to four different companies. At the end I only contacted The Pragmatic Bookshelf since they were the only publisher that offered a royalty rate that felt fair. At the Pragmatics you get 50 percent of the gross profits on your book. Other publishers typically offer a royalty rate of 7-15 percent, perhaps with a possible advance pay.

Of course, the royalty rate is just one factor to consider; Remember, you're unlikely to make much financially from the sales alone. Most of the money you make will come from indirect sources like your stronger personal brand, related training courses or increased salary since you're now a published author.

Get a contract

Once you've decided upon a publisher you need to put together a proposal for your book. The publisher typically specifies what your proposal should include. In general, they want the following:

- Your idea: Provide a short overview of the topic you plan and how you want to approach it. Draft a table of contents with all key chapters. You'll probably change it later, but this helps making your idea more concrete to the people that will review it.
- A writing sample: To write a book you need to show your publisher that you can, well, write. Make sure to polish your sample. Once you have a draft, put it away for some days before you re-visit it. That simple trick gives you a more unbiased view of your writing and you're likely to discover gaps and improvements that you had overlooked the first time.
- Motivate the book: You need to tell them why your book is a good idea. Speculate a bit about the potential market and make sure to communicate why you're the best possible author to capture this idea.

Your Code as a Crime Scene wasn't my first proposal to the Pragmatics. Actually, I'd sent them *Lisp for the Web* and offered to do an extended version of that book. Now, something unexpected happened. The publisher replied that they weren't interested in my Lisp book but that they liked the writing style. They also showed that they cared by doing some quick research on me. That's where they found my online plans for this strange code as a crime scene stuff. So, the publisher asked if I wanted to propose that book instead. I did and got my contract.

The moral of this story is that even if you get a rejection, good things may happen. Don't give up. Consider it a starting point instead. Re-shape your book, take a different angle and try again. The publishers need you so they're happy to get proposals. Remember, you as a potential author is the one that's adding the most value. Without authors there wouldn't be anything to publish.

Master the writing process

Writing is a skill like so much else. There are several good books that help you improve as a writer. My personal favorites are *The Elements of Style* (Strunk & White) and *Keys to Great Writing* (Wilbers). I can't compete with those, so let's look at the surrounding activities instead.

Chose your tools wisely

Writing a book is quite similar to programming. In fact, I recommend the same tools. Write in a pure text format, for example markdown, and use a text editor that supports your format.

You'll find that you spend a lot of time re-writing. In order to do that efficiently you want to put your book under version-control. If you stick to my advice and use a simple text format for markup, you'll be able to compare revisions and rollback edits that didn't work well.

Whatever you chose, don't make the same mistake as I did initially and think that a word processor like Word or OpenOffice is good for, well, word processing. They're not. Today's books are delivered in a variety of formats like MOBI, EPUB, HTML and the humble dead tree. Converting from a Word document to one of those formats is just painful. Don't go there.

Plan your book

Start with a detailed outline of your book. I can't stress the importance of this enough. In the past, I used to skip this step and every time I did it came back to bite me. The result is a trail of abandoned book projects. You don't want that.

When I say 'detailed outline' I mean it. Write a short overview of each chapter and capture its key points and take-aways. This step helps you fit the individual pieces into a coherent whole.

You'll probably find that some chapters don't quite fit the overall structure and tone of the book. In that case, just drop them and keep the material for other purposes. You also have to be prepared to change course and narrow the scope. Since you have it all in version-control, you can always go back if you need to.

It's during this planning stage that writing departs from programming; Writing a book is more predictable. There are less things to discover. As such, the more time you spend up front, the easier the rest of your writing. I promise, you'll still have plenty of opportunities to develop ideas that pop-up and the new connections you make as you progress.

Stay on track

Remember, writing a book is time consuming. I often got the advice to reserve dedicated writing time. Sure, it's possible to spare a day or two but it's not enough to make any real progress. If too much time passes between your writing sessions you'll lose good ideas, have a harder time maintaining flow once you write and, worse, it gets easier to just drift away; Before you realize it, a whole month has passed without a written word. That's how book projects get abandoned.

So do reserve the days you can. Do invest some of your holiday to move your book forward. But you need more. I recommend a complementary approach that lets you make progress on your book while at the same time balancing work and other obligations.

The way I do it is to print a calendar and nail it to the wall as you see in the picture below. Every day that I write at least 30 minutes on the book I'm allowed to cross out that day in the calendar. The goal is to build the longest, unbroken chain of productive days. (Figure 2)

The key here is to set a realistic goal. I often wrote several hours a day, but my goal was a modest 30 minutes. That's really nothing and it's always do-able. I wrote in airports, on trains and in cars (you see – if you ever

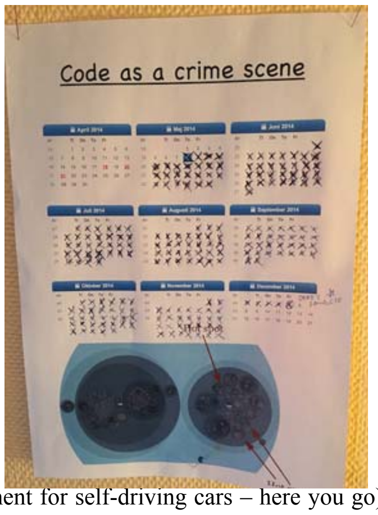


Figure 2

needed an argument for self-driving cars – here you go). I wrote on my birthday, I wrote with fever after catching the 'flu. I even wrote on the day our cellar got flooded in the worst rain of the past 150 years. After pumping out water and losing a fight to mother nature I was exhausted. But still – 30 minutes? I can do that.

When you write every day you stay on track. You make progress and keep your ideas alive in your head. Before you know it you've completed one more chapter and moved closer to your goal.

Reflect your learning

Once you have finished your book you'll look differently upon your material. Remember, writing is a learning experience and you have a much deeper understanding of your topic afterwards. You want to reflect that learning in the early chapters you wrote. That's why it is important to go back and improve your material.

Of the whole writing process, this is the hardest step. Not from a technical point of view but from a motivational. You're done and you're probably mentally exhausted from the whole project. Yet you'll find that now is the opportunity to take your book to the next level. To move beyond good enough to just great.

Market and promote your work

If you have a publisher they'll help you market your work. But even in that case much of the responsibility is yours.

There are several things you can do to spread the word about your work. Give away free copies, lots of them. Provide free material on the book's homepage. Talk at conferences and publish articles that cover some of the highlights of your book. Record podcasts and make short videos to demonstrate some core ideas.

The whole point is to get people to talk about what you have done. Since I personally love to speak at developer conferences and share what I've learned, that's what I focus on. That may not be for everyone. But whatever you chose, make sure you enjoy it.

Let's get started

We're through our quick tour of the book writing landscape now. Since you've made it this far, I conclude that you're serious about your book project. That's great! It's well-worth the effort as long as you manage to find a sustainable pace. Remember, the book's going to consume much of your time. It's going to be hard and frustrating. It's also one of the most rewarding things you can do as a programmer. ■

References

[1] https://leanpub.com/patternsinc
[2] http://www.adamtornhill.com/articles.htm
[3] https://leanpub.com/lispweb
[4] https://pragprog.com/
[5] https://pragprog.com/book/atcrime/your-code-as-a-crime-scene

Split and Merge – Another Algorithm for Parsing Mathematical Expressions

Vassili Kaplan presents an alternative to Dijkstra's algorithm.

To parse a string containing a mathematical expression (for example, $3 + 4 * 2 / (1\text{-}5)^2$) a Shunting-yard algorithm [1] created by Edsger Dijkstra is often used. It converts the expression to Reverse Polish notation and then uses the postfix algorithm [2] to evaluate it.

In this article we present another algorithm for parsing a mathematical expression. It consists of two steps as well – firstly creating a list of structures, each containing a number and an action, and secondly merging all the structures together to get the expression result.

Even though this algorithm has same time complexity as Dijkstra's, we believe it is simpler to code. We will discuss some examples of using it and present its implementation in C++.

Splitting an expression to a list of structures

The result of the first step of our algorithm is a list of structures. Each structure consists of two elements – a real number and an action. An action can be any of add, subtract, multiply, divide, power, or 'no action'. The special 'no action' is assigned to the last member of the whole expression or to the last member before a closing parenthesis. For convenience we denote this 'no action' with the symbol ')'.

Example 1

Suppose the expression string is "3 + 2 * 6 - 1". Then the result of the first step will be:

Splitting ("3 + 2 * 6 - 1") → [3, '+'], [2, '*'], [6, '-'], [1, ')']

Note that action (operation) priorities are not taken into account at this step.

If one of the tokens is a function (e.g. sine, cosine, log, etc.), a special constant (e.g. pi) or an expression in parentheses then the whole algorithm (both steps) will be applied to this token first, which is then replaced by the calculated result. This will be done recursively to all nested functions and expressions.

Therefore, after finalizing this first step, all of the functions, constants and parentheses will be eliminated.

Merging the elements of the created list of structures

The second step is an actual calculation done by merging the elements of the list of structures (each one consisting of a number and an action) created in the first step.

We attempt to merge each structure in the list with the next, taking into account the priority of actions.

Priorities (from highest to lowest) of the actions are:

- Power ('^')
- Multiplication ('*') and division ('/')
- Addition ('+') and subtraction ('-')
- 'No action' (')')

VASSILI KAPLAN

Vassili Kaplan has been a Software Developer for almost 15 years, working in different countries and different languages (including C++, C#, and Python). He currently resides in Switzerland and can be contacted at vassilik@gmail.com.

Merging can be done if and only if the priority of the action in the first structure is not lower than the priority of the second structure. If this is the case, merging consists of applying the action of the first structure to the numbers of both structures, thus creating a new structure. The resulting action of this new structure will be the one from the second structure.

E.g. merging [5, '-'] and [4, '+'] will produce [5 - 4, '+'] = [1, '+'].

But if the second structure priority is higher than the first one we cannot merge the current structure with the next one. What happens then?

Then we merge the next (second) structure with the structure next to it, and so on, recursively. Eventually the merging will take place since the last element in the stack always has a 'no action' which has the lowest priority. After merging the next element with the structures following it, we re-merge the current element with the newly created next structure (see example 2 below).

Example 2

Let's merge the elements created in example 1:

[3, '+'], [2, '*'], [6, '-'], [1, ')']

We always start with first element in the list, in this case element [3, '+']. Note that [3, '+'] cannot be merged with [2, '*'] since the action '+' has a lower priority than '*'.

So we go to the next element, [2, '*'], and try to merge it with [6, '-']. In this case the multiplication has higher priority than the subtraction so we can merge the cells. The action of the resulting structure will be the one of the second structure, i.e. '-'. So the result of merging [2, '*'] and [6, '-'] will be [2 * 6, '-'] = [12, '-'].

This resulting structure will be merged with the next structure, which is [1, ')']. Since the multiplication has a higher priority than 'no action', we can merge these two elements:

Merging ([12, '-'], [1, ')']) → [12 - 1, ')'] = [11, ')'].

Now we can remerge this result with the first structure [3, '+']. Remember that we could not do the merge first because the addition action of the first structure had a lower priority than the multiplication action of the second. But after merging the second structure with the rest, its action became the 'no action', so now the merging may take place:

Merging ([3, '+'], [11, ')']) → [3 + 11, ')'] = [14, ')'].

We are done since only one structure is left after merging.

So the final result of applying two steps of the Split and Merge algorithm is:

"3 + 2 * 6 - 1" → [3, '+'], [2, '*'], [6, '-'], [1, ')'] → [14, ')'].

Therefore, 3 + 2 * 6 – 1 = 14.

Complete processing examples

If the expression contains functions or other expressions in parentheses then those expressions will be calculated recursively first. Let's see how it is done it in the following two examples.

Example 3

Let's calculate the expression "2 ^ (3 - 1)" using this algorithm.

1. Splitting ("2 ^ (3 - 1)") → [2, '^'], [Calculate "3 - 1", ')']

Now we need to calculate "3 – 1" and then return to where we were.

To calculate it we need to apply both steps of the algorithm, merging and splitting:

1.1 Splitting ("3 - 1") → [3, '-'], [1, ')'].

1.2 Merging ([3, '-'], [1, '^']) → [3 - 1, ')'] = [2, ')'] → 2.

Now we put this result back to 1:

[2, '^'], [Calculate "3 - 1", ')'] → [2, '^'], [2, ')']

2. Merging ([2, '^'], [2, ')']) → [2^2, ')'] = [4, ')'] → 4.

Therefore the result of the expression is: "2 ^ (3 - 1)" → 4.

Example 4

Let's calculate the expression "2 * cos (pi)" using this algorithm.

1. Splitting ("2 * cos (pi)") → [2, '*'], [Calculate "cos(pi)", ')']

Calculating "cos(pi)":

1.1 Splitting ("cos (pi)") → [cos(pi), ')'] → [cos(3.14…), ')'] → [-1, ')']

1.2 Merging ([-1, ')']) → -1

Note that pi is one of the 'special' constants hardcoded in the splitting module. It is replaced by its numerical value in the splitting part of the algorithm.

Replacing the expression we've split with the result of -1, we get:

[2, '*'], [Calculate "cos (pi)", ')'] → [2, '*'], [-1, ')']

2. Merging ([2, '*'], [-1, ')']) → [2 * (-1), ')'] = [-2, ')'] → -2.

Therefore the result of the expression is: "2 * cos (pi)" → -2.

Implementation in C++

The entry point is the following **`process()`** function which calls preprocessing of the passed string and then **`loadAndCalculate()`** function which actually implements the splitting of the passed string into a list of structures and then merging the elements of this list.

```
double EZParser::process(const string& data)
{
  string cleanData;
  EZParser::preprocess(data, cleanData);
  size_t from = 0;
  return loadAndCalculate(cleanData, from, '\n');
}
```

The preprocessing just removes the blank spaces from the passed string and checks that the number of opening parenthesis matches the number of closing ones. We will skip this function for brevity. We'll also skip the **`CalculationException`** class which is a simple wrapper over the standard exception.

The main function that performs splitting into the list of structures and then calling the merging function is **`loadAndCalculate()`**. First it creates and fills the list, implemented as a vector, and then calls the **`merge()`** function to process it further (see Listing 1).

The **`stillCollecting()`** function just checks that we are still collecting the current token (i.e. there is no new action and no new parentheses in the next character) (see Listing 2).

`UpdateAction()` function looks for the actual action corresponding to the last extracted token. This is not always the next character since there can be one or more parenthesis after the token. In case there is no action, the closing parentheses will be returned. (See Listing 3.)

The implementation of the second part of the algorithm, merging the cells created in the first part is below. Note that this function calls itself recursively in case the priority of the current element is less than the priority of the next (Listing 4).

When extending this implementation, e.g. with '||', '&&', etc., the two functions in Listing 5 must be adjusted.

An arbitrary number of functions and constants may be added to the implementation. In this implementation we use an 'identity' function to calculate an expression in parenthesis. Listing 6 contains just some of the functions.

Listing 1

```
// data contains the whole expression, from is an
// index pointing from which character the
// processing should start. The end character is
// either '\n' if we want to calculate the whole
// expression or ')' if we want to calculate an
// expression in parentheses.
double EZParser::loadAndCalculate
  (const string& data, size_t& from, char end)
{
  if (from >= data.size() || data[from] == end)
  {
    throw CalculationException
      ("Loaded invalid data " + data);
  }
  vector<Cell> listToMerge;
  listToMerge.reserve(16);
  string item;
  item.reserve(16);

  do
  { // Main processing cycle of the first part.
    char ch = data[from++];
    if (stillCollecting(item, ch))
      { // the char still belongs to
        // the previous operand
        item += ch;
        if (from < data.size() && data[from]
          != end)
        {
          continue;
        }
      }
      // We finished getting the next meaningful
      // token. The call below may recursively
      // call the function we are in. This will
      // happen if the extracted item is a
      // function itself (sin, exp, etc.) or if
      // the next item is starting with a '('.
      double value = m_allFunctions.getValue
        (data, from, item, ch);
      char action = validAction(ch) ? ch
        : updateAction(data, from, ch);
      listToMerge.push_back(Cell(value, action));
      item.clear();
  }

  while (from < data.size() && data[from] != end);
  if (from < data.size() && data[from] == ')')
  { // End of parenthesis: move one char forward.
    // This happens when called recursively.
     from++;
  }
  Cell& base = listToMerge.at(0);
  size_t index = 1;
  return merge(base, index, listToMerge);
}
```

The functions in Listing 7 are used in the first step of the algorithm via the **`getValue`** function, when creating the list of structures (see function **`loadAndCalculate()`** in Listing 1 above).

Complexity and conclusions

Each token in the string will be read only once during the first, splitting step.

Suppose that there are *n* characters in the expression string and that the first step leads to *m* structures for the second step. Then in the second step there

Listing 2

```
bool EZParser::stillCollecting
  (const string& item, char ch)
{
  return (item.size() == 0 &&
    (ch == '-' || ch == ')')) ||
    !isSpecialChar(ch);
}
bool EZParser::isSpecialChar(char ch)
{
  return validAction(ch) || ch == '(' ||
    ch == ')';
}
bool EZParser::validAction(char ch)
{
  return ch == '*' || ch == '/' || ch == '+' ||
    ch == '-' || ch == '^';
}
```

Listing 3

```
char EZParser::updateAction(const string& item,
  size_t& from, char ch)
{
  if (from >= item.size() || item[from] == ')')
  {
    return ')';
  }
  size_t index = from;
  char res = ch;
  while (!validAction(res) && index < item.size())
  { // We look to the next character in string
    // until we find a valid action.
    res = item[index++];
  }
  from = validAction(res) ? index
     : index > from ? index - 1
     : from;
  return res;
}
```

Listing 4

```
double EZParser::merge(Cell& current,
  size_t& index, vector<Cell>& listToMerge)
{
  if (index >= listToMerge.size())
  {
    return current.value;
  }
  while (index < listToMerge.size())
  {
    Cell& next = listToMerge.at(index++);
    if (!canMergeCells(current, next))
    { // If we cannot merge cells yet, go to the
      // next cell and merge next cells first.
      // E.g. if we have 1+2*3, we first merge
      // next cells, i.e. 2*3, getting 6, and then
      // we can merge 1+6.
      merge(next, index, listToMerge);
    }
    mergeCells(current, next);
  }
  return current.value;
}

bool EZParser::canMergeCells(const Cell& base,
  const Cell& next)
{
  return getPriority(base.action) >=
    getPriority(next.action);
}
```

Listing 5

```
size_t EZParser::getPriority(char action)
{
  switch (action)
  {
    case '^': return 4;
    case '*':
    case '/': return 3;
    case '+':
    case '-': return 2;
  }
  return 0;
}
void EZParser::mergeCells(Cell& base,
   const Cell& next)
{
  switch (base.action)
  {
    case '^': base.value = ::pow(base.value,
      next.value);
      break;
    case '*': base.value *= next.value;
      break;
    case '/':
      if (next.value == 0)
      {
        throw CalculationException
          ("Division by zero");
      }
      base.value /= next.value;
        break;
      case '+': base.value += next.value;
        break;
      case '-': base.value -= next.value;
        break;
  }
  base.action = next.action;
}
```

Listing 6

```
class EZParserFunctions
{
public:
  typedef double (EZParserFunctions::*ptrFunc)
    (const string& arg, size_t& from);
  EZParserFunctions();
  double getValue(const string& data,
    size_t& from, const string& item, char ch);
  double identity(const string& arg,
    size_t& from);
  double sin(const string& arg, size_t& from);
  double pi(const string& arg, size_t& from);
  // …
  static double strtod(const string& str);
protected:
  map<string, ptrFunc> m_functions;
};
EZParserFunctions::EZParserFunctions()
{
  m_functions["sin"]   = &EZParserFunctions::sin;
  m_functions["pi"]    = &EZParserFunctions::pi;
  // …
}
// This function is used to process the contents
// between ().
double EZParserFunctions::identity
  (const string& arg, size_t& from)
{
  return EZParser::loadAndCalculate(arg, from,
    ')');
}
```

Using 32-bit COM Objects from 64-bit Programs

Roger Orr explains how to cross the platform boundary in COM libraries.

Microsoft introduced COM – the 'Component Object Model' – in the early 1990s and it has proved to be remarkably resilient even though the computing world has changed quite a bit since those days – for example most people who run Windows are now running it on 64-bit hardware. However, a large number of the programs and components that run in such environments are still 32-bit for a variety of reasons, some better than others.

While Microsoft have done a pretty good job of integrating a mix of 64- and 32-bit applications there can be a problem when it comes to using older 32-bit COM components in a 64-bit application as Windows does not allow a mix of 32-bit and 64-bit user code in a single application. This means a 32-bit COM DLL cannot be used by a 64-bit application (although a 32-bit COM EXE can.)

There are a variety of solutions to this problem. The best solution is to use a 64-bit version of the COM DLL as this will provide the best integration with the 64-bit application. However this requires that a 64-bit version exists, or can be produced; this is not always the case.

An alternative solution is to host the 32-bit COM DLL inside a 32-bit application and use this hosting application from the 64-bit application. Microsoft provide an easy-to-use standard example of such a solution in the form of a DllHost process.

This article describes how to use this DllHost mechanism and highlights a couple of potential issues with this as a solution that you need to be aware of.

Demonstration of the problem

We can quickly show the problem by writing an example 32-bit COM DLL in C# and trying to use it with the 32-bit and the 64-bit scripting engines (Listing 1). (In general the 32-bit COM DLLs likely to need this solution are probably written in some other language, such as C++, but the C# example has the benefit of being short!)

This can be turned into a COM object from the Visual Studio 'Developer Command Prompt' using:

```
csc /t:library CSharpCOM.cs
```

and then registered (from an administrator command prompt) using:

```
RegAsm.exe CSharpCOM.dll /codebase
```

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

Split and Merge (continued)

Listing 6 (cont'd)

```
double EZParserFunctions::sin(const string& arg,
  size_t& from)
{
  return ::sin(EZParser::loadAndCalculate(arg,
    from, ')'));
}
double EZParserFunctions::pi(const string& arg,
  size_t& from)
{
  return 3.141592653589793;
}
```

will be at most $2(m-1) - 1 = 2m - 3$ comparisons (in the worst case) to merge all of the structures. Therefore the time complexity will be:

$$O(n + 2m - 3) = O(n).$$

The main difference between the Split and Merge and the Dijkstra algorithm is that the former uses potentially many recursive calls whereas the later uses no recursion. So the actual performance gain will depend on a particular implementation language and the compiler, in particular how well the recursion is managed. ■

Listing 7

```
double EZParserFunctions::getValue
   (const string& data, size_t& from,
    const string& item, char ch)
{
  if (item.empty() && ch == '(')
  {
    return identity(data, from);
  }
  map<string, ptrFunc>::const_iterator it
    = m_functions.find(item);
  // The result will be either a function or a
  // real number if there is no function.
  return it == m_functions.end() ? strtod(item)
    : (this->*(it->second))(data, from);
}
double EZParserFunctions::strtod(const string&
str)
{
  char* x;
  double num = ::strtod(str.c_str(), &x);
  if (::strlen(x) > 0)
  {
    throw CalculationException
      ("Could not parse token [" + str + "]");
  }
  return num;
}
```

References

[1] Shunting-yard algorithm, http://en.wikipedia.org/wiki/Shunting-yard_algorithm

[2] Reverse Polish notation http://en.wikipedia.org/wiki/Reverse_Polish_notation

Listing 1

```
namespace CSharp_COMObject
{
  using System.Runtime.InteropServices;

  [Guid("E7C52644-7AF1-4B8B-832C-23816F4188D9")]
  public interface CSharp_Interface
  {
    [DispId(1)]
    string GetData();
  }

  [Guid("1C5B73C2-4652-432D-AEED-3034BDB285F7"),
  ClassInterface(ClassInterfaceType.None)]
  public class CSharp_Class : CSharp_Interface
  {
    public string GetData()
    {
      if (System.Environment.Is64BitProcess)
        return "Hello from 64-bit C#";
      else
        return "Hello from 32-bit C#";
    }
  }
}
```

(You may need to provide the full path to `RegAsm.exe`, such as: C:\Windows\Microsoft.NET\Framework\v4.0.30319\RegAsm.exe)

Note: you are likely to get a warning about registering an unsigned assembly – we are not concerned about that for the purposes of this demonstration.

Next create a VB script that loads and uses this COM object:

```
Set obj = WScript.CreateObject
  ("CSharp_COMObject.CSharp_Class")
WScript.Echo obj.GetData
```

and invoke it with the **32-bit** scripting engine:

```
c:\windows\syswow64\cscript.exe /nologo
exercise.vbs
```

If all has gone according to plan this will produce the output:

```
Hello from 32-bit C#
```

Now try to invoke it using the **64-bit** scripting engine:

```
c:\windows\system32\cscript.exe /nologo
exercise.vbs
```

This will fail with the error message:

```
WScript.CreateObject: Could not locate automation
class named "CSharp_COMObject.CSharp_Class".
```

This demonstrates the typical issue when trying to use a 32-bit COM object from a 64-bit application (or vice versa). While the precise error message may vary a little depending on the hosting application, the basic problem is that the 64-bit application cannot locate the 32-bit COM object.

Configuring the COM object to use DllHost

In order to make use of the standard DllHost mechanism we simply need to add a couple of registry keys to the system. Microsoft have used the term ‘DllSurrogate’ for this mechanism and this name is used in one of the values written into the registry.

We need to add a new value to the CLSID for the class(es) affected and this change must be made in the **32-bit** area of the registry. We must then also define a new application with a blank DllSurrogate.

In our example the target CLSID is **`{1C5B73C2-4652-432D-AEED-3034BDB285F7}`** (the GUID we provided for our COM class in the C# source code) and we need to create a new GUID for our DLL surrogate. (Strictly speaking you do not need a new GUID for the AppID; some people recommend that you re-use the CLSID. I prefer using a new GUID for clarity.) I’ve split the lines up with the **^** continuation character for ease of reading:

1. Get a new GUID for the DllSurrogate

```
C:> uuidgen -c
308B9966-063A-48B8-9659-4EBA6626DE5C
```

2. Add an AppID value to the (32-bit) class ID

```
c:\windows\syswow64\reg.exe add ^
HKCR\CLSID\{1C5B73C2-4652-432D-AEED-3034BDB285F7}^
/v AppID /d {308B9966-063A-48B8-9659-4EBA6626DE5C}
```

3. Create a new AppID

```
reg.exe add HKCR\AppID\ ^
{308B9966-063A-48B8-9659- 4EBA6626DE5C} ^
/v DllSurrogate /d ""
```

Now we can successfully use the COM object from the 64-bit scripting engine:

```
c:\windows\system32\cscript.exe /nologo
exercise.vbs
Hello from 32-bit C#
```

Removing the COM object and the surrogate

If you like keeping your computer tidy and would like to unregister the COM object, you can do this by reversing the installation above with the following commands:

```
reg.exe delete HKCR\AppID\ ^
{308B9966-063A-48B8-9659-4EBA6626DE5C}

c:\windows\syswow64\reg.exe delete ^
HKCR\CLSID\{1C5B73C2-4652-432D-AEED-3034BDB285F7}^
/v AppID

regasm.exe CSharpCOM.dll /u
```

How does it work?

If you use the task manager or some similar tool to examine the processes running in the computer you will see an instance of the Dllhost.exe process start and then disappear after a few seconds.

The command line for DllHost contains the CLSID of the target class and this (32-bit) process actually creates the 32-bit COM object using this CLSID and makes it available to the calling application, using cross-process marshalling for the calls between the 64-bit application and the 32-bit target object.

When the COM object is destroyed, DllHost hangs around for a few seconds, ready for another process to create a COM object. If none does it times out and exits.

Some differences from in-process COM objects

There are a number of differences between a ‘normal’ in-process COM object and this externally hosted object.

The first obvious difference is that each call into the COM object now has to marshall data between the calling process and the DllHost process. This is obviously going to introduce a performance overhead. It is hard to tell how significant this performance overhead will be; it depends on the ‘chattiness’ of the COM interface and the sizes of the data items being passed around.

Some COM interfaces pass non-serialisable values around (for example, pointers to internal data structures) and these will require additional support of some kind to work successfully in the Dll surrogate environment.

Another difference is that the same DllHost process may host COM objects for a number of different applications at the same time. While this is perfectly valid according to the rules of COM, there are a number of cases where problems can arise – for example if the implementation makes

Raspberry Pi Linux User Mode GPIO in C++ – Part 1

Ralph McArdell expands the Raspberry Pi with a C++ library.

I started experimenting with general purpose input/output (GPIO) with a Raspberry Pi using Python [1, 2] which enabled the reading and writing of Boolean values representing on/off states (in fact high/low voltage states) – reading switch states, turning lights on and off and the like. Shortly after I had the Python Raspberry Pi GPIO library up and running the original Gertboard [3] kit was released. The Gertboard interfaces to the Raspberry Pi via its P1 IO expansion header connector (J8 on later B+/A+/2.0 models) and provides a smorgasbord of hardware devices that use a variety of peripheral interfaces. So I ordered and built one enabling me to play with peripheral IO beyond basic GPIO. At the time the Gertboard had test and example C code available for Linux [4], particularly Raspbian the Raspberry Pi specific Linux distribution. Similar to the Python case I felt the provided C code could be expressed more cleanly. I thought it would be interesting to see what advantages C++, hopefully C++11, features and idioms might provide.

A thought: germ of a library

The originally provided Gertboard test and example C code, in a fashion similar to the Wiring Pi library [5], manages the Raspberry Pi's BCM2835 processor's peripherals by directly accessing peripherals' memory mapped control registers from user mode by using **`mmap`** to map sections of the **`/dev/mem`** physical memory device. A single function sets up **`mmap`**-ed regions in a rather long winded and repetitive way for all supported peripherals – even if only one or two peripheral regions are required. Oddly, for each mapped register-block a block of free-store is

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.

Using 32-bit COM Objects from 64-bit Programs (continued)

assumptions about being able to share internal data between separate COM objects.

Additionally, some in-process COM objects are written with some assumptions that the calling application code is in the same process as the COM object. For example, the name of a log file might be based on the name of the application. This may be an insurmountable problem for using this solution, although if the COM interface is in your control you might be able to enhance it to provide additional methods to support out-of-process operation.

Multiple classes

If your COM DLL provides a number of different COM classes, you might wish to host them all inside the same DllHost instance. You can do this by using the same AppID for each class ID; the first COM object created will launch the DllHost and each additional object created by the application will simply bind to an object in the same DllHost.

Calling 64-bit COM objects from 32-bit applications

The commonest use case for Dll surrogates is allowing a legacy 32-bit COM object to used in a new 64-bit application, but the same mechanism does work in the other direction, although it is less common for this to be necessary.

You need to add the same registry keys as above, but this time the CLSID will be the one in the 64-bit registry hive so you will use the 64-bit reg.exe from the normal directory C:\Windows\System32.

A note on 64-bit .Net COM objects

In this article I have used a simple C# COM object to demonstrate the problem of 64-bit programs using 32-bit COM objects and how to solve this by using a Dll surrogate.

A native COM DLL, for example written in C++, is built for either 32-bit or 64-bit use and can only be loaded by a program of the same bitness. In this case the Dll surrogate or an equivalent is the only way that the native COM DLL can be used by a program of different bitness.

C# programs, though, are 'bit-size agnostic' by default – the same C# program can run in 32-bit or 64-bit mode and the same C# assembly can be used from both 32-bit and 64-bit programs. (This works because there are two .Net virtual machine implementations, one for 32-bit programs and one for 64-bit ones.)

So the DLL surrogate approach used in this article is actually required only in the case of legacy COM DLLs, as .Net COM objects can operate in this dual-mode fashion.

The fundamental reason why our C# COM object could be used by 32-bit programs but not by the 64-bit scripting engine was because it had been registered with the 32-bit **regasm** program, which only writes entries to the areas of the system registry read by 32-bit programs. If you register the C# COM DLL with the 64-bit regasm, for example:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
RegAsm.exe CSharpCOM.dll /codebase
```

then the C# COM object will be usable directly by a 64-bit program without needing to use a Dll surrogate. However this is only true for .Net assemblies, whereas the Dll surrogate approach works with native DLLs as well.

Summary

While many users of Windows will be able to make use of 64-bit applications with 64-bit COM objects it is good to know that, subject to a few restrictions, you can make use of a mix of 64-bit and 32-bit components by setting only a few registry keys. ■

Further information

There is some Microsoft documentation about all this at:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms695225%28v=vs.85%29.aspx

While the technique has been available for some time, there was initially a lack of documentation about the process and, in my experience anyway, few people are aware of the existence of this technique. ■

first allocated which is larger than required to ensure a correctly aligned block can be passed to `mmap`. When done of course all these resources have to be explicitly released.

It occurred to me that in C++ I would wrap mapped of parts of `/dev/mem` in a resource managing type that used the RAII (Resource Acquisition Is Initialisation) [6] idiom. It was this thought that was to lead to a Raspberry Pi user mode C++ library. So putting some code where my brain was, so to speak, I went ahead and implemented such a beast. The result was a class template called `phymem_ptr`, with most of the implementation being taken care of by the non-template base class `raw_phymem_ptr` which works with un-typed pointers (`void*`) .

Having implemented, and tested, `phymem_ptr` using Phil Nash's Catch library [7], I re-implemented one of the simpler Gertboard example programs – `ocol.c` – in C++ using `phymem_ptr`. I felt the result was a definite improvement but more could be done and the obvious next step seemed to be to start with basic GPIO and, following my thinking with the Raspberry Pi GPIO Python library, model the abstractions after C++ library IO. Of course as this would be solving essentially the same problem as the Python GPIO library there were bound to be other concepts that would transfer to the C++ implementation – a notion of GPIO pin number as represented in the Python library as the `PinId` type for example.

In the beginning...

Having decided to write more than `phymem_ptr`, its tests and the `ocol`-in-C++ example it was time to bootstrap a project. Things such as project name, licensing terms, directory structure and what development process to utilise had to be decided.

The name `rpi-peripherals` was chosen for the library (though embarrassingly due to a typo it was `rpi-periphals` for quite a while – doh!). The longer term 'peripherals' was chosen rather than 'gpio' as the intention was for the library to provide support for more than just basic GPIO. As there was no reason to do otherwise the same dual BSD/GPL licensing used for the Python Raspberry Pi GPIO library was selected.

A fairly standard project directory structure was devised having few files in the project root and source, build and outputs etc. in various subdirectories such as `src`, `bin`, `lib`. At the time of writing all the source with the exception of header files required for the public interface of the library is in the `src` subdirectory with no further division. On the other hand all test and example source are in subdirectories `tests` and `examples` of `src` respectfully.

I did not want the library to require any development tools other than what could be expected to be installed on a typical Raspberry Pi Raspbian installation (plus git – keep reading…). That meant using the Raspbian distribution's default version of g++ (4.6.3) and I fell back to using `make` – well GNU `make` – and wrote a `Makefile` for (and placed in) each directory in which code was to be compiled, as well as a top level `Makefile` as a driver and a common file, `makeinclude.mak`, that defined common values such as directory and file names and tool names and flags. Code was developed on a Raspberry Pi running Raspbian, edited from a Windows session in Notepad++ with builds, tests etc. run via `ssh` accessed from a Ubuntu Desktop Linux virtual machine. The project was placed under git source control on GitHub as dibase-rpi-peripherals [8]. As previously mentioned Phil Nash's Catch library was used for tests, which was added to the repository as a git sub-module.

Rather than use GitHub's repository wiki for documentation, as was done for the Python Raspberry Pi GPIO library, this time Doxygen commenting was used in header files, and a Doxygen project configuration `Doxyfile` was added to the project.

What did you want for starters?

The scope for the initial functionality was mostly a subset of that of the Python Raspberry Pi GPIO library:

- Targeting only Raspberry Pi Linux distributions, especially Raspbian
- Single GPIO pin input and output, with no edge event support
- A pin id abstraction
- Internal pin in use / pin free resource management
- Library specific exception types

Since the development of the Python Raspberry Pi GPIO library a major Raspberry Pi board revision had taken place that changed some of the pin assignments on the Raspberry Pi P1 connector and added a new P5 connector with additional GPIO pins. This meant that converting a P1 connector pin number to a BCM2835 GPIO pin number was no longer a simple fixed mapping. It was now dynamically dependent on which board revision the code was running on. Additionally, of course, the newer boards had a P5 to BCM2835 pin mapping. Hence additional support for determining a board revision was needed.

Following the initial development of the library there have been further additions to the Raspberry Pi family in the form of the A+, B+, compute module models and now the Raspberry Pi 2.0 B model. I am yet to find the time to completely update the library to support all these board versions' GPIO specifics.

First cut

The library usage would be broadly similar to the Python Raspberry Pi GPIO library: an object representing an input or output pin would be associated with a GPIO pin by opening the required pin which would be specified as a pin id type instance. If valid and the pin is not in use the open request would be successful and the GPIO pin object could be used to either read or write Boolean values from/to the associated GPIO pin. On GPIO pin object destruction the pin would be marked as free for further use.

Here we have several types of object: input and output pin objects, pin id objects, some form of GPIO pin allocation management object and of course direct access via a `phymem_ptr` to the GPIO peripheral's registers.

Rather than mapping the BCM2835 GPIO peripheral's registers as an array of structure-less 32-bit words (i.e. unsigned integers), as had been done for the original `ocol`-in-C++ example, it seemed more reasonable to provide a type reflecting the BCM2835 GPIO peripheral's register layout that could be used to specialise `phymem_ptr`.

This rough structure outline is shown in Figure 1.

The `ipin`, `opin` and `pin_id` types are intended as the public interface to the library while the rest would be internal to the library – in fact during refactoring after the initial development phase such code was placed into a separate inner namespace called `internal` and the header files for the public entities were moved from the project `src` to the `include` directory. The 'o' and 'i' prefixes to `ipin` and `opin` were taken from C++ IOStream library `std::istream` and `std::ostream` naming.

So how do I register?

In order to create a structure representing the GPIO peripheral's register-block detailed knowledge of register layout, usage and location are obvious requirements. Such detailed technical information is to be found

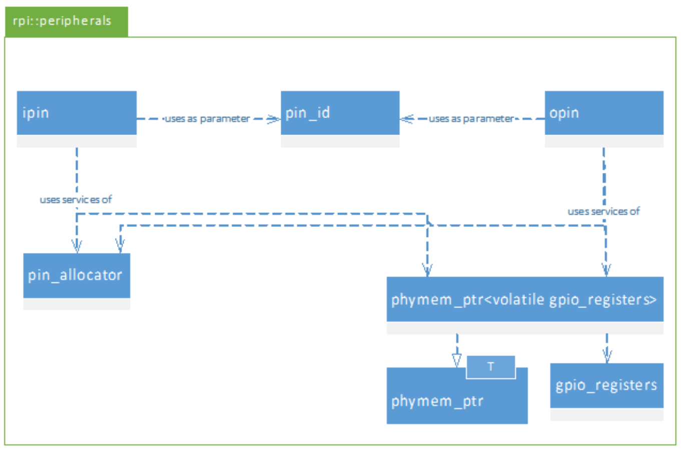


Figure 1

in the Broadcom BCM2835 ARM Peripherals document [9], although as it contains errors and omissions the errata page at eLinux [10] is also useful.

As an aside: when perusing the Raspberry Pi's Linux kernel source code [11] you will find very little specific BCM2835 support [12]; most chip-specific support appears to be provided by the BCM2708 specific code [13]. It is to be assumed that this chip is very similar in its peripheral support to the Raspberry Pi's BCM2835, so detailed BCM2835 specifics may well apply to the BCM2708 as well.

Separating the mapping of peripheral register block addresses with **`phymem_ptr`** and the register block layout representation as manifested by the **`gpio_registers`** class type for the GPIO peripheral allows instances of register layout representation types to be arbitrarily created in memory – as function-scope automatic objects for example. This allows easy unit testing of the representation type to ensure it is sane before having to rely on it when mapped over a peripheral's register block and fiddling with real peripheral registers.

It seemed that adding operations (member functions) for manipulating the fields of the GPIO peripheral's registers would be convenient for users of the **`gpio_registers`** type and would allow such operations to be fully unit tested. However, a balance had to be struck between spending time developing an all-singing all-dancing set of operations and just getting things done. The balance point I chose was to implement operations to access single fields. Many peripheral fields are bit fields and so in certain scenarios reading or writing to multiple bits fields in one go could be more efficient. I left this as optimisation territory, facilitated only by having all register data members be public. In fact the **`gpio_registers`** class type was implemented as a **`struct`**. I felt this was justified as it was intended to be a low-level type internal to the library.

I decided against using C/C++ bit fields – instead using 32-bit unsigned integer types as register data members with access via individual member functions. I suppose I could have looked up the specifics of the GCC bit field implementation, checked whether they were compatible (e.g. guaranteed 32-bit reads and writes for underlying 32-bit integer field types) and if they were then utilised unions to allow accessing whole registers or bit field parts thereof. Such a scheme would probably cover many, maybe even most, cases.

As the **`gpio_registers struct`** was a direct reflection of the GPIO peripheral's registers and fields I decided that all registers would be represented by data members and operation member functions would be provided to access all fields even if they were not known to be needed. One quite major upside of this decision is that it made me really read and understand the associated documentation – making me more able to later decide on what would be needed!

What pin was that?

The need to identify and validate GPIO pin numbers and their encapsulation in the **`pin_id`** type has been mentioned in connection with the public **`ipin`** and **`opin`** types. However, it turns out, unsurprisingly when you think about it, that the GPIO peripheral has many register-sets containing fields for each of the BCM2835's 54 GPIO pins. Thus operations often need to specify which GPIO pin they were to be applied to. As the **`pin_id`** type would obviously be useful as a GPIO pin id parameter type in these cases its implementation occurred earlier than I originally anticipated.

The basic idea behind the **`pin_id`** type is that instances may be explicitly constructed from a suitable integer type, and may be converted back to an integer type. However during construction the range of the integer passed is validated to ensure it is in the required range. If it is not a **`std::invalid_argument`** exception is thrown. Hence, barring deliberate malice, if you have a **`pin_id`** object you know it represents a valid GPIO pin number. Only allowing explicit construction makes clear what the number represents.

To support using Raspberry Pi P1 or P5 connector pin values the base **`pin_id`** type is sub-classed to provide the logic for mapping from one value to another by the rather clumsily named **`rpi_version_mapped_pin_id`** which provides a single, rather cumbersome, constructor. This class is not meant to be used directly, rather its two sub-classes **`p1_pin`** and **`p5_pin`** are designed to be used directly and provide constructors that require only a connector pin number as parameter, which is passed along with a bunch of fixed values relevant for each connector to their base **`rpi_version_mapped_pin_id`** constructor. Note that this is a case where a base class destructor does *not* need to be virtual as by design all **`pin_id`** sub-types add no extra state that might require cleaning up. Once successfully constructed a **`pin_id`** or sub-type instance always represents a valid BCM2835 GPIO pin number value.

The **`rpi_version_mapped_pin_id`** type's constructor takes a 2-dimensional array mapping connector pin numbers to BCM2835 GPIO pin numbers for each supported major Raspberry Pi board revision – termed versions, along with dimension size values: the number of pins and number of versions supported by the mapping array. A major revision here is one that changes available connectors and/or GPIO pin associations with connector pins.

The implementation logic makes use of the **`rpi_info`** type that lives in the outer **`rpi`** namespace and provides an interface to obtain information on the Raspberry Pi system the code is running on. It currently only provides Raspberry Pi raw revision and version information, derived from the **`Revision`** entry in the Raspbian `/proc/cpuinfo` pseudo file. There is no formula to map revisions to versions and relies on published information [14]. Of course this means **`rpi_info`** code will require updating to take advantage of future major board revisions. However the mapping arrays used by **`p1_pin`** and **`p5_pin`** would also require updating, and of course support for any new connectors would need to be added. This is an area of the code base that is likely to change while adding support for all the various Raspberry Pi models now available.

To support referring to Raspberry Pi P1 and P5 connector pins by name I created a set of static global constant **`pin_id`** objects. These are created by specialisations of special static pin id type templates **`static_pin_id`**, **`static_p1_pin`** and **`static_p5_pin`** that contain no data providing just a single conversion operator member function that converts to a **`pin_id`**. Each template takes a single integer template parameter representing a connector pin or BCM2835 GPIO pin number. Each conversion function defines a static object of **`pin_id`** type or associated sub-type. For the P1 pins that changed between board versions and the P5 pins, if none are used then the **`rpi_info`** machinery should not be required. Those P1 pins that did not change between board versions are defined directly as their associated BCM2835 GPIO pin value by **`static_pin_id`** objects. The case for using **`static_pin_id`** objects over just defining global constant **`pin_id`** objects is not as great as for the use of **`static_p1_pin`** and **`static_p5_pin`** objects. The main benefit would be ensuring initialisation before use of the function-scope static **`pin_id`** object.

A future direction for **`pin_id`** *et al.* would be to investigate whether use could be made of constant expressions (**`constexpr`**).

Excuse me, is that pin free?

GPIO pins are a resource and as such I had to decide on an allocation policy. The easy part was deciding that a pin should not be permitted to be used by more than one object at a time within the same process. However the inter-process policy is not so easy. For a start there is no magic system wide registry of in use GPIO pins, so any policy implemented would have to rely on other processes doing likewise. There are various possible policies, including:

- Ignoring the issue and only track pin usage within a process
- Require pins to be marked as allocated beforehand by some other agent, and presumably marked as free after executing by a similar external agent
- Require that pins are marked as free for use before allocation and mark them as allocated/free as required.

How to determine whether a pin was in use or free was another choice. This boiled down to either devising some sort of custom registry which would not be too useful if only my library used it or use some pre-existing indicator that might also allow such in use/is free information to work between libraries.

The most obvious policy, and the one I decided on, was to follow that which I had used for the Python Raspberry Pi GPIO library: check to see if a pin was in use by checking to see if it was exported from the `/sys/class/gpio/` driver. If not then export it to acquire the pin and un-export the pin to release it. This scheme is in no way water tight. For a start any process with sufficient access rights can un-export an 'allocated' pin at any moment. Then there is the matter of un-clean exit from a process leaving pins marked as unavailable because the un-exporting de-allocations failed to run – the main problem here is with signals, especially **`SIGKILL`**, that cannot be caught and handled to force de-allocation before unceremonious exit.

it would be possible to create a specialist piece of hardware – a custom circuit board or similar – designed specifically to support the sort of testing performed by the interactive tests

This is all taken care of by the **`pin_allocator`** type. Or to be precise the **`pin_allocator`** type alias as it is a **`typedef`** for a specialisation of the **`pin_cache_allocator`** class template that provides the intra-process allocator logic by deferring decisions initially to an alternate allocator whose type is provided by the single template type parameter and caches the results. The **`pin_allocator`** type alias specialises **`pin_cache_allocator`** with the **`pin_export_allocator`** that provides support, with the help of a `sys` file system module, for allocating by exporting from `/sys/class/gpio/` and de-allocating by un-exporting.

It's a wrap!

The services of **`gpio_registers`** and **`pin_allocator`** types are wrapped up behind the public interface types **`ipin`** and **`opin`**. As mentioned previously the **`ipin`** and **`opin`** types provide an abstraction modelled after the C++ IOStream library. In particular those of the **`std::ifstream`** and **`std::ofstream`** types (or any other specialisation from the underlying basic-templates). Operations on single bits are quite limited compared to those on streams of characters so other than providing open and close semantics the only other operations from the IOStreams types that really applied were **`get`** for **`ipin`** to read the Boolean value of a GPIO pin and **`put`** for **`opin`** to write, or set, the Boolean value of a GPIO pin.

There is quite a lot of common functionality around the implementation of the open and close operations of **`ipin`** and **`opin`**. This was pulled out into a common base type **`pin_base`** having a totally protected interface. Following the **`std::ifstream`** and **`std::ofstream`** examples I originally provided explicit **`open`** and **`close`** operations which were called as appropriate by **`ipin`** and **`opin`** constructors and destructors. Later, having observed that a purer form of RAII that only allowed resources to be acquired through construction was proving to be a Good Thing™, I refactored **`ipin`**, **`opin`** and **`pin_base`** to remove the default constructors and the **`open`** and **`close`** member functions which considerably simplified the implementation.

Each **`ipin`** and **`opin`** instance needs to acquire a GPIO pin for use if available, access the GPIO register block and, of course, to release the pin when done. Unsurprisingly **`pin_id`** objects are used to pass around GPIO pin values. As they appear in the public interface of the public library types **`ipin`** and **`opin`**, **`pin_id`** and related types are also part of the library's public API.

Each BCM2835 has only one set of GPIO pins so there need only be one instance of the **`pin_allocator`** type. Likewise, they only have one GPIO peripheral register block hence there only needs to be a single instance of **`gpio_registers`** mapped to the relevant physical memory address as a **`phymem_ptr<volatile gpio_registers>`** (the type pointed to by the **`phymem_ptr`** specialisation is qualified **`volatile`** because the hardware at the mapped locations are *not* memory but device registers that have to be accessed in the prescribed way and so the compiler is not free to assume it knows what is going on).

So without getting overly sophisticated the obvious pattern to use here was SINGLETON (I'll wait until some of you have recovered your composure…).

For those wondering about systems using multiple BCM2835 chips that as they have their peripheral registers' memory locations hard-baked into the silicon such systems would not be feasible, GPIO and peripherals wise, without doing something exotic.

So what was needed was a singleton class containing a **`pin_allocator`** and a **`phymem_ptr<volatile gpio_registers>`** with the single instance being accessed via an **`instance`** member function that returned a reference to the function local static instance. Singletons of this form are sometimes known as Meyers Singletons [15] after advice given in Scott Meyer's book *Effective C++* [16]. In fact this type was implemented as a **`struct`** called **`gpio_ctrl`** that allows direct access to the singleton's allocator and GPIO registers members. The implementation was initially placed within, and local to, the implementation file for the **`ipin`**, **`opin`** and **`pin_base`** classes as there was at the time no need for anything else to access it. This changed later on and at that time **`gpio_ctrl`** was moved out into its own library-internal header and implementation file.

Testing, testing

Unit testing has already been mentioned in relation to the **`gpio_registers`** type. However, after a while the project settled into three classes of tests:

- Unit tests that relied on nothing else and could in theory be built and executed on another platform.
- Platform tests, a form of integration tests that required a Raspberry Pi and/or Linux/Raspbian operating system services such as `/dev/mem` or `/sys/class/gpio/`.
- Interactive tests, a form of platform integration tests that additionally require some user action. Usually this meant ensuring the hardware was suitably connected for the tests and often that the tester perform a requested action with the test hardware such as closing or opening a switch or confirming that the expected result such as a lamp lighting or turning off occurred.

Each class of test has its own executable. Unit tests do not require any special access to execute but the two integration test varieties generally require root access via **`sudo`** or similar – as do applications – including examples – built with the library.

Unit and platform tests can be run quickly as regression tests and could be run automatically. Interactive tests take a bit more time and care and obviously need to be run manually. The Catch test runner feature allowing the use of wildcards in the specification of which tests to execute is especially useful for only running the group of interactive tests that are currently of interest, especially as the hardware to support other interactive tests may not be wired up at the time.

It has occurred to me that it would be possible to create a specialist piece of hardware – a custom circuit board or similar – designed specifically to support the sort of testing performed by the interactive tests. It might even be possible to arrange for many tests to be performed and verified automatically. Such a device combined with the test software would I suppose be similar to ATE – Automated Test Equipment. It would however be quite involved and would almost certainly be overkill. ■

Standards Report

Mark Radford reports the latest from the C++ Standards meetings.

Hello and welcome to my latest standards report. Not only is it my latest report but, it will also be my last, because I'm not continuing as Standards Officer after the forthcoming AGM (that is, forthcoming at the time of writing, it will have happened by the time this report appears in print).

The next full C++ WG21 (ISO) committee meeting is not until May. It will be held from the 4th – 9th May in Lexana, KS, USA. Unfortunately the pre-Lexana mailing isn't yet available – and probably won't be by the time I have to submit this report for publication – so I won't be able to discuss it. The only thing that's happened since my last report is the meeting held in Cologne, Germany, in the final week of February, by the Library Working Group (LWG). Therefore, while I have a little information on C standardisation progress, the majority of this report will be concerned with the Cologne LWG meeting.

The Cologne LWG meeting

The Cologne meeting set out to review all the papers that make up the inputs into both the Concurrency TS and the Library Fundamentals TS v2. The objective being to move things along by having the papers reviewed and updated in time for the Lexana meeting. If this could be achieved, then further work on the papers at Lexana will be minimised or (ideally) not necessary at all. Note that there was much more on the agenda for Cologne. However, I can't report everything, so I'll focus on the Concurrency TS

MARK RADFORD
Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

Raspberry Pi Linux User Mode GPIO in C++ – Part 1 (continued)

References

[1] Raspberry Pi Linux User Mode GPIO in Python, *CVu*, Volume 27 Issue 1, March 2015
[2] Github repository for the Python Raspberry Pi GPIO library: https://github.com/ralph-mcardell/dibase-rpi-python
[3] Gertboard Raspberry Pi IO expansion board: http://www.raspberrypi.org/archives/411
[4] Available for download as the ZIP file gertboard_sw_20120725.zip: http://www.element14.com/community/servlet/JiveServlet/download/38-101479
[5] The Wiring Pi Library: https://projects.drogon.net/raspberry-pi/wiringpi/
[6] See for example: http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization
[7] Catch: C++ Automated Test Cases in Headers https://github.com/philsquared/Catch
[8] GitHub repository for the C++ rpi-peripherals library: https://github.com/ralph-mcardell/dibase-rpi-peripherals
[9] BCM2835 ARM Peripherals: http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf
[10] eLinux BCM2835 ARM Peripherals errata page: http://elinux.org/BCM2835_datasheet_errata
[11] Raspberry Pi Linux source code Github repository: https://github.com/raspberrypi/linux
[12] Raspberry Pi Linux kernel source BCM2835 specific support: https://github.com/raspberrypi/linux/tree/rpi-3.10.y/arch/arm/mach-bcm2835 (note: for the rpi-3.10.y branch, use branch selection dropdown to select another)
[13] Raspberry Pi Linux kernel source BCM2708 specific support: https://github.com/raspberrypi/linux/tree/rpi-3.10.y/arch/arm/mach-bcm2708 (note: for the rpi-3.10.y branch, use branch selection dropdown to select another)
[14] eLinux RPi HardwareHistory page: http://elinux.org/RPi_HardwareHistory
[15] See for example the second variant presented in: http://www.devartplus.com/3-simple-ways-to-create-singleton-in-c/
[16] See Scott Meyer's site at: http://www.aristeia.com/books.html

and Fundamentals TS v2 because these are listed in the meeting agenda as being the highest priorities for the week.

The inputs into the Concurrency TS consist of three papers: ‘C++ Latches and Barriers’ (D4281), ‘Atomic Smart Pointers’ (N4162), and ‘Improvements to the Concurrency Technical Specification (Working draft)’ (N4123). Inputs into the Fundamentals TS consist of six papers: ‘Generic Scope Guard and RAII Wrapper for the Standard Library’ (N4189), ‘Extending make_shared To Support Arrays’ (N3939), ‘Multidimensional bounds, index and array_view’ (N4346), ‘Const-Propagating Smart Pointer Wrapper’ (N4372), ‘make_array’ (N4315), and ‘Source Code Information Capture’ (N4129).

Concurrency TS

‘C++ Latches and Barriers’ (D4281) was a paper that received quite a lot of time. The version under discussion had the number N4392 at the top of it, so I’m assuming that’s the number it will go by in the pre-Lexana mailing, although it was discussed as the (not publicly available) draft paper D4281. This paper has been through a few revisions, the most recent to appear in a mailing being N4204 in the pre-Urbana mailing. D4281 focuses on wording for the Concurrency TS. Further, Concepts have been removed from this edition.

I’ll give a short summary of what latches and barriers are. A *latch* is a synchronisation object which maintains a counter and blocks all participating threads until the counter is decremented to zero, at which point all the threads are unblocked (see N4204 for an example). A *barrier* causes a number of threads to block until all threads catch up, at which point all the threads are unblocked. There are two types of barrier: `barrier` and `flex_barrier`. In the latter case, when all the threads catch up and before all threads proceed, one thread is released to execute the *completion phase*, i.e. code passed to the `flex_barrier`’s constructor in the form of a function object.

The group discussed this paper four times during the week, with some of the discussion being at a very detailed level. Also, the draft changed during the week because the authors were able to provide updates in response to comments passed on to them from the meeting. At the end of the Friday session it was decided to hand the paper back to SG1 (Concurrency and Parallelism) for them to review it once more with the recent changes, and to move it to ‘ready’ status assuming they are satisfied.

‘Improvements to the Concurrency Technical Specification (Working draft)’ (N4123) was discussed on Wednesday and Thursday. Much of the discussion centred on getting the wording right. Actually the paper being reviewed wasn’t strictly N4123, rather it was N4107 with N4123 merged in i.e. the meeting reviewed the working draft of the Concurrency TS plus the improvements. Unfortunately it is not clear to me, from the minutes of the discussion, what the actual conclusion was. However, it is listed as having ‘revisions expected’ on a document status page (on the WG21 internal wiki). I’m assuming this means a revised paper, which addresses issues raised in Cologne, is expected to be available for Lexana.

‘Atomic Smart Pointers’ was not discussed as it was moved to ‘ready’ in Urbana i.e. it was not necessary for the Cologne meeting to discuss it further.

Library Fundamentals TS

‘Generic Scope Guard and RAII Wrapper for the Standard Library’ (N4189) takes the well known RAII idea and generalises it. There are two variations: one just executes a function on destruction, while the other holds a resource for which it executes a ‘deleter’ on its destruction. The discussion of this paper that went into some detail. For example: the headers it describes are quite fine grained, whereas the library has traditionally preferred coarser headers. The reason for this is build speed: touching the file system will slow down the build. The conclusion was to have just one header. This may seem picky, but it is an example of the responsibility the standards committee has i.e. its decisions affect all C++ users. In any case there were several issues that lead the meeting to conclude that this paper is unlikely to be ready in time to make it into Fundamentals v2. The conclusion was to return the paper (with comments) to the Library Evolution Working Group (LEWG) for further work.

‘Multidimensional bounds, index and array_view’ (N4346) proposes library components to enable contiguous memory to be viewed as multidimensional. Note the more recent versions have concentrated on proposing formal wording without giving an overview. To get the overview you need to go back to an older version, N3851, which can be found in the pre-Issaquah (January 2014) mailing. Again, the discussion was detailed and several issues were raised. However, the conclusion was that the paper was in good shape to make progress at the forthcoming Lexana meeting, provided it is revised to address the issues raised in Cologne.

‘Const-Propagating Smart Pointer Wrapper’ (N4372) offers help with the problem that it is possible, in a `const` member function, to call non-`const` member functions on some (smart) pointer types (because C++ considers the underlying pointer type for treatment as `const`, not the object it points to). It proposes `propagate_const`, a wrapping smart pointer that has `const` and non-`const` `operator->` overloads, that return a `const` pointer and non-`const` pointer, respectively. This paper was discussed three times during the week: on Monday when some issues were raised, on Tuesday after the author (Jonathan Coe, who was at the meeting) had made some revisions, and again on Wednesday following further revisions. At the end of the Wednesday discussion the meeting concluded that the paper was ready to make progress at Lexana.

‘make_array’ (N4315) proposes a simple utility for constructing a `std::array` (the paper points out that we have `make_pair()` and `make_tuple()`, but we lack the analogous creation function for `std::array`). In this case the discussion was short and and conclusion was that this is another paper in good shape for Lexana.

‘Source Code Information Capture’ (N4129) is a proposal for a library class (well, struct) `source_context`: when `source_context` is constructed, by way of implementation ‘magic’, it gets information such as the line number, the file name and function name of its construction. The idea is to avoid having to sprinkle macros (e.g. `__FILE__`) throughout the code, in order to have this information available in the code for logging purposes, for example. This paper was not discussed in Cologne because the author (Robert Douglas, who was unable to attend) intends to revise the paper and present the revision at the Lexana meeting.

‘Extending make_shared To Support Arrays’ (N3939) was not discussed as the LWG is awaiting a further revision.

C standardisation

Before I finish, I have a little information on C standardisation progress.

There has been a slight hiccup with part 2 of their floating point TS. There were some staff changes at ISO, and this lead to confusion about the status of this TS, with the result that it was published in error. The published version has not been reviewed by the floating point study group, or by WG14. When this reviewing has been done, a corrected version will be published. Meanwhile parts 3 and 4 (of this TS) are going through the DTS ballot. An early draft of part 5 (N1919) is now available.

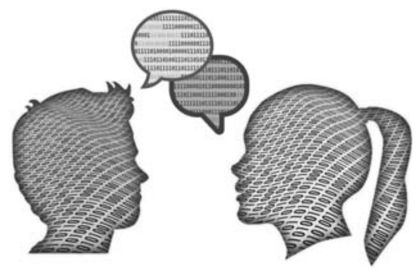
Finally

That brings me to the end of my final standards report. Thank you to those people who have provided me with support and feedback over the three years I’ve been writing these reports. I assume Jonathan Wakely, as the only candidate, will be elected to take over as Standards Officer. I wish him well and step down knowing the standards reports are in safe hands.

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

Code Critique Competition 93

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'm trying to use the new(ish) **`unordered_map.`** and my simple test program works with some compilers and options but not with others. I can't see why it wouldn't work consistently – are some of the compilers just broken? I do get a compiler warning about conversion from string constant to cstring – is that what's wrong?

Listing 1

```
#include <iostream>
#include <string>
#include <unordered_map>
typedef char *cstring;
typedef std::unordered_map<cstring, cstring>
AddressMap;
// Hold email addresses
class Addresses
{
public:
  // add addr for name
  void add(cstring name, cstring addr)
  {
    addresses[name] = addr;
  }
  // get addr from name or null if not found
  cstring get(cstring name)
  {
    if (addresses.count(name))
    {
      return addresses[name];
    }
    return 0;
  }
private:
  AddressMap addresses;
};

int main()
{
  Addresses addr;
  addr.add("roger",
    "rogero@howzatt.demon.co.uk");
  addr.add("chris",
    "chris@gmail.com");
  cstring  myadd = addr.get("roger");
  if (myadd == 0)
  {
    std::cout << "Didn't find details"
      << std::endl;
    return 1;
  }
  std::cout << "Found " << myadd << std::endl;
}
```

Can you explain what is wrong and help this programmer to fix (and improve) their simple test program? The code is in Listing 1.

Critiques

Tom Björkholm <accuml@tombjorkholm.se>

There are several problems with this code in Code Critique 92.

The main problem is a confusion about values and pointers. The way the code is written it stores pointers (addresses to memory locations) not string values in the unordered map. This is a fundamental problem. The C++ standard containers are designed to hold values not pointers.

More specifically the values stored in the container **`addresses`** are the pointers (memory address values) not the strings at those memory locations. This is fundamentally flawed as in any real program those memory locations would probably be used for other data later in the life of the program, causing the program to crash (or exhibit other strange behavior such as incorrect results).

So why does this work at all? Why doesn't this program crash at once? The reason is that only compile time constant strings are used as data in this example. The strings **`"roger"`**, **`"chris"`**, etc. in the main program are all compile time constant strings. These strings are stored by the compiler in a (usually read-only) data segment. Thus, in this example the data on the memory locations used, does not change.

The fact that the compiler is supposed to store these compile-time constant strings in read-only memory is the reason for the compiler warning. Having a non-**`const`** pointer to data that is a compile-time constant is not good. The warning can be fixed by changing

```
typedef char *cstring;
```

to

```
typedef const char *cstring;
```

With this change, there are no compiler warnings, but the program is still fundamentally broken as the key used for lookup in the map is the value of a memory address, not the name string.

So why does this produce the expected result on some compilers? Some compilers notice that the identical string **`"roger"`** appears both in one of the **`add()`** function calls and in the **`get()`** function call. The compiler is then free to store the string only once in memory, and use the same address (i.e. pointer value) in both function calls. However, the compiler is also allowed to store both strings **`"roger"`** in memory at different addresses. With a compiler that stores the string **`"roger"`** at only one memory location, the program produces the expected result. But that is because of a comparison of memory addresses used as keys, not because of a comparison on key strings.

The correct way to fix this program is to use **`std::string`** instead of character pointers. Then the values stored in the **`unordered_map`** are real string values, and the keys that are compared are the strings (not the memory locations).

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

The `get` function can also be improved. As it is written now the lookup is done twice, once to count the number of matching elements (zero or one) and once to find the element to return. (In general it is a good habit to avoid `count()` and `size()` if the task can be solved by using `empty()` or `find()` instead.)

When `std::string` is used for the keys and values, it will no longer be possible to indicate 'not found' as a null pointer. Not found can either be indicated using an empty string, or the `get()` method could return a `struct` with a string value and a `bool` flag.

Personally I would have used a `using` alias inside the `Addresses` class instead of a global `typedef`, but this is a matter of personal style preferences. To keep the program recognizable to the original author, I have opted to not make changes based on personal style preferences. I have opted to let the empty string indicate not found here (as no valid email address can be the empty string). The complete fixed program would then be:

```
  #include <iostream>
  #include <string>
  #include <unordered_map>
  typedef std::unordered_map<std::string,
          std::string> AddressMap;
  // Hold email addresses
  class Addresses{
  public:
    // add addr for name
    void add(const std::string & name,
            const std::string &addr){
      addresses[name] = addr;
    }
    // get addr from name or null if not found
    std::string get(const std::string & name) const
    {
      const AddressMap::const_iterator it =
          addresses.find(name);
      if (addresses.end() != it) {
          return it->second;
      }
      return "";
    }

  private:
    AddressMap addresses;
  };

  int main() {
    Addresses addr;
    addr.add("roger",
             "rogero@howzatt.demon.co.uk");
    addr.add("chris",
             "chris@gmail.com");
    std::string myadd = addr.get("roger");
    if (myadd == "") {
      std::cout << "Didn't find details"
                << std::endl;
      return 1;
    }
    std::cout << "Found " << myadd
              << std::endl;
  return 0;
}
```

Simon Brand <simonrbrand@gmail.com>

Note: Any standards references are from C++ draft N4296 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf).

You are coming across an issue because you are attempting to use `cstrings` as keys.

Why is this an issue? Because `cstrings` are simply pointers to a contiguous area of memory which is null-terminated. If you attempt to check two `cstrings` for equality using `operator==` you are checking if they point to exactly the same area of memory, not if the data they point to is the same.

To see why this causes an issue with `unordered_map`, we need to know a little about how it is implemented. An `unordered_map` is an implementation of a hash map (§23.2.5.9) (some nice explanations here: http://stackoverflow.com/questions/730620/how-does-a-hash-table-work), so it needs a hash function to convert a key into a bucket index, then a comparison function to resolve collisions where keys hash to the same bucket.

§23.5.4 outlines the `unordered_map` class template, which can optionally take in functors as template arguments to use as the hashing and equality functions. Now we can see that `unordered_map` uses `std::equal_to<Key>` as its default equality function and `std::hash<Key>` as the hashing function. Unless otherwise specialised, `equal_to` simply invokes `operator==`, and it has no specialisation for `char*` (§20.9.6). This means that whenever you try and perform a key comparison in the map, you are just comparing the pointers. `std::hash` is implemented such that if `k1 == k2`, `hash(k1) == hash(k2)` (§23.2.5.1.4), so the same applies. As such, using `char*` as the key type in `unordered_map` means that the keys must be pointing to the same part of memory.

So why does this work for some compilers and not for others? The answer is optimisation. Most compilers will see that you have used the same string literal twice and so will just use the same memory for any instances of that literal. In this example, both times you use `"roger"`, you could be getting pointers to completely different areas of memory, but the compiler optimises and just reuses the same data. Now when you hash or compare your keys, you just so happen to be passing the same pointer in due to a compiler optimisation, so you get the expected result. If you were, for example, reading the string in at runtime, this would fail every time on a conforming compiler.

By now you might be thinking "Why didn't I notice I was comparing pointers?". I think your problem is that you `typedef`'d `char*` to `cstring`. Hiding implementation details like that is often a good idea, but using raw pointers is often error prone, so you just make yourself complacent by trying to forget that you're using pointers.

In the case of `AddressMap`, this abstraction is a really good idea because if you want to change the implementation of your class to use ordered maps instead, you just need to change it in one place. A possible improvement to this would be:

```
  class Addresses
  {
  private:
    using map_type =
      std::unordered_map<cstring, cstring>;
  }
```

This is called a nested type name (§9.9) and is preferable to your version because it doesn't pollute the global namespace. Making it private ensures that you don't leak implementation details. Substituting the `typedef` for an alias declaration (§7.1.3.2) is mostly just for consistency here. This syntax is more powerful as it allows you to use templates in your type alias, which you can't do with `typedef`s. Because it provides a superset of functionality, I prefer to use it everywhere I'd usually use a `typedef` (Scott Meyers discusses this in Item 9 of his book *Effective Modern C++*).

I'll now outline a couple of ways to fix your code.

Version 1

As a preface to this fix, this is more an example for completeness of how you could use a `char*` in an `unordered_map`. As I'll show later, you should not do this and use `std::string` instead.

As stated above, the issue is that `unordered_map` compares your `char*`s by their pointer, not by the string they point to. In order to modify this, we can pass a hash and comparison function in to our map as either a template or constructor argument.

```
struct AddressHash
{
  //taken from Stroustrup's The C++
  //Programming Language
  size_t operator()(const char* ptr) const {
    size_t h = 0;
    while (*ptr)
    {
      h = h << 1 ^ *ptr;
      ++ptr;
    }
    return h;
  }
};

struct AddressComparison {
  bool operator()(const char* str1,
                  const char* str2) const {
      return strcmp(str1, str2) == 0;
  }
};
```

This all looks fine, but because we are using pointers, this will fail if we add a pointer in which is later invalidated. For example:

```
void addstuff(Addresses &addr)
{
  char rog[6] = "roger";
  addr.add(rog,
    "rogero@howzatt.demon.co.uk");
}
```

That call to **addr.add** will decay the **char[]** to a **char***, copy the pointer (not the data) and store it in the map. Then when the function exits, the **char[]** which holds the string is destroyed, so trying to access it through the pointer we saved is undefined. Oops! We could probably solve this issue by adding a level of abstraction and managing copies to these strings, but therein lies madness. If only there were a class which is like a **char*** but has sensible value semantics...

Version 2

I know, we'll use **std::string** instead!

std::string already has the comparison and hashing functions we need, so simply modifying your interface like so fixes most of our issues:

```
typedef  std::unordered_map<std::string,
         std::string> AddressMap;

class Addresses {
public:
  void add(const std::string &name,
    const std::string *addr)
  std::string &get(const std::string &name)
};
```

Another problem is the use of flag values for the return of **Addresses::get**. Returning a value which means an error has occurred and assuming that the client will remember to check for it is usually ill-advised. It's better to throw an exception so that they **need** to deal with it unless they want their application to crash and die. Fortunately, **unordered_map::at** already does this, throwing an **out_of_range** exception if the key doesn't exist. Now we can modify our **get** function like so:

```
std::string &get(const std::string &name)
{
  return addresses.at(name);
}
```

You should always ensure that your functions are named in a way which correctly reflects their semantics. The function **add** is maybe a bit misleading, because you could pass in a key which already has a value and it would be updated rather than adding a new entry. We could call it something like **addOrUpdate**, but there's already a more natural version:

```
std::string &operator[] (
  const std::string& key)
{
  return addresses[key];
}
```

Now we can update our code like so:

```
int main()
{
  Addresses addr;
  addr["roger"] =
    "rogero@howzatt.demon.co.uk";
  addr["chris"] = "chris@gmail.com";
  try
  {
    std::string &myadd = addr.get("roger");
    std::cout << "Found " << myadd
      << std::endl;
  }
  catch (const std::out_of_range& oor)
  {
    std::cout << "Didn't find details"
      << std::endl;
    return 1;
  }
}
```

We should now check our functions to ensure that they are const-correct, flexible and fast. Our **operator[]** takes a **const std::string&**, returns a **std::string&** and is not marked **const**. The lack of **const** is correct, because calling **operator[]** on an **unordered_map** adds a new key if one does not exist. Returning a **std::string&** is correct, because it allows us to update the object held in the map. Taking the argument as a **const** reference, however, can be improved. Imagine we are populating our map with data parsed from a large external file like this:

```
while (stillHaveData())
  addr[getNextKey()] = getNextValue();
```

The only use for the return value of **getNextKey()** is to store it in our internal map. It can't possibly be used anywhere else, because we don't store a reference or pointer to the data. Unfortunately, because **operator[]** is taking its argument as a **const&**, we are going to be copying every single key string into the map. This could get expensive; we'd much rather just reuse the data rather than copying it. To achieve this we can implement perfect forwarding (http://thbecker.net/articles/rvalue_references/section_07.html) on **operator[]**: if it is passed an **lvalue**, copy it; if it is passed an **rvalue**, move it.

```
template <typename T>
std::string& operator[] (T&& key)
{
  return addresses[std::forward<T>(key)];
}
```

This is much more efficient, as **std::forward** (§20.2.4) ensures that the reference type is preserved through our call, so the correct version of **std::unordered_map::operator[]** is called and the copy is avoided if necessary. We get this efficiency for free with the actual assignment, because **std::string** provides a move constructor which will be called when we attempt to assign an **rvalue**.

Now for our get function. It takes a **const std:string&**, returns a **std::string&** and is not marked **const**. We want to be able to check the contents of the map when we have a **const Addresses** object, but also to use it to update values when it's non-**const**, so we need a **const** and non-**const** version of the function. Taking the argument by **const&** is fine here, as we aren't going to be doing a copy. We could optimize by using **std::experimental::string_view**, and you should certainly have a look at that class, but you can be forgiven for waiting until it's actually in the standard. Returning by **std::string&** is correct as we want to be able to update our value if possible, but it needs to be a **const&** when using a **const Addresses**. Now the function looks like:

```
const std::string &get(
  const std::string &name) const
{
  return addresses.at(name);
}
std::string &get(const std::string &name)
{
  return addresses.at(name);
}
```

Now the final version of our class:

```
class Addresses
{
public:
  template <typename T>
  std::string &operator[] (T&& rhs)
  { return addresses[rhs]; }
  const std::string &get(
    const std::string &name) const
  { return addresses.at(name); }
  std::string &get(const std::string &name)
  { return addresses.at(name); }
private:
  using map_type = std::unordered_map<
     std::string, std::string>;
  map_type addresses;
};
```

Jim Segrave <jes@j-e-s.net>

The problem of compiler errors is trivial to fix – string literals have the type **`char const *`**, but the **`add()`** member function expects arguments of **`char *`**. A **`char const *`** can only be used with a cast. Changing the **`typedef`** of **`cstring`** to be **`typedef char const *`** will stop the compiler errors and the program will work.

But there are larger problems.

The keys to the map are pointers to **`char`** arrays. Two separate character arrays with identical contents will produce two separate entries in the map. For example, a **`get()`** for the name **`"roger"`** which uses a different character array to hold the text **`"roger"`** than the one used for inserting will not find the entry. The key needs to be the actual value, not a pointer to the value. Further the key has to remain valid and unchanged as long as the map exists, which is not a guarantee that a pointer can make.

The values are also stored as pointers to character arrays, which opens new opportunities for failures. What happens if the array passed as a value for the address goes out of scope? You now have an invalid pointer in the map and if you do a lookup and get that pointer returned, any use of it leads to undefined behaviour. What happens if you look up an entry and keep the pointer to the address character array and that array goes out of scope? You now have a dangling pointer which is an invitation to undefined behaviour.

My updated version stores a copy of the text of the name, so the lifetime of the name passed to the **`add()`** member function no longer affects the map. Since I'm copying the text, using a **`std::string`** instead of a character pointer adds flexibility – the name can be a string literal, a **`char`** array or a **`std::string`**.

I changed the map so that the address stored is a **`shared_ptr`** to a newly allocated copy of the string originally passed to the **`add()`** member function. Now if the string passed to **`add()`** is deleted, the contents of the map are still valid. If you use **`get()`** to obtain a pointer to an entry in the map, that pointer will remain valid even if the map itself is deleted. Only when the map is deleted and all the pointers to data that was in the map are deleted do all the address strings get removed, but you don't have to do the bookkeeping to make that happen.

In order to test the map fully, I added a function to exercise the map. It uses an array of **`struct`**s holding a name and address, so adding additional test cases requires simply inserting a {name, address} braced pair into the array. It inserts all the pairs from the array, then looks up each name in the array to see that the address data is correct. It correctly flags when a name is entered twice with different addresses.

It then looks up a name known not to be in the map to test the not-found behaviour. It also deliberately updates an entry with an address taken from a local variable which will go out of scope when the function returns. In **`main`**, that entry is looked up and the address is checked to see that it matches what was in the local variable.

My updated version

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <memory>
using std::string;
using shp = std::shared_ptr<string>;
typedef std::unordered_map<string, shp>
  AddressMap;
// Hold email addresses
class Addresses
{
public:
  // add addr for name
  void add(const string& name,
    const string& addr)
  {
    addresses[name] =
    std::make_shared<string>(addr);
  }
// get shared ptr to addr from name or null
// if not found
shp get(string name)
{
  if (addresses.count(name))
  {
    return addresses[name];
  }
  return shp(nullptr);
}

private:
  AddressMap addresses;
};

// pairs of names and addresses for testing
struct test_val {
  string name;
  string addr;
};
test_val tests[] = {
  {"roger", "rogero@howzatt.demon.co.uk"},
  {"chris", "chris@gmail.com"},
  // duplicate name with different address
  {"roger", "rogero@gmail.com"},
};

// test insertions and lookups, including a
// deliberate lookup failure
// Update an entry with local auto strings for
// the name and address
void test_map(Addresses& a) {
  // remember longest name so we can create a
  // name known not to be in the map
  string longest = "";
  for(auto test: tests) {

  if(longest.size() < test.name.size()) {
      longest = test.name;
  }
  a.add(test.name, test.addr);
}
```

```
  // check that all the names have been
  // inserted correctly
  // (the duplicated name will generate an
  // error report)
  for(auto test: tests) {
    shp addr = a.get(test.name);
    if(addr == nullptr) {
      std::cout << "Insert of " << test.name
                << " failed" << std::endl;
    }

    if(*addr != test.addr) {
      std::cout << "Expected lookup of "
                << test.name
                << " to return " << test.addr
                << " but returned "
                << *addr << std::endl;
    }
  }
  // update an entry using a local auto
  // variable
  string new_val{"auto_variable"};
  string new_name{"test_auto"};
  a.add(new_name, new_val);

  // generate a name known not to be in the
  // map
  if(a.get(longest + "x") != nullptr) {
    std::cout << "Map reported that "
              << longest << " is in the map, "
                 "although it should not be"
                 << std::endl;
  }
}

int main()
{
  Addresses addr;
  test_map(addr);
  // at this call to addr.get, the string used
  // to set the address is gone
  shp check = addr.get("test_auto");

  if(check == nullptr) {
    std::cout << "Failed to find entry for"
                 " \"test_auto\""
              << std::endl;
  }

  if(*check != "auto_variable") {
    std::cout << "Expected lookup of"
                 " \"test_auto\" to be "
              << "\"auto_variable\""
              << " but got "
              << *check << std::endl;
  }
}
```

James Holland <James.Holland@babcockinternational.com>

I can understand the programmer being baffled by the program behaving differently when using different compilers. After all, although it may not be perfect, the source code seems reasonably well constructed and should work as expected. So what is going on? Before we dig deeper, let's get rid of the warning message issued by some compilers.

Literal strings such as `"roger"` are considered to be constant and, therefore, should be incapable of being modified. The warning is saying that a pointer of type `char *` is being assigned the address of the first character of a constant string thus allowing the string to be modified by use of the pointer. This makes a mockery of the fact that the string is meant to be constant. Such assignments have been deprecated since C++98 and so compilers should, at least, issue a warning. Sadly, not all do. What is required is a pointer that is incapable of changing what it is pointing to. This is easily achieved, in this case, by adding a `const` to the declaration of `cstring` giving `typedef const char * cstring`. This will keep compilers happy.

Unfortunately, removing the warning has not altered the behaviour of the code. It still produces different results on different compilers. So something else must be causing the problem. It turns out to be a combination of the type of the unordered map key and the way in which some compilers optimise the code. Let's start with the key type.

From the sample code, it can be seen that `Addresses` member functions `add` and `get` have parameters of type `cstring` where `cstring` is a `typedef` for `const char *` (the `const` has just been added as described above). The strings that are passed to the functions are of type `const char[n]` where `n` is the length of the string. For example, `"roger"` is of type `const char[6]`. This difference in type is permitted because array types decay into pointers as they are assigned to the function parameters. The body of the functions, therefore, deal with pointers to strings.

Also, the unordered map has been defined to have both key and value of type `cstring`. In other words, the key is a pointer to a string and not the string itself. This is important because the unordered map determines whether an entry already exists by discovering whether any of the stored pointers have the same value as the address of the string passed (in our case) to its `operator[]` function. The address of a string is simply the location in memory where it is stored. So, the question is at what memory location is a string constant stored. This brings us back to compiler optimisations.

One compiler optimisation technique is called string pooling. This involves placing only one copy of identical string constants in memory instead of having multiple copies of the same string. If this optimisation takes place, p and q in the following code would be equal in value.

```
const char * p = "This is a string";
const char * q = "This is a string";
```

If this optimisation is not performed, the value of `p` and `q` would be different as two separate, but identical, strings would be stored.

This optimisation is critical to the program as to whether strings will be found. If string pooling takes place, the string `"roger"` used in the add function will have the same address as the identical string used in the `get` function. The unordered map's count function will, therefore, consider the string as found. If string pooling does not take place, the two strings, despite being identical in value, will have different addresses and so a match will not be found.

Incidentally, and ignoring the identified problems for a moment, there is an inefficiency in the `get` function. Two searches of the unordered map are made; one in the `count` function and one in the `operator[]` function. The hash calculation is, therefore, performed twice. Clearly, it would be more efficient if the hash value was calculated once. This can be achieved by using the unordered map's `find` function as follows.

```
cstring get(cstring name)
{
  return addresses.find(name) ==
    addresses.end() ? nullptr : name;
}
```

Clearly, it is not acceptable for the behaviour of the program to depend on compiler optimisations. Fortunately, there are a few things that can be done to remedy the situation.

As stated above, the unordered map compares the value of pointers to strings to determine whether strings are identical and (as also explained above) this only works when string pooling takes place. What is really wanted is to compare the strings themselves, not the pointers to the strings. Doing this would result in the software behaving as expected irrespective of whether string pooling was in effect. This can be achieved by providing a user-defined hash function and equivalence criterion. These take the form of function objects that are passed to the constructor of the unordered map.

A suitable function object that defines the hash function is shown below.

```
class String_hash
{
  public:
  std::size_t operator()(cstring s) const
  {
    return std::hash<std::string>()(s);
  }
};
```

Providing a good hash function can be difficult so I have cheated somewhat by using the hash function supplied by the standard library. The library provides hash functions for most common types. In this case I have used **`std::string`**. The constructor of **`std::string`** takes the pointer to a **`char`** and creates an **`std::string`** object that is passed to **`std::hash`** from which the hash code is generated.

A suitable equivalence criterion function object is shown below.

```
class String_equal
{
  public:
  bool operator()(cstring s1, cstring s2)
    const
  {
    return strcmp(s1, s2) == 0;
  }
};
```

The **`strcmp`** function takes a pointer to each of the strings to be compared and returns a value of zero if the two strings are equal. This value is compared with zero to give a return value of **`true`** if the two strings are identical and **`false`** otherwise. Note that the **`strcmp`** function is made available by adding **`#include <cstring>`** to the program.

The two function objects are passed, as template arguments, to the unordered map constructor as shown below.

```
typedef std::unordered_map<cstring, cstring,
String_hash, String_equal> AddressMap;
```

The program will now work as expected and does not depend on any compiler optimisation techniques. It has, however, been quite an effort to get to this stage and the modifications were not all that intuitive. There must be a simpler way of achieving our goals.

The problem lies in not choosing language features that provide the right level of abstraction. The student programmer has chosen **`cstring`**s (as provided by the original C programming language) to represent ordinary textual strings. While these can be used, they do have disadvantages. One problem is that they do not behave quite as the novice may expect. For example, I am sure the student was under the impression that the strings themselves were being stored in the map. This illusion may have been reinforced by naming the first deftype **`cstring`**. The name implying, perhaps, that strings are being stored, whereas the deftype actually refers to a pointer to **`char`**s. Also, the student included the unnecessary library header **`<string>`** thus giving the impression that strings were being manipulated.

The standard library provides a string class (**`std::string`**) that is designed to be intuitive and to present few or no problems to the user. It is this class that should be used in place of **`cstring`**s. The **`std::string`** class is a more appropriate level of abstraction for this application (and just about all others).

Rewriting the program to use **`std::string`**s gives the following.

```
#include <iostream>
#include <string>
#include <unordered_map>

typedef std::unordered_map<std::string,
  std::string> AddressMap;
class Addresses
{
  public:
  void add(std::string name, std::string addr)
  {
    addresses[name] = addr;
  }
  std::string get(std::string name)
  {
    return addresses.find(name) ==
      addresses.end() ? "" : name;
  }
  private:
  AddressMap addresses;
};

int main()
{
  Addresses addr;
  addr.add("roger",
    "rogero@howzatt.demon.co.uk");
  addr.add("chris", "chris@gmail.com");
  std::string myadd = addr.get("roger");

  if (myadd == "")
  {
    std::cout << "Didn't find details"
      << std::endl;
    return 1;
  }
  std::cout << "Found " << myadd << std::endl;
}
```

Making the interface of **`Addresses`** as similar as possible to the original has resulted in slightly awkward code in the **`get`** function. The unordered map's **`find`** function returns an iterator that is used to determine whether the string was found. If the iterator has a value of 'one passed the end' (indicating that the string was not found), a null string is returned from **`get`**; otherwise the located string is returned. In the **`main`** program, the string returned from **`get`** is compared with a null string to determine whether the email details were found. Despite this, the software is easy to understand and behaves as expected regardless of compiler optimisations; something that could not be said for the original version.

Paul Floyd <paulf@free.fr>

The non-answer which fixes the warning is to change the **`cstring typedef`** to use **`const char*`**.

The fundamental issue is that the **`std::unordered_map`** does not have a hash function that hashes the string content of pointers to character arrays (C strings if you prefer). It will hash pointers and **`std::string`**s. So in this case, it is the string pointer that is being hashed. Assuming that it's the C string contents that you want to use as the key, then this will work as long as there is a 1:1 relationship between the pointer and the string. If string literals (as in this case) are de-duplicated, then it will work. This is not guaranteed, resulting in implementation defined behaviour. I did try this with a few compilers (and debug/optimized modes), and it worked in all cases.

To make this work reliably, use a **`std::string`**, which does have a hash function defined for it, e.g.,

```
typedef std::unordered_map<
  std::string, std::string> AddressMap;
```

I don't think that the **`add`** function serves much purpose.

The **`get`** function does avoid inserting/returning a default element as **`operator[]`** would do. However, it could also be modified to avoid two lookups:

```
std::string get(const std::string& name)
{
  AddressMap::const_iterator citer =
     addresses.find(name);
```

```
    if (citer != addresses.end())
    {
      return citer->second;
    }
    else
    {
       return std::string();
    }
  }
```

Incidentally, I don’t like inline bodies in class definitions. I always try to put them outside as ‘inline’.

It isn’t clear to me whether empty strings should be allowed in the map. If so, this would cause problems as here an empty string is being used to test whether the key was found in the map. Here the **add** function could be useful, by not allowing empty strings to be added.

Testing wise, at least one test that is expected to fail should be added.

Commentary

This problem amused me because the code worked because of string pooling – this is a well known problem with Java code (where using **==** on strings has the same sort of issues as in this case) but less common in C/C++!

I don’t think there is much left to add to the comments made in the critiques above. I was interested to notice that after, converting the solution to use **std::string**, two of the entries check for failure of **get()** by using **if (value == "")** – I feel in this case it is more readable to use **if (value.empty())** but ‘your mileage may vary’.

I was a little surprised that none of the entries added any input validation to the **add** method – in particular it would seem sensible to prevent adding the ‘not found’ value!

The winner of CC92

There were some good critiques in this batch and it wasn’t easy to decide which one was best. Nearly everyone explained about the need for adding **const** to the **typedef**, but then explained that this wouldn’t solve the underlying issue. I particularly liked Tom’s well-placed reminder that the ‘standard containers are designed to hold values not pointers’ as this gives the programmer a higher level explanation of what the issue is.

Jim’s approach of using a **shared_ptr** was quite elegant, and nicely solves the problem of reporting failure from **get**, but I fear this makes the solution harder to use. Is in this case an empty email address seems a perfectly good way to indicate ‘not found’. I liked Simon’s approach to handling failure in the **get** function by using the **at** method in **std::unordered_map**, this also allowed him to support **const** correctness which was something other solutions didn’t explain (even when **const** was added to a signature.)

Overall I felt Simon’s solution was the most complete (although it did omit explaining the compiler warning) and so I have awarded him the prize against fairly stiff competition. Thanks to all the entrants!

Code Critique 93

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

> I’m trying to write a simple program to demonstrate the Mandelbrot set and I’m hoping to get example output like this Figure 1, but I just get 6 lines of stars and the rest blank. Can you help me get this program working?"

Can you find out what is wrong and help this programmer to fix (and improve) their simple test program? 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.

Figure 1

```
                                     *
                                   ****
                                   ****
                                   ****

                             *  **********
                            ******************
                             *****************
                            *****************
                           *******************
                          **********************
                          *********************
                *  **     **********************
                *******  **********************
               ********* **********************
               ********* **********************
             * ********* *********************
*********************************************
             * ********* *********************
               ********* **********************
               ********* **********************
                *******  **********************
                *  **     **********************
                          *********************
                          **********************
                           *******************
                            *****************
                             *****************
                            ******************
                             *  **********

                                   ****
                                   ****
                                   ****
                                     *
```

Listing 2

```
#include <array>
#include <complex>
#include <iostream>

using row = std::array<bool, 60>;
using matrix = std::array<row, 40>;

std::ostream& operator<<(std::ostream &os, row
const &rhs)
{
  for (auto const &v : rhs)
    os << "* "[v];
  return os;
}
std::ostream& operator<<(std::ostream &os, matrix
const &rhs)
{
  for (auto const &v : rhs)
    os << v << '\n';
  return os;
}

class Mandelbrot
{
  matrix data = {};
  std::complex<double> origin = (-0.5);
  double scale = 20.0;
public:
  matrix const & operator()()
  {
    for (int y(0); y != data.size(); ++y)
    {
      for (int x(0); x != data[y].size(); ++x)
```

Bookcase

The latest roundup of book reviews.

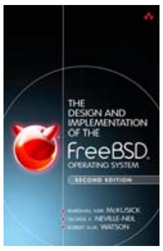
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.

Astrid Byro (astrid.byro@gmail.com)

The Design and Implementation of the FreeBSD Operating System (2nd Edition)



By McKusick, Neville-Neil and Watson, published by Addison Wesley, ISBN: 976-0-321-96897-5

Reviewed by Alan Lenton

There are some books that the word 'comprehensive' doesn't even come near to describing. This is one such book! If you want to know the details of any part of the FreeBSD operating system then this, together with the source code, is the reference book for you.

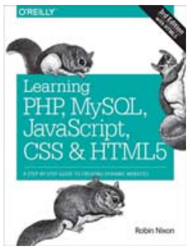
The book doesn't just cover the workings of the kernel, it also goes into details of the I/O systems, IPC, and startup/shutdown (`init`, of course. If you want the newfangled, monolithic, `systemd`, you need to look elsewhere). I found the IPC section particularly useful. I have other books that cover the issue, but I found the exposition in this book very clear and in depth.

'Since FreeBSD is a 'nix, much of what is in this book is relevant to other variants of Unix, and while an application programmer might not need to know what's going on under the hood, any more than you need to be a car mechanic to drive a car, it certainly helps to write efficient programs.

For someone studying operating systems at college, this book should be high on the 'must have' list, you would have to buy several other books to cover the topics in the depth it does. Even then, the coverage wouldn't have the cohesiveness this book has.

All in all, I would definitely recommend this book to anyone with an interest in modern Unix operating systems. You may be able to get cheaper books, but you won't get one that's so comprehensive!

Learning PHP, MySQL, JavaScript, CSS & HTML5 3rd Edition



By Robin Nixon

Reviewed by Ian Bruntlett

The 4th edition of this book has now been published.

This is a big book – both in size (676 pages plus index) and content – 3 programming languages and 2 markup languages. It has taken me fairly intensive reading since June to finish this book. Whilst it is pretty much self contained, I found that I needed to already have systems administration and HTML experience. Which I had and augmented with some questions answered for me on accu-general. I feel that the book would have better served the reader if it had listed key reference websites for the systems covered by this book.

A lot of semi-colons are involved with the programming languages. I think Shakespeare had it nailed when he wrote: "Some are born with semi-colons, some achieve semi-colons and some have semi-colons thrust upon them".

For those not familiar with LAMP (Internet) development, there is server stuff (Administration, Apache, MySQL) and client side stuff (JavaScript, HTML and CSS). Most of which is covered by this book.

I have found that my intention to read this book seven days a week was a bit misguided. I discovered that if you don't take time off, your body will force you to take time off.

The initial chapters cover a bit about HTTP, setting up a development server using the Zend Server install files. I use Ubuntu these days so I much preferred to use its built-in package

Code Critique Competition 93 (continued)

Listing 2 (cont'd)

```
      {
        std::complex<double>
          c = (xcoord(x), ycoord(y)), z = c;
        int k = 0;
        for (; k < 200; ++k)
        {
          z = z*z + c;
          if (abs(z) >= 2)
          {
            data[y][x] = true;
            break;
          }
        }
      }
    }
    return data;
  }
private:
  double xcoord(int xpos)
  {
     return origin.real() + (xpos -
       data[0].size()/2) / scale;
  }
  double ycoord(int ypos)
  {
     return origin.imag() + (data.size()/2 -
       ypos) / scale;
  }
};
int main()
{
  Mandelbrot set;
  std::cout << set() << std::endl;
}
```

management system instead. I managed to get it to work with that.

The PHP section spans five chapters (Introduction. Expressions and Control Flow, Functions and Objects, Arrays, Practical things).

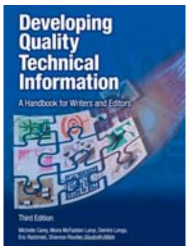
The next two chapters cover MySQL (Introduction, Mastering). My go-to facility for databases is no longer `fopen`, `fseek`, `fread`, `fclose`. There are a couple of appendices on MySQL as well (FULLTEXT stop words, a selection of MySQL functions) .Chapter 10 covers accessing MySQL using PHP. This is where this book earns its keep. Chapter 11 covers using MySQLi but the book's website has an updated chapter 11 which tells you how to use the MySQLi PHP object oriented system in preference to the older system.

Most of the remaining chapters of this book cover client side stuff (HTML forms, Cookies, Sessions and Authentication, JavaScript (Exploring , Expressions and Control flow, Functions, Objects and Arrays). There is a chapter that brings JavaScript and PHP validation together, followed by an Ajax chapter, CSS chapters and a guide to new things in HTML5.

Finally, the book has a "Bringing it all together" chapter which implements a social networking site.

Conclusion. I have gone from a person who looks at websites and asked "How do they do this?" to "A-ha!, I know how to do that!". Once I got going, I really enjoyed this book. Highly Recommended.

Developing Quality Technical Information: A Handbook for Writers and Editors, 3rd Ed.



By Michelle Carey, Moira McFadden Lanyi, Deirdre Longo, Eric Radzinski, Shannon Rouiller, Elizabeth Wilde, published June 2014, ISBN: 9780133118971, 587 pages

Reviewed by Paul Floyd

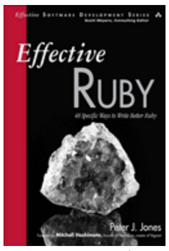
Recommended

Well, I'm neither a writer (the odd thing for the ACCU journals apart), nor an editor. Nevertheless, I would recommend this book for anyone who writes software that has any sort of user interface. I was expecting a fairly dry text, reflecting the dry text that is most technical documentation. That isn't what I found. I had jumped to the conclusion that 'Technical Information' equates to 'User Guide and Technical Reference'. This book does cover 'Technical Information' in the widest sense. Early in the book, the authors state that the large majority of users don't read the manuals, and either ask colleagues or search for information on Google. So in order to convey the information required to use software a significant part of this book covers GUI design and usability (with the emphasis on style, consistency, progressive disclosure, tooltips and contextual help). Another chapter that is 'modern' and far from dusty printed manuals is one covering search and information retrieval.

The tone is not prescriptive. There are a lot of guidelines, and many examples, in particular examples of poor interfaces and documentation followed by one or two improved versions, with the accompanying text pointing out the problems in the original and why the changes improve things. There is a fairly strong emphasis on conveying information in a way that is task oriented and minimalist. The final chapter on testing and reviewing was a nice rounded ending to the book.

There are a couple of chapters on clarity and style that fell into the classic tech writing stereotype that I half expected, but even these were concise and quite pleasant to read. Altogether a thoroughly professional piece of work.

Effective Ruby: 48 Specific Ways to Write Better Ruby



By Peter J. Jones, published by Addison-Wesley, 211 pages, ISBN-13: 978-0-13-384697-3

Reviewed by Simon Sebright

Highly recommended

This is another book in the Effective series, started by Scott Meyers and lives up to that tradition well. Material is presented in 48 'Items', each of which is a nice readable and understandable chunk from just over a page to several pages in length. Items are well-grounded and technically proficient. You can dip in and out – there is no need to read them strictly in order.

The items are grouped into the following Chapters (themes):

1. Accustomising Yourself to Ruby (things like equality, constants and warnings)
2. Classes, Objects, and Modules
3. Collections
4. Exceptions
5. Metaprogramming
6. Testing
7. Tools and Libraries
8. Memory Management and Performance

The book is aimed at people who already know ruby to a reasonable level, it is not an introduction to the language or for beginners in programming. I fitted into this category and found the book not only very readable (well written and with a good sprinkling of humour), but also extremely informative and useful in how I should use the language.

Although there is a chapter on tools, this book will not introduce to Rails, or any of the bigger topics – for that you'll need to find another book!

I highly recommend this book to anyone using Ruby regularly who cares about how they write software and wants to get better at it.

SOA With Java: Realizing Service-Orientation with Java Technologies

By Thomas Erl, Andre Tost, Satadru Roy, Philip Thomas, edited by Thomas Erl, published by Prentice Hall, ISBN-13: 978-0-13-385-903-4

Reviewed by Neil Youngman

SOA with Java bills itself as "The definitive guide to building service oriented solutions with lightweight and mainstream Java technologies". The foreword states that this is a self contained book, suitable for a complete novice, but according to the introduction it assumes a basic knowledge of fundamental service orientation. While the pre-requisites are not entirely clear, this book should not be approached without a basic understanding of Java and XML.

There are a large number of SOA related standards defined for Java and without a good guide it can be quite hard to understand the relationship between the many different standards. SOA with Java tries to walk the reader through all the major standards, giving their historic context and examples. It also introduces the Glassfish, WebSphere and Weblogic platforms. As it only has 400 pages, excluding appendices, to achieve this, it is quite a challenging goal.

The writing is quite dense, but mostly clear. At times it can be too abstract and jargon laden and the jargon is not always explained beforehand, but, if you stick with it most of the jargon is explained eventually.

When discussing specific technologies there are plenty of examples, however they are quite brief, and running through the book are case studies, which I assume are imaginary, which also help to illustrate the intended use of the technologies.

Chapters 1 and 2 introduce you to the book and the case studies. Chapters 3 to 6 make up Part 1I– fundamentals and run through terminology, Java and XML standards and APIs, and the basics of SOAP and REST technologies. Part II – Services consists of Chapters 7 to 9 and takes the reader through service orientation principles and various different types of service. Part III – Service Composition and Infrastructure contains chapters 10 to 12 and covers task services, service composition and the use of Enterprise Service bus technologies. Finally the appendices are grouped as Part IV.

Overall this book seems to me to offer a good overview of the SOA standards for Java. It doesn't provide sufficient detail to be a reference for any of the Java SOA standards and frameworks, but it gives a good overview which could be a good starting point for selecting technologies of interest for further investigation.

View from the Chair

Alan Lenton
chair @accu.org

This is quite a difficult chair's report to write – because of the lead time on the production of *CVu* I'm writing it in the week before Conference and the ACCU AGM. Sometimes things just don't work out conveniently!

Looking at things in a wider context, it's interesting to note that March this year marked the 30th anniversary of the publication the GNU Manifesto by Richard Stallman. Its publication drew a line in the sand, on one side open source software, on the other proprietary software.

The rest of us have had to make compromises on this over the succeeding 30 years, but Stallman never backed down. As he said at the time in *Dr Dobb's Journal*:

> I consider that the Golden Rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement.

The deed followed the word and Stallman went on to set up the Free Software Foundation, which brought us all those goodies like the gcc compiler suite, emacs (no, I'm not going to get into an argument about whether Vi is better), and the Bash shell.

When in 1992 Linus Torvalds released the Linux kernel under the GNU GPL, the stage was set for an open source operating system running on cheap commodity hardware that would completely change the nature of the server market, and the way people thought about and used open source.

And the rest, as they say, is history.

As the Grateful Dead so aptly put it "Lately it occurs to me what a long, strange trip it's been."

I wonder what the next 30 years is going to be like?

Bookcase (continued)

Java SE 8 for Programmers (Deitel Developer)

By Paul Deitel and Harvey Deitel, published by Prentice Hall, ISBN-13: 978-0-133-89138-6

Reviewed by Stefan Turalski

Sooner or later most coder-readers will bump into a work by Paul and Harvey Deitel (Deitel & Associates Inc). The father and son duo published a few dozens of books and recently ventured into creating video tutorials, focusing on a range of mainstream programming languages (usual mix of C, C++, C#, Java, Visual Basic with an odd publication on Python, Perl, even Swift). The target reader of Deitels' 'for programmers' book series is a 'programmer with a background in high-level language programming'. A person willing to invest time in going over a 1000-pages volume covering every aspect of a given language from fundamentals into intermediate subjects, studying numerous listings of small programs which illustrate core concepts, bare UML diagrams etc. In other words, someone, who is comfortable with a pre-Internet era style textbooks and learns best from this type of publication.

With the above prelude, I recommend the *Java SE 8 for Programmers* as a rather good book, especially for me, a developer working with C-languages for the last decade or so, who dabbling in Hadoop and Spark needs to catch up with developments in Java camp. As we are dealing with the 3rd edition of the book, the fundamentals are covered clearly and concisely and I found no issues in that department. Therefore, I will focus on what might interest ACCU readers the most – coverage of Java 8 futures. Regrettably, that is where the book falls a bit short.

I could forgive few chapters on Swing and limited focus on JavaFX, after all probably most developers are still supporting older applications. Similarly, I would not expect to see coverage of Oracle Nashorn (a new JavaScript engine within JVM) in an intermediate level book. In fact, new features, such as lambdas and streams got appropriate coverage, concurrent collections got mentioned (albeit very briefly), and so did parallel sorting. However, a new date-time API was introduced only in passing, which is rather strange as the previous API is now marked as deprecated. Deitel admits that concurrency is best left to experts, therefore I would not expect to see stamped locks (which allow writing really fast code, but may also lead to writing a self-deadlocking threads). However, in my humble opinion, the new concurrency adders (`LongAdder`) should be featured as these simplify complex code. Similarly, I was nicely surprised by the attention given to `SecureRandom`, which seriously simplify random number generation for non-critical purposes, yet I could not find adequate coverage of the new optional references (`Optional<T>`), and I do not recall new 'exact' methods on the Math interface (which will throw exceptions if the results overflow).

Despite these few rather minor shortcomings, *Java SE 8 for Programmers* is a serious offer, less dry than the Java SE 8 edition of *The Java® Language Specification* by Gosling, Joy, Steele, Bracha, Buckley or the 9th edition of *Java The Complete Reference* by Schildt. Still, I would recommend the last position, if you were looking for a reference that will serve you for a bit longer. Additionally, if your goal is to just get up to speed with Java 8, I would suggest Horstmann's *Java SE 8 for Really Impatient*, Warburton's *Java 8 Lambdas* if you are after the functional programming perspective as well, Liguori&Liguori's *Java 8 Pocket Guide* if you are after, well… pocket guide, and finally Urma, Fusco and Mycroft's *Java 8 in Action*, which is a really serious book if you are willing to dive a bit deeper.

Nonetheless, if you are in the market for an intermediate Java book that will introduce you to modern Java in an informative and engaging style, and you are ready to just skim a first few introductory chapters, the *Java SE 8 for Programmers* is a really good textbook; I can fairly recommend it.