the magazine of the accu

www.accu.org

{cvu}

Volume 26 • Issue 4 • September 2014 • £3

Features

Regulars

{cvu}
Volume 26 Issue 4
September 2014
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Pete Goodliffe,Paul Grenyer, Matthew Jones, Chris Oldwood, Roger Orr, Mark Radford, Alex Paterson

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

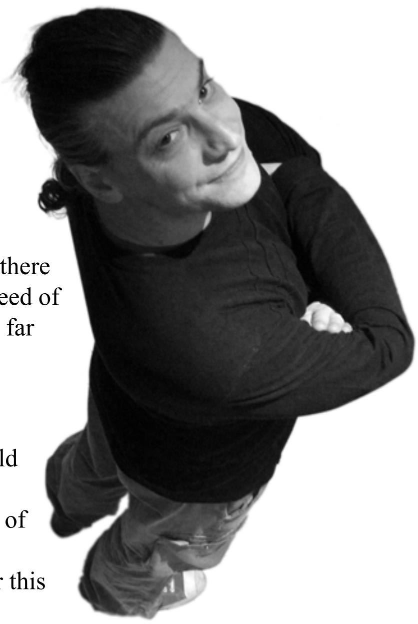
Cover Art
Pete Goodliffe

Print and Distribution
Parchment (Oxford) Ltd

Design
Pete Goodliffe

accu

Look but don't touch



There are a couple of news stories in the technology world causing a bit of a hoo-hah at the moment.

On the one hand, there is the growing 'Internet of Things', household and other common (but not obviously computer-based) items such as heating systems, refrigerators, car instrumentation and even guidance systems, and so on, that are coming 'Online', becoming controllable from – and reporting status to – personal computers and phone-apps. On the other hand, there is the growing maturity of quantum computers, a new breed of super computer potentially capable of processing speeds far exceeding that which today's technology can manage.

To take them in reverse order, the potential for quantum algorithms to easily crack today's strong encryption techniques is causing concern. RSA is the commonly held example, because its security is based on it being computationally infeasible to crack. Quantum computers of sufficient power and size *might* defeat public-key encryption relatively easily. The reason for concern over this should be obvious.

The other hand's concern over the Internet of Things is that even basic security doesn't seem to have entered into the minds of those designing many Internet-connected thermostats, traffic lights, and refrigerators. In fact, many common 'Internet of Computers' devices like home routers and the like seem to come with hard-wired default administrator passwords, and broadcast their willingness to communicate to anyone who knows how to listen.

I'm probably sounding very like a member of the tin-foil-hat brigade by now, but I do try to take my online privacy relatively seriously, so it matters whether my router is susceptible to buffer overflow attacks, and perhaps even more whether my car can be tricked by a man-in-the-middle attack. I also don't particularly like the idea of someone being able to use a known admin password to log into my house thermostat – if only because from there they could possibly stage to one of the real computers!

The logical conclusion of these two things together is that all security and privacy needs to be protected by quantum cryptography, meaning that maybe quantum programming would need to become more popular. I wonder what a quantum high-level programming language looks like?

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 26.5: 1st October 2014
C Vu 26.6: 1st December 2014

Overload 124: 1st November 2014
Overload 125: 1st January 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

COPYRIGHTS AND TRADE MARKS

After all, you'd be testing your code as you write it, anyway.

> We need tests at all levels of the software stack and development process. However, programmers particularly require tests at the smallest scope possible, to reduce the feedback loop and help develop high-quality software as quickly and easily as possible.

Types of test

There are many kinds of test, and often when you hear someone talk about a 'unit test' they may very likely mean some other kind of code test. We employ:

- Unit tests

  Unit tests specifically exercise the smallest 'units' of functionality *in isolation*, to ensure that they each function correctly. If it's not driving a single unit of code (which *could* be one class or one function), in isolation (i.e. without involving any other 'units' from the production code), then it's not a unit test.

  This isolation specifically means that a unit test will not involve any external access: no database, network, or file system operations will be run.

  Unit test code is usually written using an off-the-shelf 'xUnit' style framework. Every language and environment has a selection of these, and some have a de-facto standard. There's nothing magical about a testing framework and you can get a long way writing unit tests with just the humble `assert`. We'll look at frameworks later.

write unit tests during, or very shortly after, a section of code

- Integration tests

  These tests inspect how individual units integrate into larger cohesive sets of cooperating functionality. We check that the integrated components glue together and interoperate correctly.

  Integration tests are often written in the same unit test frameworks; the difference is simply the scope of the system under test. Many people's 'unit tests' are really integration-level tests, dealing with more than one object in the SUT. In truth, what we call this test is nowhere near as important as the fact the test exists!

- System tests

  Otherwise known as *end-to-end* tests, these can be seen as a specification of the required functionality of the entire system. They run against the fully integrated software stack, and can be used as acceptance criteria for the project.

  System tests can be implemented as code that exercises the public APIs and entry points to the system, or they may drive the system from outside using a tool like Selenium, a web browser automator. It can be hard to realistically test all of an application's functionality through its UI layer, in which case we employ *subcutaneous tests* that drive the code from the layer just below the interface logic.

  Because of the larger scope of system tests, the full suite of tests can take considerable time to execute. There may be much network traffic involved or slow database access to account for. The set-up and tear-down costs can be huge to get the SUT ready to run each system test.

Each of these levels of developer test establishes a number of facts about the SUT, and constructs a series of *test cases* that prove that these facts hold.

There are different styles of test-driven development. A project can be driven by a unit-test mentality: where you would expect to see more unit tests than integration tests, and more integration tests than system tests. Or it may be driven by a system-test mentality: the reverse, with far fewer unit tests. Each kind of test is important in its own right, and all should be present in a mature software project.

When to write tests

The term TDD (that is, *Test-Driven Development*) is conflated with *test-first* development, although there really are two separate themes here. You can 'drive' your design from the feedback given by tests without religiously writing those tests first.

However, the longer you leave it to write your tests, the less effective those tests will be: you'll forget how the code is supposed to work, fail to handle edge cases, or perhaps even forget to write tests at all. The longer you leave it to write your tests, the slower and less effective your feedback loop will be.

The test-first 'TDD' approach is commonly seen in XP circles. The mantra is: *don't write any production code unless you have a failing test.* The test-first TDD cycle is:

1. Determine the next piece of functionality you need. Write a test for your new functionality. Of course, it will fail.
2. Only then implement that functionality, in the simplest way possible. You know that your functionally is in place when the test passes. As you code, you may run the test suite many times. Since each step adds a small new part of functionality, and therefore a small test, these tests should run rapidly.
3. *This is the important part that's often overlooked:* now tidy up the code. Refactor unpleasant commonality. Restructure the SUT to have a better internal structure. You can do all this with full confidence that you won't break anything, as you have a suite of tests to validate against.
4. Go back to step 1 and repeat until you have written passing test cases for all of the required functionality.

This is a great example of a powerful, and gloriously short, feedback loop. It's often referred to as the r*ed-green-refactor* cycle in honour of unit test tools that show failing tests as a red progress bar, and passing tests as a green bar.

Even if you don't honour the test-first mantra, keep your feedback loop short and write unit tests during, or very shortly after, a section of code. Unit tests really do help 'drive' our design: not only does it ensure that everything is functionally correct and prevent regressions, it's also a great way to explore how a class API will be used in production: how easy and neat it is. This is invaluable feedback. The tests also stand as useful documentation of how to use a class once it's complete.

> Write tests *as you write* the code under test. Do not postpone test-writing, or your tests will not be as effective.

This test-early/test-often approach can be applied at the unit, integration, and at the system level. Even if your project has no infrastructure for automated system tests, you can still take responsibility and verify the lines of code you write with unit tests. It's cheap and, given good code structure, it's easy. (Without good code structure, an attempt to write a test will help drive you towards better code structure.)

Another essential time to write a test is when you have to fix a bug in the production code. Rather than rush out a code fix, first write a failing unit test that illustrates the cause of the bug. Sometimes the act of writing this test serves to show other related flaws in the code. Then apply your bugfix, and make the test pass. The test enters your test pool, and will serve to ensure the bug doesn't reappear in the future.

When to run tests

You can see a lot by just looking.
~ Yogi Berra

Clearly, if you develop using TDD, you will be running your tests *as* you develop each feature to prove that your implementation is correct and sufficient.

But that is not the only life of your test code.

Add both the production code *and* its tests to version control. Your test is not thrown away, but joins the suite of existent tests. It lives on to ensure that your software continues to work as you expect. If someone later modifies the code badly, they'll be alerted to the fact before they get very far.

All tests should run on your build server as part of a *Continuous Integration* toolchain. Unit tests should be run by developers frequently on their development machines. Some development environments provide shortcuts to launch the unit tests easily; some systems scan your filesystem and run the unit tests when files change. However I prefer to bake tests right into the build/compile/run process. If my unit test suite fails, the code compilation is considered to have *failed* and the software cannot be run. This way, the tests are not ignorable. They run *every* time the code is built. When invoked manually, developers can forget to run tests, or will 'avoid the inconvenience' whilst working.

Injecting the tests directly into the build process also encourages tests to be kept small, and to run fast.

Encourage tests to be run early and often. Bake them into your build process.

Integration and system tests may take too long to run on a developer's machine every compilation. In this case, they may justifiably run only on the CI build server.

Remember that code-level, automated testing doesn't remove the need for a human QA review before your software release. Exploratory testing by real testing experts is invaluable, no matter how many unit, integration, and system tests you have in place. An automated suite of tests avoids introducing those easily fixable, easily preventable mistakes that would waste QA's time. It means that the things the QA guys do find will be *really* nasty bugs, not just simple ones. Hurrah!

Good development tests do not replace thorough QA testing.

What to test

Test whatever is important in your application. What are your requirements?

Your tests must, naturally, test that each code unit behaves as required, returning accurate results. However, if performance is an important requirement for your application, then you should have tests in place to monitor the code's performance. If your server must answer queries within a certain timeframe, include tests for this condition.

You may want to consider the *coverage* of your production code that the test cases execute. You can run tools to determine this. However, this tends to be an awful metric to chase after. It can be a huge distraction to write test code that tries to laboriously cover every production line; it's more important to focus on the most important behaviours and system characteristics.

Good tests

Writing good tests requires practice and experience; it is perfectly possible to write bad tests. Don't be overly worried about this at first – it's most important to actually *start* writing tests than to be paralysed by fear that your tests are rubbish. Start writing tests and you'll start to learn.

Bad tests become baggage: a liability rather than an asset. They can slow down code development if they take ages to run. They can make code modification difficult, if a simple code change breaks many hard-to-read tests.

The longer your tests take to run, the less frequently you'll run them, the less you'll use them, the less feedback you'll get from them. The less value they provide.

code-level, automated testing doesn't remove the need for a human QA review before your software release

I once inherited a codebase that had a large suite of unit tests; this seemed a great sign. Sadly, those tests were effectively worse *legacy code* than the production code. Any code modification we made caused several test failures in hundreds-of-lines-long test methods that were intractable, dense and hard to understand. Thankfully, this is not a common experience.

Bad tests can be a liability. They can impede effective development.

A good test:

- has a short, clear, name, so when it fails you can easily determine what the problem is (for example: *new list is empty*)
- is maintainable: it is easy to write, easy to read, and easy to modify
- runs quickly
- is kept up to date
- runs without any prior machine configuration (e.g. you don't have to prepare your file system paths or configure a database before running it)
- does not depend on any other tests that have run before or after it; there is no reliance on external state, or on any shared variables in the code
- tests the *actual* production code (I've seen 'unit tests' that worked on a *copy* of the production code – a copy that was out of date. Not useful. I've also seen special 'testing' behaviour added to the SUT in test builds; this, too, is not a test of the real production code.)

These are some common smells of badly constructed tests:

- tests that sometimes run, sometimes fail (often this is caused by the use of threads, or racy code that relies on specific timing, by reliance on external dependencies, the order of tests being run in the test suite, or on shared state)
- tests that look awful and are hard to read/modify
- tests that are too large (large tests are hard to understand, and the SUT clearly isn't very isolatable if it takes hundreds of lines to set up)
- tests that exercise more than one thing in a single test case (a 'test case' is a *singular* thing)
- tests that attack a class API function-by-function, rather than addressing individual behaviours
- tests for third party code that you didn't write (there is no need to do that unless you have a good reason to distrust it)
- tests that don't actually cover the main functionality or behaviour of a class, but that hide this behind a raft of tests for less important things (if you can do this, your class is probably too large)
- tests that cover pointless things in excruciating detail, e.g. property getters and setters
- tests that rely on 'white-box' knowledge of the internal implementation details of the SUT (this means you can't change the implementation without changing all the tests)
- tests that only work on one machine

Sometimes a bad test smell indicates not (only) a bad test, but also bad code under test. These smells should be observed, and used to drive the design of your code.

Listing 1

```
@Test
public void stringsCanBeCapitalised()
{
  String input    =
    "This string should be upper case.";   ①
  String expected =
    "THIS STRING SHOULD BE UPPER CASE.";

   String result = input.toUpperCase();    ②

   assertEquals(result, expected);         ③
}
```

What does a test look like?

The test framework you use will determine the shape of your test code. It may provide a structured set up, and tear down facility, and a way to group individual tests into larger *fixtures*.

Conventionally, in each test there will be some preparation, you then perform an operation, and finally validate the result of that operation. This is commonly known as the *arrange-act-assert* pattern. For unit tests, at the assert stage we typically aim for a single check – if you need to write multiple assertions then your test may not be performing a single test case.

Listing 1 shows an example Java unit test method that follows this pattern:

① Arrange: we prepare the input

② Act: we perform the operation

③ Assert: we validate the results of that operation

Maintaining this pattern helps keep tests focused and readable.

Of course, this test alone does not cover all of the potential ways to use and abuse String capitalisation. We need more tests to cover other inputs and expectations. Each test should be added as a new test method, not placed into this one.

Test names

Focused tests have very clear names, that read as simple sentences. If you can't easily name a test case, then your requirement is probably ambiguous, or you are attempting to test multiple things.

The fact that the test method *is* a test is usually implicit (because of an attribute like the `@Test` above), so you needn't add the word `test` to the name. The example above need not be called `testThatStringsCanBeCapitalised`.

Imagine that your tests are read as specifications for your code; each test name is a statement about what the SUT does, a single fact. Avoid ambiguous words like 'should', or words that don't add value like 'must'. Just as when we create names in our production code, avoid redundancy and unnecessary length.

Test names need not follow the same style conventions as production code; they effectively form their own domain-specific language. It's common to see much longer method names and the liberal use of underscores, even in languages like C# and Java where they are not idiomatic (the argument being `strings_can_be_capitalised` requires less squinting to read).

The structure of tests

Ensure that your test suite covers the important functionality of your code. Consider the 'normal' input cases. Consider also the common 'failure cases'. Consider what happens at boundary values, including the empty or zero state. It's a laudable goal to aim to cover all requirements and all the functionality of your entire system with system and integration tests, and cover all code with unit tests. However, that can require some serious effort.

Do not duplicate tests: it adds effort, confusion and maintenance cost. Each test case you write verifies one fact; that fact does not need to be verified again, either in a second test, or as part of the test for something else. If your first test case checks a precondition after constructing an object, then you can assume that this precondition holds in every other test case you write – there is no need to reproduce the check every time you construct an object.

A common mistake is to see a class with five methods, and think that you need five tests, one to exercise each method. This is an understandable (but naïve) approach. Function-based tests are rarely useful since you cannot generally test a single method in isolation. After calling it, you'll need to use other methods to inspect the object's state.

Instead, write tests that go through the specific behaviours of the code. This leads to a far more cohesive and clear set of tests.

Maintain the tests

Your test code is as important as the production code, so consider its shape and structure. If things get messy, clean it, and refactor it.

If you change the behaviour of a class so its tests fail, don't just comment out the tests and run away. Maintain the tests. It can be tempting to 'save time' near deadlines by skipping test cleanliness. But rushed carelessness here *will* come back to bite you.

On one project, I received an email from a colleague: *I was working on your XYZ class, and the unit tests stopped working, so I had to remove them all*. I was rather surprised by this, and looked at what tests had been removed. Sadly, these were important test cases that were clearly pointing out a fundamental problem with the new code. So I restored the test code and 'fixed' the bug by backing out the change. We then worked together to craft a new test case for the required functionality, and then re-implemented a version that satisfied the old tests and the new.

> Maintain your test suite, and listen to it when it talks to you.

> **we aim to place truly isolated units of code into the 'system under test'**

Picking a test framework

The unit/integration test framework you use shapes your tests, dictating the style of assertions and checks you can use, and the structure of your test code (e.g. are the test cases written in free functions, or as methods within a test fixture class?).

So it's important to pick a good unit test framework. It doesn't need to be complex or heavyweight. Indeed, it's preferable to not chose an unwieldy tool. Remember, you can get very very far with the humble `assert`: I often start testing new prototype code with just a `main` method and a series of `assert`s.

Most test frameworks follow the 'xUnit' model which came from Kent Beck's original *Smalltalk SUnit*. This model was ported and popularised with JUnit (for Java) although there are broadly equivalent implementations in most every language, for example NUnit (C#) and CppUnit (C++). This kind of framework is not always ideal; xUnit style testing leads to non-idiomatic code in some languages (in C++, for example it's rather clumsy and anachronistic; other test frameworks can work better; check out *Catch* [2] as a great alternative).

Some frameworks provide pretty GUIs with red/green bars to clearly indicate success or failure. That might make you happy, but I'm not a big fan. I think you shouldn't need a separate UI or a different execution step for development tests. They should ideally be baked right into your build system. The feedback should be reported instantly like any other code error.

System tests tend to use a different form of framework, where we see the use of tools like Fit [3] and Cucumber [4]. These tools attempt to define tests in a more humane, less programmatic manner, allowing non-programmers to participate in the test/specification-wring process.

No code is an island

When writing unit tests, we aim to place truly *isolated* units of code into the 'system under test'. These units can be instantiated without the rest of the system being present.

A unit's interaction with the outside world is expressed through two contracts: the interface it provides, and the interfaces it expects. The unit must not depend on anything else – specifically not on any shared global state or singleton objects.

> Global variables and singleton objects are an anathema to reliable testing. You can't easily test a unit with hidden dependencies.

The interface that a unit of code *provides* is simply the methods, functions, events and properties in its API. Perhaps it also provides some kind of callback interface.

The interfaces it *expects* are determined by the objects it collaborates with through its API. These are the parameter types in its public methods or any messages it subscribes to. For example: an `Invoice` class that requires a `Date` parameter relies on the date's interface.

The objects that a class collaborates with should be passed in as constructor parameters, a practice known as *parameterise from above*. This allows your class to eschew hard-wired internal dependencies on other code, instead having the link configured by its owner. If the collaborators are described by an *interface* rather than a concrete type, then we have a seam through which we can perform our tests; we have the ability to provide alternative test implementations.

This is an example of how tests tend to lead to better factored code. It forces your code to have fewer hard-wired connections and internal assumptions. It's also good practice to rely on a minimal interface that describes a specific collaboration, rather than on an entire class that may provide much more than the simple interface required.

> Factoring your code to make it 'testable' leads to better code design.

When you test an object that relies on an external interface, you can provide a 'dummy' version of that interface in the test case. Terms vary in testing circles, but often these are called *test doubles*. There are various forms of doubles, but we most commonly use:

- Dummys

  Dummy objects are usually empty husks – the test will not invoke them, but they exist to satisfy parameter lists.

- Stubs

  Stub objects are simplistic implementations of an interface, usually returning a canned answer, perhaps also recording information about the calls into it.

- Mocks

  Mock objects are the kings of test double land, a facility provided by a number of different mocking libraries. A mock object can be created automatically from a named interface, and then told up-front about how the SUT will use it. A SUT test operation is performed, and then you can inspect the mock object to verify the behaviour was as expected.

  Different languages have different support for mocking frameworks. It's easiest to synthesize mocks in languages with reflection.

  Sensible use of mock objects can make tests simpler and clearer. But, of course, you can have too much of a good thing. Tests that are encumbered by complex use of many mock objects can become very tricky to reason about, and hard to maintain. *Mock mania* is another common smell of bad test code, and may highlight that the structure of the SUT is not correct.

Conclusion

If you don't care about quality,
you can meet any other requirement
~ Gerald M. Weinberg

Tests help us to write our code. They help us to write *good* code. They help maintain the *quality* of our code. They can *drive* the code design, and serve to document how to use it. But tests don't solve all problems with software development. Edsger Dijkstra said: Program testing can be used to show the presence of bugs, but never to show their absence.

No test is perfect, but the existence of tests serves to increase confidence in the code you write, and in the code you maintain. The effort you put into developer testing is a trade-off; how much effort do you want to invest in writing tests to gain confidence? Remember that your test suite is only as good as the tests you have in it. It is perfectly possible to miss an important case; you can deploy into production and still let a problem slip through. For this reason, test code should be reviewed as carefully as production code.

Nonetheless, the punchline is simple: if code is important enough to be written, it is important enough to be tested. So write development tests for your production code. Use them to *drive* the design of your code. Write the tests *as* you write the production code. And automate the running of those tests.

Shorten the feedback loop.

Testing is fundamental and important. This article can only really scratch the surface, encourage you to test, and prompt you to find out more about good testing techniques. ■

Questions

1. Which is the best development test technique: test-first, or test (very shortly) after coding? Why? How has your experience shaped this answer?
2. Why do QA departments not traditionally write much test code, and generally focus on running through test scripts and performing exploratory testing?
3. How can you best introduce test-driven development into a codebase that has never received automated testing? What kind of problems would you encounter?
4. Investigate *Behaviour-Driven Development*. How does it differ from 'traditional' TDD? What problems does it solve? Does it complement or replace TDD? Is this a direction you should move your testing in?

Acknowledgments

With thanks to Seb Rose, Chris Oldwood and Steve Love for their valuable and timely input into this article.

References

[1] Janzen & Saiedian. 'Test-driven development concepts, taxonomy, and future direction', 2005. Published in: *Computer* (Volume 38, Issue 9)
[2] The Catch unit test framework. Available from: http://github.com/philsquared/Catch
[3] Fit. Available from: http://fit.c2.com
[4] Cucumber. Available from: http://cukes.info

A Secure Data Centre in the Heart of... Bowthorpe

Paul Grenyer takes a tour of MigSolv's facility.

I've been a fan of *Blake's 7* [1] since I was a little boy. The producers always used the most fantastic locations for filming the futuristic science fiction series. Part of the 'Children of Auren' was even filmed at the Metropolitan University near where I lived in Leeds.

Little did I know there is now an ideal *Blake's 7* set in Norwich!

Silicon Broads

The term 'Silicon Broads' started being bandied around shortly before SyncNorwich's *TechCrunch* [2] event in November 2013. Around the same time I was contacted by a PR company working for a company called MigSolv [3], who were looking to do a story on whether Norwich could build another Tech City or Silicon Fen.

Part of the discussion included an invite to visit and tour MigSolv's high security data centre in Bowthorpe. It was more than six months before Sean Clark, who has recently taken over the *Norfolk Tech Journal*, and I were able to take advantage of the generous offer...and we weren't disappointed.

> **it's far more environmentally efficient to provide air conditioning locally to the racks, rather than to the entire room**

About two years ago, MigSolv acquired the site from Aviva and invested £12 million refurbishing it. It consists predominantly of two large halls which can hold rack upon rack of computers. A bomb proof mound stands between them. Six generators, which can be spun up in seconds to take over from the UPSs, ensure that in the event of a major power cut the site keeps running. The site is served by two different mains power feeds and a number of different internet connections converge on the site from different directions and providers. It is monitored 24 hours a day from an onsite control room.

Security measures

Security is tight. You cannot visit the site without an appointment and you must provide photo ID on arrival. There's a perimeter fence tens of metres from the main buildings that is topped with barbed wire. Entry is through a manned gatehouse through a turnstile.

We were met by Jacob Barreth who works in sales at MigSolv. Jacob is an extremely well informed and articulate individual. He doesn't just know about his company and their site in Norwich, but clearly understands the market, MigSolv's place in it and the forces affecting the clients who use the site now and in the future. Jacob also has a keen interest in digital rights.

The tour

After a long chat (that I could happily have enjoyed all afternoon) about MigSolv's business, their facility in Norwich and the technical community in Norwich, Sean and I were taken on a tour of the fascinating site. It started outside where we were shown the generators and talked through many of the external security features. We were then taken through the various staging areas through which a customer would receive, unpack and provision their hardware before it is installed in one of the data centre halls. The final door into each hall requires an iris scan.

The halls are immense dust free environments. They were much warmer than I expected because, as Jacob explained, it's far more environmentally efficient to provide air conditioning locally to the racks, rather than to the entire room, (and it costs less which means their customers pay less).

In the background there was the constant hum of the air extractors and the light was subdued but ample. Each bank of racks is fed power and networking from beneath the floor and kept locked. We wouldn't have been able to gain access to any individual bank. At either end of the halls are cages which are sectioned off areas for the telecommunications providers that provide the Internet connections and are designed so that only they can access them from doors and passageways external to the main hall.

Jacob talked us through the fire suppression system which uses gas to put out any fire detected in plant areas and fine sprinklers in the main data halls. Let's face it, in a data centre there's not much that will actually burn, so if you do have a fire you've already got huge problems.

Final thoughts

Before we knew it the full three hours allocated had passed and, after an exchange of business cards, Jacob took us back outside the windowless building and pointed us towards the turnstiles that would take us back to security. It turns out that the turnstiles themselves were a bit of an ingenuity test and it took us a few minutes to work out how to get out!

I came away very impressed by the facility and the attitude of MigSolv themselves.

The halls are vast and I can easily imagine Avon and the rest of the Blake's 7 crew charging through them hotly pursued by the federation. MigSolv are keen to engage with the local tech community and in particular with start-ups and SMEs to demonstrate the services they provide.

I am sure we, as the local tech community, will be hearing much more from them, starting in the very near future. ■

References

[1] http://blakes7.com/
[2] https://www.youtube.com/user/SyncNorwich/videos
[3] http://www.migsolv.com/

PAUL GRENYER

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

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?

What Do People Do All Day?

Matthew Jones gives some insight as to what his job involves.

This is the first in what I hope will be a series of 'What Do People Do All Day' articles [1] written by us, the members of ACCU. As the title suggests, this is about what we do, day-to-day, in our jobs. It was called into existence by Chris Oldwood, on accu-general [2] – see sidebar. I hope there will be many more, much like the 'Desert Island Books' series in *CVu*. Whether this actually happens is down to you!

Software spans a huge variety of languages, tools, market sectors, and countries. I have vague notions of what people in the banks in Canary Wharf do, but I bet I'm mostly wrong. I would like to know, and I suspect most readers would too. I know Chris does.

There are several aspects to why we might want to know. It is partly just curiosity, or nosey-neighbour curtain twitching trying to see what's going on next door. We always want validation: proof from others that we're not alone in working the way we do, and that our problems and struggles are not unique. And I suspect there's always a secret hope that we will find that our job actually *is* the best in the world, and we really *are* better at software than everyone else.

Being the first, this stands as a template for others to follow, if you want. But think for yourselves! You're all individuals.

Background

By most people's standards I am a 'real time' and 'embedded' software engineer. But I spend most of my time writing medium to high level, portable, generic code. I usually work on large systems rather than 'proper' embedded stuff (by which I mean 8 bit micros, assembler/C, JTAG debuggers and the like).

At the moment I am working as a permie at a small company, the software director of which is an old friend from university. I came here a year ago because I had been toying with the idea of working somewhere small for years. I had a rose tinted day-dream of no bureaucracy, transparent communication and everyone pulling their weight.

Before this I had worked in a series of large, usually multinational, companies; in multi-person teams on projects that usually spanned years rather than months. It has been interesting, but frustrating, as most people in that situation will agree.

The team

At the top we have the software director. He is also the general manager. He founded the company about 10 years ago, based on software that he had started writing 10 years before that. As general manager he looks after most of the day-to-day operation of the company including the production team who assemble, pack and ship the various products. This often prevents him from coding, so every now and then he either works from home, or takes himself off for a week or so, part holiday and part coding retreat. This is where some new features come from – straight from his head into the product.

Out of a staff of a dozen or so, there are 4 software engineers. One joined straight from university, about 5 years ago. Although he is the most junior, he's been there the longest, and is actually the mainstay of the team. He's also the IT guy, and the website guy, and the e-shop guy, and the Linux expert and so the list goes on. He is critical to the operation of the development team, and the business as a whole.

One of the developers works on truly embedded stuff: he writes VHDL as well as C, and works on PCB design, firmware and code. He also works from home in a different city so we only see him occasionally: every month or so we have progress meetings which he attends.

The inspiration behind the series...

This article was inspired – you could even say 'requested' – by Chris Oldwood's post on accu-general:

> **A plea – more articles on what kind of programming you do**
>
> Hi All,
>
> Something I've tried to do in my own articles in *C Vu* and *Overload* is to try and give some context to the kind of programming I do, because I know it's different from what many others do. In particular I'm interested to know what kinds of constraints, or lack of them others have to put up with.
>
> For example, I remember an accu-general thread about unit testing and how that might/might not be as easy to apply in the gaming industry. Those in the embedded market have historically avoided C++, when perhaps it's certain features of C++ they couldn't afford. Floating-point maths is apparently a no-no in finance, unless you're doing large volumes of risk calculations and then performance trumps precision (well, in the bits of finance I worked in). How does the inability to patch a video game because it's delivered on a read-only cartridge affect the development process?
>
> I don't know about anybody else in ACCU, but I want to know more about the kinds of stuff other people do. And in particular what makes it different to what I do. I appreciate it's often tricky to know what's different (unconscious incompetence) but in those cases when you have had to make a trade-off – what was it and why? When have you read a blog post or book about some cool technique and then shouted at it because it has no place in your industry/organisation/etc?
>
> I'm sure the editors of our *C Vu* and *Overload* journals would be more than happy to receive more content...

Hopefully, this will be the first of a series.

Apart from me, the other developer is a contractor who mostly works for us but goes off for weeks at a time on non-development jobs relating to our industry. This gives him excellent experience with both our products, and the competition, and he can wear the customer hat when we're discussing requirements or solutions to problems.

Lastly, there's me. On paper I have the most experience of anyone, having worked in a broader range of industries, but this is worth little in practice, except under the blanket of 'with age comes experience'. Luckily for me I was put to work on a new product which sits alongside the existing range. This was a double edged sword; although I had a clean sheet, I also had to learn everything from scratch.

The software

We have one code base. As I mentioned above, parts of it are now getting on for 20 years old. Unsurprisingly this shows; it is 90% old-school C, with the attendant style and design failings. There is a lot of cut-and-paste (it is the fastest way for the director to add new features). We have 1000 line `switch` statements. We have about 3 main include files, each of which

MATTHEW JONES

Matthew started programming with BBC Basic, and then learned C during a VI form summer job. He has been programming professionally for over 20 years, having moved on to C++, and usually works on large embedded systems. He can be contacted at m@badcrumble.net

tops 1000 lines. The application state is spread all over the place. This is what happens when one person develops by themselves for years, without keeping up with current thinking. Having said all that, the code is 'brutally fast' (to quote a colleague), and it works.

The old code is hard to understand, and fragile if you don't know it well. It follows its own coding standards and conventions, but these are mostly rules and techniques in the original author's head. When he follows them, preconditions are met, variable names make sense and the code works. When anyone else starts making changes things can go wrong, and fast. So we have quite naturally fallen into an arrangement where each developer usually sticks to their own code, and therefore their own area of expertise. But this also means we have different coding styles and language dialects. We are not at the point where this can be homogenised; the old code is not going to get changed without good reason, and the director readily confesses to only really understanding a core subset of C, let alone C++.

I felt the need to write some unit tests for some critical parts of my code that I just couldn't properly test manually

The main software runs in our physical products, which use touch screens and a variety of input devices, and which use medium to high-end embedded processors running Linux. There is also a software only version which has graphical emulation of the product's front panel. This runs on Windows, Mac and Linux. We manage all this using the Qt framework [3].

We use Subversion [4] for revision control, and Buildbot [5] is triggered by commits to the trunk, whereupon it builds all the different flavours of the product, resulting in installers that are ready to release, should we feel the need. Once we have a green build releasing it is simply a case of copying the installers onto the download website and updating the webpage.

Releases are ad-hoc. There is a product roadmap but it is a sketch rather than something to frame on the wall and point at when we're late with a feature. We tend to release several times a month, usually when there is a new feature or two worthy of release. Bug fixes that have happened along the way are released by default, although sometimes we make specific releases to fix critical issues. There are no code branches; we release from the trunk, and all products run the same version of software (subject to the users upgrading). Considerable effort goes into ensuring backward, and forward, compatibility with configuration data and the like. Above the product roadmap we have the calendar with two large trade shows a year. These are the deadlines for new products or major new features.

Testing is manual. We are expected to test our own code changes, and we have a couple of support people who are meant to put each release candidate through its paces. This is patchy and unreliable because they have plenty of other things to do (like answering customers' calls). I felt the need to write some unit tests for some critical parts of my code that I just couldn't properly test manually. These will become automated when they reach the top of my todo list.

The process

We have no formal development process. However, we are pretty agile (although only a few of us would even know the term!).

Being all in one office, we all know what is going on in the development team. With forthcoming work we are expected to tell, rather than ask. There's always plenty to do and I've never had to ask for a task. We state our forthcoming work in weekly email reports, which allows the director to comment on priority and direction (he is the director after all!). Sometimes on a Monday he might say, "This week you need to work on X because I did Y over the weekend," where Y could be 'thought about it', 'fixed a bug', 'added a feature', 'spoke to a customer', 'got a support call' and so on. It's never quite a case of dropping everything, but we do make 90 degree turns sometimes.

Requirements come from the directors (HW and SW) and are usually verbal. As we work on a feature the requirements will be thrashed out amongst ourselves. When I started I wrote a SW requirement specification mainly as a tool to aid my own understanding, and to trigger review and debate. Since then I haven't maintained it because I've learned the ropes and we haven't had anything new enough to require significant up-front thinking. We rarely have design meetings, and I have never seen a SW design document. Mainstream features evolve from the existing code and rarely involve significant re-writing or re-design. Because each of us tends to work alone on an area of the code, most of the time we quietly 'just do it'.

To many who are used to working in large companies with QA departments, processes, reviews and project plans, this might feel like the wild west. In many ways it is, but it works pretty well. We just don't seem to need those trappings. There is sufficient (but never enough!) communication, and the team is experienced enough to turn out code that does the job. It can be ugly at times, but it hides inside a product that is functional, reliable, established, and well respected by its users.

The future

When I arrived I was pleasantly surprised to find Buildbot running automatically. Of course this was the work of the youngest developer. I was amazed to find that we had something like 3000 compiler warnings. A lot of them were silly things, but hiding amongst them (and going unnoticed) were a few serious ones. I couldn't stand this because the noise constantly distracted me when writing new code, so I lead a crusade. Now we have green builds, and `-Werror` in the makefile [6].

Everybody understands that we have to isolate the old code, and with it the director's involvement in the day-to-day development of 'the new stuff'. We know that the long term goal is to extricate the true core from what is currently thought of as 'the core', but in reality is a mix of core logic, GUI, and application state machines. We have had a number of casual discussions (waiting for the kettle to boil – we don't have a water cooler!) about gradually re-writing the GUI using the full power of Qt, and hiding the core using MVC [7] and its ilk. We know the core will never be modernised, but it can be distilled.

Whatever we do there is enormous inertia in the existing product, and the force that four employees can apply is small. The needs of the business heavily outweigh any ideals. Being a small company there is a constant tension between the needs of the old code, adding features cost effectively, and the staff wage bill. I retain my sanity by retreating to my code which is clean [8], moderately SOLID [9] and where all is beauty and light. Actually I'm starting to revisit my own legacy code from a year ago, when I was effectively prototyping the product, and not liking what I see! It just goes to show how your own style and ideas change as you work. ■

References

[1] http://www.goodreads.com/book/show/313375.Richard_Scarry_s_What_Do_People_Do_All_Day_ https://en.wikipedia.org/wiki/Richard_Scarry
[2] 8 March 2014: Oldwood, Chris on accu-general: A plea - more articles on what kind of programming you do
[3] http://qt-project.org/
[4] http://subversion.apache.org
[5] http://buildbot.net
[6] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
[7] https://en.wikipedia.org/wiki/MVC_model
[8] http://blog.cleancoder.com/
[9] https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

What do *you* do all day? If you want to share with us the great way you go about your day-to-day work, or the frustrations of working in a dysfunctional environment, or anything in between, get in touch! Send your experiences to cvu@accu.org

Beware setlocale Behaviour in Visual C++ 2012 & 2013

Alex Paterson investigates a bug in the Visual C++ runtime library.

This article describes how investigating a crash in a popular open source library led to the discovery of a change in behaviour in the C runtime library implementation that could have a significant detrimental impact on applications using C runtime locales across multiple threads that are compiled with Visual C++ 2012 & 2013.

Debugging a crash in GDAL

A crash was reported in a project using the Geospatial Data Abstraction Library (GDAL), a popular library for handling geospatial data and reading/writing various geospatial data formats. [1] [2]

The person experiencing the crash had recently upgraded their build platform to use a new version of Visual Studio (from 2005 to 2012) and had also upgraded to a new build of GDAL. At some point they found their previously stable program often crashed, producing the same stack trace each time. They investigated the problem and came up with a small test program that could replicate the problem; it did some basic reading of geospatial data on multiple threads and I used it to replicate the crash for myself. The test application was investigated further by using Microsoft's excellent Application Verifier [3] tool and, after rebuilding the entire GDAL library in debug mode (requiring some build file tweaks [4]), it became apparent that the problem was in the area of locale handling.

Isolating the problem

Another test program was created, this time replicating the locale behaviour of the GDAL code without directly using it.

This program corrupts memory on Visual C++ 2012 and 2013, but not on 2010 and earlier. Increasing **`THREAD_COUNT`** increases the frequency of corruption events. These issues can be detected in the debugger by running the program with the Application Verifier 'Basics' checks enabled. Listing 1 shows code isolating the problem with **`setlocale`**.

Listing 1

```
#include <locale.h>
#include <process.h>
#include <stdio.h>
#include <windows.h>

const int THREAD_COUNT = 2;

void setlocale_loop(void*)
{
  while(1)
    setlocale(LC_NUMERIC, NULL);
    //query the current locale
}

int main()
{
  for (int i = 0; i < THREAD_COUNT; i++)
    _beginthread(&setlocale_loop, 0, NULL);

  Sleep(60 * 1000);
  /*sleep main thread for 60 seconds*/
}
```

This program starts two workers threads, each of which calls **`setlocale()`** to query the current locale in a tight loop (**`setlocale`** being used to both change and query the current locale). Sounds simple, but even this small test program was hitting an Application Verifier breakpoint for heap corruption. The stack trace (Listing 2, overleaf) was indicating that the problem was in the part of the runtime library code that was updating the locale, which was surprising because my test program wasn't updating the locale.

As Microsoft include the source code of their C runtime library in their compiler distribution, examining the code revealed that even just *querying* the locale appears to cause some shared state to be modified.

What is a locale?

Due to the wonderful variety of different cultures we have on our planet, data presentation varies between regions, using different formatting and/or ordering according to local standards and customs. As an example, ask yourself what the value of the following is:

```
1.125
```

If you live in the UK, you're likely to read it as floating point value between 1 and 2, but if you live in mainland Europe, you might read it as one thousand, one hundred and twenty five. This presents a problem for any code that needs to deal with translating this information from human-readable form (i.e. strings) to internal representation (e.g. a variable of type float), or vice-versa.

A locale in the C runtime sense is essentially a set of data that describe the data presentation standards for a given region/culture. They are essential in not only being able to present information on screen to a user, but also in reading data. If I wanted to read a file that included coordinate information, then I need to know what locale the data is stored in so that I can interpret the decimal point correctly.

Finding a hypothesis

In an attempt to take an easy step to rule out any problems introduced by the change of compiler version, I tried my test program in Visual C++ 2005, the version that was being used before the upgrade. To my surprise, my test program did not crash, suggesting a change in behaviour of the runtime library and/or compiler. So what could Microsoft have done to affect C locale handling between these two different versions of their compiler? Comparing the setlocale runtime library code between versions 2005 and 2012 of Visual C++ showed significant changes.

To give a quick bit of background, the C standard defines the following method for setting the locale: **`char * setlocale (int category, const char * locale)`**. In the Microsoft world, there are two variations of this method, one standard following the above specification and one to support wide characters: **`wchar_t * _wsetlocale (int _category, const wchar_t * _wlocale)`**. So we

ALEX PATERSON

Alex Paterson is a programmer working mostly with C++ and PL/SQL in the field of GIS and Intelligent Transport Systems. He lives and works in the North East of England and can be contacted at alex@tolon.co.uk.

Listing 2

```
ntdll.dll!_RtlReportCriticalFailure
ntdll.dll!_RtlpReportHeapFailure
ntdll.dll!_RtlpLogHeapFailure
ntdll.dll!_RtlpCoalesceFreeBlocks
ntdll.dll!@RtlpFreeHeap
ntdll.dll!_RtlFreeHeap
kernel32.dll!_HeapFree
msvcr110.dll!free(void * pBlock=0x00769650) Line 51
msvcr110.dll!__freetlocinfo(threadlocaleinfostruct * ptloci=0x00740000) Line 202
msvcr110.dll!_updatetlocinfoEx_nolock(threadlocaleinfostruct * * pptlocid, threadlocaleinfostruct *
ptlocis) Line 250
msvcr110.dll!_wsetlocale(int _category=4, const wchar_t * _wlocale=0x00000000) Line 569
msvcr110.dll!setlocale(int _category=4, const char * _locale=0x00000004) Line 50
SetLocaleThreadSafety.exe!setlocale_loop(void * __formal=0x00000000) Line 9
msvcr110.dll!_callthreadstart() Line 255
msvcr110.dll!_threadstart(void * ptd) Line 237
kernel32.dll!@BaseThreadInitThunk
ntdll.dll!___RtlUserThreadStart
ntdll.dll!__RtlUserThreadStart
```

have two different setlocale methods, one for **char** and another for **wchar_t**. In Visual C++ 2005 **setlocale()** implements the querying and changing of the locale as expected, but the **_wsetlocale()** variation performs some **wchar_t *** to **char *** string conversion and then delegates the work to **setlocale()**. So despite there being two different functions for changing the locale there is really only one implementation, which seems like a good approach and follows the DRY [5] philosophy.

In Visual C++ 2012, the situation is reversed. **_wsetlocale()** does the actual work and **setlocale()** delegates to it; different editions of Microsoft's Visual C++ compiler were examined and it appears that the former implementation was last shipped in 2010 and the latter shipped in both 2012 and 2013.

Verifying the hypothesis

To verify that the problem was due to this change, I wrote another test program using **_wsetlocale**, which this time crashed when built with 2005, but ran without a problem when built with 2012. (Listing 3 is test program code using the **wchar_t** variation of **setlocale**.)

So in Visual C++ 2005, a very similar problem existed, but it was in the non-standard **_wsetlocale** method.

The crux of the problem seems to be that the **setlocale** method modifies the global locale storage even though only a query is requested. Coupled with reference-counted storage of the locale data and some gaps in the locking mechanism means that calling **setlocale** concurrently from multiple threads in VC++ 2012/2013 (or **_wsetlocale** prior to this) should be treated with caution.

This change in behaviour is most unfortunate, as the problem has moved to the standard **setlocale** method, which I believe will be used by far more applications than the **_wsetlocale** version (most C code aiming to be platform independent will be using **setlocale**). I shudder to think of how many programs and libraries that require localisation are now running the risk of hitting this problem.

Listing 3

```
/* This crashes in MSVC 2010 and earlier.
Note the use of _wsetlocale, the wide character
variant of setlocale. */

void setlocale_loop(void*) {
  while(1)
    _wsetlocale(LC_NUMERIC, NULL);
}

int main() {
  for (int i = 0; i < 2; i++)
    _beginthread(&setlocale_loop, 0, NULL);

  Sleep(60 * 1000);
  /*sleep main thread for 60 seconds*/
}
```

It is difficult to know how widespread this issue really is, but it seems to me that any cross-platform C or mixed C/C++ program might be susceptible to the same problem if they support locales across multiple threads. One of the more likely candidates for hitting this problem are those applications that are concerned with processing data that may be localised. GDAL falls into this category as when it is reading data from the various geospatial formats it supports, it primarily needs to understand the decimal point representation (dot vs comma).

What does the Standard say?

In the C99 [6] and C11 [7] standards, **setlocale** is defined in §7.11.1.1. In C99, paragraphs 2, 5 and 7 are most relevant to this discussion:

- Para 2 - The **setlocale** function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof...
- Para 5 - The implementation shall behave as if no library function calls the **setlocale** function.
- Para 7 - A null pointer for locale causes the **setlocale** function to return a pointer to the string associated with the category for the program's current locale; the program's locale is not changed.

In C11, paragraphs 2 and 7 are the same, but paragraph 5 is modified to the following:

- Para 5 - A call to the **setlocale** function may introduce a data race with other calls to the setlocale function or with calls to functions that are affected by the current locale. The implementation shall behave as if no library function calls the setlocale function.

This update to paragraph 5 seems sensible when considering calls that modify the locale, but surely it shouldn't apply to calls that are merely querying the current program locale? Paragraph 5 states that concurrent calls to **setlocale** may introduce a data race, but paragraph 7 tells us that when querying the locale, the program's locale is not changed. Surely if we're not modifying the program's locale, there shouldn't be any possibilty of a data race?

Digging deeper / a curious comment

It seems that these crashes only happen when our concurrent calls to query the locale include the delegating version of the method. Examining the code of the delegating methods gives us a couple of good clues that the runtime coders aren't too confident about what is going on! The code includes some resource locking and reference-counted storage, which

Listing 4

```
...
/*
 * Note that we are using a risky trick here.  We are adding this
 * wlocale to an existing threadlocinfo struct, and thus starting
 * the wlocale's wrefcount with the same value as the whole struct.
 * That means all code which modifies both threadlocinfo::refcount
 * and threadlocinfo::lc_category[]::wrefcount in structs that are
 * potentially shared across threads must make those modifications
 * under _SETLOCALE_LOCK.  Otherwise, there's a race condition
 * for some other thread modifying threadlocinfo::refcount after
 * we load it but before we store it to wrefcount.
 */
...
```

makes sense as some form of locking and reference-counting is required in order to prevent shared locale resources being changed whilst being used.

Comments taken from the Microsoft C runtime library implementation of **_wsetlocale** in Visual C++ 2005 (wsetloca.c) are shown in Listing 4.

Workaround

The workaround added to GDAL is to use a synchronisation method to prevent it from making concurrent calls to either **setlocale** or **_wsetlocale**. This has been done by providing a wrapper method which uses critical section semantics to prevent concurrent calls to the underlying setlocale method. This is far from ideal as it only prevents concurrent setlocale calls in one library; the workaround would need to be implemented in all susceptible libraries until such a time that the underlying issue in the runtime library has been resolved.

Conclusion

Put simply, beware **setlocale** in Visual C++ 2012 and 2013!

There is a race condition in Visual C++ 2010 and earlier when querying the locale using the non-standard **_wsetlocale** function, but not the standard **setlocale** one. In Visual C++ 2012 and 2013 this situation is reversed; there is a race condition when querying the locale using **setlocale**, but not **_wsetlocale**. This change in behaviour may have an impact on any programs compiled with the Microsoft C runtime library that use locales across multiple threads.

The problem may have previously existed for **_wsetlocale**, but I believe that this non-standard version is called far less often than **setlocale** (i.e. most platform-independent code is probably using **setlocale** in all cases rather than having special cases to call **_wsetlocale** on Windows). The change in behaviour has the unfortunate effect that when some libraries and programs are recompiled in Visual C++ 2012 and later, the stability may not be as good as when they were compiled in previous versions.

Recommendations

The implementation of **setlocale** and **_wsetlocale** in the Visual C++ C runtime library should be addressed to remove the current race condition. The C standard states that querying the locale should not change the program's locale, so any **setlocale** queries should not have problems, even if they are concurrent. A shared synchronisation primitive between the **setlocale** and **_wsetlocale** implementation could solve the problem, preventing any concurrent execution across both methods. There are other routines that must use the current locale, but these do not exhibit the same race condition. For example, using **sprintf** concurrently to write float values into a string does not yield any problems in either VC++ 2005 or VC++ 2012.

With regard to the 'risky' comment. In the 2012 and 2013 editions of Visual C++, the curious comment appears in both the **setlocale** and **_wsetlocale** methods, suggesting both an overuse of copy-and-paste and perhaps a lack of diligence when reviewing code. In code as critical as this, I am surprised that such as comment is present, but I do have a lot of respect for the programmer who wrote it in the first place; their comment is essentially a red flag telling us that they have written some code or have found some code that has a bit of a smell and it really should be sorted out at some point. [8]

Code reviews are an important tool for software developers, but they should not be over-used. However, in something like a runtime library, especially one as mature as Visual C++, I would expect every change to be reviewed. Either this is not happening or the quality of the code review appears to be questionable.

The issue with concurrent setlocale query calls has been raised with Microsoft. [9]

Further work

In this case the 'risky' code could have been found by searching the runtime library source code for the string "risk". If we were to extend this search to include some other 'code smell' phrases, such as "don't know" (as in "I don't know what this code does", which I have seen in the wild, although not on my present project I hasten to add), or "to do", then this may be a useful method of detecting some areas of the codebase that require further attention.

So my suggestion is that you occasionally search your own codebases for some of these words and phrases. I have done this myself with some interesting results, but I won't bore you with the self-incriminating details... ■

References

[1] GDAL Ticket 5366: Access violation (Crash) with OGRCreateCoordinateTransformation (http://trac.osgeo.org/gdal/ticket/5366)

[2] GDAL - Geospatial Data Abstraction Library (http://www.gdal.org)

[3] Microsoft Application Verifier (http://msdn.microsoft.com/en-us/library/aa480483.aspx)

[4] GDAL can now produce PDB files for proper debugging on Windows using the WITH_PDB=1 flag (http://trac.osgeo.org/gdal/ticket/5420)

[5] Orthogonality and the DRY Principle (http://www.artima.com/intv/dry.html)

[6] C99 Language Specification with TC changes, final draft (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)

[7] C11 Language Specification, final draft (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf)

[8] A 'code smell', as defined in *Clean Code, A Handbook of Agile Software Craftsmanship*, Robert C. Martin, 2009, Prentice Hall.

[9] Bug report logged with Microsoft (https://connect.microsoft.com/VisualStudio/feedback/details/794122)

Revisiting the Gang of Four

Chris Oldwood reflects on things missed first time around.

My education as a programmer recently took another unexpected turn as quite by accident I discovered the difference between types and classes. Whilst I was aware there was a difference, I had never felt that my skills as a programmer were sufficiently impaired to feel the need to go out and discover the answer proactively.

This serendipitous moment came by way of a link in a tweet to a StackOverflow question written way back in 2012. It involved a comment made by James Coplien about the book *Design Patterns* and asked "Are there any patterns in GoF?" [1]. The accepted answer, which finally came some 18 months later from James Coplien himself, caused me to go back and read the 'Introduction' chapter of the seminal *Design Patterns* book [2]. When I reached Section 1.6, 'How Design Patterns Solve Design Problems', I found the difference between class and type described clear as day under the subsection 'Class versus Interface Inheritance':

> It's important to understand the difference between an object's class and its type…The class defines the object's internal state and the implementation of its operations. In contrast, an object's type only refers to its interface…

It turns out Section 1.6, which is only 16 pages long, is an absolute goldmine of information on the object-orientated (OO) paradigm, containing such gems as the sections: 'Programming to an Interface, not an Implementation', 'Inheritance versus Composition', 'Delegation, Inheritance versus Parameterized Types' and 'Designing for Change'.

In some respects I find this sudden gain in clarity a little disturbing because I know I read that section for (at least) a second time, as I always read my books from cover-to-cover (eventually). So why did I not remember this particular nugget within the book, even if it was 20 years ago, as I do so many other gems in other books? For example, the discussion of noumena and phenomena and the essence of objects in the weighty tome on OLE by Kraig Brockschmidt [3] is far more tenuous, and yet apparently far more memorable.

At the start of your professional programming career you just don't have the capacity to take in everything that you consume

I think Emyr Williams probably hit the spot in his recent *Becoming a Better Programmer* blog post 'Concepts Not Syntax' [4]. At the start of your professional programming career you just don't have the capacity to take in everything that you consume, especially when you're being paid to write code. And unless you come from a Computer Science background in the first place you need to make a choice about what you focus on. I now realise I unconsciously chose to focus on learning the technology –platforms and programming languages – which means I'm now paying catch up to understand what it all means.

This isn't the first time either. My career is littered with examples of where I finally really managed to grok something just as it goes out of fashion, e.g. COM, batch files, etc. With functional programming grabbing the headlines and object-orientation being given its last rites it's somewhat apt that now is the time I start to discover what I probably should have known 20 years ago.

Irrespective of whether James Coplien is right or not on whether they are really idioms, and not patterns as Alexander intended, the book still contains some thoroughly useful knowledge about learning the principles behind OO. ■

CHRIS OLDWOOD

Chris is a freelance developer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros; these days it's C++ and C#. He also commentates on the Godmanchester duck race. Contact him at gort@cix.co.uk or@chrisoldwood

References

[1] http://stackoverflow.com/questions/12981021/are-there-any-patterns-in-gof

[2] *Design Patterns: Elements of Reusable Object-Orientated Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, published by Addison Wesley, 1994

[3] *Inside OLE* by Kraig Brockschmidt, published by Microsoft Press, revised edition 1995

[4] 'Concepts Not Syntax' by Emyr Williams in his *Becoming a Better Programmer* blog: http://becomingbetterdotnet.wordpress.com/2014/04/26/concepts-not-syntax/

Talk in Code

Andy Balaam presents some tips on clear communication.

Last week we had an extended discussion at work about how we were going to implement a specific feature.

This discussion hijacked our entire Scrum sprint planning meeting [1] (yes, I know, we should have time-boxed it [2]). It was painful, but the guy who was going to implement it (yes, I know, we should all collectively own our tasks) needed the discussion: otherwise it wasn't going to get implemented. It certainly wasn't going to get broken into short tasks until we knew how we were going to do it.

Anyway, asides aside, I came out of that discussion bruised but triumphant. We had a plan not only on how to write the code, but also how to test it. I believe the key thing that slowly led the discussion from a FUD-throwing [3] contest into a constructive dialogue was the fact that we began to talk in code.

There are two facets to this principle:

1. Show me the code

As Linus once said, "Talk is cheap. Show me the code." [4].

If you are at all disagreeing about how what you're doing will work, open up the source files in question. Write example code – modify the existing methods or sketch a new one. Outline the classes you will need. Code is inherently unambiguous. White board diagrams and hand-waving are not.

Why wouldn't you do this? Fear you might be wrong? Perhaps you should have phrased your argument a little less strongly?

Is this slower than drawing boxes on a whiteboard? Not if you include time spent resolving the confusion caused by the ambiguities inherent in line drawings.

Does UML [5] make whiteboards less ambiguous? Yes, if all your developers can be bothered to learn it. But why learn a new language when you can communicate using the language you all speak all day – code?

2. Create a formal language to describe the problem

If your problem is sufficiently complex, you may want to codify the problem into a formal (text-based) language.

In last week's discussion we were constantly bouncing back and forth between different corner cases until we started writing them down in a formal language.

The language I chose was an adaptation of a Domain-specific language [6] I wrote to test a different part of our program. I would love to turn the cases we wrote down in that meeting into real tests that run after every build (in fact I am working on it) but their immediate value was to turn very confusing 'what-if's into concrete cases we could discuss.

Before we started using the formal language, the conversations went something like this:

Developer: If we implement it like that, this bad thing will happen.

Manager: That's fine – it's a corner case that we can tidy up later if we need it.

Developer: (Muttering) He clearly doesn't understand what I mean.

Repeat

After we started using the formal language they went something like this:

Developer: If we implement it like that, this bad thing will happen.

Me: Write it down, I tell you.

Developer: (Typing) See, this will happen!

Manager: That's fine – it's a corner case that we can tidy up later if we need it.

Developer: (Muttering) Flipping managers.

Summary

The conversation progresses if all parties believe the others understand what they are saying. It is not disagreement that paralyses conversations – it is misunderstanding.

To avoid misunderstanding, talk in code – preferably a real programming language, but if that's too verbose, a text-based code that is unambiguous and understood by everyone involved. ■

References

[1] http://www.scrumalliance.org/pages/scrum_ceremonies
[2] http://en.wikipedia.org/wiki/Timebox
[3] http://en.wikipedia.org/wiki/Fear,_uncertainty_and_doubt
[4] http://lkml.org/lkml/2000/8/25/132
[5] http://en.wikipedia.org/wiki/Unified_Modeling_Language
[6] http://www.martinfowler.com/bliki/DomainSpecificLanguage.html

Whiteboards

You can't copy and paste them, and you can't (easily) keep what you did with them, and you can't use them to communicate over long distances.

And don't even try and suggest an electronic whiteboard. In a few years they may solve all of the above problems, but not now. They fail the "can I draw stuff?" test at the moment.

Even when electronic whiteboards solve those problems, they won't solve the fact that lines and boxes are more ambiguous and less detailed than code in text form.

If you all know and like UML, that makes your diagrams less ambiguous, but still they often don't allow enough detail: why bother?

ANDY BALAAM

Andy is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

Standards Report

Mark Radford reports current discussions in C++ Standardisation.

Hello and welcome to my latest standards report.Since my last report, the ISO C++ committee has met (Rapperswil, 16th–21st June), and the BSI C++ Panel had its post-Rapperswil meeting on 21st July. Also, the post-Rapperswil mailing has been published [1].

For reasons I'll discuss at the end of this report, no work was done on the C++ working draft in Rapperswil. Instead, the meeting focussed its attention on doing work on the TS (technical specification) documents that are currently under development, as well as addressing defect reports. The status of each TS is reported in the Rapperswil meeting minutes (N4053), but I'll just mention that the Filesystem TS has made significant progress, with work being done on ballot resolution in order to progress it to the DTS (draft technical specification) stage.

Of all the things that were discussed in Rapperswil, I've singled out two to go into more detail about: the removal of the 'Executers and Schedulers' section from the concurrency TS, and the modules SG discussions. The former because of the potentially significant step it represents in the development of the concurrency TS. The latter because, although there are no important decisions to report, the addition of modules to C++ would represent a big change to the language. I also want to talk about two discussions from the BSI C++ Panel meeting: firstly, the proposal to add a 'const-propagating wrapper' to the standard library, and second, the proposal to consider allowing return expressions in curly braces to be subject to conversions even if the conversion is explicit. In the former case this is because the proposal is a very simple one, but it's so useful. In the latter case, because of the impact the proposal could have on the language, including the potential for silent changes to existing code.

The first discussion from Rapperswil I want to talk about is that concerning the paper *Working Draft, Technical Specification for C++ Extensions for Concurrency* (N4107). Compare this with its predecessor version (N3970, which can be found in the previous mailing), and you may notice a whole section has been removed. As I mentioned above, I am referring to the section that was entitled 'Executers and Schedulers'. The 'Executers and Schedulers' section was based on the proposal in 'Executors and schedulers, revision 3' (N3785), a proposal that has been around for a while and has been through several revisions. When the BSI Panel first discussed N3378 (the original version of N3785) opinions were mixed. Reasons for not favouring it varied, but many of them are summarised in Christopher Kohlhoff's paper *Executors and Asynchronous Operations* (N4046) which can be found in the pre-Rapperswil mailing. N4046 was discussed by the Concurrency and Parallelism Study Group (SG1) in Rapperswil, and it was met with a positive reception. SG1 are now considering which of the two alternatives to adopt. One thing that came out of the Rapperswil discussions is that SG1 would like to see wording for N4046. Wording is not supplied by N4046 because the author wanted to see what kind of reception it received before doing further work. Given the response from Rapperswil, I'm assuming there will be a revision of the paper that supplies wording (as well as taking account of any other feedback the author has received). The BSI C++ Panel conducted a follow up discussion on this paper at the meeting on July 21st. Given the interest in N4046, Christopher Kohlhoff (its author) was urged to attend the next ISO meeting (Urbana, IL, in November), given the contributions he could make to the discussion on this proposal.

SG2, the Modules Study Group, were once again active at the Rapperswil meeting, the last time being in Chicago last year. At the Chicago meeting they discussed some things in general as well as some high level goals. In Rapperswil they were discussing *A Module system for C++* (N4047). As far as I can see the first appearance of this paper was in the pre-Rapperswil mailing, so it won't have been discussed (by the ISO committee) before. Looking at the minutes of SG2s discussion, I can't see any firm conclusions I can report. However, from the discussion, it is clear that some problems are far from easy to solve. For example, could pre-processor macros be exported from a module?

Moving on to the BSI C++ Panel meeting, the first discussion I want to talk about concerns the paper *A Proposal to Add a Const-Propagating Wrapper to the Standard Library* (N4057), authored by Jonathan Coe and Robert Mill. This proposal addresses the behaviour of const member functions when they access const pointer members of the class. Specifically, if a member pointer is declared `const type *p;` then `p` is treated as `const`, but what it points to is not (note that this is the case even for some types of smart pointer, as illustrated in the paper). The paper proposes a `propagate_const<T>` wrapper that propagates the 'constness' through to the object pointed to. I feel this paper is proposing something that is long overdue. It was well received in the Panel discussion.

A paper that received a very different reception by the Panel is Herb Sutter's *Let return {expr} Be Explicit, Revision 2* (N4074). This paper proposes that an expression surrounded by curly braces, in a function return statement, should be considered explicit. That is to say: if a conversion is required, and the conversion is declared with the explicit qualifier, then in this special case the conversion can be executed without any explicit conversion in the return statement. Herb Sutter argues that this (inability to perform the conversion automatically) results in the most hated kind of error message that (to quote the paper) effectively says: "I know exactly what you meant. My error message even tells you exactly what you must type. But I will make you type it". That sounds logical on the surface, but there's more to it. In fact, there is so much more to it, that this is one of those cases where others have felt compelled to produce what you might call a 'rebuttal' paper. The problem is that ignoring the requirement for an explicit conversion can lead to unintended loss of information. Howard Hinnant and Ville Voutilainen give several such examples in their paper *Response To: Let return {expr} Be Explicit* (N4094) which gives the counter arguments and examples. The Panel felt that the change proposed by N4074 would be a bad thing and that the more support gained by N4094, the better.

That almost concludes this edition of my standards report, but just before I sign off, I'll give an update on where we are with C++14. The current draft is now at its DIS (draft international standard) stage until 15th August, awaiting yes/no votes from national bodies (a recommendation has been passed on to the BSI that the UK should vote 'yes'). What happens next depends on whether or not there are any 'no' votes. If there are none, the DIS is passed in its current form (without the need for a further round of votes) and will become an international standard. If there are any 'no' votes, the draft will be updated to take comments into account, before moving to its FDIS (final draft international standard) stage.

MARK RADFORD

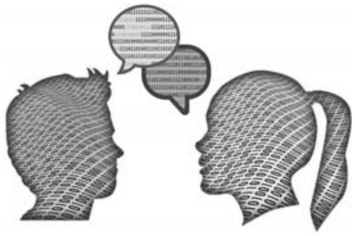
Mark Radford has been developing software for twenty-five years, and has been a member of the BSI C++ Panel for fourteen of them. His interests are mainly in C++, C# and Python. He can be contacted at mark@twonine.co.uk

References

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/#mailing2014-07

Code Critique Competition 89

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.

Last issue's code

> I'm trying to write a simple program to read and process lines of text from the console. I've got a problem – and have stripped the program down to a small demonstration of the it. If I run the program and type
>
> **`add 1 2`**
>
> it prints, as I'd expect:
>
> **`add( 1 2)`**
>
> and is back ready to read the next line and I can, for instance, type **`subtract 2 3`** and it echoes back **`subtract( 2 3)`**.
>
> But if I type just
>
> **`add`**
>
> then the program prints
>
> **`add(`**
>
> and although it seems to read more lines it no longer seems to process them.
>
> Additionally, with one compiler (MSVC), I get this warning and don't know why: **`cast between different pointer to member representations, compiler may generate incorrect code`** – which worries me.
>
> To be honest I don't really know why the **`static_cast`** is needed but I can't get it work any other way.

Can you help fix the problems presented and perhaps suggest some other improvements? The code is in Listing 1.

Listing 1

```
#include <iostream>
#include <map>
#include <sstream>
#include <string>
class Processor;
typedef void (Processor::*Pmf)(
  std::istream &is);
class Processor
{
public:
  void run(std::istream & is);
  void addCmd(std::string const &cmd, Pmf pmf)
  {
    cmds[cmd] = pmf;
  }
protected:
  Processor() = default;
private:
  std::map<std::string, Pmf> cmds;
};
void Processor::run(std::istream &is)
{
  std::string line;
  while (std::getline(is, line))
  {
    std::istringstream iss(line);
```

Listing 1 (Cont'd)

```
    std::string cmd;
    iss >> cmd;
    Pmf pmf = cmds[cmd];
    if (pmf) (this->*pmf)(iss);
  }
}
class Identity
{
  int id = getNext();
  static int getNext()
  {
    static int seed;
    return ++seed;
  }
public:
  int getIdentity() { return id; }
};
// (Stripped down code)
class Calculator : public Processor,
  public virtual Identity
{
public:
  Calculator();
private:
  void add(std::istream &is);
  void subtract(std::istream &is);
  // etc
};
Calculator::Calculator()
{
  addCmd("add",
    static_cast<Pmf>(&Calculator::add));
  addCmd("subtract",
    static_cast<Pmf>(&Calculator::subtract));
}
void Calculator::add(std::istream &is)
{
  // stub ...
  std::cout << "add("
    << is.rdbuf() << " )\n";
}
void Calculator::subtract(std::istream &is)
{
  // stub ...
  std::cout << "subtract("
    << is.rdbuf() << " )\n";
}
int main()
{
  Calculator calc;
  calc.run(std::cin);
}
```

ROGER ORR

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

Critiques

No-one took up the challenge this time; so we have no critiques. Those of you who meant to write one but didn't quite get round to it can try to put finger to keyboard for this one instead!

Commentary

The code has two main problems; the first the presenting problem where the program stops responding to user input and the second one related to the mysterious error message.

This is a stripped down version of a large program and so the actual content of the methods is a bit simplistic. However, it is a common enough pattern for many classes of program where you want to read input from the user and perform different operations, each of which requires slightly different additional input to be supplied.

One common problem with such programs is getting the input back to a known state when the user provides incorrect data. At first sight it seems that this program has avoided the problem since it reads input, one line at a time, into an **std::string** (the variable line in **Processor:::run**), so bad input provided on one line shouldn't affect the processing of the next line. So why does the program appear to be ignoring further input?

In fact, it isn't: what's happening here is in fact that the output stream is 'poisoned' while handling the bad input. The program writes the remainder of the input stream argument is to **std::cout** by passing it the **rdbuf()** from the input stream. Unfortunately for us the C++ standard for this output operation states: "If the function inserts no characters, it calls **setstate(failbit)**". So when the input stream read buffer is empty **cout** will be left in 'fail' state – the program in fact continues to process input from the user; it has simply stopped writing any output to **std::cout**!

This is different behaviour from what happens when streaming an empty **string**, for example:

```
Std::string emptyString("");
std::istringstream emptyStream("");

std::cout << emptyString.str(); // no output
assert(std.cout); // cout is still OK

std::cout << emptyStream.rdbuf(); // no output
assert(cout.fail()); // in 'fail' state
```

I'm not sure why this is the behaviour – I think it is something to do with the fact that writing a **streambuf** is treated as an unformatted I/O operation – but the end result in the case of this program is to mark **std::cout** in error and so subsequent output operations all fail.

So how can we fix this problem? There are several possible approaches we could take; for example we could check the input stream is not empty before trying to write the **rdbuf()**, we could set **std::cout** to throw an exception on fail state, or we could clear the fail bit using **std::cout.clear()** after each line of output. The right response will depend on the functional requirements of our program: in this case I might simply check there are characters available (using **!is.eof()**) before streaming the **rdbuf** to **cout**.

The second problem is the casting of the pointer to member function: we are trying to cast a pointer to a member of the **Calculator** class to a pointer to a member of **Processor**, a base class.

There are two problems with this. This first problem is that the resulting pointer to member can be invoked against any object of the **Processor** class, or classes derived from it, which includes ones that aren't instances of a **Calculator** and so don't contain the target method.

What would happen with code like this?

```
Pmf p = static_cast<Pmf>(&Calculator::add));
Processor baseOnly;
(baseOnly.*p)(is); // oh dear ...
```

We are trying to invoke the **add** method against an object of the **Processor** type, which has no such method. That explains why the code needs to use a **static_cast<>** to convert to the target type as what we are doing is type-unsafe. (Note that it ***is*** safe to cast in the reverse direction: if we take a pointer to a member of the base class and cast it to a pointer to a member of a derived class there is no potential for harm since the instance of the derived class *will* contain every method in the base class: either it is inherited from the base class or it is overridden somewhere in the class hierarchy.)

The second problem is more subtle: the **static_cast<>** is not guaranteed to *work*. The internal representation of pointer-to-member must be capable of containing the information necessary, when it is invoked, to obtain a **this** pointer from the supplied target pointer and to locate the target function. Since the target function may be a virtual function this is not a simple as merely storing the function address.

Implementations vary in how they achieve this: one technique (used by Microsoft Visual Studio) utilises *different sized* objects depending on how complicated the class hierarchy is and whether virtual functions are involved. The consequence is that a pointer-to-member of a fairly simple base class may be too small to hold the all the information required for a pointer-to-member of derived class with a complicated class hierarchy. You can see this at work when using the Microsoft compiler by printing the result of **sizeof** on the two pointer-to-member types. For example with a 64bit build:

```
sizeof(&Calculator::add): 16
sizeof(Pmf): 8
```

Other implementations, notably gcc, use a fixed sized object for all pointer-to-member functions and generate an internal helper function to 'wrap' the target member function for the more complex cases. While this may be very slightly more wasteful of memory it does mean that casting between pointer-to-members of different classes in a hierarchy is safe *whenever* the actual type of the target object is compatible with the member function held in the pointer-to-member.

This divergence in implementation means the mechanism used in this example is not guaranteed to work. In practice, as long as the base class has a 'similar-enough' layout to the derived class, it does work. In this example we have a derived class with a virtual base class and consequently the pointer-to-member needs to contain information about how to generate the **this** pointer for the correct sub-object. However, the offset conversion for the **this** pointer between **Calculator** and the **Processor** base class is 0 and so, in this specific example, we can call members of the **Calculator** class from pointer-to-members of the base class in the MSVC case. If we make **add** into a **virtual** function then the program crashes when we try to invoke it using a pointer-to-member of **Processor** as the format used in this class (which has no virtual members) does not hold enough information to be able to find the virtual function to call.

An alternative approach is to avoid the problem by using one of the more recent techniques added to the language, such as **std::function**, and not using the pointer-to-member mechanism directly. For example:

```
typedef
std::function<void(std::istream &is)> Func;
```

Then change **Pmf** to **Func** in the **addCmd** method and in the type of the field **cmds** to:

```
void addCmd(std::string const &cmd, Func func)
{
  cmds[cmd] = func;
}
std::map<std::string, Func> cmds;
```

And then populate it with:

```
Calculator::Calculator()
{
  using namespace std::placeholders;
  addCmd("add",
    std::bind(&Calculator::add, this, _1));
    ...
```

Finally, you can call the function using:

```
Func func = cmds[cmd];
if (func) func(iss);
```

I find **`std::function`** is simpler than trying to use pointers-to-members directly and you also gain flexibility if, for some reason, one of the methods you wish to invoke is in a different object completely.

This final small nit is that using **`cmds[cmd]`** *modifies* the map if an unknown command is encountered by adding a new entry. I would prefer using **`find`** and testing the item exists.

```
void Processor::run(std::istream &is)
{
  ...
    iss >> cmd;
    auto iter = cmds.find(cmd);
    if (iter != cmds.end())
    {
      iter->second(iss);
  ...
}
```

Code Critique 89

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

It must be time for a C one, I think.

> I'm trying to write a simple program to shuffle a deck of cards, but it crashes. What have I done wrong?

The code is in Listing 2.

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum
{
  black,
  red
};

enum
{
  Hearts,
  Diamonds
};

enum
{
  Clubs,
  Spades
};
```

Listing 2 (cont'd)

```
typedef struct Card
{
  int color;
  int suit;
  int value;
} Card;

typedef Card Deck[52];

void LoadDeck(Deck * myDeck)
{
  int i = 0;
  for(; i < 51; i++)
  {
    myDeck[i]->color = i % 2;
    myDeck[i]->suit = i % 4;
    myDeck[i]->value = i % 13;
  }
}

void PrintDeck(Deck * myDeck)
{
  int i = 0;
  for(;i < 52; i++)
  {
    char *colors[] = {"black", "red"};
    char *suits[][2] =
       {{"clubs", "spades"},
        {"hearts", "diamonds"}};
    printf("Card %s %d of %s\n",
      colors[myDeck[i]->color],
      myDeck[i]->value,
      suits[myDeck[i]->color]
           [myDeck[i]->suit]);
  }
}

void Shuffle(Deck * myDeck)
{
  int i = 0;
  for (; i < 52; i++)
  {
    int n = sizeof(Card);
    int to = rand() % 52;
    Card tmp;
    memcpy(&tmp, myDeck[i], n);
    memcpy(myDeck[i], myDeck[to], n);
    memcpy(myDeck[to], &tmp, n);
  }
}

int main()
{
  Deck myDeck;
  memset(&myDeck,0,sizeof(Deck));
  LoadDeck(&myDeck);
  PrintDeck(&myDeck);
  Shuffle(&myDeck);
  PrintDeck(&myDeck);
  return 0;
}
```

ACCU Information

Membership news and committee reports

accu

View from the Chair

Alan Lenton
chair@accu.org

Well, I'll start by thanking those of you who turned out to vote for me. I know there wasn't a choice, but it's nice to know that ACCU members are prepared to turn out and vote. Predictably no one (except committee members) turned up to the Special General Meeting – those who would normally come had already cast their vote online, and there was no other business to draw people to the meeting.

That being the case we had a committee meeting immediately after the Special General Meeting. The draft minutes should be available for members by the time you read this.

Last issue I asked if there was anyone who could help out with publicity.Unfortunately no one flocked to answer my request, so we are still looking.

And while I'm on the subject of the last 'View from the Chair', I haven't exactly been inundated with suggestions about how we could make the web site better. Which reminds me, the committee also asked me make an appeal for help migrating the membership system. So that's three things: publicity, web site improvements, and membership system migrations.

Ask not what ACCU can do for you, but what you can do for ACCU (apologies to John F. Kennedy).

OK, moving on, the question of the day is 'Whither ACCU'. An important question, and one that's going to present us with some stark choices in the next few years. For a number of years we have been faced with a slow, but steady, decline in membership. If this continues, at some stage we will become unviable as an organization. The failure to elect two of the officers at the last Annual General Meeting was a wake up call.

In view of this the committee has decided to hold a meeting in the autumn at which the only item on the agenda will be a discussion on how to reverse this situation. This is not to assume that only the committee has ideas on how to tackle this problem! We would be happy – very happy – to consider suggestions from members. I know for a fact that a significant number of members have their own ideas about where we should be going and how to grow. Drop me an email (chair@ACCU.org) if you have an opinion on this matter, and I'll collate the suggestions for the committee to consider. Whatever conclusion the committee comes to will go out to the membership long before any decisions are made at the AGM, and there will be time to put alternatives.

On a happier note, details of the keynote speakers for the 2015 Conference are out – Andrei Alexandrescu, James Coplien, Alison Lloyd, and Alex Neumann from CERN. Keep an eye on the web site for more details. By the time you read this magnificent magazine (hi Steve, can I have somewhere other than the last page in future, please?) the call for papers should be out. Go for it!

Well, I guess that's about all for this issue. Happy programming, and may all your compiler template errors be less that six screens long for each error...