the magazine of the accu

www.accu.org

{cvu}

Volume 30 • Issue 1 • March 2018 • £3

Features

Regulars

CONFIRMED KEYNOTE SPEAKERS


SEB ROSE


LISA LIPPINCOTT


GEN ASHLEY


HADI HARIRI

"Great conference with fantastic speakers and amazing subjects. I enjoyed it thoroughly."

ACCU 2017 Delegate

400+ attendees

4 days

60+ speakers

5 parallel streams

ACCU 2018

FOR FURTHER INFORMATION AND TO REGISTER, PLEASE VISIT https://conference.accu.org/

2018.4.11-14

pre-conference tutorials 2018.4.10

www.accu.org/conference

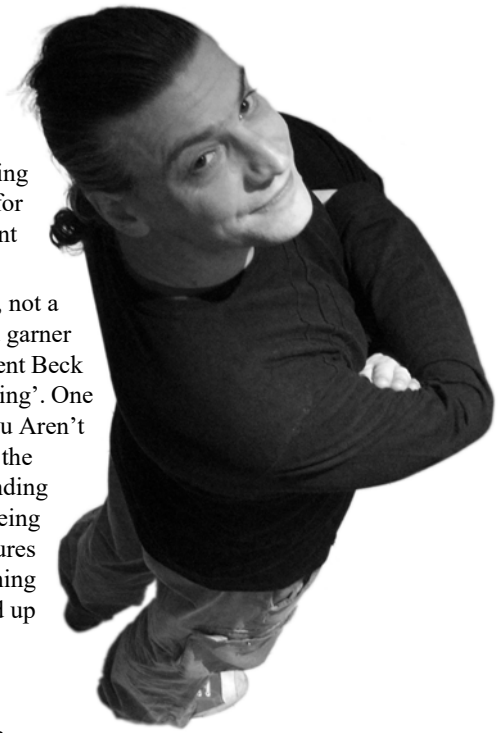
Follow @ACCUConf

Tweet #ACCUConf

BRISTOL MARRIOTT HOTEL CITY CENTRE

{cvu}
Volume 30 Issue1
March 2018
ISSN 1354-3164
www.accu.org

On Being Lazy



I sometimes talk to people about being a 'lazy' programmer. Laziness in this case isn't about laziness for its own sake, but it **is** about not doing things if you either don't have to, or if you know it'll save you effort in the long run. An example of the latter is often about identifying manual, repetitive tasks and automating them to reduce time, as well as the opportunity for mistakes. It's the former way of being lazy I want to look at here.

Not doing things you don't have to is, of course, not a new idea – in programming, or elsewhere. It did garner a kind of popularity about 20 years ago, when Kent Beck and others were advocating 'Extreme Programming'. One of the principles behind that was 'YAGNI' – You Aren't Gonna Need It. It was mainly a reaction against the prevailing ideas of big up-front design, and spending time dreaming up features that ended up never being used. It's a worthy cause, for sure, and also features in other lazy principles like doing the simplest thing that could possibly work (tm). It can be summed up as only implementing those features for which a concrete requirement (not just a wish) has been identified.

On the face of it, this seems fine, except that software development is never so neat and tidy. There are some non-functional or hidden requirements that are rarely considered by anyone except developers. One of these is application configuration. If a system isn't built from the beginning to be configurable so it can (for example) be run on a developer's workstation, or in any one of many deployment environments, it can be detrimental to the overall effort. It's certainly the case that *designing* a system to be flexible like this can be difficult and time consuming, but I think it's well-spent if it makes life easier down the road. It's rare that something like this is identified as a formal requirement though, and back-fitting it can be very difficult.

There are many examples like this that seem to fly in the face of the YAGNI principle. Please write in with your own!

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 30.2: 1st April 2018
C Vu 30.3: 1st June 2018

Overload 144: 1st May 2018
Overload 145: 1st July 2018

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

Testing Times (Part 2)

Pete Goodliffe continues the journey into software testing.

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

In the previous column, we started a journey into the world of software testing; we considered why it's important, why it should be automated, and who should be doing this. We looked at the types of test we perform, when we *write* them, and when we *run* them.

So, let's round this off by looking at what should be tested, and what good tests look like.

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 time frame, 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.

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.

These are the characteristics of a good test:

- Short, clear name, so when it fails you can easily determine what the problem is (e.g., *new list is empty*)
- Maintainable: it is easy to write, easy to read, and easy to modify
- Runs quickly
- Up-to-date
- Runs without any prior machine configuration (e.g., you don't have to prepare your filesystem 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 descriptions 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 or 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 work on only 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.

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 is an example Java unit test method that follows this pattern, and

PETE GOODLIFFE

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

Listing 1

```
@Test
public void stringsCanBeCapitalised()
{
  String input = "This string should be uppercase."; <1>
  String expected = "THIS STRING SHOULD BE UPPERCASE.";

  String result = input.toUpperCase();               <2>

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

- <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` we saw earlier), so you needn't add the word `test` to the name. The preceding example 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, as 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 reimplemented a version that satisfied the old tests and the new.

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

Picking a test framework

The unit or 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 choose 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* as a great alternative [1]).

Some frameworks provide pretty GUIs with red and 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 [2] and Cucumber [3]. 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 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 hardwired 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:

- Dummies

  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

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 chapter can only really scratch the surface, encourage you to test, and prompt you to find out more about good testing techniques. ■

Questions

- How can you best introduce test-driven development into a codebase that has never received automated testing? What kind of problems would you encounter?
- 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?
- If you don't already, start to write unit tests for your code today. If you already use tests, pay attention to how they inform and drive your code design.

References

[1] The Catch unit test framework (available from http://github.com/philsquared/Catch).
[2] Fit: http://fit.c2.com/
[3] Cucumber: http://cukes.info

Quaker's Dozen

The Baron once again invites us to take up a challenge.

Sir R-----, my fine friend! The coming of spring always puts one in excellent spirits, do you not find? Speaking of which, come join me in a glass of this particularly peaty whiskey with which we might toast her imminent arrival!

Might I tempt you with a little sport to quicken the blood still further?

It lifts my soul to hear it Sir!

I have in mind a game that I learned when in passage to the new world with a company of twelve Quakers. I was not especially relishing the prospect of yet another monotonous transatlantic crossing and so you can imagine my relief when I spied the boisterous party embarking, dressed in the finest silks and satins and singing a bawdy tavern ballad as they took turns at a bottle of what looked like a very fine brandy indeed!

I ingratiated myself with them in short order and, to my delight, discovered during our first night's meal at sea that they were no less keen sportsmen than I. Naturally, we followed each evening's repast at the captain's table with cigars, brandy, tales of derring-do and, most importantly, dice. Given my natural talent for wager, I was most surprised that I found myself unable to best them; at least until the final night of our voyage.

After I had retired with a small loss at the table the previous night, my fellow travellers continued their revels well into the following day. In consequence, they did not quite have their wits about them and at the evening's close I had had for my prize the contents of their purses, the entirety of their cargo and, indeed, the very shirts and blouses off of their backs.

For the sake of their modesty, I procured for them some plain labourer's clothing from the purser; they cut a sorry sight disembarking in quiet contemplation of their folly, I must say!

Here, take another dram and I shall explain the rules of their game!

Your goal is to cast a higher score with this pair of dice than I shall with a single twelve sided die and if you do so you shall have one coin for every point greater that your score is than mine. Now it shall cost you two coins and twelve cents to play but if you are dissatisfied with your dice you may cast them again for a further cost of one coin and twelve cents, and again and again for the same cost each time until you are satisfied, at which point I shall cast mine.

- Sir R-----'s First roll

- Sir R----- Rolls again

- Sir R----- Sticks and wins

- Sir R-----'s payoff

When I told that godforsaken student, who fate has cruelly decreed that I cannot evade, he began cooing about a gal named Constance with whom he had become quite struck. That the coming of spring will surely invigorate saplings as well as oaks must come as cold comfort to the poor benighted maiden.

But let us not dwell upon her misfortune! Let me refresh your glass whilst you decide whether to play! ■

Baron M

Courtesy of www.thusspakeak.com

BARON M

In the service of the Russian military the Baron has travelled widely in this world, and many others for that matter, defending the honour and the interests of the Empress of Russia. He is renowned for his bravery, his scrupulous honesty and his fondness for a wager.

The Expressive C++ Coding Challenge in D

Sebastian Wilzbach presents a D language solution to a C++ problem.

You might have seen that I have been coding a lot in D lately and as a few weeks ago there was the 'Expressive C++17 Coding Challenge' [1] with its solution in C++ [2] and Rust [3] now being public, I thought this is an excellent opportunity to show why I like D so much.

The requirements

Let me first recap the requirements of this challenge:

This command line tool should accept the following arguments:

- the filename of a CSV file,
- the name of the column to overwrite in that file,
- the string that will be used as a replacement for that column,
- the filename where the output will be written.

```
./program <input.csv> <colum-name> <replacement-string> <output.csv>
```

Example input

Given this simple CSV as input

```
name,surname,city,country
Adam,Jones,Manchester,UK
Joe,Doe,Cracow,Poland
Michael,Smith,Paris,France
Alex,McNeil,Gdynia,Poland
```

the program called with:

```
./program input.csv city London output.csv
```

should write:

```
name,surname,city,country
Adam,Jones,London,UK
Joe,Doe,London,Poland
Michael,Smith,London,France
Alex,McNeil,London,Poland
```

Sounds fairly trivial, right?

Please have a short look first at the 'best' C++ [4] and Rust [3] solutions before you look at the D solution.

The solution

Okay, so Listing 1 is one way to solve this in D.

If you are scared at the moment – don't worry. I will explain it line by line below.

So how does this compare to C++17 and Rust?

Language	LoC	Time
C++	125	15s
Rust	83	6s
D	19	12s
D (slightly tweaked)	25	6s

Later in the article I will present a solution which only uses 12 lines, but it also uses the built-in `std.csv` module and I think D doesn't need to cheat.

I used the following D script to generate a simple CSV file with 10 fields and 10 million lines:

Listing 1

```
#!/usr/bin/env rdmd
import std.algorithm, std.exception, std.format,
std.range, std.stdio;
void main(string[] args) {
  enforce(args.length == 5, "Invalid args\n" ~
  "./tool <input.csv> <colum-name>
    <replacement-string> <output.csv>");
  auto inFile = args[1], columName = args[2],
    replacement = args[3], outFile = args[4];
  auto lines = File(inFile).byLine.map!(a
    => a.splitter(","));
  auto colIndex = lines.front.countUntil(
    columName);
  enforce(colIndex >= 0,
    "Invalid column. Valid columns: %(%s,
    %)".format(lines.front));
  auto os = File(outFile, "w");
  os.writefln("%(%s, %)", lines.front);
  foreach (line; lines.dropOne) {
    auto r = line
      .enumerate // iterate with an (index,
                 // value) tuple
      .map!(a => a.index == colIndex
         ? replacement : a.value)
      .joiner(",");
    os.writeln(r);
  }
}
```

```
rdmd --eval='10.iota.map!(
  a=> "field".text(a)).join(",")
.repeat(10_000_000).joiner("\n").writeln'
  > input_big.csv
```

The resulting `input_big.csv` has a size of 668M.

Aren't you concerned that Rust is faster in this benchmark?

Not at all. The challenge was to write *expressive* code.

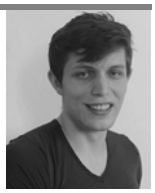
When performance really matters, D provides the same tools as C or C++ and D even supports native interoperability with C and most of C++.

In this example, however, I/O is the bottleneck and D provides a few convenience features like using locked file handles, s.t. accessing files is thread-safe by default, or supporting unicode input.

However, it's easy to opt out of such productivity features and use other tricks like memory mapped files [5]. For interested readers, I have included a slightly optimized version at the end.

In addition, if you are interested in performance, Jon Degenhardt (member of eBay's data science team), has made an excellent performance benchmark [6] between eBay's tsv-utils and existing CSV/TSV processing tools written in C, Go, and Rust.

SEBASTIAN WILZBACH



Sebastian is a D enthusiast who spends his nights submitting pull requests to make the D language even better. In his other life, he studies computational biology in Munich and loves rock climbing. He can be contacted at seb@wilzba.ch.

1) What is #!/usr/bin/env rdmd?

One of my favorite parts of D is that it has a blazingly fast compiler. Period. I can compile the entire D front-end of the compiler (~200 kLoC) in less than two seconds or the entire standard library with lots and lots of compile-time function evaluation and templates and > 300 kLoC in 5 seconds *from scratch without any cache or incremental compilation*.

This means that the compiler is almost as fast as an interpreter and the `rdmd` tool allows you to use D as 'pseudo-interpreted' language. You can invoke `rdmd` with any file and it will automatically figure out all required files based on your dependencies and pass them to the compiler.

It's very popular in the D community because for small scripts one doesn't even notice that the program is compiled to real machine code under the hood. Also if the shebang header is added and the file is executable, D scripts can be used as if they were script files:

```
./main.d input.csv city London output.csv
```

2) So you import a bunch of libraries. What do they do?

```
import std.algorithm, std.exception, std.format,
std.range, std.stdio;
```

In short `std.stdio` is for input and output, `std.range` is about D's magic streams called 'ranges' and `std.algorithm` abstracts on top of them and provides generic interfaces for a lot of sophisticated algorithms.

Moreover, `std.exception` offers methods for working with exceptions like `enforce` and finally `std.format` bundles methods for string formatting.

Don't worry – the functionality imported from these modules will be explained soon.

3) Your program has a main function. What's so special about it compared to C or C++?

```
void main(string[] args) {
  …
```

For starters, arrays in D have a length. Try:

```
args[5].writeln;
```

Compared to C/C++ null-terminated strings and arrays, it won't segfault. It would just throw a nice Error (see Figure 1).

Figure 1

```
core.exception.RangeError@./main.d(10): Range violation
----------------
??:? _d_arrayboundsp [0x43b622]
prog.d:9 void main.foo(immutable(char)[][]) [0x43ac93]
prog.d:4 _Dmain [0x43ac67]
```

Oh so D performs automatic bounds-checking before accessing the memory. Isn't that expensive?

It's almost negligible compared to the safety it buys, but D is a language for everyone, so the people who want to squeeze out the last cycles of their processor can do so by simply compiling with `-boundscheck=off` (for obvious reasons this isn't recommended).

In D, strings are arrays too and there's another nice property about D's arrays. They are only a view on the actual memory and you don't copy the array, but just the view of the memory (in D it's called a slice).

Consider this example:

```
int[] arr = [1, 2, 3];
auto bArr = arr[1 .. $];
bArr[] += 2; // this is a vectorized operation
arr.writeln; // [1, 4, 5]
```

There many other things D has learned from C and C++.

Walter has recently written a great article [7] on how D helps to *vanquish forever these bugs that blasted your kingdom*, which I highly recommend if you have a C/C++ background.

4) What's up with this 'enforce'?

```
enforce(args.length == 5, "Invalid args.\n" ~
"./tool <input.csv> <colum-name> <replacement-
  string> <output.csv>");
```

I have never seen the ~ operator before!

It's the string concatenation (or more general array concatenation) operator. How often how you encountered code like `a + b` and needed to know the types of `a` and `b` to know whether it's a addition or concatenation?

Why don't you use an `if` *statement and terminate the program explicitly?*

```
if (args.length < 5) {
  writeln("Invalid args.");
  writeln("./tool <input.csv> <colum-name>
    <replacement-string> <output.csv>");
  return 1;
}
```

That's valid D too. D allows a lot of different programming styles, but this article is intended to highlight a few specific *D styles* like `enforce`.

`enforce` [8] is a function defined in `std.exception` and throws an exception if its first argument has a falsy value.

Hmm, I looked at the documentation [8] and saw this monster. I thought it simply throws an exception?

```
auto enforce(E : Throwable = Exception, T)
  (T value, lazy string msg = null, string file =
  __FILE__, size_t line = __LINE__)
```

I don't have the time to fully dive into D's syntax, but `auto` instructs the compiler to infer the return type for you. This leads to the interesting *Voldemort* return types [9] as they can't be named by the user, but that's a good topic for another article.

The next part looks a bit complicated `(E : Throwable = Exception, T)`, but don't worry yet. It means that `E` is a template parameter which needs to inherit from `Throwable` (the root of all exceptions), and is by default `Exception`. `T` is the template type of `value`.

Wait. I just instantiated a template without specifying its template parameters?

Yes, the D compiler does all the hard work for you.

The technical term is Implicit Function-Template Instantiation [10] (IFTI). Of course, we could have instructed `enforce` to throw a custom exception, but more on template instantiation later.

Alright. So this function takes a generic `value` *and a* `msg`*, but a* `lazy string msg`*?*

`lazy` is a special keyword in D and tells the compiler to defer the evaluation of an argument expression until is actually needed.

I don't understand. `msg` *seems to be a string concatenation of two strings. Isn't this done* ***before*** *the* `enforce` *is called?*

```
"Invalid args.\n" ~ "./tool <input.csv>
  <colum-name> <replacement-string> <output.csv>"
```

No, `lazy` is lazy and the string concatenation doesn't happen at the caller site, but can be requested explicitly by the callee.

It gets a bit clearer if we look at the second `enforce`:

```
enforce(colIndex < 0, "Invalid column name. Valid
  are: %(%s, %)".format(lines.front));
```

`format` and all the expensive work of formatting the error message is never done on the default path, but only if an exception actually gets thrown. Ignore the `%(%s, %)` formatting string for a bit, it will be explained soon.

Ok, but how does that work?

In short: the compiler performs some smart formatting for you and creates an anonymous lambda. For more details, see this advanced article about D's `lazy` [10].

But there's more magic here. What's `__FILE__` *and* `__LINE__`*?*

```
string file = __FILE__, size_t line = __LINE__
```

Remember that D is a compiled language and accessing the stack isn't as easy as asking the interpreter nicely. These two default arguments are automatically set by the compiler with the file and line number of the caller. This is important for logging or throwing exceptions like we have done here.

So an API author can simply say "Hey, I would like to know the line number of my caller." and doesn't depend on the user hacking the replacements as you'd have to do in C/C++ with the preprocessor macros:

```
#ifdef SPDLOG_DEBUG_ON
#define SPDLOG_DEBUG(logger, ...) logger-> \
  debug(__VA_ARGS__)  << " (" << __FILE__  \
  << " #" << __LINE__ <<")";
#else
#define SPDLOG_DEBUG(logger, ...)
#endif
```

In fact, D doesn't even have a preprocessor.

5) auto and a statically typed language

```
auto inFile = args[1], columName = args[2],
  replacement = args[3], outFile = args[4];
```

Hmm, but what's `auto`*? I thought D was a statically typed system?*

Yes D is statically typed, but the compiler is pretty smart, so we can let it do all the hard for us. `auto` is a filler word for the compiler that means 'whatever the type of the assignment, use this as type of this variable'.

6) What the hell is UFCS?

```
auto lines = File(inFile).byLine.map!(a =>
  a.splitter(","));
```

One of the major features of D is the Unified Function Call Syntax (UFCS) [11]. In short, the compiler will look up a function in the current namespace if it's not found as a member function of a type, but let's go through this step by step.

I looked at the documentation of `File` *[12] and it has a method* `byLine` *[13]. So where's the magic?*

Have another look at `map` [14], it's located in `std.algorithm`.

Okay, wait. How does this work?

The compiler internally rewrites the expression `File.byLine.map` to the following:

```
map(File.byLine());
```

Missing parenthesis are allowed too – after all, the compiler knows that the symbol is a function.

Okay, but what's up with this `!(a => a.splitter(","))`*?*

`!` is similar to C++/Java's `<>` and declares a template. In this case it's a lambda function of `a => a.splitter(",")`. Notice that for `splitter` [15], UFCS is used again and your brain might be more used to reading `splitter(a, ",")` for now.

7) Ranges

Okay to recap, we have taken the input of a file, and split lines using commas `,`.

Wouldn't this result in a lot of unnecessary allocation?

The short answer is: D uses 'iterators on steroids' which are lazy and work is only done when explicitly requested. Usually range algorithms don't even require any heap allocation as everything is done on the stack.

For example, in the next line `.front` returns the first line through which `countUntil` explicitly iterates:

```
auto colIndex =
  lines.front.countUntil(columnName);
```

So `lines.front` looks something like:

```
["name", "surname", "city", "country"]
```

`countUntil` will return the line of the first match or `-1` otherwise. It's a bit similar to an `indexOf` function familiar from, for example, JavaScript, but it accepts a template. So we could have supplied a custom predicate function:

```
lines.front.countUntil!(a => a.endsWith("ty"));
```

8) std.format: and compile-time checking of parameters

The next lines are:

```
enforce(colIndex >= 0, "Invalid column name.
  Valid are: %(%s, %)".format(lines.front));
auto os = File(outFile, "w");
os.writefln("%(s, %)", lines.front);
```

I have never seen `writefln("%(s, %)")` *before. What happens here?*

`writefln` [16] is just a handy wrapper around D's `format` [17] function.

`format` itself provides a lot of options for serialization, but it's very similar to `printf`, although it does provide a few goodies like the special syntax for arrays `%(s, %)`.

This syntax opens an array formatting 'scope' delimited by `%(` and closes it with `%)`. Within this array 'scope' the elements should be formatted with `s` (their `string` serialization) and use `,`, a delimiter between the element.

It's a shorthand syntax that often comes in handy, but if you don't like it there are many other ways to achieve the same result. For example, `joiner`:

```
lines.front.joiner(",").writeln;
```

What would such an error message look like?

It would look like Figure 2.

Okay, but isn't `printf` *bad and unsafe? I heard that languages like Python are moving away from C-like formatting.*

A Python library can only realize that arguments and formatted string don't fit when it's called. In D the compiler knows the types of the arguments and if you pass the format string at compile-time, guess what, the format can be checked compile-time. Try to compile a format string that tries to format strings as numbers:

```
writefln!"%d"("foo");
```

The compiler will complain:

```
/dlang/dmd/linux/bin64/../../src/phobos/std/
stdio.d(3876): Error: static assert  "Incorrect
format specifier for range: %d"
onlineapp.d(4):        instantiated from here:
writefln!("%d", string)
```

Wow, that's really cool. How does this work?

```
  object.Exception@./main.d(9): Invalid column name. Valid are: "name", "surname", "city", "country"
----------------
??:? pure @safe void std.exception.bailOut!(Exception).bailOut(immutable(char)[], ulong,
const(char[])) [0x7a34b57e]
??:? pure @safe bool std.exception.enforce!(Exception, bool).enforce(bool, lazy const(char)[],
immutable(char)[], ulong) [0x7a34b4f8]
??:? _Dmain [0x7a34b17f]
```

Figure 2

D has another unique feature: compile-time function evaluation (CTFE) that allows to execute almost any function at compile-time. All that happens is that **writefln** is instantiated at compile-time with the string as template argument and then it calls the same **format** function that would normally be called at run-time with the known string. The coolest part about this is that there's no special casing in the compiler and everything is just a few lines of library code.

9) Let's parse the file

Now that we have found the index of the replacement column, have opened the output csv file and have already written the header to it, all that's left is to go over the input CSV file line by line and replace the specific CSV column with the **replacement** (Listing 2).

One of the cool parts of D ranges is that they are so flexible. You want to do everything in a functional way? D has you covered (Listing 3).

There's another cool thing about D – **std.parallelism**. Have you ever been annoyed that a loop takes too long, but didn't know a quick way to parallelize your code? Again, D has you covered with **.parallel** [18]:

```
foreach (line; lines.parallel)
  // expensive operation in parallel
```

No way. I don't believe this can be so simple.

Just try it yourself [19].

The Garbage Collector (GC)

On the internet and especially on reddit and HackerNews there's a huge criticism of D's decision to do use a GC. Go, Java, Ruby, JavaScript, etc. all use a GC, but I can't better phrase it than Adam D. Ruppe:

> D is a pragmatic language aimed toward writing fast code, fast. Garbage collection has proved to be a smashing success in the industry, providing productivity and memory-safety to programmers of all skill levels. D's GC implementation follows in the footsteps of industry giants without compromising expert's ability to tweak even further.

So ask your question:

Okay, "ability to tweak even further" sounds a bit vague, what does this mean? I can tweak the memory usage?

Well, of course you can do that, but that's something most languages with a GC allow you to do. D allows you to get the benefit of both worlds: profit from the convenience of the GC *and* use manual allocation methods for the hot paths in your program. This is great, because you can use *the same language* for prototyping and shipping your application.

A short and simplified summary of allocation patterns in D:

- **malloc** and friends are available in D (everything from C is)
- RAII is supported (e.g. **File** you saw earlier is reference-counted and automatically deallocates its buffer and close the file once all references are dead)
- there's **std.experimental.allocator** for everyone with custom allocation needs

Listing 2

```
foreach (line; lines.dropOne)
  // remove the header
{
  auto r = line
    .enumerate // iterate with an (index, value)
               // tuple and lazily map a different
               // value for the specific CSV column
    .map!(a => a.index == colIndex ? replacement
      : a.value),
    .joiner(","); // join the lines back to a CSV
                  // format
    os.writeln(r);
}
```

Listing 3

```
alias csvPipe = pipe!(enumerate,
  map!(a => a.index == colIndex ? replacement
    : a.value), partial!(reverseArgs!joiner,
    "_"),);
lines.dropOne.map!csvPipe.each!(
  a => os.writeln(a));
```

- **std.typecons** provides a lot of library goodies like **Unique**, **Scoped**, **RefCounted** for **@nogc** allocation.

Mike Parker has recently started an extensive GC Series [20] on the DBlog which I recommend to everyone who prefers performance over convenience.

Other goodies

std.csv

Hey, I saw that there's **std.csv** *in D, why didn't you use it?*

Simple – it felt like cheating. See Listing 4.

std.getopt

One of the reasons why this challenge used positional arguments and no flags is that argument parsing is pretty hard in C++. It's not in D. **std.getopt** provides convenience for everything out of the box. See Listing 5.

DMD, LDC and GDC

One of the things that newcomers are often confused by is that D has three compilers. The short summary is:

- DMD (DigitalMars D compiler) – latest greatest features + fast compilation (= ideal for development)
- LDC (uses the LLVM backend) – battle-tested LLVM backend + sophisticated optimizers + cross-compilation (= ideal for production)
- GDC (uses the GCC backend) – similar points as LDC

Benchmark and performance

Benchmarking a language compiler is a bit tricky as very often you end up benchmarking library functions. In general, D code can be as fast as C++ and often is even faster – after all the LDC and GDC compilers have the same backend as clang++ or g++ with all its optimization logic. If you are interested to see how D programs perform against similar programs written in other languages, checkout Kostya's benchmarks [21].

There's also an excellent performance benchmark [6] from Jon Degenhardt (member of eBay's data science team) on how eBay's **tsv-**

Listing 4

```
import std.algorithm, std.csv, std.functional,
  std.file, std.range;

void main(string[] args)
{
  auto inputFile = args[1], columnName = args[2],
    replacement = args[3], outputFile = args[4];
  auto records = inputFile.readText.csvReader!(
    string[string])(null);
  outputFile.write(records.map!((r) {
    r[columnName] = replacement;
    return r;
  }).pipe!(rows => records.header.join(",") ~
    "\n" ~ rows.map!(
    r => records.header.map!(
    h => r[h]).join(",")).join("\n")
  ));
}
```

Listing 5

```
import std.getopt;
int main(string[] args)
{
  string input, output, selectedColumn,
    fill = "FOO";
  auto opts = getopt(args,
    "i|input", &input,
    "o|output", &output,
    "s|select", "Select a column to overwrite",
      &selectedColumn,
    "f|fill", "Overwrite (default: FOO)", &fill,
  );
  if (opts.helpWanted || input.length == 0) {
    defaultGetoptPrinter("./program",
      opts.options);
    return 1;
  }
  return 0;
}
```

`utils` [6] compare against existing CSV/TSV processing tools written in C, Go, and Rust.

@safe

Even though D is a system programming language that allows you to mess with pointers, raw memory and even inline assembly, it provides a sane way to deal with the dirty details. D has a `@safe` subset [22] [23] of the language in which the compiler will enforce that you don't do anything stupid thing and shoot yourself in the feet with e.g. accessing undefined memory.

Unittest

One strategic advantage of D is that unit-testing is so easy as it's built-in in the language and compiler. This is a valid D program:

```
unittest {
  assert(1 == 2);
}
```

And with `-unittest` the compiler can be instructed to emit a unittest block to the object files or binary. Here, `rdmd` is again a friendly tool and you can directly go ahead and test your line with you this:

```
rdmd -main -unittest test.d
```

No advanced tooling setup required. Of course, this also means that it's particulary easy to automatically verify all examples that are listed in the documentation, because they are part of the testsuite. I even went one step further and made it possible to edit and run the examples on dlang.org [23].

Other cool D features

There are many other cool features that D offers that didn't make it into this article, but as a teaser for future articles:

- Code generation within the language (cut down your boilerplate)
- Strong and easy Compile-Time introspection (Meta-programming)
- `alias this` for subtyping
- `-betterC` (using D without a runtime)
- `mixin` for easily generating code
- A module system that doesn't suck
- `debug` attribute to break out of `pure` code
- Built-in documentation
- Contracts and invariants
- `scope(exit)` and `scope(failure)` for structuring creation with its destruction
- Native interfacing with C (and most of C++)
- `with` for loading symbols into the current name

For a full list, see the 'Overview of D' [24] and don't forget that the full language specification [25] is readable in one evening.

Downsides

Okay, so you say D is so great, but why hasn't it taken off?

There's a lot more to a programming language than just the language and compiler. D has to fight with the problems all young languages have to deal with e.g. small ecosystem, few tutorials/sparse documentation and occasional rough edges. Languages like Kotlin, Rust or Go have it a lot easier, because they have a big corporate sponsor which gives these language a big boost.

Without such a boost, it's a chicken/egg problem: if nobody is learning D, it also means that no one can write tutorials or better documentation. Also many people have learnt a few languages and use them in production. There's little incentive for them to redesign their entire stack.

However, things improved greatly over the last years and nowadays even companies like Netflix, eBay, or Remedy Games use D. A few examples:

- the fastest parallel file system for High Performance Computing [26] is written in D
- if you drive by train in Europe, chances are good that you were guided by D (Funkwerk [27] – the company that manages the transport passenger information system – develops their software in D)
- if you don't use an Adblocker, chances are good that algorithms written in D bid in real-time for showing you advertisement (two of the leading companies in digital advertising (Sociomantic and Adroll) use D)

The 'organizations using D' page [28] lists more of these success stories.

Of course, D – like every other language – has its 'ugly' parts, but there's always work in progress to fix these and compared to all other languages I have worked with, the ugly parts are relatively tiny.

Where to go from here?

Okay that sounds great, but how do I install D on my system?

Use the install script [29]:

```
curl https://dlang.org/install.sh | bash -s
```

or use your package manager [30].

And start hacking!

It's possible to do three easy tweaks to make I/O faster in D:

- disabling auto-decoding with `byCodeUnit`
- non-thread-safe I/O with `lockingTextWriter`
- use of `std.mmfile` [5]

Acknowledgements

Thanks a lot to Timothee Cour, Juan Miguel Cejuela, Jon Degenhardt, Lio Lunesu, Mike Franklin, Simen Kjærås, Arredondo, Martin Tschierschke, Nicholas Wilson, Arun Chandrasekaran, jmh530, Dukc, and ketmar for their helpful feedback.

References

[1] https://www.fluentcpp.com/2017/09/25/expressive-cpp17-coding-challenge/
[2] https://www.fluentcpp.com/2017/10/23/results-expressive-cpp17-coding-challenge/
[3] https://gist.github.com/steveklabnik/ad0a33acc82e21ca3f763e4278ad31a5#file-main-rs-L36
[4] http://coliru.stacked-crooked.com/a/70f762ee7f9c2606
[5] Memory mapped files: https://dlang.org/phobos/std_mmfile.html
[6] Performance benchmark: https://github.com/eBay/tsv-utils-dlang/blob/master/docs/Performance.md

Getting Personal

Chris Oldwood considers the effect of personal choice on delivering software.

The history of software development is littered with wars between factions who have differing opinions on various aspects of the way it should be analysed, designed, presented, tested, etc. Aside from the tabs vs spaces debate the other giant flame war is almost certainly around the choice of text editor – vi vs emacs. While the devotees of each side remain steadfast in their cause there are other groups who seek to douse the flames either by making the point a moot one (the 'pragmatists') or by taking choice away altogether by mandating a preference. This latter approach can, and does make sense in various circumstances, but it often gets taken too far so that even when it doesn't matter the only choice is effectively Hobson's choice – no choice at all.

Consistency – the sacred altar

When it comes to tooling, most notably in the enterprise arena, the organisation will classically play its trump card – consistency. In its endeavour to treat IT as a cost centre, and therefore be made as efficient as possible, their desire appears to be to limit the choice of tooling as much as possible on the premise that it's then easier to move 'resources' around because there will be a shorter learning curve, at least technology wise. Hence technology stacks tend to be limited to a set of core languages and products, both for the system being built and the supporting development 'infrastructure', i.e. CI service, test framework, deployment tools, etc.

CHRIS OLDWOOD

Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise-grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

The Expressive C++ Coding Challenge in D (continued)

Listing 6

```
#!/usr/bin/env rdmd
import std.algorithm, std.exception, std.format,
  std.mmfile, std.range, std.stdio, std.utf;

void main(string[] args) {
  enforce(args.length == 5, "Invalid args\n" ~
  "./tool <input.csv> <colum-name>
  <replacement-string> <output.csv>");
  auto inFile = args[1], columName = args[2],
    replacement = args[3].byCodeUnit,
    outFile = args[4];
  scope mmFile = new MmFile(args[1]);
  auto lines = (
    cast(string) mmFile[0..mmFile.length])
    .splitter('\n').map!(
    a => a.byCodeUnit.splitter(","));
    auto colIndex =
      lines.front.countUntil!equal(columName);
    enforce(colIndex >= 0,
      "Invalid column. Valid columns: %(%s, %)"
      .format(lines.front));
    auto os = File(outFile, "w");
    os.writefln("%(%s, %)", lines.front);
    auto w = os.lockingTextWriter;
    foreach (line; lines.dropOne) {
      auto r = line
      .enumerate // iterate with an
                 // (index, value) tuple
     .map!(a => choose(a.index == colIndex,
       replacement, a.value))
     .joiner(",".byCodeUnit);
    w.put(r);
    w.put('\n');
  }
}
```

[7] https://dlang.org/blog/2018/02/07/vanquish-forever-these-bugs-that-blasted-your-kingdom/
[8] https://dlang.org/phobos/std_exception.html#enforce
[9] https://wiki.dlang.org/Voldemort_types
[10] https://dlang.org/articles/lazy-evaluation.html
[11] https://tour.dlang.org/tour/en/gems/uniform-function-call-syntax-ufcs
[12] https://dlang.org/phobos/std_stdio.html#.File
[13] https://dlang.org/phobos/std_stdio.html#.File.byLine
[14] https://dlang.org/phobos/std_algorithm_iteration.html#.map
[15] https://dlang.org/phobos/std_algorithm_iteration.html#splitter
[16] https://dlang.org/phobos/std_stdio.html#.File.writefln
[17] https://dlang.org/phobos/std_format.html#.format
[18] https://dlang.org/phobos/std_parallelism.html#.parallel
[19] https://run.dlang.io/is/9fbtpQ
[20] https://dlang.org/blog/the-gc-series/
[21] https://github.com/kostya/benchmarks
[22] https://dlang.org/articles/safed.html
[23] https://dlang.org/blog/2017/03/08/editable-and-runnable-doc-examples-on-dlang-org/
[24] https://dlang.org/overview.html
[25] https://dlang.org/spec/spec.html
[26] https://www.theregister.co.uk/2017/12/22/a_dive_into_wekaios_parallel_file_system_tech/
[27] https://dlang.org/blog/2017/07/28/project-highlight-funkwerk/
[28] https://dlang.org/orgs-using-d.html
[29] Install script: https://dlang.org/install.html
[30] Package manager: https://dlang.org/download.html

Don't get me wrong, consistency of tooling is undeniably important. The modern 'virtual machine' approach to runtimes such as the JVM and .Net allow for different components to be written in entirely different languages and paradigms, but actually using a multitude of languages within the same application is likely to cause more friction than it solves unless the team are all seriously experienced polyglots. Similarly, maintaining your own personal build scripts because you don't like the team's is unlikely to be the best use of one's time either. One size never fits all but a few well picked sizes might fit most problems well enough that the cost/benefit ratio is favourable without being over constraining.

Essentially we are talking about architectural level stuff here – the kind of choices where changing your mind or going against the grain is likely to incur some non-trivial expense, usually in the form of time, and therefore indirectly in money.

Room for personal taste

Consistency however is also not the end of the argument. Just as my choice of desktop wallpaper or text editor font or colour scheme does not impact what I deliver, so the same goes for a number of tooling choices in the overall development process. Sadly there appears to be a lack of recognition that software development is far more than simply churning out source code with an IDE. There are numerous additional activities whilst programming such as research, design, documenting, testing, support, etc. that demand very little consistency in tooling as the tools themselves provide no material impact on the delivered artefacts (source code, documentation, etc.), particularly when the artefact is of a common, interoperable file format such a text file.

In an earlier episode of this column [1] I described how the answer to the simple question of 'what tool do you use to find text' is a complex affair with a dizzying array of answers that all depend heavily on the context of the problem and the familiarity of the tools at hand. I am far more au fait with the command line switches of the classic Unix grep tool than I am of the Windows built-in find and findstr despite spending virtually my entire career to date on that one platform. Over the years Microsoft have added a few more command line tools here and there but many essentials, like curl and tar are only just seeing the light of day which means we have to rely on third parties to fill the void; or create our own [2]. There is an air of irony about being labelled 'inconsistent' when you prefer to use the same tools as the wider development community over the smaller organisational sized one.

Pairing/mobbing

There is perhaps one area where differences in tooling could generate some unnecessary friction and that is when pairing or mobbing on a problem. The entire premise of the exercise is to share one machine and focus on solving the problem together by allowing anyone to just get on and move the solution further forward by being able to take over control of the keyboard and 'just start typing'. Putting a copy of Vim in front of an entrenched Visual Studio developer or a Cygwin bash prompt towards a CMD prompt die-hard is not going to be a recipe for rapid progression if the impedance mismatch is a continual distraction.

That's the theory, but I'm happy to report that I've found it doesn't really play out like that in practice, at least not with those people I've paired with. Whilst the keyboard may not have moved around quite as freely as you might hope, it's fairly easy to remain focused on the problem and type enough of a snippet to get your point across without having to go on and produce production ready refactored code or a blazing one-liner; the 'driver' can easily adapt and finish off your 'scribbles'.

If anything I prefer watching other people using other tools because that's when you get to see what you're missing out on. A perfect example of this was multi-cursor support in Sublime Text.

Common battlegrounds

The following non-exhaustive list contains some of the bigger areas of contention that have arisen in the past for which I personally feel are outside the scope of effecting delivery and provide a significant enough impact on productivity that it's worth fighting for.

Version control system client

The war on version control systems is over – git won. Unfortunately, like databases, teams rarely switch their VCSs at the flick of a switch. Consequently the long tail contains CVS, Visual SourceSafe, Subversion, TFS, etc. Even Microsoft has seen the light of day and git is now promoted to a first class citizen in its ongoing transformation. What this means is that for those of us who have already experienced the benefits of a distributed VCS, such as git or Mercurial, will find any means possible to continue using it even if the back-end isn't either of those.

One of the earliest bridges to be included with git was for Subversion and it's obvious to see why as Subversion was itself a logical step on from CVS for many organisations. If you've ever messed up your working copy trying to integrate the latest changes from the repo, 'git svn' is worth the price of admission alone to avoid that expensive mistake. (Arguably smaller commits also minimises the loss and is better from an integration perspective.)

In a similar vein 'git tfs' is a bridge for talking to Microsoft's heavily ridiculed Team Foundation Server. If you're using TFS without gated check-ins it pretty much works out of the box and understands the normal branching convention (like the Subversion bridge), although personally I have always avoided branching with TFS. If you do have gated check-ins then, when you push, it will invoke the standard TFS commit dialog to invoke its custom workflow. It also supports automatically attaching commits to work items using the normal '#<id>' convention in the commit message.

Code analysis & refactoring

One assumes that the starting point for this reluctance to allow programmers the freedom to use their own choice of tools stems from the need to restrict the download and installation of any third party software. Whilst I understand some of the reasons why – security and licensing [3] – they do not make it easy for those of us who wish to use our own purchases of professional, licensed software tools on site (license permitting, of course). It would be ludicrous to expect a carpenter or plumber to arrive at your house and fix your problem only with the tools laying around the garage or kitchen, but for some contract positions that scenario is not quite as absurd as you might think.

One category of tools that falls squarely into this category are those plug-ins and extensions to common major development products, such as IDEs, which perform on-the-fly code analysis and refactoring. In particular Whole Tomato's Visual Assist and JetBrains' ReSharper are two tools which have fast become essential tools for the modern developer that likes to write clean, maintainable code.

For me this particular battle started back in my C++ days with the venerable PC-Lint and Visual Lint tools which, along with Visual Assist provided a useful arsenal of tools for tracking down and fixing those kinds of bugs which C++ is sadly all too famous for. On one contract I was rejected a company licence for PC-Lint (a few hundred dollars back then) and refused the option to use my own license too, but in a delicious twist of irony a few months later I discovered that a colleague (a fellow contractor) had wasted a day and a half tracking down an initialisation bug that would have been picked up by PC-Lint. At contractor rates that bug alone cost the company more than twice the license cost of the tool it refused me access to and it wasn't the only painfully buried initialisation bug that surfaced later either.

Log viewer

Another tool I shelled out for because there were no decent open source alternatives on Windows at the time was a log file viewer called BareTail. With sterling incredibly favourable against the dollar and the pro version offering all sorts of nice filtering and highlighting options it became my tool of choice. Naturally log files are only read by this tool and not

written, and therefore it makes very little sense to control the exact choice and, given than support is usually of a time sensitive nature, I would have thought familiarity with tooling was commendable.

There are a number of excellent freeware tools available now on Windows that have since surpassed BareTail and with logs often being pumped directly into cloud-based logging services the goalposts have moved on significantly here from monitoring a handful of servers. Even so, due to cost reasons these may not be available outside of the production environment and therefore you might still be trawling log files in development and test environments with a traditional set of tools.

File & folder comparison

I've never been a big fan of the classic patch file format for displaying diffs in text files, and even less so for doing that in a command prompt window. The 'diff' tool is another one of those bread-and-butter tools which is in continual use and yet it has no impact on the delivered artefacts – source files, mostly. Many years ago the diff tool bundled with the version control system you were using was fairly limited – they might only show the two files side-by-side and block colour the edits. Even simple features like ignoring changes due to whitespace were missing.

Once again this is an area where there are number of excellent freeware and open source alternatives that provide a pretty good experience. And there are even products like Semantic Merge which aim to go beyond simpler textural diffs and apply more far intelligence to changes it detects. But even simple features like Beyond Compare's CSV diff could make a difference to your productivity without resorting to an 'untrusted' third party tool.

The only reason I can imagine any company would stop you using products like those from genuine software houses is to avoid any jealousy your co-workers might have.

Web browser

I suspect the only modern battle that can outshine the war around text editors is the one for your favourite web browser. For those actually developing web based services it's the bane of their life – having to support so many different flavours, even with third party libraries attempting to plug the gaps. Despite being dominant for so many years the tyranny of IE 6 is finally behind us and even the enterprise has graciously accepted that oranges are not the only fruit.

Once again I absolutely understand that if you are developing an in-house web application the business is entirely justified in only putting its money behind supporting the preferred 'house' option. However many of us write traditional, non-web based applications and services or even RESTful services and therefore our choice of browser is likely of very little consequence to delivering our goals. In fact, once again, I suspect familiarly is of more benefit, not for the core browser itself but the bells and whistles.

For those outside the world of IT I suspect their view of the web browser is still rooted in the simple display of text and images / videos. Consequently they would have little cause to investigate the proliferation of extensions and plug-ins the modern browser affords. If you store documentation in your source code repo using some kind of mark-up language then you'll probably want to view them in their rendered state when reading and browser extensions are one possible choice here. When working on web APIs I found myself trying out a whole bunch of different REST clients, such as Postman and Advanced REST Client, to see which I preferred or had features I found useful. That didn't cause me to stop using CURL, but they make chaining together REST requests into a 'journey' far less painful with their support for variables and response parsing.

Text editors & IDEs

You would think that the problem of editing text was a solved one by now and yet the last few years have still seen remarkable growth in the market with VS Code and JetBrains' Rider proving that the hearts and minds of developers are still yet to be won.

It's hard to talk about text editors without crossing over into IDE territory as the face of the IDE is pretty much the text editor. In my couple of decades as a programmer I've seen the humble text editor grow into the behemoth that is the modern, bloated IDE as more and more of the build and debug tooling has been grafted on. At the same time the need to automate builds and deployments has also meant that this tooling has had to remain at arm's length to some degree and therefore it's still possible to work with just a shell and simple text editor. The folly of proprietary binary project files came and went with JSON now the in-vogue choice for some languages, despite its shortcomings.

The enterprise IDE must seem like a bean counter's dream – 'a text editor, build system, debugger and deployment tool all in one package!' If you've worked on a codebase of any appreciable size that only uses the tools that come with the enterprise package you'll eventually discover their limitations, especially if you want to adopt a heavily automated approach to testing. Ever tried merging or automating testing of an old-fashioned DTS package? This isn't the product's fault, but it does put the onus on us to recognise when we've outgrown it either because we want more from our delivery process or the technology is moving too fast for it to keep up.

The rise of the polyglot and concepts like 'infrastructure as code' means that we spend far more time editing source files that are not part of the core product we're delivering. Although you might be writing your REST API in C# you might be using Terraform for your infrastructure, Go for your deployment tools, and a range of batch files, PowerShell scripts and Bash scripts for gluing it all together. Your documentation could be in Markdown, your database support queries in N1QL and have a variety of configuration files using XML, JSON and the classic .ini format.

The natural fallout of all this is that although I might be forced into using the IDE of the company's choice for the core product, there is still plenty of room to use both open source offerings like VS Code and licensed products such as Sublime Text because they offer certain essential features which are missing from the jack-of-all-trades offering, e.g. multi-cursor support or a plug-in that integrates the Go toolchain directly into it. Editing documentation in markdown format for instance is so much easier when you have the classic source + preview two-pane view like you have in VS Code; and these days I expect everywhere that I have to type any natural language to support spelling and grammar checking too.

If you think we've now reached 'peak IDE' know that JetBrains has launched a commercial Go-centric IDE for those that need more than what they're getting from the current crop of plug-ins. VS Code happens to suffice for my Golang purposes at the moment but it's good to know this is one battle that's still raging on.

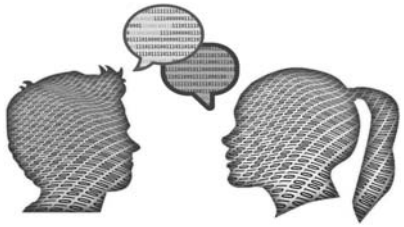
Other stuff

That list has covered all the big ones that immediately spring to mind and yet I can still think of many other areas where I have a different personal taste to my colleagues, such as still using the old fashioned Windows command prompt in preference to PowerShell. Consequently the multi-tabbed ConEmu fills a void you don't have with its more modern counterpart which, incidentally, shipped with its own IDE. And we haven't even touched on how the new Windows Subsystem for Linux is going to shake things up. I may finally give up on Gnu on Windows (my preferred bundle of the core Unix command line tools such as grep, sed, awk, etc.) as I'll have the real deal to use instead.

I don't currently do much in the way of web based stuff myself but I know from osmosis that the world of transpilers and Node.js adds another heap of tooling choices into the melting point, some of which form part of the end product, whilst others are alternative, portable implementations of existing native tools. Despite what I said about personal build scripts earlier I did once keep my own PSake (PowerShell) based build script alongside the real Gulp (Node.js) one because it saved me a couple of

Code Critique Competition 110

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



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

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

Last issue's code

Thanks to Jason Spencer for the random number generator suggestion leading to this issue's critique!

> I'm trying to write a very simple dice game where the computer simulates two players each throwing dice. The higher score wins and after a (selectable) number of turns the player who's won most times wins the game. (I'm going to make the game cleverer once it's working.) But the games always seem to be drawn and I can't see why. Here is what the program produces:

```
dice_game
Let's play dice
How many turns? 10
Drawn!
How many turns? 8
Drawn!
How many turns? ^D
```

Listing 1

```
// Class to 'zip' together a pair of iterators
template <typename T>
class zipit : public std::pair<T, T>
{
  zipit &operator+=(std::pair<int,int> const
&rhs)
  {
    this->first += rhs.first;
    this->second += rhs.second;
    return *this;
  }
```

What's going wrong, and how might you help the programmer find the problem? As usual, there may be other suggestions you might make of some other possible things to (re-)consider about their program.

- Listing 1 contains `zipit.h`
- Listing 2 contains `dice game.cpp`

Listing 1 (cont'd)

```
public:
  using std::pair<T, T>::pair;
  zipit &operator+=(int n)
  {
    return *this += std::make_pair(n, n);
  }
  zipit &operator-=(int n)
  {
    return *this += std::make_pair(-n, -n);
  }
  zipit &operator++()
  {
    return *this += 1;
  }
  zipit &operator--()
  {
    return *this += -1;
  }
  auto operator*()
  {
    return std::make_pair(
      *this->first, *this->second);
  }
```


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

Getting Personal (continued)

minutes every build not waiting for 'npm install' to check its cache. (The team switched over permanently soon after.)

(Executive) Summary

It's not just in IT that we have our own personal choices in tooling; the same is true in other walks of life. For instance nobody tells you what kind of pen you have to use or whether you should be using lined or clear paper to write your notes on. I happen to favour the gel pens but I know others that still prefer to use traditional ink. I think it's fair for any organisation to expect me to speak and write documentation in the English language if that's the one most commonly by them.

What I find objectionable is the lack of distinction between when a choice of tooling affects the deliverable itself as opposed to only affecting the means of delivery, especially when the personal choice would be more beneficial to the team or organisation. Consistency within tooling is not an all or nothing affair – we need to question our choices when delivery might be impacted but also embrace different approaches when possible so that we can learn from our colleagues. ■

References

[1] 'In The Toolbox – Finding Text', *C Vu* 27.6, http://www.chrisoldwood.com/articles/in-the-toolbox-finding-text.html

[2] 'In The Toolbox – Home-Grown Tools', *C Vu* 28.4, http://www.chrisoldwood.com/articles/in-the-toolbox-home-grown-tools.html

[3] 'Developer Freedom', *C Vu* 26.1, http://www.chrisoldwood.com/articles/developer-freedom.html

Listing 1 (cont'd)

```
  auto operator*() const
  {
    return std::make_pair(
      *this->first, *this->second);
  }
  // Hmm, operator-> ??
};
template <typename T>
auto begin(T one, T two)
  -> zipit<typename T::iterator>
{
  return {one.begin(), two.begin()};
}
template <typename T>
auto end(T one, T two)
  -> zipit<typename T::iterator>
{
  return {one.end(), two.end()};
}
```

Critiques

James Holland <James.Holland@babcockinternational.com>

In `main()`, the user has created an object named `generator` of type `randomize` and then has sensibly passed it by reference to `play()`. Within `play()` the `randomize` object is then passed to the two `generate()` functions by value. Passing by value makes a copy of the object and this where problems begin. The `randomize` objects contain state that is used in a predetermined way to generate the pseudo random numbers. Two `randomize` objects that start in the same state will produce the same sequence of numbers. The two calls to `generate()` each make use of identical copies of `randomize` and so both will produce exactly the same sequence of random numbers. This is why all runs of the dice game end in a draw.

What we would like to happen is for both calls of `generate()` to use the same `randomize` object. In this way when one call to `generate()` has obtained a set of random numbers, the other call to `generate()` will continue to use the same `randomize` object and so will obtain a different sequence of numbers.

One way to achieve this is to pass the `randomize` object to `generate()` by reference thus ensuring there is only ever one `randomize` object. This can be done by employing the standard library reference wrapper `ref()`. This wrapper conveys references to function templates that take the parameters by value. Fortunately, `generate()` is such a function. The calls to `generate()` simply become as shown below.

```
std::generate(player1.begin(), player1().end(),
  std::ref(generator);
std::generate(player2.begin(), player2().end(),
  std::ref(generator);
```

While we are talking about random number generators, it is recommended not to use one of the standard library generators, such as `MT19937`, on its own but instead to use it with a distribution, also available from the standard library. There are problems associated with dividing the output from a generator by a constant and taking the reminder. Using a distribution will ensure a high quality series of random numbers.

Running `dice_game` will now probably produce the desired result. I say 'probably' because the student's software contains a serious flaw. The free functions `begin()` and `end()` pass their values by value. This means their parameters are copied. The functions `begin()` and `end()` construct a `zipit` object from iterators that point to copies of `player1` and `player2` arrays. When the functions return, the copies of `player1` and `player2` will be destroyed. The `zipit` object will be returned to its caller but the iterators it contains will be invalid as they point to objects that no longer exist. It may be the case that the memory locations where the copies of `player1` and `player2` existed maintain their values for as long as the program requires them. It is in this way that the program appears to work

Listing 2

```
#include <algorithm>
#include <iostream>
#include <random>
#include "zipit.h"
class randomize
{
  std::mt19937 mt;
public:
  int operator()() { return mt() % 6 + 1; }
};
void play(int turns, randomize &generator)
{
  std::vector<int> player1(turns);
  std::vector<int> player2(turns);
  std::generate(player1.begin(),
    player1.end(), generator);
  std::generate(player2.begin(),
    player2.end(), generator);
  int total{};
  for (auto it = begin(player1, player2);
       it != end(player1, player2); ++it)
  {
    if ((*it).first != (*it).second)
    {
      auto diff = *it.first - *it.second;
      total += copysign(1.0, diff);
    }
  }
  if (total > 0)
  {
    std::cout << "Player 1 wins\n";
  }
  else if (total < 0)
  {
    std::cout << "Player2 wins\n";
  }
  else
  {
    std::cout << "Drawn!\n";
  }
}
int main()
{
  randomize generator;
  int turns;
  std::cout << "Let's play dice\n";
  while (std::cout << "How many turns? ",
    std::cin >> turns)
  {
    play(turns, generator);
  }
}
```

correctly. The behaviour of software should not be left to chance and so this bug must be corrected. Fortunately, this is a relatively simple matter.

Instead of passing `player1` and `player2` to `begin()` and `end()` by value, they should be passed by reference. In this way no copies are made and the iterators returned in the `zipit` object will point to the original (and still existing) versions of `player1` and `player2`. The program is now running with a firm footing!

Although the software now works correctly, it seems quite complicated for what it does. Perhaps there is another way of structuring the program that is shorter and easier to understand. The student's software involves the creation of two vectors that contain the results of rolling the dice, one for each of the players. There is also quite an elaborate arrangement of operator functions that are required to manipulate iterators that point to the two vectors. Instead of having a pair of vectors, each element of which contains a single value, I propose having a single vector that contains a pair

of dice values, one for each player. Iterating through a single vector is much simpler that trying to iterate through two vectors at the same time. There is no danger of getting out of step. I suggest my version of the software is also considerably simpler as none of the code within `zipit.h` is required. I have kept the interface to **`Play()`** the same as in the student's code and so **`main()`** can remain the same. I have modified the **`Randomize`** class to take advantage of the standard library's uniform integer distribution as shown below.

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <random>
#include <vector>
class Randomize
{
  public:
  Randomize():d(1, 6){}
  std::pair<int, int> operator()()
  {
    return {d(e), d(e)};
  }
  private:
  std::mt19937 e;
  std::uniform_int_distribution<> d;
};
void play(int turns, Randomize & generator)
{
  std::vector<std::pair<int, int>>
    players(turns);
  std::generate(players.begin(),
    players.end(), std::ref(generator));
  int total{};
  for (const auto & turn : players)
  {
    const auto p1_score = turn.first;
    const auto p2_score = turn.second;
    if (p1_score != p2_score)
    {
      const auto diff = p1_score - p2_score;
      total += diff > 0 ? 1 : -1;
    }
  }
  if (total > 0)
  {
    std::cout << "Player 1 wins\n";
  }
  else if (total < 0)
  {
    std::cout << "Player 2 wins\n";
  }
  else
  {
    std::cout << "Drawn!\n";
  }
}
int main()
{
  Randomize generator;
  int turns;
  std::cout << "Let's play dice\n";
  while (std::cout << "How many turns? ",
    std::cin >> turns)
  {
    play(turns, generator);
  }
}
```

`Randomize` now simulates rolling the dice for both players within one call of **`operator()()`**. I felt that **`copysign()`** is a slightly strange function to use in this context as it converts its integer parameters to **`double`**s before copying the sign of the second parameter to the first. The resulting double is then converted back to an integer when added to **`total`**. The same effect can be simply achieved by a conditional operator as I have used. No conversion to and from **`double`** is required.

Paul Floyd <paulf@free.fr>

There are two main problems with this code. The first is the cause of the draws, the second results in crashes.

For the cause of the draws, we have a misuse of the interface of **`std::generate`**:

```
template< class ForwardIt, class Generator >
void generate( ForwardIt first,
  ForwardIt last, Generator g );
```

(from http://en.cppreference.com/w/cpp/algorithm/generate)

So in the original code, where the call to this is

```
std::generate(player1.begin(),
  player1.end(), generator);
std::generate(player2.begin(),
  player2.end(), generator);
```

then the variable **`generator`** is passed to **`std::generate`** by value (the template deduces a type of **`randomize`**).

This is borne out by **`nm`** (output slightly reformatted for printing):

```
nm -C cc109 | grep generate
00000000004013d3 W void std::generate
<__gnu_cxx::__normal_iterator<int*,
 std::vector<int, std::allocator<int>>>,
 randomize>(
  __gnu_cxx::__normal_iterator<int*,
  std::vector<int, std::allocator<int>>>,
  __gnu_cxx::__normal_iterator<int*,
  std::vector<int, std::allocator<int>>>,
  randomize)
```

This means that the original copy does not get updated by the first call, so the second call to **`std::generate`** gets a copy of the same object. The fix for this would be either to make the call to the template function explicit (not nice) or to just use **`std::ref`** so that the template will be initialised with a reference type rather than passing by value. Now with a reference the variable **`generator`** does have its state modified by the first call.

Here's **`nm`** again when using **`std::ref`**:

```
nm -C cc109 | grep generate
00000000004013ca W void std::generate
<__gnu_cxx::__normal_iterator<int*,
 std::vector<int, std::allocator<int>>>,
 std::reference_wrapper<randomize>>(
  __gnu_cxx::__normal_iterator<int*,
  std::vector<int, std::allocator<int>>>,
  __gnu_cxx::__normal_iterator<int*,
  std::vector<int, std::allocator<int>>>,
  std::reference_wrapper<randomize>)
```

At first I thought that was the end of the story. But then I ran the code through AddressSanitizer (compiled with clang++ and the **`-fsanitize=address`** option), and I saw:

```
Let's play dice
How many turns? 10
=================================================
==2657==ERROR: AddressSanitizer: heap-use-after-
free on address 0x6040000003d0 at pc 0x0001085aeca3
bp 0x7ffee7654010 sp 0x7ffee7654008
READ of size 4 at 0x6040000003d0 thread T0
    #0 0x1085aeca2 in zipit<std::__1::
__wrap_iter<int*> >::operator*() .zipit.h:32
    #1 0x1085ad4f0 in play(int, randomize&)
dice_game.cpp:30
    #2 0x1085af3e6 in main dice_game.cpp:59
```

```
    #3 0x7fff5c710114 in start (libdyld.dylib:
x86_64+0x1114)
0x6040000003d0 is located 0 bytes inside of 40-byte
region [0x6040000003d0,0x6040000003f8)
freed by thread T0 here:
    #0 0x1086226ab in wrap__ZdlPv
(libclang_rt.asan_osx_dynamic.dylib:
x86_64h+0x646ab)
    #1 0x1085af8f4 in std::__1::__vector_base<int,
std::__1::allocator<int> >::~__vector_base()
vector:445
    #2 0x1085af4c4 in std::__1::vector<int, std::
__1::allocator<int> >::~vector() iterator:1386
    #3 0x1085ae384 in std::__1::vector<int, std::
__1::allocator<int> >::~vector() iterator:1386
    #4 0x1085ad0b5 in play(int, randomize&)
dice_game.cpp:27
    #5 0x1085af3e6 in main dice_game.cpp:59
    #6 0x7fff5c710114 in start (libdyld.dylib:
x86_64+0x1114)

previously allocated by thread T0 here:
    #0 0x1086220ab in wrap__Znwm
(libclang_rt.asan_osx_dynamic.dylib:
x86_64h+0x640ab)
    #1 0x1085b0279 in std::__1::vector<int, std::
__1::allocator<int> >::allocate(unsigned long)
vector:925
    #2 0x1085b2631 in std::__1::vector<int, std::
__1::allocator<int> >::vector(std::__1::vector<int,
std::__1::allocator<int> > const&) vector:1200
    #3 0x1085ae35c in std::__1::vector<int, std::
__1::allocator<int> >::vector(std::__1::vector<int,
std::__1::allocator<int> > const&) vector:1193
    #4 0x1085acfcd in play(int, randomize&)
dice_game.cpp:27
    #5 0x1085af3e6 in main dice_game.cpp:59
    #6 0x7fff5c710114 in start (libdyld.dylib:
x86_64+0x1114)
```

So there is an error in the dereference operator for a vector that is created and destroyed on the same line. That smells like a local variable being created and destroyed.

The problem is with the zipit **begin()** and **end()** functions.

```
template <typename T>
auto begin(T one, T two)
  -> zipit<typename T::iterator>
...
```

This is instantiated here

```
for (auto it = begin(player1, player2);
```

Where **player1** and **player2** are the vectors of **int**. So **T** is a vector of **int**, and **one** and **two** are passed by copy. The local copy gets destroyed at the end of the call, leaving the returned pair of iterators dangling.

A third opinion from **nm**:

```
nm -C cc109 | grep begin
00000000004015a0 W zipit<std::vector<int,
  std::allocator<int>>::iterator>
begin<std::vector<int, std::allocator<int>>>(
  std::vector<int, std::allocator<int>>,
  std::vector<int, std::allocator<int>>)
0000000000401b9c W std::vector<int,
  std::allocator<int>>::begin() const
0000000000401356 W std::vector<int,
  std::allocator<int>>::begin()
```

The fix for this is to make the **begin**/**end** arguments references:

```
template <typename T>
auto begin(T& one, T& two)
  -> zipit<typename T::iterator>
{
  return {one.begin(), two.begin()};
}
template <typename T>
auto end(T& one, T& two)
  -> zipit<typename T::iterator>
{
  return {one.end(), two.end()};
}
```

OK, now for the minor nits. I get a warning about the unused **cout** in the **while** condition. This can be silenced by a cast

```
while((void)(std::cout << "How many turns? "),
  std::cin >> turns)
  {
    play(turns, generator);
  }
```

A bit ugly.

There are no include guards, and the cpp file does not include `zipit.h` before the other files. This means that users that include `zipit.h` will have the misfortune of playing 'guess the header dependency' (or fixing the header).

Lastly, the read of the **int** for the number of turns does not prevent against reading negative values. This then gets converted to **size_t** in the vector constructor. If the allocation succeeds then it's going to be a very long game.

Russel Winder <russel@winder.org.uk>

First and foremost why try to invent a custom zip capability, and risk getting it wrong? Where are the tests? I am sure there is a good criticism to be made of the code in `zipit.h`, but I shall just circumvent all that and say: use **boost::combine** to undertake the zip and delete `zipit.h`.

The original code also has output scattered around. OK so only in two functions, but still, the design principle applies: keep all input and output in one place and have functions return values to represent the result. Doing this enhances testability, so must be a good thing. In this case **play** should not do output but should return a 'who won' result. Seems like a good place for an enumerated type. The output of the result then happens in **main**.

So ignoring the time it took to make those changes and get the code to actually work (because the developer failed to read the documentation correctly), we find it still delivers a draw in all cases. A bit of a WTF moment, but surely one it was intended one would have. So what is the problem? If you print out the value of the two **std::vector**s of rolls, you find that you always get the same sequence of values. Always. Fairly obvious then the same start state is the case for all calls to the Mersenne Twister generator. It turns out that **std::generate** seems to clone the generator passed so the player1 generation and the player2 generation are identical since the **randomize** class creates a new generator for each instance initialised exactly the same. A cure for this is to make the **mt** instance shared by making it static. Since this is a single threaded application this does not seem to be a problem, even if global shared state is deemed anathema by some.

With this change in place, and resulting in the program shown in Listing 1, the program seems to work as originally intended.

Of course, one has yet again to ask the question whether C++ is the best programming language to use for this application. Let us assume no. So what to choose? Well to investigate this let's look at Python, D and Rust.

Python, see Listing 2, has a built in **zip** function. Also the **sum** function acting over an iterable makes for a much nicer expression of the core part of the algorithm - which is rolling the dice and checking for who wins. Even though Python is a dynamic language at run time, type annotations have been used so that using the mypy command, we can be sure that all the function calls are correctly statically typed. The random numbers are provided via the **random** package, using a Mersenne Twister algorithm, which is 'global' so there are no problems about generating the same sequence of random numbers. It would seem that Python makes it harder to make errors in this sort of application.

D, see Listing 3, also has a builtin **zip** function, so no difficulties on that front. Comparing the C++, Python, and D, one gets the sneaking suspicion that D is the best language for writing this code: less faff. Well except maybe for the random number generator. Rather than using a random number generator creating integers from the set {1, 2, 3, 4, 5, 6} with each integer result assumed to have equal probability, The **std.random dice** function is used to create a random number generator of the values with an explicit probability. Equal probabilities are used here, but it allows for experimentation with 'loaded dice'. Another 'feature' of the code worth noting is the function call chaining approach as opposed to function application as used in C++ and Python. This is blurring the 'function application' and 'method call on an object' that is quite rigid in C++ and Python. It is an aspect of trying to be as declarative as possible in what is an imperative language.

Of course Rust, see Listing 4, is reputedly the language of the moment, set to replace C, and perhaps C++, as the language of first choice for new projects. Immediately obvious is the very different way enumerations work. C++, Python, and D have very similar ways of defining and working with enumerations, Rust takes a very different approach. In particular an enumeration value can carry data of undefined value, just defined type. Rust, like D, emphasises being declarative, hence the creation of pipelines of data manipulation as seen in the D code. Of course Rust tries to go further and mandates the use of monads to create what is arguably a functional programming language (disguising the imperative language). This is seen most clearly in **main**. The code seems to a bit bigger, but there an appeal that error handling is enforced but in a quite pleasing way; especially highlighting the nice use of enumerations.. After the interesting dice emulation of D, in this Rust code returns to the generate an integer in a range assuming equal probabilities. The random number generator is not global, but it is shared, so no possibility of generating the same sequence for each player.

So is C++ the best language for this application? Probably not, but for something like this using the language you are most 'at home' with is likely the best choice. Unless you are challenging yourself as a programmer by using a programming language you are not entirely familiar with. Personally I quite favour the D code.

Why is Go not in this list of languages likely to be better than C++? It should be, but I ran out of time to complete that version of the code. Bad project management I admit.

Nota bene: the code presented here does not represent a reasonable formatting of the code, and indeed violates many *de facto* and *de jure* code formatting guidelines. However, the journal formatting structure requires badly formatted code to be presented. :-(

PS There is clearly a difference in the languages behaviour here since the different languages produce different frequencies of draws. Hummm… [Editor: since the programs all model the same die throwing game then statistically significant differences imply at least one is buggy.]

Listing 1

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <numeric>
#include <random>
#include <boost/range/combine.hpp>
#include <boost/tuple/tuple.hpp>
class randomize {
  private:
    static std::mt19937 mt;
  public:
    int operator()() { return mt() % 6 + 1; }
};
std::mt19937 randomize::mt {
  (unsigned long int)std::chrono::system_clock::now()
  .time_since_epoch()
  .count()
};
enum class result {
  draw = 0,
  player1_win = 1,
  player2_win = 2,
};
typedef boost::tuple<int, int> int_pair;
result play(int const turns, randomize & generator) {
  std::vector<int> player1(turns);
  std::vector<int> player2(turns);
  std::generate(player1.begin(), player1.end(), generator);
  std::generate(player2.begin(), player2.end(), generator);
  auto pairs = boost::combine(player1, player2);
  int_pair const t_t = std::accumulate(
    pairs.begin(),
    pairs.end(),
    int_pair{0, 0},
    [](int_pair t, int_pair i) {
      int a, b;
      boost::tie(a, b) = i;
      return int_pair{
      boost::get<0>(t)
        + (a > b ? 1 : (a < b ? -1 : 0)),
                      0};
            });
    int const total = boost::get<0>(t_t);
    return total == 0
      ? result::draw
      : (total > 0 ? result::player1_win
      : result::player2_win);
}
int main() {
    randomize generator;
    int turns;
    std::cout << "Let's play dice" << std::endl;
    while (
            std::cout << "How many turns? ",
            std::cin >> turns) {
        auto const result = play(turns, generator);
        if (result == result::draw) {
            std::cout << "Drawn!" << std::endl;
        } else {
            std::cout
              << "Player "
              << (int)result
              << " wins"
              << std::endl;
        }
    }
}
```

Listing 2

```
#!/usr/bin/env python3
from enum import Enum
import random
from typing import Tuple
class Result(Enum):
    draw = 0
    player1_win = 1
    player2_win = 2
def player1_win(p: Tuple[int, int]) -> int:
    if p[0] > p[1]:
        return 1
    elif p[0] < p[1]:
        return -1
    return 0
def play(turns: int, gen) -> Result:
    player1_wins = sum(
        player1_win(p) for p in zip(
            (gen() for _ in range(turns)),
            (gen() for _ in range(turns))
        ))
```

```
    if player1_wins > 0:
        return Result.player1_win
    elif player1_wins < 0:
        return Result.player2_win
    return Result.draw
def main() -> None:
    print("Let's play dice")
    random.seed()
    while True:
        try:
            turns = int(input('How many turns? '))
        except ValueError:
            return
        result = play(
            turns,
            lambda: random.randint(1, 6)
        )
        if result == Result.draw:
            print('Drawn!')
        else:
            print('Player {} wins.'.format(result.value))
if __name__ == '__main__':
    try:
        main()
    except (EOFError, KeyboardInterrupt):
        print()
```

Listing 3

```
#!/usr/bin/env rdmd
import std.algorithm: map, sum;
import std.array: array;
import std.conv: to;
import std.random: dice;
import std.range: iota, zip;
import std.stdio: readln, write, writefln, writeln;
import std.string: strip;
enum Result {
    draw = 0,
    player1_win = 1,
    player2_win = 2,
}
Result play(uint turns, uint delegate(uint) generator) {
    auto player1_wins =
      zip(
          iota(turns).map!generator(),
          iota(turns).map!generator())
      .map!"a[0] > a[1] ? 1 : (a[0] < a[1] ? -1 : 0)"
      .sum;
    return player1_wins == 0
      ? Result.draw
      :  player1_wins > 0
        ? Result.player1_win
        : Result.player2_win;
}
void main() {
    writeln("Let's play dice");
    try {
        while (true) {
            write("How many turns? ");
            uint turns = to!uint(readln().strip());
            auto result = play(turns, delegate uint(uint) {
                    return to!uint(
                            dice(0.0,
                                 16.67,
                                 16.67,
                                 16.67,
                                 16.67,
                                 16.67,
                                 16.67));
                });
            if (result == Result.draw) {
                writeln("Drawn!");
            } else {
                writefln("Player %d wins.", result);
            }
        }
    } catch (Exception) {
        writeln();
    }
}
```

Listing 4

```
extern crate itertools;
extern crate rand;
use std::io::{self, Write};
use itertools::Itertools;
use rand::distributions::{Range, Sample};
enum Result {
    PlayerWon(u8),
    Draw
}
fn play(
    turns: u32,
    dice_roll: &mut FnMut()->u8) -> Result
{
    let player1_wins: i32 =
        itertools::repeat_call(dice_roll)
        .tuples::<(_, _)>()
        .map(|p|
             if p.0 > p.1 { 1 }
             else if p.0 < p.1 { -1 }
             else { 0 }
        )
        .take(turns as usize)
        .sum();
    if player1_wins != 0 {
        Result::PlayerWon(
            if player1_wins > 0 { 1 }
            else { 2 }
        )
    }
    else { Result::Draw }
}
fn main() {
    println!("Let's play dice");
    let mut dice = Range::new(1u8, 7u8);
    let mut rng = rand::thread_rng();
    let mut buffer = String::new();
    loop {
        print!("How many turns? ");
        io::stdout()
            .flush()
            .expect("Could not flush stdout.");
        buffer.clear();
        match io::stdin().read_line(&mut buffer) {
            Ok(_) => match buffer.trim().parse::<u32>() {
                Ok(turns) => match play(
                    turns,
                    &mut ||{ dice.sample(&mut rng) }
                ) {
                    Result::PlayerWon(p) =>
                        println!("Player {} wins.", p),
                    Result::Draw =>
                        println!("Drawn!"),
                },
                Err(_) => break
            },
            Err(_) => break
        }
    }
}
```

Jason Spencer <contact+pih@jasonspencer.org>

The reason for all games ending in a draw is that the random number generator actually creates a deterministic sequence of numbers, and since the calls to `std::generate` copy the generator object, that is to say its internal state, the die throws generated for both players are identical. There is, however, another bug with invalidated iterators that potentially corrupts the result calculation, and may cause a crash or non-draw result.

The issues with the code are as follows:

zipit.h

- The file requires the following includes: `<utility>` for `std::pair` and `std::make_pair`. Also `<iterator>` for `std::advance`, `std::begin` and `std::end`, which will be added next. An include guard for the whole file is also a good idea.
- class `zipit` is derived from `std::pair<T,T>` with public access. Consider making it protected, otherwise a user can access `.first` and `.second` and change their values, putting them out of sync. This might be a wanted behaviour, however. I think it breaks encapsulation and the inheritance should be protected.
- Consider renaming the template parameter from `T` to `Iter`, to express the meaning. `T` is used so often that it can get confusing.
- Two `operator*` functions (one const, one non-const) aren't required, only one – the function doesn't change the `zipit` instance – although dereferencing may then lead to a change in the pointed to variable, but that doesn't mean the `zipit` instance changes (compare `const char * ptr` and `char * const ptr`). So only the const version is needed, and the non-const should be removed.
- `operator*` at the moment returns an rvalue, so `zipit` only models an input iterator. We may want to return an lvalue to model an output iterator. See the discussion and implementation later.
- `zipit &operator+=(std::pair<int,int> const &rhs)` should use `std::advance` instead of `+=` to advance `this->first` and `this->second`. If the underlying iterator is not a random access iterator it may not have an `operator+=` overload. `std::advance` will either use `operator++` or `operator+=`, preferring the latter as it has constant time complexity, while the former is linear.
- `operator->` is indeed required to at least satisfy the requirements of `InputIterator` (see later for discussion of iterator types).
- The `begin(T,T)` and `end(T,T)` free functions should both have their arguments passed by non-const reference. Both of the functions extract iterators from the arguments passed and store them in a `zipit` instance and return that. However, since the arguments passed to these functions are currently temporary copies the iterators are invalid as soon as the function returns. Since the score is later calculated from dice throws stored in memory that has been deallocated the program either crashes or has invalid results.
- In `begin(T,T)` and `end(T,T)` the use of `T::iterator` assumes that the typedef exists in `T`. By using `std::begin(one)` instead of `one.begin()` and changing the return type of these functions they can be adapted to also support C-style arrays, thusly (just s/begin/end/g for the `end` function):

```
template <typename T>
auto begin(T & one, T & two) ->
zipit<decltype(std::begin(std::declval<T&>()))>
{
  return { std::begin(one), std::begin(two) };
}
```

dice_game.cpp

- In class `randomize` the `mt19937` instance should not be left unseeded. Either pass the seed as an argument to the constructor or call `mt19937::seed` in the `randomize` constructor, otherwise every time the program is executed the numbers returned by generator will be the same (and therefore the game results will be the same for the same number of games played). I propose two constructors for class `randomize` – a default constructor which will use a source of entropy (some OS or hardware source, but worst case the current time) to seed, and another constructor that takes the seed as an argument. See below for a sample implementation and later for a discussion on seeding.
- In `randomize::operator()` the use of `%6+1` to scale the output of `mt()` may produce a non-uniform distribution. While it is good enough for this use, there's a very tiny bias. If `mt()` were to generate numbers uniformly distributed over the interval [0,7] then taking modulo 6 of these numbers 0 would map to 0, 1=>1, 2=>2, 3=>3, 4=>4, 5=>5, 6=>0, 7=>1, so the probability of getting a 0 or 1 output is twice the probability of the other numbers. In our case, however, `mt()` will produce numbers in the range $0–2^{32}$-1, so the bias is unnoticeable with a $1.4 * {}^{10\text{-}7}$% higher probability of output die rolls 1,2,3. Even so, prefer using `std::uniform_int_distribution` to change the distribution of the `mt19937` instance output – it doesn't have the bias issue and is more descriptive in terms of intent.

  The randomize class then would look something like this:

```
class randomize {
   std::mt19937 rng;
   std::uniform_int_distribution<int> D6;
public:
   randomize(decltype(rng)::result_type seed)
   : rng(seed), D6(1,6) { }
   randomize ()
   : randomize ( std::random_device()() ) { }
   int operator()() { return D6(rng); }
};
```

  It might be an idea to log the seed value generated by `std::random_device`, in case the program has to be debugged later.
- In `play(...)` the last argument of both `std::generate` calls should be `std::ref(generator)`. Since `std::generate` takes the third argument by copy `std::ref` should be used to wrap a reference to generator in an `std::reference_wrapper` instance. As discussed later the random number generator actually produces a deterministic sequence, rather than true unpredictable random numbers. Without the wrapper the first call to `std::generate` makes a temporary copy of the generator object for use within the function, and all the internal state is copied with it, afterwards in `play(...)` the internal state of the generator hasn't changed, so in the second call to `std::generate` the sequence of numbers is repeated. The result is that both players get the exact same die throws, hence the repeated draws for the headline bug.
- In `play(...)` this is personal choice, but I prefer `int total = 0` over `int total{}` as the expected value is clear.
- In `play(...)` the use of an `if` and then a `copysign` call is perhaps a little confusing. The intent is to do a three way comparison. While there's a proposal for such a feature in C++ [1] it doesn't currently exist. There are a number of ways to do this but my preferred approach is a ternary conditional expression operator – it is the most expressive, and simple to execute:

```
total += (diff<0) ? -1 : ( (diff==0)? 0 : 1 );
```

  Here the intent and the result values are explicit (-1,0,1) and appear in order. If `diff` is more likely to be in one of the states, 0 for example, then the two ternary conditionals could be re-ordered for performance, so the most common case is dealt with first and the second comparison doesn't have to be executed, but that's not the case here.
- `copysign` is presumably from `math.h`, but the header isn't included, yet this compiles (g++ 7.3), so I'm not sure where it's from. Consider using `std::copysign` and include `cmath` if

copysign must be used. Otherwise I'd recommend the ternary operator approach.

- In **play(...)** consider replacing the whole **for** loop over the zip iterator with **std::accumulate**:

```
auto get_score_from_throws = [](auto p1,
  auto p2)
{
  auto diff = p1 - p2;
  return (diff<0) ? -1 : ( (diff==0)? 0 : 1 );
};
total = std::accumulate (
  begin(player1, player2),
  end(player1, player2), 0,
  [&get_score_from_throws](auto sum,
    const auto & it) {
    return sum +=
    get_score_from_throws(it.first,
      it.second);
  }
);
```

or use **std::inner_product**:

```
  total = std::inner_product (
    std::begin(player1), std::end(player1),
    std::begin(player2), 0, std::plus<>(),
    get_score_from_throws );
```

The accumulate version uses the zip iterator, and the function name is more self explanatory. The **inner_product**, while confusingly named, achieves the same thing, and *doesn't even need the zip iterator*.

Alternatively **std::reduce** or **std::transform_reduce** could be used, but neither libstdc++ (the STL used by g++) [2], nor libc++ (the STL used by CLang) [3] have yet to support these C++17 features, so I couldn't produce sample code.

- In **play(...)** consider making the turns argument unsigned instead of **int**. You cannot play a negative number of games.
- In **main(...)** turns should be bounds checked

So that should get the right result with the existing program design, but I'd make some design changes also.

- If a zip iterator is still required consider using **Boost.ZipIterator** instead – it supports the zipping of more than two iterators and is tested and known to work.
- **zipit::operator==** is a potential source of errors. Since there isn't an explicit comparison operator the inherited one is used. This is typically the correct behaviour, however if the vectors passed to **begin(T,T)** and **end(T,T)** are different lengths there will never be a time when the end zipit iterator is reached, since both **first** and **second** have to match. Since they're incremented together they'll never both match when there are different length vectors. Consider tweaking **end(T,T)** to set **.first** and **.second** to the same offset from the beginning of the vectors. The offset should be the length of the shorter vector.
- With its current functionality the **randomize** class isn't actually needed at all. Assuming no other features are required the following is a drop-in replacement:

```
auto generator = std::bind (
std::uniform_int_distribution<unsigned>(1,6),
  std::mt19937(rng_seed) );
```

For fun, you could also create a loaded die that has double the chance of rolling a six:

```
auto loadedD6 = std::bind(
  std::discrete_distribution<unsigned>{
    0,1,1,1,1,1,2},
  std::mt19937(rng_seed));
```

Of course, for one player to have the advantage you'd need each player to have different dice types.

With respect to seeding – it might be an idea to make the seed a command line argument, so that you can have reproducible results in testing, something along the lines of:

```
void usage(char * execname) { std::cerr <<
  "Usage: " << execname <<
  "[--seed <unsigned integer>]\n"; }
int main(int argc, char *argv[]) {
  using namespace std::string_literals;
  using rng_t = std::mt19937;
  rng_t::result_type rng_seed =
    std::random_device()();
  if ( argc == 3 ) {
    if("--seed"s != argv[1]) {
      usage(argv[0]); return -1;
    }
    // could throw invalid_exception or
    // out_of_range
    int seed = std::stol(argv[2]);
    if(seed < 0) throw std::out_of_range(
      "Seed value must be a positive"
      " integer");
    rng_seed = seed;
  }
  else if(argc!=1) {
    usage(argv[0]); return -1;
  }
  std::cout << "RNG seed = " <<
    rng_seed << '\n';
  auto D6 = std::bind(
    std::uniform_int_distribution<unsigned>
    (1,6), rng_t(rng_seed));
...
```

- Something to consider is whether we actually need to keep the die throws in memory. No game is dependant on the result of any other, so there's no need to store the throws or even the score per round. A loop could make die throws, calculating the score and keeping a running score, without any vectors.

  Of course it can be shown that for two N sided fair dice the chance of throwing the same value is 1/N, the chance the first die throws a smaller number is (N-1)/2N, same for the second die. So we could get the game result with the following generator:

```
const unsigned die_sides = 6;
auto game_point_generator = std::bind ( \
  std::discrete_distribution<unsigned>{ 1.0,
(static_cast<double>(die_sides)-1.0)/2.0,
(static_cast<double>(die_sides)-1.0)/2.0 }, \
  std::mt19937(std::random_device()()) \
  );
```

  A call to **game_point_generator()** will generate 0 if the game is a draw, a 1 if player1 wins, and 2 if player 2 wins.
- If the die throws do have to be kept in memory then consider a vector of **std::pair**s to store the results – this way there is no chance of one vector in the zip iterator being smaller than the other. We also don't need the zip iterator then.
- **zipit** could be generalised by using a **tuple** to allow more than two iterators to be zipped; and there's no reason for all iterator types to be the same.
- **zipit** should consider SFINAE/**enable_if** to enable the operator overloads based on the type of the zipped iterators.
- **std::iterator_traits<zipit>** should be implemented so that **zipit** can be better integrated into the STL (for example if it happens to model a random access iterator and states so via **iterator_traits** then **std::advance** will be optimised).

- In the spirit of C++20 **zip_it** can be implemented with constexpr modifiers.

Points for discussion

Random number generation

Random number generators fall into two categories – deterministic and non-deterministic. Since CPUs are deterministic state machines they cannot, without special hardware or external input, produce true entropy. 'Random' numbers are therefore typically produced by pseudo-random algorithms that are actually deterministic. If you know the internal state and the algorithm, you can predict the next number exactly. This is how the **rand()** library call works (that's typically a linear congruential generator [4]), as well as **mt19937** which is available since C++11. To change the sequence produced every time the program is run it is typical (although perhaps not always correct) to bootstrap the generator from the current time. Since C++11 **std::random_device** can be used to seed the deterministic generator. **std::random_device** takes entropy from an implementation dependant source (for example RDRAND on Intel/AMD, or `/dev/urandom` under linux – both use external events such as interrupts to accumulate entropy) to try and generate non-deterministic random numbers, but the non-determinism is not guaranteed by **std::random_device** (26.5.6.2 in [5]).

Creating custom iterators

We're not so much creating a custom iterator here, as we are an iterator adaptor. And we also cannot assume the underlying iterator type – it may be a forward input, forward output, bidirectional, or random iterator [6]. And then there are the const and reverse variants. But I think the intention of **zipit** is still to behave like an STL iterator. One thing is for sure though, writing iterators isn't easy. There's a fair amount of responsibility that goes into writing one.

If we try to model an input iterator then according to [6] **zipit** is still missing a post-increment operator and an **operator->**. The post-increment operator isn't that difficult and can be expressed in terms of the pre-increment operator:

```
zipit operator++(int)
{
  auto temp = *this;
  ++*this;
  return temp;
}
```

operator-> makes my head hurt, however. It should allow **iter->member** ... but how do we do that while managing two iterators?

```
zipit<Iter> const * operator->() const
{
  return this;
}
```

works, but then usage is ugly:

```
std::vector<std::string> sv1 = {"a","b","c"};
std::vector<std::string> sv2 = {"c","b","a"};
auto sv_zip = begin(sv1,sv2);
std::cout << sv_zip->first->length() << '\n';
```

and possibly incorrect. Should usage be **sv_zip->first.length()**? Or **sv_zip.first->length()**? The latter won't call **operator->** and is available for free as long as the inheritance from **std::pair** is public.

Alas it's hard to know what to return from a zip iterator when pointers or references are involved.

If we want to take it further, and model an output iterator then we must make the iterator dereferenceable as an lvalue (i.e. **operator*** returns a type convertible to **T &**). This leads to a similar problem.

Bear in mind also that the iterator we're storing is not necessarily a pointer, it's something that behaves like a pointer, but may not be a raw pointer. Case in point – **vector<bool>::iterator** is a proxy class that accesses the underlying storage through something sufficiently advanced (magic?). Dereferencing it returns a **vector<bool>::reference**.

Using this for inspiration we can make an attempt at **operator*** for output iterators:

```
// references cannot be stored in std::pair,
// so we create a wrapper
template <typename T> class ref_wrap {
  private:
    T & p;
  public:
    explicit ref_wrap (T & t) : p{t} {}
    operator T & () const noexcept {
      return p; }
    T & operator= (const T & t) const {
      p = t; return p; }
};
// convert the iterator type to a reference
// type
template <typename T> struct
  iter_reference_type { typedef typename
  std::iterator_traits<T>::reference type; };
template <typename T> struct
  iter_reference_type < T * > {
    typedef T & type; };
// optionally wrap the reference in our
// wrapper
template <typename T> struct
  wrapped_reference_type { typedef T type; };
template <typename T> struct
  wrapped_reference_type < T & > { typedef
    ref_wrap< T > type; };
template <typename Iter> struct
  wrapped_iter_reference_type {
  typedef typename wrapped_reference_type< \
    typename iter_reference_type<Iter>::type \
    >::type type; };
// a second template parameter is added to
// zipit in case the reference deduction
// doesn't work for some custom iterator
template <typename Iter,
  typename wrapped_reference = typename
wrapped_iter_reference_type<Iter>::type >
class zipit : public std::pair<Iter, Iter>
{
...
public:
  typedef wrapped_reference reference;
  std::pair<reference, reference> operator*()
    const {
    return std::make_pair(
      reference(*this->first),
      reference(*this->second) );
  }
...
};
```

Here we have a simple class to wrap a reference, which allows us to store it in an **std::pair**. **std::reference_wrapper** cannot be used since it is implicitly convertible from **T** and any assignment then simply changes the reference, not the referred to value.

The SFINAE is required to extract the true value reference type. **T *** converts to **T &**, **std::vector<T>::iterator** converts to either the proxy object, or if it is a raw pointer to **T &**. Raw references then get wrapped and packed in a pair. Proxy objects don't get wrapped, they just get put into a pair.

The same wrapper could be used to implement **operator->** but since we have to return a pointer to the **std::pair<reference, reference>** we might have to allocate the pair object on the heap and return as a **unique_ptr**. And that's really ugly. I want no part of it.

…and of course plenty more features are needed if the iterator is to model one of the other iterator types.

…and don't forget to specialise **std::iterator_traits** (deriving from **std::iterator** was deprecated in C++17).

…and since C++11 the iterators must be swappable through **std::swap**… so specialise that too.

All in all, implementing iterators can lead to bouts of anxiety and heavy drinking. So drop **zipit** from the code – just use **Boost.ZipIterator** and accept its semantics, or **std::inner_product** without a zip iterator, or **std::vector<std::pair<int,int>>**, or don't store the dice throws.

References

[1] http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0515r0.pdf

[2] https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.2017

[3] http://libcxx.llvm.org/cxx1z_status.html

[4] https://en.wikipedia.org/wiki/Linear_congruential_generator

[5] *The C++11 Programming Language standard* – ISO/IEC 14882:2011(E)

[6] http://www.cplusplus.com/reference/iterator/

Balog Pal <pasa@lib.hu>

This entry starts with a really good teaser. Dice game that is always drawn? In real life I would ask "did you check your RNG sequence? I'd bet you either keep generating the same number or the exact same sequence for the two players." It's worth a look just to prove my gut right or wrong.

Looking at the code I see some iterator magic that I save for later. Then more magic with the new **<random>** that I'm not up to date with. However, in the past I read a lot of problems related to RNG. It reminds me an old story from around the millennium. I had to write an application that used winsock sockets for communication. When it was 'ready' I went to the forums to ask some peers' opinion. And the people, instead of commenting on the code just pointed me to fetch the 'winsock lame list' and serve myself. As anyone would guess I found a good amount of matches… And **random** looks like a similar can of worms, maybe someone compiled a similar list? Let's try a web search for 'lame list for rng'. Wow, guess what, first hit is titled 'The C++ <random> Lame List – Elbeno' [1]. It even refers to the ws lame list as the muse. Unfortunately, it lacks the explanations but even just the points can help a lot.

Matching the code against the list I see hits on 6, 7, 11, 14 and we get half-dozen good hints on what to avoid as we try to fix. And especially 14 looks like a killer that may match my first clue. But let's stash that too, and put the code in a compiler so we can instrument it with some output at interesting points. And while at it, try some static analysis.

In Visual Studio 2017 (15.5.5) default W3 compile gives one warning on **total +=** – conversion **double** to **int**. Maybe Wall tells more. Huh, indeed. 900 warnings! Too bad all but 4 come from the standard headers. I wonder if the VS team ever tried the dogfood method that others use to create actually usable stuff. The 3 warnings on our code are **int signed/unsigned** mismatch that are not very interesting till we not press the bounds. But the **int turns** that comes directly from user input is just used to size the vector… we'll try to play with -1 later. <evil grin>

Now let's see analyze. We're warned that **begin()** and **end()** can be declared **noexcept**. Also 'The reference argument **generator** for function **play** can be marked as const". That in general we treat by adding the **const**, but for this case it is an indication of a problem: we expect the generator to mutate, if it could be **const**, we messed up. Must be that missing **ref()** from point 14.

Lastly, a warning "Do not dereference a invalid pointer (lifetimes rule 1). **return of +=** was invalidated by **end of scope for allotemp.2**" that honestly beats me, especially as **op -=** has identical code without warning.

Clang invoked through the 'clang power tools' extension also picked that ref should be **const** [see the clang-tidy check google-runtime-references] and said that in call **play()** the first arg is not initialized, what is not really true for our case, if we got there, **cin** reported okay, so it did complete **>>** properly.

Let's just run it as is to see those draws. No luck here, we hit an assert: "vector iterator not dereferencable" in **operator***, debugger shows **this->first** being **null**. Wow. While at it, I take a peek into the sequences sitting in player 1: 3 1 3 6, and 2: same. Was I right or was I right? But I really want to see the output, maybe in release build that assert will not stand in the way? Indeed, now the program executes, and I get … 'Player 1 wins' on all my tries. Let's hope we didn't also launch some nukes on another continent.

Originally, I did not want to bother with the code before **class randomize** at all, but must push back the related rant even further, to at least get UB out of the way. We don't have too much code before the assert, just the **begin()** so let's look at that: guess what, it takes out vectors by value, gets the iterator, stores it, then the copies are destroyed for good leaving us with the invalid iterators. Same story with **end()**. Let's add the missing **&**s and see what happens. Yey, no more **assert**, and we're finally drawn for all attempts. And if we use **std::ref(generator)** in the two **generate()** calls, then we get various results too.

So, once we have the program working as desired, what's left besides to throw it ALL away and create properly? From scratch, as we learnt from the food industry, that if you already mixed the bad things with the good, it is hopeless to separate them later, instead we only add what is good. It does not stop us from reusing what we like from the original source be it idea or a code fragment.

For start we paste the original **main()**. It looks okay-ish for the original aim. We just need a bounds check on **turns** before passing it on. Call to **play()** is fine, just make sure **generator** is passed by ref. And we will need the random source. Having it on the stack will trigger #11. We might ignore that reasoning it is a single case and we are positive to have enough stack space. Or use some alternative, like making it static or have a dynamic instance with **make_unique**. This program does not want threads, so we can ignore the related points, but commit them to memory.

Now let's make a proper random source. While at it, let's also rename it accordingly, **random_source** is way better description than **randomize**. The list approves **mt19937** as a good start, I have to look up how to use it. Along with other hints that we'll need **random_device** and **uniform_int_distribution**. The latter on cppreference.com has the code for our case. All together I came up with this:

```
class random_source
{
  static unsigned int get_seed()
  {
    try {
      std::random_device rd;
      return rd();
    }
    catch (std::exception const&)
    {
      return static_cast<unsigned int> (time(
        nullptr));
    }
  }
  std::mt19937 mt{get_seed()};
  std::uniform_int_distribution<> dis{1, 6};
public:
  int operator()() {
    return static_cast<int>(dis(mt)); }
};
```

The description of **random_device** is pretty vague, most of it is left to the implementation, but it is intended to do the best to summon some entropy. If it can't and we get the exception the OP should decide what to do, I just dumped in an alternative source that is less lame as the 2nd violin. Time returns unspecified type, but it is numeric and for unsigned anything goes. The other cast in the **op()** is benign as the number is in the known range. I was thinking to change the return type instead to **unsigned**, but

for the use plain **int** is the natural thing. Or is it? Maybe **byte** is more fitting. That can be rearranged later. Moving on to **play()**.

The aim was to do play N rounds, in each make both player throw dice, decide the round outcome and accumulate the result. One way to do that is

```
int turn(random_source &generator)
// returns score for player 1: -1 0 or +1
{
// return sign( generator() - generator() );
// too bad no stock sign function....
  auto const d1 = generator();
  auto const d2 = generator();
  return d1 < d2 ? -1 : d1 > d2;
}
int game(int turns, random_source &generator)
// the difference between player 1 and 2's wins
{
  int total = 0;
  for (int i = 0; i < turns; ++i)
    total += turn(generator);
  return total;
}
```

And in the original **play()** function we just

```
auto const total = game(turns, generator);
```

and print the result. And we are done. For good. No kidding. Fun thing that it now even works with the original **main** without bound check and prints draw for -1 instead of crash or assert.

Someone may object that we were supposed to criticize or fix **zipit**… Were we actually? It breaks one of my fundamental rule for reviews (and dev process): application programmers are forbidden to write blacklisted stuff, that starts with known timewasters like 'logger class', 'string class' 'refcounting pointer', and also contains 'collection' and 'iterator'. For the collection I can be convinced in theory if someone has a problem that requires a really new type of collection not covered by either standard or the many libraries we work with. And for iterator if one opens with "I have this code full of algorithm use, I needed an iterator for that special collection to drive them." Not the case here. Even for that case I'd ask whether it is really needed right now or we may wait a little till ranges finally arrive.

If the more advanced version of the program have actual need to keep record of turns, it can be simply done in a **pair<int, int>** per round, collection of those and running whatever processing with tools that work (WELL!) out of the box.

And those who really need to implement their own, can start with web search for 'how to write c++ iterator', the first hit [2] is more than promising.

An extra stab at header separation: this simple code does not need a header at all, now, or probably ever. But if some parts are extracted, the header shall be self-containing, have all needed **#include** lines, not rely on the client.

References

[1] www.elbeno.com/blog/?p=1318
[2] https://stackoverflow.com/questions/3582608/how-to-correctly-implement-custom-iterators-and-const-iterators

Commentary

There's not much left to say so my commentary is brief. The presenting problem contained undefined behaviour (UB), which in this case meant different people found the code worked differently depending on the compiler and flags they used. On of the nasty problems with UB is this unpredictability: code containing UB may work perfectly for years and then stop working when something changes – sometimes something very small is enough to trigger it.

Fortunately the tools are improving at detecting UB statically or dynamically – it is well worth investigating what is available for your target platform(s).

In this case, the UB came from the poorly written iterator class. As several entries pointed out, it's generally better to try and find an available library for such components than to write it yourself, at least initially. However, if you do fail to find a suitable pre-written class, it is worth searching for help online. Last millennium there was a great book written by Matt Austern (*Generic Programming and the STL: Using and Extending the C++ Standard Template Library*) – but I'm not aware of a similar publication for modern C++.

It is important for C++ programmers to remember that the standard library is generally value-based and so objects may be copied which is what happens here to the arguments given to the **generate()** call

The Winner of CC 109

The four critiques remaining in the C++ world all found and fixed the two main issues. However, there are other problems with random number generation: both the distribution and the seed, as James noted and Pal and Jason demonstrated.

The solution seemed slightly over-engineered as there didn't really seem to be any good reason to create two collections of the scores and then zip them together. James and Pal both pointed to simplifications that remove the need to use zipit at all. I think this is a more helpful direction for the writer of the code than trying to fix the problems with their implementation, but Jason did a pretty thorough job of listing a number of the things that would need considering if a 'proper' iterator is actually required.

Russel's approach was very different; seeing what changes and what stays the same when one re-implements similar logic in a different language is very interesting. As he says, the right language to use for a particular problem depends heavily on what language(s) you are 'at home' in.

There were several good critiques this time – thank you to all those who spent time putting together their entry! Given that his suggestion was used in this critique Jason has requested, in the interests of fairness, that he would like not to be considered for the prize. Of the others, I have awarded the prize for this issue to Balog Pal (I particularly liked his phrasing about mixing good and bad code).

Code Critique 110

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

> I've written a simple program to print the ten most common words in a text file supplied as the first argument to the program. I've tried to make it pretty fast by avoiding copying of strings. Please can you review the code for any problems or improvements.

What would you comment on and why?

Listing 3 contains the code. (Note: if you want to try compiling this on a pre-C++17 compiler you can replace **string_view** with **string** and most of the issues with the code remain unchanged.)

Listing 3

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <map>
#include <sstream>
#include <string_view>
#include <unordered_map>
#include <vector>

int main(int argc, char **argv)
{
  std::unordered_map<std::string_view, size_t>
    words;
  std::ifstream ifs{argv[1]};
  std::string ss{
    std::istreambuf_iterator<char>(ifs),
    std::istreambuf_iterator<char>()};
```

ACCU London

Frances Buontempo reports from the London chapter.

On January 10th 2018 we started the new year with a joint meetup with C++ London [1], organised by Phil Nash. There were two talks, jointly titled 'Origami and state machines'.

- Vittorio Romeo gave an 'Introduction to C++ origami'.

  Fold expressions, introduced in C++17, allow us to easily generate code that combines variadic template arguments together or performs an action on them.

  After a brief overview of the feature's syntax, this short talk showed some cool and useful utilities that can be implemented using fold expressions.

  Some familiarity with variadic templateswas required – knowledge of C++17 features is not required.

- Andrew Gresyk then talked on 'Practical HFSM'.

  He gave an update on the progress made on the Hierarchical Finite State Machine library [2] since his April 2017 talk, and a live demo showcasing its usage in production-like code. These were hosted at skillsmatter and the recording is available [3].

On 12th February we invited Jason Gorman, Managing Director of Codemanship [4] to talk about .Net Code Craft. We fed him a couple of pints first and he asked not to be recorded, so if you weren't there you've missed it!

He talked about the five factors that make code more difficult and expensive to change, and explored how we can write code that delivers value today, and leaves the door open to delivering more value tomorrow.

We saw a hands-on demo, refactoring some unpleasant code to make it easier to work with.

Both sessions were really well attended. We're looking forward to the rest of the talks this year. If you'd like to speak, put a message on our meetup page [5] or get in touch with Ralph (ralph@dibase.co.uk), Fran (frances.buontempo@gmail.com) or Steve (steve@arventech.com).

It would be wonderful if someone from each of the local groups sent CVu@accu.org a short write up of when their talks/workshops etc. happened and what they were about. Don't wait for the organisers to do this – anyone can write it up!

Don't forget to check the local groups page on our website [6] to see if there's a group near you. If there isn't why not contact Nigel at nigellester99@gmail.com to find how to start one?

References

[1] https://www.meetup.com/CppLondon
[2] https://github.com/andrew-gresyk/HFSM
[3] https://skillsmatter.com/explore?q=C%2B%2B+London+January
[4] http://www.codemanship.com
[5] https://www.meetup.com/ACCULondon
[6] https://accu.org/index.php/accu_branches

FRANCES BUONTEMPO

Frances has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

Code Critique Competition (continued)

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

Listing 3 (cont'd)

```
auto *start = ss.data();
bool inword{};
for (auto &ch : ss)
{
  bool letter = ('a' <= ch && ch <= 'z' ||
    'A' <= ch && ch <= 'Z');
  if (inword != letter)
  {
    if (inword)
    {
      std::string_view word(
        start, &ch - start);
      ++words[word];
    }
    else
    {
      start = &ch;
    }
    inword = !inword;
  }
}
```

Listing 3 (cont'd)

```
std::map<size_t, std::string_view> m;
for (auto &entry : words)
{
  auto it = m.lower_bound(entry.second);
  if (it != m.begin() || m.empty())
  {
    m.insert(it,
      {entry.second, entry.first});
    if (m.size() > 10)
    {
      m.erase(m.begin());
    }
  }
}
for (auto &entry : m)
{
  std::cout << entry.first << ": "
    << entry.second << '\n';
}
}
```

Report on Challenge 2

Francis Glassborow presents the answers to his last challenge and gives us a new one.

There was a gratifyingly varied set of solutions offered. Some were ingenious and surprised me. Here are the ones I received together with my comments. At the end you will find my decision as to the 'winner' along with my next challenge.

From Alex Kumaila

```
// Casting to long to avoid assigning a sum that is
// larger than the max signed integer, to a signed
// integer.
long sumInts(const int x, const int y){
  return (long)x + (long)y;
}

void add(const int a, const int b, long *const c){
  if(a<0 && b<0) { // Both negative,
                   // therefore decrement.
    do {
      (*c)--;
    } while(*c > sumInts(a,b));
  } else if((a<0 && b>0)||(a>0 && b<0)) {
// Different signs, therefore decrement to the
// negative value, and then increment to the
// positive.
    do {
      (*c)--;
    } while(*c > std::min(a,b));
// min(a,b) returns the negative value,
// given the above predicate.
    do {
      (*c)++;
    } while(*c < sumInts(a,b));
  } else {
// Both positive, including both zero,
// therefore increment.
    while(*c < sumInts(a,b)){
      (*c)++;
    };
  }
}

int main() {
  static long c;
//Static long initialises to zero.
//Test cases assuming a 4 byte signed integer.
std::cout
  << "Integer size on repl.it (should be 4): "
  << sizeof(int) << "\n";
//endl doesn't appear to work on repl.it
  add(0,0,&c);
std::cout << "Both zero: " << c << "\n";
  add(1,2,&c);
std::cout << "Sum of two small (+ve) numbers: "
<< c << "\n";
  add(-2147483647,-2147483647,&c);
std::cout << "Both lower (-ve) bound: " << c
  << "\n";
  add(2147483647,2147483647,&c);
std::cout << "Both upper (+ve) bound: " << c
  << "\n";
  add(-2147483647,2147483647,&c);
std::cout
  << "Different signs at lower/upper bound: "
  << c << "\n";
  add(2147483647,-2147483647,&c);
std::cout
  << "Different signs at upper/lower bound: "
  << c << "\n";
}
```

FG: The odd reference to *repl.it* is because the code is loaded there so that you can test it. The link is https://repl.it/repls/WideeyedRepulsiveHoatzin. I leave it to the reader to discover what a Hoatzin is (there really is such a beast).

FG: The code in `main` tests several corner cases as well as one straight forward addition.

FG: As you can see, this solution is a pure solution that avoids any form of assignment though that is rather severe (and was not intended). It also results in rather long run times for large absolute values of the operands.

Note that the basic solution will work in C if the output lines are rewritten to use `printf`.

FG: However, the code can have undefined behaviour on any platform where `int` and `long` have the same range (allowed by the standard)

From Silas Brown

Hi Francis, please find a response below.

Thanks.

Silas

FG: A great pleasure to see a contribution from you. (For readers who do not know, Silas joined ACCU whilst he was still at school and served as our disabilities officer for many years.)

One obvious way to do assignment without `=` is to use constructors in C++:

```
#include <iostream>
using namespace std;
int main() {
  int a(1),b(2);
  int c(a+b);
  cout << c << endl;
}
```

Apart from that, C++ before C++17 also supports `or_eq` as a 'trigraph' [**FG:** No, that is an alternative token, not a trigraph] for `|=`, which I'm not sure should qualify for the challenge, because it's technically the same as using `|=`, it's just representing it differently in the source file:

```
static int c;
c or_eq (a+b);
```

FG: Using the alternative tokens is fine and is one of the solutions I expected someone would come up with. These are still valid in C++ 17. What has been removed is the trigraphs. Those were a ghastly fix C invented to deal with some keyboards that lacked some keys for characters C uses. I heard one

FRANCIS GLASSBOROW

Since retiring from teaching, Francis has edited C Vu, founded the ACCU conference and represented BSI at the C and C++ ISO committees. He is the author of two books: *You Can Do It!* and *You Can Program in C++*.

person opine that it would have been cheaper to give them all new keyboards than the cost of supporting those rarely used alternatives.

This relies on **static** being initialised to **0** by default. Without the **static**, it might work anyway but this is not guaranteed (and it also might result in reformatting your hard drive). The above will not work if the code is called more than once in the same program, unless you first clear **c** by calling **memset()** or similar.

FG: Actually you can set a variable to zero by using **xor_eq** (**a xor_eq a** will set **a** to zero). The problem is that you need to initialise variables before use or suffer the potential for undefined behaviour. Static and global variables get default initialised to zero.

FG: Note that what had to be avoided was the **=** symbol.

In C, you could use simple counting, but it's inefficient and it destroys the values of **a** and **b**:

```
#include <stdio.h>
int main() {
  static int a, b, c;
  a++; b++; b++;  /* so a is 1 and b is 2 */

  /* addition starts here;
     we assume a and b are both > 0 */
  while (a--) c++;
  while (b--) c++;

  printf("%d\n",c);
}
```

(The same considerations as above apply re the use of **static**.)

But I very much prefer a method that does not deconstruct the addition into increments and decrements. You didn't say if we can use any standard libraries for the addition code; there are two approaches there that spring to mind. One is **memset()** which I've already alluded to: if we are dealing with small numbers that fit into a **char**, then it's trivial:

```
#include <stdio.h>
#include <string.h>
int main() {
  char a, b, c;
  memset(&a, 1, 1);
  memset(&b, 2, 1);
  memset(&c,a+b,1);
  printf("%d\n",c);
}
```

But you never said the integers won't overflow the bounds of **char**. We could extend the above to larger numbers if we know how the platform represents integers, e.g. on a 32-bit little-endian system:

```
#include <stdio.h>
#include <string.h>
int main() {
  int a,b,c;
  /* ... set a and b ... */
  memset(&c,(a+b)&0xFF,1);
  memset(((char*)(&c))+1,((a+b)>>8)&0xFF,1);
  memset(((char*)(&c))+2,((a+b)>>16)&0xFF,1);
  memset(((char*)(&c))+3,((a+b)>>24)&0xFF,1);
  printf("%d\n",c);
}
```

But I don't like introducing this dependency on the underlying hardware, nor repeating the addition so much (although an optimising compiler would likely re-use the intermediate result).

The other 'standard library' approach that springs to mind is using **sprintf** and **sscanf**:

```
#include <stdio.h>
int main() {
  int a,b,c;
  sscanf("1","%d",&a);
  sscanf("2","%d",&b);
  char buf[22]; /* sufficient for 64-bit */
  sprintf(buf,"%d",a+b);
  sscanf(buf,"%d",&c);

  printf("%d\n",c);
}
```

But that has the disadvantage of converting to and from a string at runtime (not quite as bad as the counting approach, but not as fast as the C++ solutions).

Finally I have a 'cheat' answer, which is to write the C file in the UTF-7 character set, which (similar to the **or_eq** trigraph) lets you write **=** without using the **'='** byte:

```
+ACM include +ADw stdio.h +AD4
int main() +AHs
  int a +AD0 1 +ADs
  int b +AD0 2 +ADs
  int c +ADs
  c +AD0 a +- b +ADs
  printf ( +ACIAJQ-d+AFw-n+ACI, c) +Ads
+AH0
```

This can be compiled with GCC using the **-finput-charset=UTF-7** option [**FG:** Oh! Dear! You can write it but not compile it without using the **=** key] but only if your GCC has been compiled with the *iconv* library (which is not the case as standard on every GNU/Linux distribution), and it's certainly not Standard C, so it ought to be disqualified.

FG: I had wondered about whether something along the lines of your last solution was possible. Thanks for demonstrating that it is, at least using GCC.

Incidentally, this kind of challenge now has a real application in computer museums. For example, the Computing History Museum at Cambridge has a BBC Micro with a broken 'F' key. Well, you could exploit a bug in its operating system to program all 10 of its extra function keys to 'F' in just 12 keystrokes (type !2832=17937 and press Return), but other machines from the era didn't have function keys, and if you want to do a live programming demonstration on a restored mainframe with a terminal from the period, it's entirely possible you'll have to work around certain keys not working on its keyboard.

FG: Thank you Silas for a comprehensive set of solutions as well as a reason that one might actually need to do something like this (apart from doing exam questions from the dawn of computing)

From Hubert Mathews

Assigning the sum of two integer variables to a third variable without using **=** is easy in languages that don't use **=** for assignment. For instance in R:

```
a <- 10
b <- 25
c <- a+b
sprintf("%d + %d is %d", a, b, c)
```

or even in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    77 A PIC 99.
    77 B PIC 99.
    77 C PIC 99.
PROCEDURE DIVISION.
  SET A TO 10.
  SET B TO 25.
  ADD A B GIVING C.
  DISPLAY A " + " B " IS " C.
STOP RUN.
```

C++ has ways of initialising variables without using **=**, specifically using direct initialisation (either the C++98 version with parentheses or the C++11 uniform initialisation syntax with braces):

```
#include <iostream>
int main()
{
  int a(10);
  int b(25);
  int c(a+b);
  std::cout << a << " + " << b << " is "
            << c << std::endl;
}
```

Since these examples use facilities built into the language it hardly seems worth writing tests for them.

Things get more interesting for languages like C that have no obvious way of initialising variables without **=**:

```
#include <assert.h>
#include <string.h>
#define SET_VALUE(x, value)       \
  memset(&x, 0, sizeof((x)));     \
  while ((x) < (value)) (x)++;    \
  while ((x) > (value)) (x)--;
int main()
{
  int a, b, c;
  SET_VALUE(a, -4);
  SET_VALUE(b, 7);
  SET_VALUE(c, a+b);
  printf("%d + %d is %d\n", a, b, c);
  assert(!(a < -4) && !(a > -4));
  assert(!(b < 7) && !(b > 7));
  assert(!(c < 3) && !(c > 3));
  assert(!(a+b < c) && !(a+b > c));
  return 0;
}
```

This sort of code definitely requires tests as it is easy to get wrong. Tests without using **=** are even more fun and the above code uses the same technique as used by C++'s **std::map** for determining equality (not less than and not greater than). Using a macro means that there's no need to worry about accessibility of variables as there would be if using a function.

Which leads to the contortions that are necessary when trying the same in Java:

```
public class Add {
  int a, b, c;
  public void calculate() {
    while (a < 6) a++;
    while (a > 6) a--;
    while (b < 7) b++;
    while (b > 7) b--;
    while (c < a+b) c++;
    while (c > a+b) c--;
    System.out.println(String.format(
      "%d + %d is %d", a, b, c));
    assert !(a < 6) && !(a > 6);
    assert !(b < 7) && !(b > 7);
    assert !(c < a+b) && !(c > a+b);
  }
  public static void main(String... args) {
    new Add().calculate();
  }
}
```

The variables have to be fields in order that they will be initialised to zero. There is no obvious way of encapsulating the initialisation code into a function as **int**s are primitives and are passed and returned by value in Java, thus requiring an assignment and so an equals sign. Using boxed **java.lang.Integer** instead doesn't help as that would still require an assignment statement.

FG: Thanks for the language tour. I have to confess that I do not understand the Java solution.

From Pete Disdale

Hello Francis,

I will be very interested to see the 'several simple solutions' to this challenge – I have thought a fair bit about this and can come up with no more than 3 using only C! And of those, only 1 is a 'pure' solution inasmuch as the other 2 likely depend on **=** somewhere inside the library code. I also excluded **asm { .... }** code as it's not in the spirit of the challenge (and it's not C/C++ either). Perhaps there are more 'simple' solutions in C++?

Please see the attached `test.c`: **add1()** takes advantage of **sscanf** by storing the result in an **int***, **add2()** is the 'pure' C solution and works because there is no mandate to initialise a variable before use (bad, of course!) and **add3()** requires a support function, **foo()**.

Out of curiosity, I compiled this with and without optimisation, and whilst I was pleasantly reassured that the expression (a + b) was cached in the optimised code, I was really impressed that the optimiser had completely optimised away the call to **memcpy()** in **foo()**! The plain version (expected) is:

```
_foo:    pushl    %ebp
  movl    %esp, %ebp
  subl    $24, %esp
  movl    $4, 8(%esp)
  leal    8(%ebp), %eax
  movl    %eax, 4(%esp)
  movl    12(%ebp), %eax
  movl    %eax, (%esp)
  call    _memcpy
  leave
  ret
```

whilst the optimised version is:

```
_foo:    pushl    %ebp
  movl    %esp, %ebp
  movl    8(%ebp), %edx
  movl    12(%ebp), %eax
  movl    %edx, (%eax)
  popl    %ebp
  ret
```

(This with a fairly ancient version of gcc too.)

I look forward to reading all the other contributions in the next CVu.

ps. did anybody manage to find a single solution using Java?

`test.c`

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

int add1(int, int);
int add2(int, int);
int add3(int, int);

int main (int argc, char *argv[])
{
  /* int a, b; unused */
  if (argc > 2)
  {
    printf ("add1(%s, %s) gives %d\n", argv[1],
      argv[2], add1 (atoi (argv[1]),
      atoi (argv[2])));
    printf ("add2(%s, %s) gives %d\n", argv[1],
      argv[2], add2 (atoi (argv[1]),
      atoi (argv[2])));
    printf ("add3(%s, %s) gives %d\n", argv[1],
      argv[2], add3 (atoi (argv[1]),
      atoi (argv[2])));
  }
  return 0;
}
```

```
int add1 (int a, int b)
{   /* use hex for fixed length string,
       2 chars per byte, no ±, plus 1 for '\0' */
  char s[sizeof(int)*2 + 1];
  int c;

  snprintf (s, sizeof(s), "%x", a + b);
  sscanf (s, "%x", &c);

  return c;
}
int add2 (int a, int b)
{
  int c;

  if (c > a + b)
      while (--c > a + b);
  else if (c < a + b)
      while (++c < a + b);
  return c;
}
void foo (int x, int *y)
{
  // memmove ((unsigned char *) y,
  //(unsigned char *) &x, sizeof(int));
  memcpy (y, &x, sizeof(int));
}
int add3 (int a, int b)
{
  int c;
  foo (a + b, &c);
  return c;
}
```

FG: It is amazing what modern optimisers are able to do. With a bit of AI perhaps we can get them to rewrite our entire code to run more efficiently even if the resulting source code is completely incomprehensible to humans. Just joking (or am I?)

From James Holland

Francis uses the word 'assign' and so I assume initialisation doesn't count. That's a pity because statements something like `int c{a + b}` would fit the bill. Furthermore, when Francis says "without using the `=` symbol", I assume I can't use `|=` and its like. This is also a pity because I could have used two statements such as `c ^= c; c |= a + b;` to assign the sum of `a` and `b` to `c`. This gives me an idea, however. I could use the C++ 'alternative tokens'.

```
c xor_eq c;
c or_eq a + b;
c xor_eq ~c;
c and_eq a + b;
```

There are other ways of assigning the sum to the variable without using the `=` symbol, some more useful than others. We can ask the computer user for help.

```
while (a + b - c)
{
  std::cout
    << "Please type the number followed by Enter"
    << a + b << ' ';
  std::cin >> c;
}
```

Possibly not the most efficient method but at least the result can be checked. Instead of using a person, let's use the file system.

```
std::ofstream out_file("file.dat");
out_file << a + b;
out_file.close();
std::ifstream in_file("file.dat");
in_file >> c;
```

This is an improvement over the previous attempt, but we need not use the file system. We can use a string stream.

```
std::stringstream ss;
ss << a + b;
ss >> c;
```

Another way is to make use of `sscanf()` as shown below.

```
std::string s(std::to_string(a + b));
sscanf(s.c_str(), "%d", &c);
```

Perhaps we could keep incrementing the variable until it becomes the required value. This could take quite a while for a variable with a large number of bits!

```
while (a + b - c) ++c;
```

Lastly, many of the standard library algorithms can be persuaded to do the job as well.

```
int d{a + b};
std::swap(c, d);

std::fill(&c, &c + 1, a + b);

std::fill_n(&c, 1, a + b);

int d{a + b};
std::copy(&d, &d + 1, &c);

std::generate(&c, &c + 1, [a, b](){
  return a + b;});

std::iota(&c, &c + 1, a + b);
```

FG: I am not sure that any of these are more in the spirit of assignment without using an equals sign than just the simple `int c{a+b};`.

That said it is remarkable how many standard library functions can be subverted into storing the result of `a+b` in `c`.

The Winner is...

Well I am stuck because each of the entrants have strong positives.

- Alex took time to consider the corner cases and write a test to ensure they worked and then placed the test where anyone can see that it works.
- Silas walked us through several solutions and then added a motivation for this kind of problem.
- Hubert gave us a language tour and even provided a solution for Java.
- Pete demonstrated that optimisers can produce something respectable even when our code is pretty crude.
- James offered us a smorgasbord of ways to achieve our objective and actually came up with an effective solution without using initialisation.

So which do you like best? I am going to cheat (like James, asking the user to provide the answer) and ask you, the reader to email me with your choice. I will publish the voting figures in my next Challenge column.

Challenge 3

Here is an old problem that might just be new to some of you. Write a program that outputs its own source code. Please attempt this in C++. There are two categories of solution:

1. A solution that will run on your computer
2. A solution that will run on my computer.

In each case you are allowed to use standard library functions but everything but the output directed to a file must compile to produce the same executable. The first should be easy; the second may prove more challenging.

Bookcase

The latest roundup of book reviews.

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

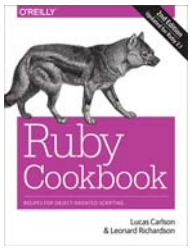
After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamorous 'not recommended' rating, you are entitled to another book completely free.

Thanks to Pearson and Computer Bookshop for their continued support in providing us with books.
Astrid Byro (astrid.byro@gmail.com)

Ruby Cookbook 2e

By Lucas Carlson & Leonard Richardson, published by O'Reilly, ISBN 978-1449373719.

Reviewed by Ian Bruntlett



Before attempting to read this book, I'd recommend reading and understanding *The Ruby Programming Language* first.

This is the first O'Reilly Cookbook I have read from cover to cover. It has 25 chapters on a variety of subjects. It has flaws – ranging from daft (e.g. ignoring `Array#to_h` and writing it from scratch) to wrong (e.g. its description of the `<=>` operator in 5.5 'Sorting an array'). It is big – just over 970 pages long. It looks like this edition is orphaned – the online errata is sparse and O'Reilly stated in an e-mail that the sample source code would not be available for download – which is a pity – O'Reilly's former reputation for maintaining errata and making examples available for download was a significant factor for me when I bought this book.

Chapter 1: Ruby 2.1. Discusses the changes made to Ruby between version 1.8 and 2.1.

The remaining chapters can be grouped by overall topic:

- Built-in data structures (6 chapters)
- Ruby idioms & philosophy (4 chapters)
- Popular ways of storing data (3 chapters)
- Network applications (4 chapters)
- Programming support (3 chapters)
- Miscellany (4 chapters)

Chapter 2 (19 Recipes): Strings. Deals with the obvious and introduces the use of irb (interactive Ruby) which is used heavily throughout the book. It covers many things that will augment the knowledge of someone with a basic grasp of Ruby strings.

Chapter 3 (17 Recipes): Numbers. Covers many things. From the obvious (converting strings to integers, comparing floating-point numbers) to more specialist stuff (matrices, logarithms).

Chapter 4 (14 Recipes): Date and Time. There are three classes in Ruby that handle time – the `Time` class (an interface to the C time libraries) and the Ruby classes (`Date` and `DateTime`). It handles the measurement of time, formatted output, Daylight Saving Time, running a code block periodically, waiting for time to elapse and adding a time-out to a long-running operation.

Chapter 5 (15 Recipes): Arrays. Starts off with the obvious (iterating over an array, building a hash from an array – note, use Ruby's `Array#to_h`). In 5.5 'Sorting an array' there is a seriously wrong explanation of the `<=>` operator's behaviour.

Chapter 6 (15 Recipes): Hashes. This is an interesting chapter both for people who have encountered hashes / dictionaries / associative arrays in other languages and those who are seeing them for the first time.

Chapter 7 (21 Recipes): Files and Directories. Covers day-to-day handling of them at a fairly high-level and O.S. independent perspective. Dig into the topic deep enough and you will uncover the underlying C / UNIX foundations.

Chapter 8 (11 Recipes): Code blocks and Iterations. This is fundamental to the idiomatic use of Ruby and this chapter is full of useful, pertinent examples. I'd read *The Ruby Programming Language* first, though.

Chapter 9 (19 Recipes): Objects and Classes. Good examples but best read by an intermediate Ruby programmer rather than a novice.

Chapter 10 (10 Recipes): Modules and Namespaces. Another interesting chapter but let down by a poor online errata. Some of its examples are useful – others are clever but fraught with pitfalls.

Chapter 11 (16 Recipes): Reflection and Metaprogramming. This is an interesting and challenging chapter, ranging from finding an object's class through to quite involved meta-programming (Aspect-Oriented Programming and enforcing software contracts).

Chapter 12 (15 Recipes): XML and HTML. The preface of this book states that this book has 'copy-and-paste code snippets'. This chapter explodes that myth. Consider 12.1 'Checking that XML is well-formed'. It casually uses the method `assert_nothing_thrown` without even telling the reader that this method belongs to a testing framework or where to find out more about it (it is discussed in Chapter 19 'Testing, Debugging, Optimizing and Documenting').

Chapter 13 (14 Recipes): Graphics and Other File Formats. Has recipes for graphics, graphs, sparklines, text (encryption, csv file handling), compressed and compressing files, YAML, PDFs and MIDI files.

Chapter 14 (17 Recipes): Databases and Persistence. Covers RDBMSs – from database-specific bindings (MySQL, Postgres) to generic RDBMS support (DBI – an idea borrowed from Perl) and an accessing an RDBMS without using SQL (ActiveRecord). A plethora of other approaches (YAML, Marshal, Madeleine, SimpleSearch, Ferret (inspired by Java's Lucene library), Berkeley DB. The chapter finishes with a section on avoiding SQL injection attacks.

Chapter 15 (20 Recipes): Internet Services. I am no expert in networking but this chapter covers a lot of ground and supports just about everything I've ever wanted to do on a network and the Internet.

Chapter 16 (23 Recipes): Web development: Ruby on Rails. I have dabbled with web development, previously, studying *Learning PHP, MySQL, JavaScript, CSS & HTML5 (3rd Edition)*. After that, I looked around for something else to study. Rails was the killer app that attracted me to Ruby. This chapter covers

View from the Chair

Bob Schmidt
chair@accu.org

This will be a short *View*, as i t is the last before the Annual General Meeting, and I don't have much to report.

2018 Annual General Meeting

This will be your last reminder – ACCU's 2018 Annual General Meeting (AGM) will be held on Saturday, April 14th, 2018, at the Marriott City Centre in Bristol, UK, in conjunction with the 2018 ACCU conference. The table at the top of the page shows the important, remaining dates associated with the AGM (the Proposal and Nomination deadline has already passed).

Please take the time to vote when your notification arrives via email.

Code of conduct

The website page previously titled 'Diversity Statement' has been renamed 'ACCU Values', and a generalized Code of Conduct has been added. This Code of Conduct was derived from the conference Code of Conduct; it is considerably shorter, since it concentrates more on our expectations of courteous conduct and less on the punitive and administrative aspects of enforcement.

All people at any event held under the auspices of ACCU are expected to abide by the Code of Conduct. The more expansive Conference Code of Conduct will be used during the ACCU Conference.

For more information, refer to the Code of Conduct [1] and the Conference Code of Conduct [2]. If you have any questions, comments, or criticisms, please contact me.

Important dates for ACCU AGM		
3 March 2018	Draft agenda	42 days prior to AGM
17 March 2018	Agenda freeze	(28 days prior to AGM)
24 March 2018	Voting opens	(21 days prior to AGM)

Web site redesign

I have received some feedback on my previous requests for comments on web site platforms. My thanks to Hubert Matthews, Russel Winder, and Martin Moene for their responses. There is certainly a lot to be considered prior to starting a project of this scope. I hope to have further discussions on the topic at the conference in April.

Web site status

After working on the backlog for several months, I now believe that the web site is up to date with respect to *CVu* and *Overload* articles and book reviews. PDF and HTML versions of all articles published since 1 July 2017 are posted, and the by-author and by-article bibliographies are caught up. In addition, I have made changes to several of the informational pages to bring them up to date.

We are still searching for a full time web editor. It has been taking me approximately three to four hours a month to get the magazines posted and the bibliographies created. If you are interested in volunteering for the position, please let me know.

I know I have said this several times over the past 8 months, but it needs repeating – I am very grateful for all of the work Martin Moene did creating multiple sets of instructions prior to his retirement from the role of web editor last year, and for all of the help he has provided me while I've stumbled around trying to keep up with the work since. My failures as interim web editor are my own, and my successes are directly related to Martin's legacy and ongoing help.

ACCU 2018

As mentioned above, the next ACCU conference will be held in Bristol, U.K., from the 11th through the 14th of April, 2018, with pre-conference workshops on April 10th. Barring some unforeseen circumstance, I will be there. Stop by the ACCU table between sessions and say hello.

References

[1] General Code of Conduct: https://accu.org/index.php/aboutus/diversity_statement
[2] Conference Code of Conduct: https://conference.accu.org/coc_code_of_conduct.html

Bookcase (continued)

a lot of Rails including basics, web application patterns and use of tools. For me, it provides a taste of things to come. I'll return to this chapter after reading a dedicated Rails tutorial.

Chapter 17 (12 Recipes): Sinatra. Sinatra is a slim web framework. I don't know enough about this sort of thing but this is another chapter that I will return to.

Chapter 18 (16 Recipes): Web Services and Distributed Programming. The previous chapters dealt with network programming (where you write software to enable people to do stuff on a network / the Internet). This chapter deals with distributed programming – where you write software to enable different computers to cooperate on a network / the Internet.

Chapter 19 (14 Recipes): Testing, Debugging, Optimizing and Documenting. This is an introduction to many useful things. I think there is the occasional typing error but I am not experienced enough in Ruby to decide that.

Chapter 20 (8 Recipes): Packaging and Distributing Software. This chapter is quite an eye-opener – Ruby has a lot of infrastructure for packaging and distributing software – typically in the form of gems. This is a particularly useful chapter.

Chapter 21 (8 Recipes): Automating tasks with Rake. I think `Rake` is a contraction of 'Ruby Make' and it performs so many repetitive tasks for Ruby programmers that it could be called the Ruby Butler. Like `make`, `Rake` is flexible and can be used for things beyond Ruby software development.

Chapter 22 (11 Recipes): Multitasking and Multithreading. Deals with threads, processes, Windows services, synchronising object access, and remote execution of code (via the net-ssh gem) and discusses various 'gotchas' that can trap the unwary. In places this chapter has been updated for Ruby 2.1

Chapter 23 (16 Recipes): User Interface. Covers the terminal / console / text UI, graphical UI, (with Tk, wxRuby, Ruby/GTK, AppleScript) and command line arguments using optparse. Other chapters cover Internet user interfaces. I didn't try all of the examples. This chapter can be a bit bleeding edge. I couldn't get the curses examples to run because support for it has been moved out of the Standard Library and I couldn't persuade the relevant gem to install.

Chapter 24 (5 Recipes): Extending Ruby with other languages. This has three recipes about accessing C functions in shared object files but also covers writing C code inline in a Ruby script. It also covers accessing Java class libraries from within a Ruby script.

Chapter 25 (11 Recipes): System Administration. This covers system administration tasks being performed by Ruby in Windows and UNIX-like systems. Personally, I believe that these tasks might be better solved by writing a script in bash or PowerShell – but, as a piece of software becomes more and more complex, Ruby becomes more appealing.

Conclusion. Software changes. Things break. This book won't protect you from that but the breadth of Ruby Programming knowledge here has proven to be useful and for that I am grateful.

VERDICT: Recommended.