The magazine of the ACCU

www.accu.org

{cvu}

Volume 23 • Issue 1 • March 2011 • £3

{cvu}

Volume 23 Issue 1
March 2011
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Stephen Baynes, Alexander Demin, Pete Goodliffe, Paul Grenyer, Richard Harris, Jon Jagger, Frances Love, Chris Oldwood, Roger Orr, Nat Pryce, Matthew Wilson

ACCU Chair
Hubert Matthews
chair@accu.org

ACCU Secretary
Alan Bellingham
secretary@accu.org

ACCU Membership
Mick Brooks
accumembership@accu.org

ACCU Treasurer
Stewart Brodie
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

What was, what is, and what may be

Jane was becoming increasingly confused. Since joining her team, she'd had a variety of little 'missions' to complete, all working towards the release of a replacement product, and (she had to admit) they'd even been *interesting* missions, technologically speaking. She'd even felt she'd had some influence on the design of the new system. Nevertheless, she was now finding it hard to escape a feeling of futility about it.

It seemed that the project was becoming bogged-down in constantly re-visiting details of design that she (and her colleagues, she felt sure) had already thought decided. No-one had a very clear idea of how the different components should communicate together, or even what those components should be. There was no clarity on some fundamental things like the domain types to be used, or the persistence mechanism for them. In short, a lack of concrete requirements. Oh sure, there was a broad agreement about what the system should 'do' – it must be be like the old system, but shinier and faster. And more extensible.

She suspected it was this last part that was causing all the trouble. Arequirements vacuum had inspired a universe of possibility in which all things were possible. Instead of 'just' replacing the existing system, it had to be able to support other output styles, manipulate new data types, received from as-yet-non-existent sources, all dynamically configurable (of course). The possibilities were endless.

Yes. Jane was **sure** that was the problem. Instead of actually asking people what they *required*, the project had become encumbered and paralysed by the dreams of *what might be*.

The question now was – what should she **do** about it?

Steve

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

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

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

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

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

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

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 23.2: 1st April 2011
C Vu 23.3: 1st June 2011

Overload 103: 1st May 2011
Overload 104: 1st July 2011

WRITE FOR C VU

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

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

ADVERTISE WITH US

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

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

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

COPYRIGHTS AND TRADE MARKS

The First Little Step into Test-Driven Development

Alexander Demin takes a good look at Google Test.

The software development world is changing rapidly – new versions of the operating systems, compilers, libraries are coming up faster and faster. It's actually great. Lots of options allow you to choose the development tools ideally fitting your personal requirements. Approaches to developing good quality software are also changing all the time. Nowadays the cool words in the programming world are object oriented design, functional programming, extreme programming and of course test-driven development (TDD).

Though I have more than ten years' experience of programming, and it covers various languages from machine code and assembler up to functional programming, I have discovered the test driven development world quite recently. Programmers are often very conservative (and quite lazy!) and they do not like to change their habits. I am a perfect example. But when I stepped over my laziness and started to use TDD I felt that my development became more predictable, more stable. I managed to split complex tasks into pieces, and manage code interdependencies significantly more easily and faster. More importantly: I have stopped repeating my coding mistakes, reintroducing already fixed bugs and now I am able to refactor my code anytime without any fear of breaking something important a day before the release. Why? All thanks to test driven development.

I would like to share my experiences on entering the wonderful world of TDD and hope to encourage somebody to join.

My main background is C and C++, so I will cover these languages, but all ideas mentioned are common for lots of modern languages (Java, C#, Python, Delphi etc).

Let's start from the beginning. Usually the first program written by a newbie is Hello World. Assume you have done it already and you want to do something more complex.

Let's assume you studied a lot of computer science and you know how to implement a very fast multiplication function. Listing 1 is what it might look like.

I want to warn the reader that this particular example is not ideal in terms of coding style and it's not clear in logic, it uses a lot of C/C++ 'cool short' expressions and so on. Also the function has some weird line with 920 and 847. This is intentional, and will be covered later.

Listing 1

```
// File: mult.h
#ifndef _MULT_H
#define _MULT_H
int mult(int a, int b);
#endif

// File: mult.cc
#include "mult.h"
int mult(int a, int b) {
  if (!a || !b) return 0;
  int r = 0;
  if (a == 920 && b == 847) r++;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

Listing 2

```
#include "mult.h"
#include <iostream>

int main(int argc, char* argv[]) {
  while(1) {
    std::cout << "enter a: ";
    int a;
    std::cin >> a;
    std::cout << "enter b: ";
    int b;
    std::cin >> b;
    std::cout << "a * b = " << mult(a, b)
       << std::endl;
  }
}
```

Now, you have done the code. You definitely know that it should work more reliably and faster because your computer science background tells you that. How can you make sure that it works correctly? The function code is quite 'non-understandable' and you cannot swear that it works correctly just by looking on the source. You have to try it on. The first and the most obvious way to create a simple example might be that shown in Listing 2.

Then you run it, play with it a bit, try a couple of examples and then come to the conclusion that it works. Later you add the `mult.cc` file to your project and probably delete the test example source because you do not need it anymore. You have linked the function into your application and you are almost happy.

Let's step back for a second now and imagine that unfortunately sometimes your application gives a wrong result or perhaps crashes and you suspect that the issue is your **`mult()`** function. You have to find your original test source or even write it again because you have lost it, then run it again under a debugger and try to find what the problem is. And now imagine you have hundreds or thousands of similar functions in your application and you have to re-test them all. It's a nightmare.

Well, let me show you another way – the test driven development way. We will use the excellent Google Test Framework 1.5.0 for that. You can download and unpack it in your working directory:

```
wget    http://googletest.googlecode.com/files/
gtest-1.5.0.tar.gz
gzip -dc gtest-1.5.0.tar.gz | tar xvf -
```

It will create `gtest-1.5.0` directory in your current folder. We will refer to this directory below so make sure that you use proper directory names in your compilation commands.

Then you create the unit test (`mult_unittest.cc`, in Listing 3).

ALEXANDER DEMIN

Alexander Demin is a software engineer with a PhD in Computer Science. Constantly exploring new technologies he is always ready to drill down into the code with a disassembler to prove that the bug is out there. He may be contacted at alexander@demin.ws.

Listing 3

```
#include <gtest/gtest.h>
#include "mult.h"

TEST(multTest, simple) {
  EXPECT_EQ(91, mult(7, 13));
}
```

Listing 4

```
#include <gtest/gtest.h>

int main(int argc, char **argv) {
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

This file contains the simple test case. The meaning of it is explained below.

Then the test main runner module (`runner.cc`, in Listing 4).

This runner will execute all declared tests in your test application. This piece of code can be almost the same for any of your unit test suites. It just parses the command line arguments and runs all tests.

Now let's compile it. If you are running Linux and have the GCC C++ compiler version 3 or later you can use the following command:

```
g++ -Igtest-1.5.0/include -Igtest-1.5.0 -o
mult_unittest gtest-1.5.0/src/gtest-all.cc
mult.cc mult_unittest.cc runner.cc
```

The mult_unittest executable should be generated. Let's run it:

```
./mult_unittest
```

It prints something like this:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 1 test.
```

Let's go back and look at it in more detail now. We have created a test case named **`multTest.simple`** in the file `mult_unittest.cc` (**`multTest`** is the test suite name and the **`simple`** is the test name in the suite) which runs your function with 7 and 13 as the parameters and checks that result is 91. The macro for the test declaration is **`TEST(...)`**. The magic happens in the **`EXPECT_EQ (...)`**. This function call has two arguments: the first one is the expected value and the second is the real one. If they are equal the function passes through quietly but if they are different it reports an error message.

The Google Test Framework provides a bunch of similar functions to check various conditions with different argument types. The **`EXPECT_*`** function family does not abort the test run. It just prints the report about a test failure and keeps going to execute other tests. The **`ASSERT_*`** functions (for example, **`ASSERT_EQ()`**) stop the test suite run and terminate the runner. They are convenient when there is no reason to continue testing on a fatal error (for example, a database is not available).

But in our case the test runner reports a successful test execution – the test case has been executed and the result is correct. That's fine but this test case is so obvious and checks only one pair of numbers. You need more. Because the **`mult()`** function has some weird checking of the argument for zero at the beginning let's test it. You add one more test case – **`multTest.zero`** (File: `mult_unittest.cc`, Listing 5).

Let's compile with the same command and run **`mult_unittest`** executable again. It should print this:

Listing 5

```
#include <gtest/gtest.h>
#include "mult.h"

TEST(multTest, simple) {
  EXPECT_EQ(91, mult(7, 13));
}

TEST(multTest, zero) {
  EXPECT_EQ(0, mult(0, 7));
  EXPECT_EQ(0, mult(7, 0));
}
```

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[ RUN      ] multTest.zero
[       OK ] multTest.zero
[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran.
[  PASSED  ] 2 tests.
```

The new test passes successfully as well and the **`mult()`** function seems to handle checking the parameter for zero correctly. But we still have an unsolved issue – your application using the function **`mult()`** fails and it means this function sometime returns wrong value. Let's add a stronger test to file `mult_unittest.cc` (Listing 6).

This test (**`multTest.all`**) checks all possible values of arguments from 0 to 999. Let's compile and run it again:

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from multTest
[ RUN      ] multTest.simple
[       OK ] multTest.simple
[ RUN      ] multTest.zero
[       OK ] multTest.zero
[ RUN      ] multTest.all
mult_unittest.cc:18: Failure
Value of: mult(a, b)
  Actual: 779241
Expected: a * b
Which is: 779240
[  FAILED  ] multTest.all
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran.
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] multTest.all

1 FAILED TEST
```

Listing 6

```
#include <gtest/gtest.h>
#include "mult.h"

TEST(multTest, simple) {
  EXPECT_EQ(91, mult(7, 13));
}

TEST(multTest, zero) {
  EXPECT_EQ(0, mult(0, 7));
  EXPECT_EQ(0, mult(7, 0));
}

TEST(multTest, all) {
  for (int a = 0; a < 1000; ++a)
    for (int b = 0; b < 1000; ++b)
      EXPECT_EQ(a * b, mult(a, b));
}
```

Listing 7

```
TEST(multTest, all) {
  for (int a = 0; a < 1000; ++a)
    for (int b = 0; b < 1000; ++b)
      EXPECT_EQ(a * b, mult(a, b))
        << "wrong result on a=" << a << " and
           b=" << b;
}
```

Wow! The test fails. It means we have found the problem. We see that in line 18 of `mult_unittest.cc` there is a test failure: the expected value is 779240 but the actual one is 779241. It's a great result, but we also need to know which exact parameters cause this error. So let's modify the test (Listing 7).

This code will also print the error message and the values of **a** and **b** on failure. The **EXPECT_EQ(...)** can be used the output stream similar to **std::cout**, for example, to print out the diagnostics on a test failure.

Compile and run it again. We should get the following result:

```
 [==========] Running 3 tests from 1 test case.
 [----------] Global test environment set-up.
 [----------] 3 tests from multTest
 [ RUN      ] multTest.simple
 [       OK ] multTest.simple
 [ RUN      ] multTest.zero
 [       OK ] multTest.zero
 [ RUN      ] multTest.all
 mult_unittest.cc:17: Failure
 Value of: mult(a, b)
   Actual: 779241
 Expected: a * b
 Which is: 779240
 wrong result on a=920 and b=847
 [  FAILED  ] multTest.all
 [----------] Global test environment tear-down
 [==========] 3 tests from 1 test case ran.
 [  PASSED  ] 2 tests.
 [  FAILED  ] 1 test, listed below:
 [  FAILED  ] multTest.all

 1 FAILED TEST
```

Now we know exactly that the function fails when **a**=920 and **b**=847. This is the problem. And now we can fix the 'problem' by removing the line **if a == 920 && b == 847) r++;** from the `mult.cc` file. Listing 8 is an error free version of the `main.cc`.

Well, now compile it and run **mult_unittest** once again. Here is the output:

```
 [==========] Running 3 tests from 1 test case.
 [----------] Global test environment set-up.
 [----------] 3 tests from multTest
 [ RUN      ] multTest.simple
 [       OK ] multTest.simple
 [ RUN      ] multTest.zero
 [       OK ] multTest.zero
 [ RUN      ] multTest.all
 [       OK ] multTest.all
 [----------] Global test environment tear-down
 [==========] 3 tests from 1 test case ran.
 [  PASSED  ] 3 tests.
```

All tests work perfectly and now you are sure that your function **mult()** is fully error free.

Let's analyse what we've done. We have created the function **mult()** and also the tests which can be used any time to prove its proper functioning. At this point test driven development strongly recommends you include the test build and execution into your project build. For example, this is the part of your myapp project makefile:

Listing 8

```
#include "mult.h"

int mult(int a, int b) {
  if (!a || !b) return 0;
  int r = 0;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

```
...
all: build

build:
  cc -o myapp main.cc mult.cc
```

You should add the test compilation and run into this makefile:

```
...
release: build test

build:
  g++ -o myapp main.cc mult.cc

test:
g++ -Igtest-1.5.0/include -Igtest-1.5.0 -o
mult_unittest gtest-1.5.0/src/gtest-all.cc
mult.cc mult_unittest.cc runner.cc

./mult_unittest
```

Why do you need this? You need this because each time you release the project (using release target) it will compile and run the test suite to make sure that the current implementation of the **mult()** function is ok and works as you expect.

Now imagine you want to check whether it is reasonable to use your own hacky implementation of the simple arithmetic operation as the multiplication. Let's run your test suite again using the command:

```
./mult_unittest --gtest_print_time
   --gtest_filter=multTest.all
```

We ask Google Test framework to print the test execution time and also we ask to run only one test using the filter by name.

The output:

```
Note: Google Test filter = multTest.all
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from multTest
[ RUN      ] multTest.all
[       OK ] multTest.all (1266 ms)
[----------] 1 test from multTest (1297 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1328
             ms total)
[  PASSED  ] 1 test.
```

It reports only one test run (**testMult.all**) and it takes 1279 ms on my Core 2 Duo laptop (timing on your machine may be different).

Now you want to try another fairly simple implementation for the **mult()** function (file `mult.cc`, in Listing 9).

Let's compile it using exactly the same command as we used for the first implementation:

Listing 9

```
#include "mult.h"
int mult(int a, int b) {
  return a * b;
}
```

```
g++ -Igtest-1.5.0/include -Igtest-1.5.0 -o
mult_unittest gtest-1.5.0/src/gtest-all.cc
mult.cc mult_unittest.cc runner.cc
```

and run it:

```
./mult_unittest --gtest_print_time
   --gtest_filter=multTest.all
```

The output should look like this:

```
Note: Google Test filter = multTest.all
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from multTest
[ RUN      ] multTest.all
[       OK ] multTest.all (1094 ms)
[----------] 1 test from multTest (1141 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1171
ms total)
[  PASSED  ] 1 test.
```

We see it takes only 1094ms on my laptop and it's faster than our original handmade implementation.

Now you know that the original implementation is not quite so good and may be optimized or replaced by a better one.

So what is that we have achieved by this entire exercise? What is the point of it?

Firstly, we have created a test mechanism for our function. This mechanism can be used at a later time to prove the function logic and it can be fully automated. Once created it can be re-used as many times as you want. You do not lose your efforts applied initially for creating the testing routine.

Secondly, we have included the test run into the project build. If the function logic is broken for some reason (you've changed the code accidentally or maybe the new version of the compiler has generated the wrong code) the test will automatically point you towards it by failing the build.

And thirdly, we tried two different implementations of the **mult()** function using the same test suite. This means you can easily refactor the code without any fear of breaking something. The tests will check the function results and the expectations from the function. You have determined the function behaviour via the test cases and from this point you can easily play with the function implementation. On top of this we have tested two different implementations for execution time and now we have enough information to choose the better one.

These are really awesome results – you have automated the error checking procedure for your project. You do not need to do any manual runs anymore, playing with parameters to make sure that everything works as expected after any recent changes. Let's imagine how just a little extra effort of writing a 5 minute test case (comparing to the original user interactive test application) gave us so much additional information and helped to create a better design for the application. It's definitely worth it.

There is probably an argument that in some cases testing can be tricky because real world applications are much more complex than this isolated example. That is 100% correct, however the answer to it is also very simple: you have to write testable code from the beginning. Every time a piece of code is done, ask yourself – how will I test it? And maybe you will write the code a bit more simply, a bit more split into simple sub-tasks, a bit more isolated from external dependencies and so on. Definitely writing testable code is a complicated issue and there are a lot of techniques for it: dependency injection, isolating the business logic from the object instantiation (**operator new**), using inheritance and polymorphism instead of overly complicated **if**/**switch** constructions and so on and so forth.

Listing 10

```
// File: mult.h
#ifndef _MULT_H
#define _MULT_H
int mult(int a, int b);
#endif

// File: mult.c (buggy version)
#include "mult.h"
int mult(int a, int b) {
  int r = 0;
  if (!a || !b) return 0;
  if (a == 920 && b == 847) r++;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

Of course I have referenced many things from the object oriented world which make it easier to use unit testing. Applications with object oriented design in most cases are quite easy to test but the classic procedural languages like C or Pascal, for example, are not out of the question either.

Let's see how to test a similar example written in ANSI C. Your sources are in Listing 10.

I will use another Google testing framework here – cmockery 0.1.2. This framework was designed to test C code and it's a very powerful framework. On top of the set of **assert_*** functions it can help to find memory leaks, and buffer under- and over-runs.

Let's get it:

```
wget http://cmockery.googlecode.com/files/
cmockery-0.1.2.tar.gz
gzip -dc cmockery-0.1.2.tar.gz | tar xvf -
```

This command will create the `cmockery-0.1.2` folder in your current directory. We will use it so do make sure you do all runs below with this as the current directory.

Let me show you the test suite with the same functionality but written in C (**mult_test.h** in Listing 11 and `mult_test.c` in Listing 12), and the runner (Listing 13).

Let's compile it with GCC version 3 or higher:

```
gcc -Icmockery-0.1.2/src/google -o mult_test
cmockery-0.1.2/src/cmockery.c mult.c mult_test.c
runner.c
```

If everything is correct you should test **mult_test** executable. Let's run it:

```
./mult_test
```

and it will print something like Listing 14.

Listing 11

```
#ifndef _MULT_TEST_H
#define _MULT_TEST_H
void mult_simple_test(void **state);
void mult_zero_test(void **state);
void mult_all_test(void **state);
#endif
```

Listing 12

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>

void mult_simple_test(void **state) {
   assert_int_equal(91, mult(7, 13));
}

void mult_zero_test(void **state) {
   assert_int_equal(0, mult(0, 7));
   assert_int_equal(0, mult(7, 0));
}

void mult_all_test(void **state) {
  int a, b;
  for (a = 0; a < 1000; ++a)
    for (b = 0; b < 1000; ++b)
      assert_int_equal(a * b, mult(a, b));
}
```

Listing 13

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>
#include "mult_test.h"

int main(int argc, char* argv[]) {
   const UnitTest tests[] = {
      unit_test(mult_simple_test),
      unit_test(mult_zero_test),
      unit_test(mult_all_test),
   };
   return run_tests(tests);
}
```

Listing 14

```
mult_simple_test: Starting test
mult_simple_test: Test completed successfully.
mult_zero_test: Starting test
mult_zero_test: Test completed successfully.
mult_all_test: Starting test
0xbe3e8 != 0xbe3e9
ERROR: mult_test.c:19 Failure!
mult_all_test: Test failed.
1 out of 3 tests failed!
    mult_all_test
```

The **`mult_all_test`** fails on line 19 and it reports that the expected value of multiplication is 0xBE3E8 (decimal 779240 = 920 * 847) but the actual one is 0xBE3E9 (decimal 779240). Now we *fix* the **`mult()`** function removing buggy line **`if (a == 920 && b == 847) r++;`**, giving Listing 15, an error-free version of `mult.c`., and run the test suite again.

Listing 15

```
#include "mult.h"

int mult(int a, int b) {
  int r = 0;
  if (!a || !b) return 0;
  do {
    if (b & 1) r += a;
    a <<= 1;
  } while (b >>= 1);
  return r;
}
```

Now it prints this:

```
mult_simple_test: Starting test
mult_simple_test: Test completed successfully.
mult_zero_test: Starting test
mult_zero_test: Test completed successfully.
mult_all_test: Starting test
mult_all_test: Test completed successfully.
All 3 tests passed
```

We see now all three tests work fine. Of course C-based unit testing is not as advanced and comfortable in terms of reporting or code organization. You have to declare your test cases in the header file and in the runner but this is a limitation of the C language. The cmockery framework from Google makes the most of what is technically possible for comfortable testing in C. But even if the reporting is not ideal you are always informed about which test fails and in which line.

Other languages have unit testing frameworks as well. jUnit for Java, pyUnit for Python and so on. The principles of unit testing are exactly the same – running small pieces of your application in isolation.

QA (Quality Assurance) testing and regression testing are separate big topic in themselves, and are handled differently. Good unit tests should be fast so they don't slow down the compilation process on the project. But sometimes you want to do stress testing for your code – maybe execute something millions of times, check memory allocation for leaks, create the test for a recently fixed bug to avoid its reintroduction later and so on. These kinds of tests can take a long time and it's not comfortable to run them on every project build. Here, QA and regression testing step onto the scene. It's also quite an interesting topic and I will try to cover it soon as well. ■

Many-festos

Pete Goodliffe crafts one manifesto to rule them all.

Confusion of goals and perfection of means seems, in my opinion, to characterise our age.
– Albert Einstein

It's becoming an epidemic! They're springing up everywhere. We've got them coming our of our ears. It's as if you can't write a line of code, kick off a development process, or even think about the act of coding without signing up to one.

With all these manifestos for software development, our profession is in danger of becoming more about politics than the actual art, craft, science, and trade of software development.

Of course, a large and important part of professional software development is the people problem. And that necessarily involves politics, to some extent. But we're making even the foundational coding principles a political battle. Is this for the best? Or is it just a fashionable, sound-bite-sized way to get your point across, and to try to garner support for your pet hobby-horse?

These 'development' manifestos are often too ambiguous for people to sign up to in any meaningful way. They're so general that they simply *must* be right. Akin to a development horoscope, if you will. Very few of them break new ground, or introduce anything genuinely radical. And, sadly, when a manifesto becomes popular we see factions form around it, leading to disputes about what the manifesto really stands for. Whole debates spring up around the exegesis of the particular manifesto items.

Software religion is alive and well.

Whether or not manifestos are a good idea, they seem to be springing up for any conceivable purpose. So, in order to stem the flow, and make it easier for future software activists who'd like to pen their own manifesto, here I present the one, the overarching, generic software development manifesto. `Manifesto<PET_SUBJECT>`, if you like.

A generic manifesto for software development

We, the undersigned, have an opinion about software development. We are concerned about the future of our profession, and our passion leads us to draw the following conclusions:

■ *We believe in*	a fixed set of immutable ideals
over	tailoring our approach to each specific situation.
■ *We believe in*	concentrating on and discussing only the things that interest us
over	the bigger problem.
■ *We believe in*	our opinion
over	the opinions and experiences of others.
■ *We believe in*	arbitrary black-and-white mandates
over	real-world scenarios with complex issues and delicate resolutions.
■ *We believe that*	when our approach is hard to follow
then	it only shows how much more important it is.
■ *We believe in*	crafting an arbitrary set of commandments
over	the realisation that it's just never that simple.
■ *We believe in*	trying to establish a movement to promote our view
over	something that will be genuinely useful.
■ *We believe that*	we are better developers than those who don't agree with us
because	they don't agree with us.

That is, we believe we're doing the right thing. And if you don't you're wrong. And if you don't do what we do, you're doing it wrong.

OK, OK

Alright. I'll admit it. I exaggerated for effect. And my tongue *is* in my cheek. Mostly. ■

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

References

The Agile Manifesto http://agilemanifesto.org/
The Craftsmanship Manifesto http://manifesto.softwarecraftsmanship.org/
The Refactoring Manifesto http://refactoringmanifesto.org/
The SOA Manifesto http://www.soa-manifesto.org/
The GNU Manifesto http://www.gnu.org/gnu/manifesto.html
The Software Testing Manifesto http://www.softwaretestingmanifesto.org/
The Library Software Manifesto http://techessence.info/manifesto/
The OpenCloud Manifesto http://www.opencloudmanifesto.org/
The Mozilla Manifesto http://www.mozilla.org/about/manifesto
The Cluetrain Manifesto http://www.cluetrain.com/
The End-User Manifesto http://alistair.cockburn.us/User+Manifesto
The Experience Design Manifesto http://www.brazandre.com/manifesto/
The Hacker Manifesto http://en.wikipedia.org/wiki/Hacker_Manifesto

Questions

1. What foundational development 'principles' do you hold dear?
2. Do you sign up to, or align yourself with, development streams like 'agile', 'craftsmanship' and so on? How closely do you agree with each of the items in their manifesto?
3. What do you think these manifestos do have to offer the development community?
4. What kinds of harm might they really be able do, if any?
5. Or do you keep your head down and ignore this kind of thing? Should you actually follow these software fashions and fads to maintain personal development?

A Game of Blockade

Baron Muncharris sets a challenge.

Good heavens Sir R----- you look chilled to the bone! Come take a seat by the fire and a glass of mulled wine to undo the ill effects of this most unseasonable weather.

To speed your recovery might I suggest a small wager to warm the blood?

Splendid fellow!

I have in mind a game invented to commemorate my successfully quashing the Caribbean zombie uprising some few several years ago. Now, as I'm sure you well know, zombies have ever been a persistent, if sporadic, scourge of those islands. On that occasion, however, there arose a formidable leader from amongst their number; the zombie Lord J------ the Insensate. [1]

Against all reason and contrary to historicity, the mindless Lord J------ was possessed of a remarkable proficiency in the art of soldiering. He rallied the shuffling horde into a single effective fighting force and proceeded to make war upon the unfortunate islanders.

I was out scouting atop the mountains in the south when I spied the zombie army marching in a westerly direction deep in a ravine beneath me. As I turned back to give report of their position, my steed lost his footing and we came sliding down the mountainside. We gathered quite the collection of boulders during our descent and it took all of my not insubstantial horsemanship to keep from falling; were I a lesser man my horse and I should surely have perished.

When we reached the bottom I realised that our motley collection of boulders had completely blocked off the eastern end of the ravine. I saw immediately what needed to be done; I took one of the larger monoliths and, employing my saddle as a sling, cast it the ten miles to the left hand slope at the westward end. As I had anticipated, its impact set off a second landslide trapping Lord J------'s forces between them.

Safe in the certitude that they could not escape their mountain gaol, I proceeded at my leisure to alert the artillery of their target.

But now I am delaying our sport!

The game shall take place upon this chessboard I have set here before you, although the outer squares shall play no part in our contest as we shall restrict our field of play to the inner six by six squares.

We shall each take a store of dominoes and take turns placing them upon the board so that each lays across two adjacent squares, either rank-wise or file-wise. You shall place the first and the game shall continue until the entire field is thusly covered.

We shall be free to choose any squares not yet occupied unless the placement would render the game inconclusive by carving out an island of empty squares odd in number.

If, at the conclusion, you have played artfully and blockaded each line running between the ranks and files of the field by ensuring that they are all straddled by at least one domino you shall have one coin of mine. Contrariwise, if you have not, I shall have two coins of yours.

When I described the manner of play of this delightful test of strategy to that wretched student with whose acquaintance I am so tediously burdened, he took it upon himself to describe, at some quite unwelcome length I might add, his newfound interest in pigeons. Now I will happily admit to some small interest in ornithology, albeit principally the noble falcon and not in any way that verminous fowl of his fancy, but quite why he imagined that I should wish to discuss the subject with him is entirely beyond me.

But this is not a fitting subject for conversation!

Come, take another draught and formulate your stratagems! ■

Note

[1] With thanks to 'zombie' Lee Jackson for this puzzle

Figures

Figure 1: Sir R----- loses!

Figure 2: An illegal move.

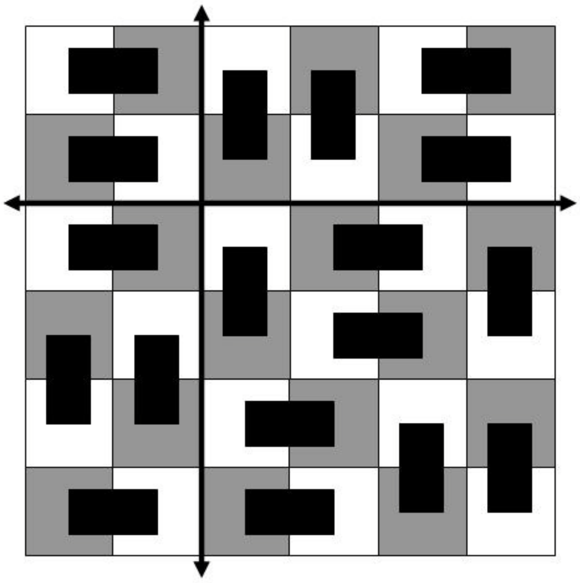


Figure 1

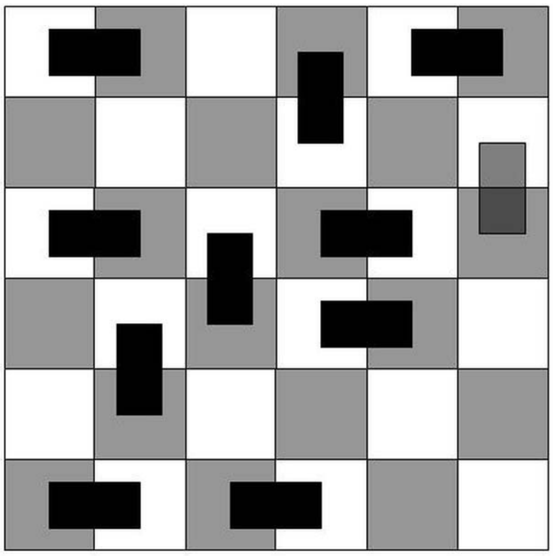


Figure 2

BARON MUNCHARRIS

In the service of the Russian military Baron Muncharris 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.

On a Game of Tug o' War

A student performs his analysis.

Recall that the Baron's game involves moving a coin on a track of 12 numbered spaces.

0	1	2	3	4	5	5	4	3	2	1	0

The leftmost 6 squares were held by the Baron and the rightmost by Sir R-----. The Baron began by casting a die and placing a coin on his square numbered with its value. If he rolled a 6 Sir R----- cast the die and placed it on his so numbered square. If Sir R----- also rolled a 6 the game was begun anew.

The game progress with the player upon whose side of the track the coin lay rolling the dice and, if it exceeded the value on the square, moving the coin one square towards his 0 numbered, or home, square or one square away otherwise.

The winner was to be the first player to move the coin to his own home square, with the Baron having 3 coins from Sir R----- if he won and Sir R----- having 9 and one quarter coins from the Baron otherwise.

The key to a hasty reckoning of the fairness of this wager is firstly in recognising that the game is symmetric; Sir R-----'s chance of winning the game if the coin lies upon any given square is equal to the Baron's if it were on the other player's equally numbered square. In consequence the Baron's chance of winning if the coin lies upon one of Sir R-----'s squares must be equal to one minus the chance of his winning if the coin lies upon on the identically numbered square of his.

Having so noted, we can express the probability of the Baron winning the game from a start upon his 5 numbered square, p_5, in terms of its own self and the probability of his winning from a start upon his 4 numbered square, p_4.

$$\begin{aligned}
p_5 &= \frac{1}{6}p_4 + \frac{5}{6}(1-p_5)\\
6p_5 &= p_4 + 5 - 5p_5\\
p_5 &= \frac{1}{11}p_4 + \frac{5}{11}
\end{aligned}$$

We can thus recursively construct the probabilities of his winning from any of his squares.

$$\begin{aligned}
p_4 &= \frac{1}{3}p_3 + \frac{2}{3}p_5 = \frac{11}{31}p_3 + \frac{10}{31}\\
p_3 &= \frac{1}{2}p_2 + \frac{1}{2}p_4 = \frac{31}{51}p_2 + \frac{10}{51}\\
p_2 &= \frac{2}{3}p_1 + \frac{1}{3}p_3 = \frac{51}{61}p_1 + \frac{5}{61}\\
p_1 &= \frac{5}{6}p_0 + \frac{1}{6}p_2 = \frac{62}{63}
\end{aligned}$$

I described these observations to the Baron when he told me of his game, although I am not entirely sure that he gave them his full attention.

Substituting p_1 into the equation for p_2, p_2 into the equation for p_3, and so on and so forth, yields

$$\begin{aligned}
p_2 &= \frac{51}{61}\times\frac{62}{63}+\frac{5}{61} = \frac{3162+315}{3843} = \frac{3477}{3843} = \frac{57}{63}\\
p_3 &= \frac{31}{51}\times\frac{19}{21}+\frac{10}{51} = \frac{589+210}{1071} = \frac{799}{1071} = \frac{47}{63}\\
p_4 &= \frac{11}{31}\times\frac{47}{63}+\frac{10}{31} = \frac{517+630}{1953} = \frac{1147}{1953} = \frac{37}{63}\\
p_5 &= \frac{1}{11}\times\frac{37}{63}+\frac{5}{11} = \frac{37+315}{693} = \frac{352}{693} = \frac{32}{63}
\end{aligned}$$

Finally, we may exploit the symmetry of the game once more to figure the probability of the Baron winning before the starting square has been picked by noting that if the Baron rolls a 6 then sir R-----'s chance of winning must be equal to the Baron's chance at the outset of the game.

$$\begin{aligned}
p &= \frac{1}{6}\times\frac{62}{63}+\frac{1}{6}\times\frac{57}{63}+\frac{1}{6}\times\frac{47}{63}+\frac{1}{6}\times\frac{37}{63}+\frac{1}{6}\times\frac{32}{63}+\frac{1}{6}\times(1-p)\\
7p &= \frac{62+57+47+37+32+63}{63} = \frac{298}{63}\\
p &= \frac{298}{441}
\end{aligned}$$

Sir R-----'s expected winnings are consequently

$$9\frac{1}{4}\times(1-p) = \frac{37}{4}\times\frac{143}{441} = \frac{5291}{1764} = 2\frac{1763}{1764}$$

and I should not therefore have advised him to enter into the Baron's wager. ■

Errata

Owing to an attack by the typo gremlins you may have found the student's analysis in the last issue somewhat confusing.

Having worked out the area of a triangle in terms of trigonometric functions of θ and α, the analysis of the second game should have proceeded as follows:

For a given θ, the average area of the triangles is thusly given by

$$\frac{1}{\pi}\times\left[\int_0^\theta(\sin\theta\cos\alpha-\sin\theta\cos\theta)d\alpha+\int_\theta^\pi-(\sin\theta\cos\alpha-\sin\theta\cos\theta)d\alpha\right]$$

$$= \frac{1}{\pi}\times[\sin\theta\sin\alpha-\alpha\sin\theta\cos\theta]_0^\theta - \frac{1}{\pi}\times[\sin\theta\sin\alpha-\alpha\sin\theta\cos\theta]_\theta^\pi$$

$$= \frac{\sin^2\theta-\theta\sin\theta\cos\theta}{\pi} - \frac{-\sin^2\theta-(\pi-\theta)\sin\theta\cos\theta}{\pi}$$

$$= 2\frac{\sin^2\theta-\theta\sin\theta\cos\theta}{\pi}+\sin\theta\cos\theta$$

To calculate the average area of any triangle we must perform a similar exercise upon this result.

$$\frac{2}{\pi^2}\int_0^\pi\sin^2\theta d\theta - \frac{2}{\pi^2}\int_0^\pi\theta\sin\theta\cos\theta d\theta + \frac{1}{\pi}\int_0^\pi\sin\theta\cos\theta d\theta$$

We have called in the gremlin catchers and hopefully the problem will be resolved shortly.

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

Further Experiments in String Switching

Matthew Wilson finds adding requirements can be agreeably easy.

In July 2010's *CVu* [1] I wrote about my experiments in simplifying the business of converting well-known command-line arguments strings into well-known integral constants. The example given from that article looked like Listing 1.

I observed two drawbacks with the implementation:

- It only worked with multibyte strings (i.e. character type = `char`)
- It only worked with C-style strings (i.e. non-null pointers to nul-terminated character arrays).

Shortly, I'll discuss how I was able to address the first of these shortcomings with relative ease; the second is not yet addressed. But before that, I want to discuss a new requirement that just cropped up, whose solution prompted me to address the string encoding restriction, and to write this follow-up article.

New requirements

In its original guise (Listing 1), `stlsoft::string_switch()` operated by assigning to `*result.` In my continuing work involving lots of command-line development, I've come across a need to have the string-switching OR one or more bit flags to an already-assigned flag variable, as in the following code, in which flags is manipulated by the detection of command-line flags (e.g. `--show-same`) as well as, via `string_switch()`, the value of the option `--compare-as` (see Listing 2).

But how to achieve this dual role? Initially, only two unpleasant alternatives sprang to mind:

Listing 1

```
language_t  language  = SSSALC_LANG_NEUTRAL;
char const* val;
if(clasp::check_option(args, "--language",
   &val, NULL))
{
  string_t  langStr;
  string_t  verStr;
  . . .
  stlsoft::split(val, ',', langStr, verStr);
  stlsoft::split(verStr, '.',
     verHiStr, verLoStr);
  . . .
  if(!stlsoft::string_switch(
      langStr.c_str()
    , &language
    , stlsoft::string_cases(
        "C",    SSSALC_LANG_C
      , "C++",  SSSALC_LANG_CPLUSPLUS
      , "C#",   SSSALC_LANG_CSHARP
      , "D",    SSSALC_LANG_D
      , "Java", SSSALC_LANG_JAVA
      )
    )
  )
  {
    ff::fmtln(std::cerr,
       "Invalid language specified: {0}", val);
    return EXIT_FAILURE;
  }
```

- Add another method, called `string_switch_or()` or `string_switch_combine()`, or some such, with the required behaviour. In the implementation shown in Listing 3 the line `*result = case_.value;` would be replaced with `*result |= case_.value;`. Although not hugely onerous, repeating almost the whole function body and whacking on a clumsy name does not appeal to my aesthetic.
- Change `string_switch()` to the new semantics, and require users to remember to set the result variable to 0 before calling. This has the strong potential to break code, and is a total non-starter.

Taking the least bad of the two options, it looked like I'd have to grin and bear the use of the cut-and-paste `string_switch_combine()` option.

Listing 2

```
enum
{
  VDD_F_SHOW_SAME                = 0x00000001,
  VDD_F_SHOW_DIFFERENT           = 0x00000002,
  VDD_F_SHOW_LEFT_ONLY           = 0x00000004,
  VDD_F_SHOW_RIGHT_ONLY          = 0x00000008,
  . . .
  VDD_F_COMPARE_AS_BINARY        = 0x00010000,
  VDD_F_COMPARE_AS_TEXT          = 0x00020000,
  VDD_F_COMPARE_AS_NOWS          = 0x00030000,
  VDD_F_COMPARE_AS_NOCOMMENT     = 0x00040000,
  VDD_F_COMPARE_AS_LEX           = 0x00050000,
  VDD_F_COMPARE_AS_TEXT_PERCENT = 0x00060000,
  . . .
  VDD_F_COMPARE_AS_MASK          = 0x000f0000,
  . . .
};
static int tool_main(
  clasp::arguments_t const* args
)
{
  int flags = 0;
  char const* compareAs = NULL;
  clasp::check_flag(args, "--show-same", &flags,
     VDD_F_SHOW_SAME);
  clasp::check_flag(args, "--show-different",
     &flags, VDD_F_SHOW_DIFFERENT);
  clasp::check_flag(args, "--show-left-only",
     &flags, VDD_F_SHOW_LEFT_ONLY);
  clasp::check_flag(args, "--show-right-only",
     &flags, VDD_F_SHOW_RIGHT_ONLY);
  . . .
  if(clasp::check_option(args, "--compare-as",
     &compareAs, "binary"))
  {
    . . // contingent report, return EXIT_FAILURE
  }
```

MATTHEW WILSON

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

Listing 2 (cont'd)

```
if(!stlsoft::string_switch(
    compareAs
  , &flags
  , stlsoft::string_cases(
      "binary",    VDD_F_COMPARE_AS_BINARY
    , "text",      VDD_F_COMPARE_AS_TEXT
    , "nows",      VDD_F_COMPARE_AS_NOWS
    , "nocomment", VDD_F_COMPARE_AS_NOCOMMENT
    , "lex",       VDD_F_COMPARE_AS_LEX
    , "percent",
       VDD_F_COMPARE_AS_TEXT_PERCENT
    )
  , flags
  )
)
{...// contingent report, return EXIT_FAILURE
}
. . .
```

Just as I was about to paste, the distaste caused a brainwave that, characteristically, seems risibly simple when explained. The new implementation upgrades the existing `string_switch()` function with the new functionality, while remaining backwards-compatible with the old, just by the addition of a default argument, which defaults to 0, and combining it with the result, as shown in Listing 4. All the existing unit-tests continue to be satisfied, and the new requirements too.

One problem remains. The template parameter R designates the type of `*result` as well as defining the type of the string cases. But when used in bitwise OR guise, this will not suffice, because the type of `*result` is `int` and, as in the motivating example above, the flag values are enumerators. In this case, the fact that C/C++ enumerators are implicitly convertible to `int` is of no use to the compiler, which gets all confused when being passed ints and enumerators in the same call. The answer to this is just as simple as the rest of this solution, and does rely on the implicit conversion: simply define an additional template parameter to act as the enumerators, as shown in Listing 5.

Widestring compatibility

I'd imagined this to be much harder than it turned out to be: abstracting the worker types and functions on character type paying dividends. All the changes are contained within the `string_switch()  function` (see Listing 6). Simply, the multibyte-specific `strcmp()` call is replaced by use of `std::char_traits<C>`'s `length()` and `compare()` methods. As a trivial bonus, the implementation is probably mildly better performing than the original in cases where the strings have a mix of lengths. ■

Listing 3

```
// in namespace stlsoft
template<
  typename C
, typename R
, size_t   N
>
inline bool string_switch(
  C const* s
, R*       result
, ximpl::string_case_item_array_t<C, R, N>
   const& cases
)
{
  { for(size_t i = 0; i != cases.size(); ++i)
  {
    ximpl::string_case_item_t<C, R>
       const& case_ = cases[i];
    if(0 == ::strcmp(case_.name, s))
    {
      *result = case_.value;
      return true;
    }
  }}
  return false;
}
```

Listing 4

```
// in namespace stlsoft
template<
  typename C
, typename R
, size_t   N
>
inline bool string_switch(
  C const* s
, R*       result
, ximpl::string_case_item_array_t<C, R, N>
   const& cases
, R        resultBase = R()
)
{
  { for(size_t i = 0; i != cases.size(); ++i)
  {
    ximpl::string_case_item_t<C, R> const&
       case_ = cases[i];
    if(0 == ::strcmp(case_.name, s))
    {
      *result = resultBase | case_.value;
      return true;
    }
  }}
  return false;
}
```

Listings

Listing 1 – Original implementation of `string_switch()`.

Listing 2 – Bitwise-OR enhanced version of `string_switch()`.

Listing 3 – Heterogeneous argument bitwise-OR enhanced version of `string_switch()`.

Listing 4 – Character-encoding agnostic version of `string_switch()`.

Listing 5

```
// in namespace stlsoft
template<
  typename C
, typename R
, size_t   N
, typename V
>
inline bool string_switch(
  C const* s
, R*       result
, ximpl::string_case_item_array_t<C, V, N>
   const& cases
, R        resultBase = R()
)
{
  { for(size_t i = 0; i != cases.size(); ++i)
  {
    ximpl::string_case_item_t<C, V>
       const& case_ = cases[i];
    if(0 == ::strcmp(case_.name, s))
    {
      *result = resultBase | case_.value;
      return true;
    }
  }}
  return false;
}
```

Using the Windows Debugging API

Roger Orr reveals the magic of Windows Debuggers.

A significant amount of most programmers' time is spent in debugging. Wikipedia defines this activity as: 'a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected.'

There are many different (overlapping) ways to debug; but one of the commonest is to use an interactive debugger. There are a large number of different kinds of interactive debuggers, such as hardware probes for testing new hardware or emulators for embedded software components, but most programmers probably think of debugging an application running on a desktop operating system. However even on a desktop operating system you may have:

- operating system level debuggers (so called 'kernel debuggers') which usually require a secondary machine to host the debugger and provide access to the entirety of the machine, device drivers, operating system facilities and application programs.
- general application debuggers, such as Visual Studio in 'Native' debugging mode on Windows or gdb on Linux
- virtual machine debuggers, such as Visual Studio in 'Managed' debugging mode for .NET programs or the Eclipse Java debugger for JVM programs.

What does a debugger do? Well, a compiler compiles programs so you might naively expect that a debugger debugs them. Sadly this is not usually the case; an interactive debugger is a tool used by programmers to find errors in their program.

I personally dislike the term 'debugger': in my experience the best tools for *automatically* debugging are static code analysis tools; but it's probably too late to change the name now.

So what does a debugger actually provide? In general, most interactive debuggers provide the ability to:

- stop a program when errors occur
- inspect the state of the program
- set breakpoints
- step through the program
- provide symbolic names and source code for objects in the program
- modify the state of the program's memory

Given this list, perhaps a better description of what they do is 'interactive tracing and visualisation'.

Many programmers are familiar with what they do, but few people seem to know how they work.

This article focuses on the Windows application debugging API that is used by the general application debuggers in the list above. (Kernel debuggers and virtual machine debuggers use very different mechanisms to perform their task.) There is a huge amount of work in a good interactive debugger and it would take several articles to describe all the features that need implementing. I am restricting this article to describing the basic debugger API, and will work through a simple example of how to use this API to trace the key events of a program's execution.

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

Further Experiments in String Switching (continued)

Reference

[1] Wilson, Matthew (2010) 'Experiments in String Switching', *CVu* 22.3, July 2010.

Figure 6

```
// in namespace stlsoft
template<
  typename C
, typename R
, size_t   N
, typename V
>
inline bool string_switch(
  C const* s
, R*       result
, ximpl::string_case_item_array_t<C, V, N>
   const& cases
, R        resultBase = R()
)
```

Figure 6 (cont'd)

```
{
  typedef std::char_traits<C> char_traits_t;
  size_t const len = char_traits_t::length(s);
  { for(size_t i = 0; i != cases.size(); ++i)
  {
    ximpl::string_case_item_t<C, V> const&
       case_ = cases[i];
    size_t const caselen =
       char_traits_t::length(case_.name);
    if( caselen == len &&
       0 == char_traits_t::compare(case_.name,
          s, len))
    {
      *result = resultBase | case_.value;
      return true;
    }
  }}
  return false;
}
```

(I hope to produce a subsequent article doing something similar with the Unix debug API which will provide a contrasting mechanism for achieving a similar end on a different operating system.)

What's in the Win32 debugging API?

The two main methods in the Win32 debug interface are **WaitForDebugEvent** and **ContinueDebugEvent**. These provide the mechanism for the debugger to be notified of debug events from the process being debugged and to resume the target process once the event has been processed.

These methods are activated when a child process is created with one of the special flags: **DEBUG_ONLY_THIS_PROCESS** or **DEBUG_PROCESS**. This sets up a communication channel between the child process (known as the 'debuggee' or 'target process') and the parent ('debugger') process.

The debugger process then receives notification events from the debuggee on:

- process start and exit
- thread start and exit
- DLL load and unload
- OutputDebugString calls
- each occurrence of an exception

In each case the event provides some additional data with the event to give the debugger (just) enough information to be able to make sense of it. For example when a DLL is unloaded the event contains the base address of the unloaded DLL.

Note that several different things are included in the category of 'exception' – such as access violations, software generated errors and break points (used in a full-scale debugger to add support for stepping through a program).

What is missing?

The Windows debug API is solely concerned with the debug events and does not of itself provide any other access to the target process. (Note that this is very different from the ptrace interface used in most of the Unix world.)

All other access to the target process is achieved using general purpose functions that are not restricted to a debugger; given suitable permissions *any* process can use these functions. So for example the debugger can use the functions **ReadProcessMemory** and **WriteProcessMemory** to read or write memory in the target process and **GetThreadContext** to read the process registers for a thread.

This has the advantage that much of the functionality of a debugger does not require the specific debugger-debuggee relationship between the processes and so a variety of tools can be written to provide specific functionality, such as visualising a program's data structures or giving a stack dump of all the active threads. The Microsoft debugger WinDbg (included in 'Debugging Tools for Windows' [1]) has a 'non-invasive' attach mode that demonstrates just how much debugging you can do *without* the debug API.

Interpreting symbolic information

The main method used by Microsoft for attaching symbolic information to executable files is the PDB ('program database') format. I wrote an earlier article (for *Overload* 67 [2]) giving an introduction to the Microsoft symbol engine and I've used the code from that article (slightly expanded) to help provide symbolic information in this example program. I'm not going to go into further details of the symbol engine implementation, apart from a note about the **getString** method (under 'DLL load and unload' below).

In the example code below the field **eng** refers to this symbol engine and the methods from it used below include **loadModule** (loads the debug information for a module), **addressToString** (converts a target address into a symbolic string) and **stackTrace** (prints the call stack).

Processtracer

I am going to use a program that traces major events in a program's lifecycle to provide the framework for exploring the Windows Debug API. Here is an example of the program in use (paths edited for clarity):

```
C:> ProcessTracer BadProgram.exe
CREATE PROCESS 4092 at 0x00401398 mainCRTStartup
f:\dd\...\crtexe.c(404)
LOAD DLL 77230000 ntdll.dll
LOAD DLL 75BC0000 C:\Windows\system32\kernel32.dll
LOAD DLL 75430000
C:\Windows\system32\KERNELBASE.dll
LOAD DLL 6BD90000 C:\Windows\WinSxS\...\MSVCR80.dll
LOAD DLL 76E40000 C:\Windows\system32\msvcrt.dll
EXCEPTION 0xc0000005 at 0x0040108C Test::doit + 12
c:\article\badprogram.cpp(10) + 12 bytes
  Parameters: 0 0
  Frame       Code address
  0x0012FF34  0x0040108C Test::doit + 12
c:\article\badprogram.cpp(10) + 12 bytes
  0x0012FF44  0x00401045 main + 21
c:\article\badprogram.cpp(16) + 15 bytes
  0x0012FF88  0x004011F7 __tmainCRTStartup + 271
f:\dd\...\crtexe.c(597) + 23 bytes
  0x0012FF94  0x75C11194 BaseThreadInitThunk + 18
  0x0012FFD4  0x7728B3F5
RtlInitializeExceptionChain + 99
  0x0012FFEC  0x7728B3C8
RtlInitializeExceptionChain + 54
EXCEPTION 0xc0000005 at 0x0040108C Test::doit + 12
c:\article\badprogram.cpp(10) + 12 bytes (last
chance)
EXIT PROCESS 3221225477
  Frame       Code address
  0x0012FF34  0x0040108C Test::doit + 12
c:\article\badprogram.cpp(10) + 12 bytes
  0x0012FF44  0x00401045 main + 21
c:\article\badprogram.cpp(16) + 15 bytes
  0x0012FF88  0x004011F7 __tmainCRTStartup + 271
f:\dd\...\crtexe.c(597) + 23 bytes
  0x0012FF94  0x75C11194 BaseThreadInitThunk + 18
  0x0012FFD4  0x7728B3F5
RtlInitializeExceptionChain + 99
  0x0012FFEC  0x7728B3C8
RtlInitializeExceptionChain + 54
```

Getting started

The first step is to create the child process (`BadProgram.exe`) with the correct flags. This is the relevant call to **CreateProcess**:

```
CreateProcess(0,const_cast<char*>(
   cmdLine.c_str()), 0, 0, true,
   DEBUG_ONLY_THIS_PROCESS, 0, 0, &startupInfo,
   &ProcessInformation)
```

We don't need the process and thread handles returned by the **CreateProcess** call as the debug API will provide them, so we close these handles immediately.

One of the things I find confusing about the debug API is remembering which handles need to be closed manually and which ones are handled by the system – we will revisit this issue later on.

The debug API is designed to support debugging multiple processes; to achieve this you should pass the **DEBUG_PROCESS** flag to **CreateProcess**. I've not done that in this example program as implementing a debugger that correctly manages multiple child processes would make the code significantly more complex without really adding much new material.

The heart of the matter

The main driving loop of ProcessTracer is the 'debug loop', which looks like Listing 1.

The first call halts the debugger until the next event is ready from the debuggee and, on a successful return, the appropriate fields of the **DebugEvent** structure will be populated. When one of the various debug events occurs in the target, the operating system blocks *all* the threads in the process and passes the appropriate debugging event and associated data to the debugger. Execution of the target will not resume until the debugger signals that it has completed its handling of the event by calling **ContinueDebugEvent**.

ContinueDebugEvent needs the process and thread ID: in this example the process ID will always be the same (but the thread ID may vary if the target process creates additional threads). The function also takes a **continueFlag** argument. This is only relevant when the event is an exception and I'll cover the use of this argument when I look at handling exception events.

This synchronous call-based mechanism of passing events between the debuggee and the debugger makes it slightly tricky to write an interactive debugger since the debugger has to be responsive to user actions via the GUI and also wait for debug events from the target process. This normally means the debug loop runs in its own dedicated thread, decoupling it from the user interface. However in the ProcessTracer example there is no UI and so the implementation can be a simple single threaded application.

Process start and stop

The first event you receive is a **CREATE_PROCESS_DEBUG_EVENT**. The code in the debug loop in ProcessTracer is shown in Listing 2.

The implementation of **OnCreateProcess** is in Listing 3.

The **CreateProcessInfo** debug event data includes a handle to the process and to the main thread. Subsequent events will not provide these handles so it is important to retain them while the process is active: I keep the process handle in a simple field and the thread handle in a map indexed by thread ID. (To my mind this is a poor API design since it forces *each* user of the API to implement a mechanism to manage mapping process and thread IDs to handles.)

Also included in the create process event data is a handle to the file containing the executable program and the base address of the image. This can be passed to the symbol engine to populate the data for the main executable. Sadly, although the event data contains a field **lpImageName** that is documented as 'may contain the address of a string pointer in the address space of the process being debugged' the string is, as far as I can tell, *always* absent. So we have a handle to the file but do not know its name – I could write another article on the various mechanisms to get the file name from a file handle but for simplicity I've simply passed an empty string as the module name. (This is another place where the debug API appears to have been poorly implemented.)

Finally note that the debug API manages the *process* and *thread* handles and we must *not* attempt to close them, but that we *are* responsible for closing the *file* handle (if it was provided) – failing to do this can result in a long running debugger leaking file handles and keeping files locked. (As I said earlier, this confusion over the ownership of open handles complicates the job of writing a debugger; I can see no good reason for this asymmetric design in the API.)

In this simple case the file handle will be provided as we create the child process using the same credentials as the parent process; in more complex uses of the debug API involving processes with different credentials you may find that the debugger process has no permission to access the file and then no file handle is provided.

When the process ends the **EXIT_PROCESS_DEBUG_EVENT** is generated as the last debug event; the process handle is then closed by the debug API. In ProcessTracer we log the event, print a stack trace using the symbol engine and then set **completed** to **true** to terminate the debug loop.

Thread start and stop

The process start event implicitly includes a thread start event of the main application thread (and the process exit event implicitly includes a thread exit event for the last thread closed). On the creation of additional threads a separate event is raised, containing the start address and thread handle for the newly created thread.

Listing 1

```
while ( !completed )
{
  DEBUG_EVENT DebugEvent;
  if ( !WaitForDebugEvent(
      &DebugEvent, INFINITE) )
  {
    throw std::runtime_error(
       "Debug loop aborted");
  }
  DWORD continueFlag = DBG_CONTINUE;
  switch (DebugEvent.dwDebugEventCode)
  {
    ... // cases elided, for now
  default:
    std::cerr << "Unexpected debug event: " <<
      DebugEvent.dwDebugEventCode << std::endl;
  }
  if (
    !ContinueDebugEvent(DebugEvent.dwProcessId,
       DebugEvent.dwThreadId, continueFlag) )
  {
    throw std::runtime_error(
       "Error continuing debug event");
  }
}
```

Listing 2

```
switch (DebugEvent.dwDebugEventCode)
{
case CREATE_PROCESS_DEBUG_EVENT:
  OnCreateProcess(DebugEvent.dwProcessId,
     DebugEvent.dwThreadId,
     DebugEvent.u.CreateProcessInfo);
  break;
  ...
```

Listing 3

```
void ProcessTracer::OnCreateProcess(
  DWORD processId, DWORD threadId,
  CREATE_PROCESS_DEBUG_INFO const &
     createProcess)
{
  hProcess = createProcess.hProcess;
  threadHandles[threadId] =
     createProcess.hThread;
  eng.init(hProcess); // Initialise the
                      // symbol engine

  eng.loadModule(createProcess.hFile,
   createProcess.lpBaseOfImage, std::string());

  std::cout << "CREATE PROCESS " << processId <<
     " at " << eng.addressToString(
     createProcess.lpStartAddress) << std::endl;

  if (createProcess.hFile)
  {
    CloseHandle(createProcess.hFile);
  }
}
```

Our code simply logs the event and adds the thread handle to the map. Listing 4 is the method called from the debug loop.

When a thread exits the associated data includes the exit code for the thread; in process tracer we simply log this and print a stack trace using the symbol engine.

We also remove the thread handle from the map since the debug API closes the thread handle for us on the next call to **`ContinueDebugEvent`**.

DLL load and unload

As a DLL is loaded into the target process a debug event is generated containing, among other things, a file handle and a pointer to the file name of the DLL (in the target address space).

The file name is usually a fully qualified path, but the *first* DLL loaded (which is always `ntdll`) has a path-less file name simply consisting of `ntdll.dll`. If you need the full file name you can use the fact that all Win32 processes load `ntdll.dll` from the same location and so obtain the full file name by using **`GetModuleFileName`** on the debugger process.

Listing 5 is the code in ProcessTracer that handles the DLL load event.

Note that closing the file handle is, once again, our responsibility.

Similarly the unload DLL event is handled by logging the event and calling **`eng.unloadModule(unloadDll.lpBaseOfDll)`**.

How long is a NUL terminated string?

One slight twist I will mention in **`getString`** is that, since the string is NUL terminated, we don't know how long the string is until we have read it. The naive implementation of just trying to read **`MAX_PATH`** characters sometimes fails since the string being read is near a page boundary. It is a shame that the debug API doesn't report the string length too.

The error code reported by calling **`GetLastError`** on a failed call to **`ReadProcessMemory`** is **`ERROR_PARTIAL_COPY`** but in fact no partial copying has been done – the **`ReadProcessMemory`** function fails if we try to read data from the target process where only some of the pages of memory are accessible. Do not be misled by the last argument to this function: **`SIZE_T *lpNumberOfBytesRead`**. This value can *only* be either 0 on failure or the full buffer size on success!

My solution is to first try and read the entire **`MAX_PATH`** buffer from the target, as this is usually successful. Should this fail I reduce the number of bytes read to the next lowest page boundary.

OutputDebugString events

Windows provides the **`OutputDebugString`** function specifically to send a string to the debugger for display. A debug event is generated when the target process calls this function and the associated debug information provides the buffer address – and length – so reading the data from the target and displaying it is very easy.

Exceptions

These are probably the most interesting debug events and there is a lot of processing that can be done here. The debug API allows the debugger a first chance to look at the exception and, if neither the debugger nor the debuggee handles the exception, a last chance to look at the unhandled exception before terminating the process.

The detailed flow of control when an exception occurs is as follows.

1. An exception occurs in the target process
2. The debugger gets a debug event with the **`dwFirstChance`** flag set.
3. The debugger calls **`ContinueDebugEvent`** with **`dwContinueStatus`** set to:

   a) **`DBG_EXCEPTION_NOT_HANDLED`** or

   `b) DBG_CONTINUE`

The following stages will then be either

4. (a – **`DBG_EXCEPTION_NOT_HANDLED`**) The debuggee follows the usual search and dispatch logic of normal exception flow
5. If no handler is found the debugger gets a second debug event – but this time with the **`dwFirstChance`** flag set to zero
6. When the debugger calls **`ContinueDebugEvent`** the target process is terminated.

or

4. (b – **`DBG_CONTINUE`**) The target process resumes execution back at the exception context as if the exception had not occurred. The context may be the next instruction (for example after a breakpoint exception) or the same instruction (for example after an access violation); in the latter case if the underlying cause of the exception hasn't changed the exception will occur again and you end up back at (1).

Microsoft Visual Studio, for example, uses the first chance exception to pop up a dialog box with the options **Break**, **Continue** or **Ignore**. **Break** leaves the debugger active and defers the **`ContinueDebugEvent`** call until later; somewhat confusingly **Continue** corresponds to option 4a (**`DBG_EXCEPTION_NOT_HANDLED`**) and **Ignore** corresponds to option 4b (**`DBG_CONTINUE`**).

Just to make things interesting, when a process is being debugged, the program loader generates a breakpoint exception event when the process start up has completed (this exception should always be continued). I think this is a false economy in the API and it would have been better to designate a specific event for dealing with this case. In particular reuse of the breakpoint event makes it hard to process any exceptions that occur while loading a DLL during process start-up. In **`ProcessTrace`** we simply test and set a boolean variable **`attached`** to detect the first exception event. In all other cases we set the **`continueFlag`** to **`DBG_EXCEPTION_NOT_HANDLED`** to trace into the handling of the exception by the target process.

Our handling of an exception looks like Listing 6.

The event data contains the code for the exception and the address where the exception occurred. Some exceptions also include additional information describing the exception; for example

Listing 4

```
void ProcessTracer::OnCreateThread(
   DWORD threadId,
   CREATE_THREAD_DEBUG_INFO const & createThread)
{
  std::cout << "CREATE THREAD " << threadId <<
    " at " <<    eng.addressToString(
    createThread.lpStartAddress) << std::endl;
  threadHandles[threadId] = createThread.hThread;
}
```

Listing 5

```
void ProcessTracer::OnLoadDll(
   LOAD_DLL_DEBUG_INFO const & loadDll)
{
  void *pString = 0;
  ReadProcessMemory(hProcess,
    loadDll.lpImageName, &pString,
    sizeof(pString), 0);
  std::string const fileName(eng.getString(
    pString, loadDll.fUnicode, MAX_PATH) );
  eng.loadModule(loadDll.hFile,
     loadDll.lpBaseOfDll, fileName);
  std::cout << "LOAD DLL " <<
     loadDll.lpBaseOfDll << " " <<
     fileName << std::endl;
  if (loadDll.hFile)
  {
    CloseHandle(loadDll.hFile);
  }
}
```

EXCEPTION_ACCESS_VIOLATION indicates the access mode that failed (read, write, execute) in the first entry of the exception information and the address of the inaccessible data in the second entry.

Changes in the debugged process

When a process is running under the debug API several things are different. It is important to be aware of these as behaviour that is different under the debugger is hard to debug!

Firstly, by default, the Windows Heap manager runs in a 'debug' mode when the process is being debugged. This adds additional checking to memory allocations and writes check data before and after each allocation to allow detection of under- and over-runs. In current versions of Windows this default behaviour can be turned off by defining the environment variable **_NO_DEBUG_HEAP**.

ProcessTracer does this automatically using:

```
  _putenv("_NO_DEBUG_HEAP=1");
```

Secondly, the **CloseHandle** function throws an exception with exception code **STATUS_INVALID_HANDLE** (0xC0000008) when an invalid handle value is closed. The theory is that this ensures bad handle values are made visible when you are debugging the program; but this can mean the behaviour of an application changes under a debugger since exception unwinding can be invoked. One option is to set the **continueFlag** to **DBG_CONTINUE** for this exception code.

Thirdly, if **SetUnhandledExceptionFilter** is used to set the unhandled exception filter for a process this will be *ignored* if the process is being debugged. This is not usually a problem, but does make debugging an unhandled exception filter troublesome.

Fourthly, text sent to **OutputDebugString** goes to the application debugger rather than to the system debugger (or a tool like SysInternals Dbgview).

Finally the action of the debugger changes the runtime behaviour of the process as each debug event involves stopping all the threads in the process and several context switches back and forth to the debugger thread. This can make some sorts of race condition hard to debug since the action of debugging the process changes the timings of the interactions between the affected threads.

There are two API calls that can be used to check if a process is being debugged: **CheckRemoteDebuggerPresent** (which checks the presence of a debug connection from another process) and **IsBeingDebugged** (which reads the value of the **BeingDebugged** flag located at byte offset 2 in the Process Environment Block).

Attaching to existing processes

The debug API can also be used to attach to an already running process by using **DebugActiveProcess**.

Most of the debug events described above are exactly the same in this case; the biggest change is that when the process start and DLL load events are generated the file handle and file name are usually both zero. However the **GetModuleFileNameEx** function in the Windows 'Process Status' library (PSAPI) can be used to get the file name in this case (unfortunately this method only works when attaching to an existing process).

Listing 6

```
void ProcessTracer::OnException(DWORD threadId,
   DWORD firstChance,
   EXCEPTION_RECORD const & exception)
{
  std::cout << "EXCEPTION 0x" << std::hex <<
     exception.ExceptionCode << std::dec <<
     " at " << eng.addressToString(
     exception.ExceptionAddress);
  if (firstChance)
  {
    if (exception.NumberParameters)
    {
      std::cout << "\n  Parameters:";
      for (DWORD idx = 0;
         idx != exception.NumberParameters;
         ++idx)
      {
        std::cout << " " <<
           exception.ExceptionInformation[idx];
      }
    }
    std::cout << std::endl;
    eng.stackTrace(
       threadHandles[threadId], std::cout);
  }
  else
  {
    std::cout << " (last chance)" << std::endl;
  }
}
```

Additionally in order to debug a process with different credentials you may need to have the **SeDebugPrivilege** privilege and appropriate permissions – the details are outside the scope of this particular article.

Finally a debugger can detach from a debuggee by using **DebugActiveProcessStop**.

Writing a fully fledged interactive debugger

I haven't covered more than the basics of a debugger and there is obviously a lot more that must be added to write a proper interactive debugger. However I hope that the overview of the debug API that I have presented here has given you some understanding of the bare bones of the interaction between the debugger and the target. ■

Acknowledgements

Many thanks to Lee Benfield and Baris Acar for reviewing this article and providing numerous useful suggestions for improvement.

References and source code

[1] WinDbg is freely available from http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx
[2] *Overload 67* http://accu.org/index.php/journals/276Source code
[3] The full source code for this article can be found at http://www.howzatt.demon.co.uk/articles/ProcessTracer.zip

Write for us!

C Vu and Overload rely on articles from members. That's you! Without articles there are no journals. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

What is in a name?

Stephen Baynes examines just how important a name is.

Choosing the name of a function is important both to improve the reader's understanding of the code and also because it can affect how the code develops as it is modified over time.

Here is a little puzzler to make you think about the impact of your choices. There are no right or wrong answers, just different ones.

The example is simple; you have to write some code. It is for an algorithm which has to first find a starting point and then process the data from there. The starting point is the largest item in the data set. You decide that locating the starting point should be in a different function from the main algorithm. So the big question is what do you call that function? Do you work top down and name it after what it is for, which is locating the starting point? Or do you name it bottom up and name it for what it does, which is finding the largest? Here are three possible solutions:

1: Top down – call it by what it is for

```
dataptr find_main_start_point(){
    ...code to locate largest in dataset...
}
void main_algorithm(){
    dataptr start_point = find_main_start_point();
    ...rest of algorithm...
}
```

2: Bottom up – call it by what it does

```
dataptr find_largest(){
    ...code to locate largest in dataset...
}
void main_algorithm(){
    dataptr start_point = find_largest();
    ...rest of algorithm...
}
```

3: Do both in a multilayered way

```
dataptr find_largest(){
    ...code to locate largest in dataset...
}
dataptr find_main_start_point(){
    return find_largest();
}
void main_algorithm(){
    dataptr start_point = find_main_start_point();
    ...rest of algorithm...
}
```

I don't know which of those you would have chosen, all are correct and there is not much at this stage to choose between them. Things really become interesting when the code is changed. Assume that the code is part of a big application, with several programmers working on it who may make calls to functions you have written, so that you have to try and avoid changes that might affect the others working on the program.

Now think what you would do if you had to change the algorithm to use the smallest data item, rather than the largest, as the starting point. If you had chosen option 1 – you just need to change the contents of the function **`find_main_start_point`**. But with option 2 and 3 you also have to change the name from **`find_largest`** as well as the content (but you can only do this if it is not used elsewhere) or introduce a new function with the name **`find_smallest`**.

Returning to the original code, think what you would do if the change had been to add elsewhere some code that also needed to locate the largest data item. With options 2 and 3 you can just call **`find_largest`**. But with option 1 you would have to write a new function to do the same work as **`find_main_start_point`** or rename it using a new naming policy.

Then think what would happen if you did both these changes, one after the other, both in the order given and in the reverse. Don't forget to try and avoid changes that might affect others working on the same program. For example renaming or changing the behaviour of a function that could have been called elsewhere.

You may have ended up with multi-layered solutions similar to 3 in all cases, or you may have ended up with several functions doing the same job, but with different names.

It is possible to draw some weak conclusions. Bottom up naming as (after what a function does) as in option 2 gives more reuse. Top down naming (after what a function is used for) as in option 3 tends to reduce coupling making the code easier to change but at the risk of repeated code. The multi layered solution (a function named for what it is used for calling a function named for what does) as in option 3 gives the best of both worlds but with additional overhead. Like many things, every situation is different, and what is best one time, may not be the best another. But what you chose may have consequences in the future and a little time spent now may save some problems later. ■

STEPHEN BAYNES

At various times, Stephen has handled requirements, been architect, developer, tester, trainer and project manager. He is now senior developer at Smoothwall Ltd and can be contacted at sbaynes@gmx.com

The Kanban Ones Games

Jon Jagger describes a game revealing team behaviour.

I invented a simple collaborative team-game to simulate some of the aspects of software development recently. I made the mistake of telling Steve Love about it at the accu Xmas get together. Of course, being a dedicated editor, Steve immediately asked if I would write a short description of the game for CVu. So here it is.

You need some paper, some pens, and some dice. Quite a lot of dice in fact. But the dice are used merely to introduce some element of randomness into the game. So if you wanted to program the dice in a small app that would be just as good.

You start by rolling two dice, and adding up the total. So suppose you roll a 3 and a 4 then that's a 7. You then write 7 on a piece of paper. That piece of paper represents a single work-item and it goes onto the backlog. You can throw two dice to create a new backlog work-item whenever you like.

There are four tables (A,B,C,D) and each table represents a single column in a kanban style workflow. There are two developers per table. Each developer has 6 dice which they throw to simulate doing work. In overview what happens is this; the work-items come off the backlog and are worked on by the developers stationed at table A. If the work-item is a 7 they need to do 7 units of work to complete it. When they complete the work-item they pass it on to table B. The developers on table B then need to do another 7 units of work to complete it, then they pass it on to table C, etc. When the developers of table D finish their 7 units of work then that work-item finally makes it to DONE.

The developers work on the work-items in iterations. There are 10 days (2 weeks) per iteration. Each day's work is simulated by the developers simply by throwing their dice. Only 1's thrown contribute to units of work. So, for example, if a developer throws their 6 dice and gets 1,5,1,6,2,3 then that 2 x 1's and so they've done 2 units of work which they can put towards a work-item, a 7 say, meaning it has 5 units of work left. At the end of an iteration any work-item in the DONE column contributes to the velocity for that iteration.

That's the basic game play but there are some rules:

- You can use up units of work across different work-items. So, for example, if table C has 3 work-items and throws 3 1's then they can reduce each work-item by 1, or reduce one work-item by 3, or any other combination.
- 1's thrown on a table can only be used on that table.
- You can use up 1 unit of work to split a work-item. So, for example, you can use up a 1 to split a 7 into a 2 and a 5. Of course, if 3 units of work have been done on the 7 when you decide to split it then the 3 has to be distributed to the 2 and 5 somehow. For example, you could put 2 on 5 work-item (meaning there is 3 left) and 1 on the 2 work-item (meaning there is 1 left). Or you could put 2 on the 2 work-item (to complete it) and put 1 on the 5 work-item (meaning there is 4 left). You will find it useful if you can visually represent the work left on a work-item in some way. One simple way is to use coins.
- Each developer starts each day with 6 dice. However, they can opt to lend some of their dice to a developer on another table. If they do this they must put aside, and not throw (for that day) an equal number of dice.
- At the end of each day, after completed work-items have been moved to the next table, you must check for bugs. To do this you throw two dice for each work-item. If the dice total 10 then 1 unit of work is added to the work-item. If the dice total 11 then the work-item goes back to the previous table and starts again (partial work is lost). If the dice total is 12 then the work-item goes all the way back to the backlog and starts again (partial work is lost).

Recap

Ten days per sprint. Each day

- add work-items to the backlog?
- help other tables by lending dice?
- roll your dice on your table.
- count your 1's
- use up your 1's (splitting if you want to)
- move completed work-items to the next table (or into DONE)
- check for bugs for each work-item still on a table.

At the end of the iteration

- record your velocity
- do a retrospective

There are numerous possible variations you can add:

- More or less tables.
- Physically separate the tables and watch how it reduces communication. This is quite amazing to watch.
- More or less dice per developer.
- When developers lend dice to another developer they don't get them back the next day.
- Record the number of days a work-item takes to get to done once it comes off the backlog. The easiest way to do this is to record the iteration and day it both leaves the backlog and then enters DONE.

This is a key measure.

- Scope an iteration to a fixed time period (e.g. 2 minutes) and let developers throw their dice as often as they like – and then watch how everyone concentrates almost exclusively simply on throwing their dice faster.
- Measure waste. For example, the number of unrolled dice each iteration (because of lending) or the number of 1's thrown but unusable (because there wasn't enough work on the table).

I've written a blog entry on the game [1].

It's fun and it's simple but is surprisingly difficult to play well and stimulates a lot of interesting discussions and behaviour!

References

[1] http://jonjagger.blogspot.com/2010/12/kanban-1s-game.html

JON JAGGER

Jon is a UK-based software consultant. His passion is helping people improve their effectiveness in this collaborative game we call software development.

Inspirational (P)articles

Dr Love introduces Chris Oldwood.

Many thanks to Chris Oldwood for sharing remembrances of his student days, inspiration on how he managed to observe what was slowing him down (the wrong tool), a better, quicker way (the right tool – in this case awk) and how to find/read the manual. Further use (and abuse) of awk can be found here: http://stackoverflow.com/questions/407518/code-golf-leibniz-formula-for-pi

So, I'm sitting there slouched in my chair staring at the Visual Studio text editor whilst it does a rather lengthy, but albeit simple transformation of a text file and I'm coming to the realisation that this is no longer just a one-off event. In fact as I think harder I realise that during the last few weeks I've had to generate a number of long simple SQL scripts by transforming the CSV format result set from SSMS[1] back into a bunch of INSERT statements. It's also suddenly become noticeable that the tool I have historically been using for these simple tasks is no longer the svelte and nimble text editor that was VC6 (nay VS98), but has instead morphed into the bloated and sluggish VC9 (VS2008). I've read many complaints in the past about its increasing slothfulness but brushed them aside as they didn't really resonate with me; but when a tool asks you if it can disable its own 'Undo' feature to improve performance you know you're in trouble…

I haven't been paid by-the-hour since before the millennium so just sitting and waiting for it to finish was no longer an option. And then I started to hear the sniggers in the background as legions of smug Unix programmers chortle to themselves and flick through their tool chest whispering 'Vi?', 'Emacs?', 'Sed?', 'Awk?'… And then I heard a chorus of disapproval as every other programmer joined in with a round of 'Use the right tool for the job – stupid'. The problem was that I haven't used any of those tools since leaving university nearly 20 years ago and my previous experience of Unix ports under DOS and 16-bit Windows was not favourable…

Still, I put my metaphorical spade down (actually I left it working away in the background as I thought I might as well make use of my multi-tasking OS) and go in search of a proper Windows port of Sed and/or Awk. I mean one that respects Windows conventions like having spaces in filenames, using backslashes in paths and understanding that it has to process wildcards passed on the command line. I also didn't want to have to install and run any kind of emulator; as Raymond Chen once said 'If the solution starts with "First Install Product X" then now you have two problems.' Fortunately I think I've found a suitable candidate in the shape of the UnxUtils project on good old SourceForge[2]. Of course Visual Studio still finished before I had unpacked the .zip file, copied the files to a folder and updated my PATH, but that's not the point – I now had the tools ready for action.

I didn't have to wait too long either as a few weeks later I needed to work out how much disk space a certain type of file was consuming in our one of our data stores. This was an awkward one as I kind of knew how to do it in PowerShell (which I'm also learning and has far more relevance for sysadmin work on Windows), but I was keen to spend a short time with Awk as I remembered that it could 'do sums' as well as transform text. A quick Google later and I had a one liner that did exactly what I wanted (albeit after some gnashing of teeth because `END` has to be in upper case it seems).

Somewhat chuffed with myself at that point I decided that if I was going to make a real go of it then I should seek out some paper based tutorial so that I could harness (some) of the power of these two little gems. Mere seconds on Amazon unearthed *Sed & Awk* (2nd Edition) by Dougherty & Robbins with a second hand copy listed for just a handful of British Pounds. I also couldn't resist the companion *O'Reilly Pocket Reference* either for a couple of quid. So a few clicks later and I'm done. The Dougherty & Robbins book was the perfect introduction for me (ok, I'll stop there and take the opportunity to write up a formal review to keep Mr Higgins off my back…) and many happy memories of my time at university came flooding back as I remembered some of the things these tools do well.

A few months later I then had one of those glorious moments where I had a problem to solve and I knew exactly which would be the best tool for the job. I needed to extract some values from a file which was formatted somewhat like an XML data document with one element per line and a key value pair as an attribute. A simple GREP wouldn't do it as the key/value pair occurred in a number of different contexts – what I needed to do was restrict the GREP to certain repeating portions of a file. And before you could say 'Address Range' I had already typed the command line and was exhibiting a grin so large it would make the Cheshire Cat look miserable. It seems this old dog can still be taught new tricks, but also very old ones too…

References

[1] SQL Server Management Studio
[2] http://unxutils.sourceforge.net

ACCU Mentored Developers Project

Growing Object-Oriented Software, Guided by Tests

The ACCU Mentored Developers will soon be embarking on their next big project, a read-through of *Growing Object-Oriented Software Guided by Tests* by Nat pryce and Steve Freeman. This will differ slightly from the traditional project structure as there are no 'items' or exercises. However, most of the chapters are only 10 to 20 pages long and can therefore be read and discussed as if they are items. We're hoping to read at a pace of 2 chapters a week. Each chapter will be reviewed and summarised by an allocated project member and serve as a starting point for group discussion.

The project is open to all ACCU members. To take part, please sign up to the list at the link below and make yourself known:

http://lists.accu.org/mailman/listinfo/accu-mentored-growing

You can take part as a project member, who will be allocated at least one chapter to review, or an observer, who doesn't review a chapter but is free to take part in the discussion. If you have any questions about the project, please feel free to email me: paul.grenyer@gmail.com.

Desert Island Books

Nat Pryce makes his selection.

I first encountered Nat Pryce, together with Steve Freeman, at the ACCU London event, 'Sustainable TDD', held at JP Morgan (remember the lifts) in February 2010. I took two of my colleagues down from Norwich for the night to see what we could learn and it turned out to be quite a lot.

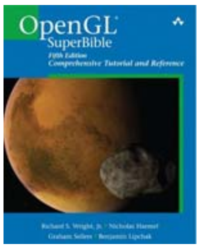
The second time, like so many others, was at the ACCU conference where, again with Steve, Nat spoke about 'TDD at the System Scale'. Again, I learnt a lot. The most recent time I met Nat was at the first Agile Cambridge event where he and Steve sat on a very informative discussion panel on agile. Steve and Nat will be giving a keynote at the conference this year and I'm quite disappointed I won't get to see it.

Nat and Steve do, of course, have a book! *Growing Object-Orientated Software, Guided by Tests* is about to be the subject of an ACCU Mentored Developers Project. This promises to be one of the best projects in terms of the scope of things to be learnt. Since its release I have heard nothing but good things about it, backed up by the strength of the material I have seen Steve and Nat present.

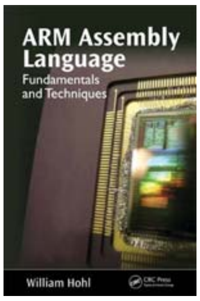
Nat Pryce

So here I am in my steamer cabin, frantically stuffing what possessions I can into my suitcase, while water sloshes over my ankles, rising towards my knees. Through the porthole I can see a desert island within swimming distance. That remote spot will be home for the foreseeable future, and I don't have much room in this suitcase. I'd better choose carefully. In go my laptop, solar charger and speakers. There's enough room for some books and CDs. But what to choose...

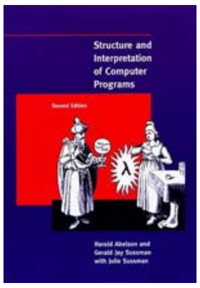
I get bored easily so if I'm going to be stuck on this island all alone, I'm going to need some intellectual stimulation. You can't beat computer programming for that, in my opinion, so first I'll grab some technical books. Since I'm going to be alone on this island, I don't need to pick anything practical. Pure (nerdy) indulgence is the order of the day!



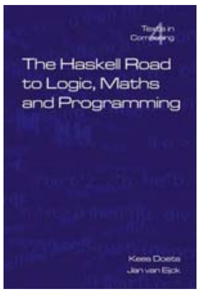
The first book I choose is the *OpenGL Superbible* by Richard S. Wright. Graphics and games attracted me to computer programming in the first place but I don't get to do much of that in my professional work. Months alone on a desert island will actually give me time to revisit the programming I really enjoy and let me wallow in nostalgia for my younger days, when I wrote games on my ZX Spectrum. And I'll be able to noodle about with the latest technologies, such as programmable graphics pipelines.



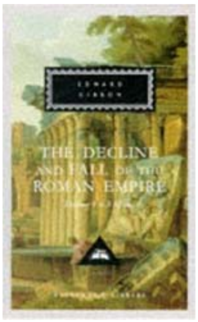
My second choice is *ARM Assembly* by William Hohl. Some of my favourite jobs have involved programming close to the machine in various dialects of C. The only assembly programming I've done has been for the Z80 microprocessor, the PDP11 and the Motorola 68000 (I'm showing my age!). If I'm going to be stuck in the middle of nowhere, I'd like to rekindle my interest in low-level fundamentals and learn some modern assembly language. I've picked ARM because ... well... I have the choice! Why would I choose to program the x86 for fun?



My third choice is *The Structure and Interpretation of Computer Programs* by Abelson, Sussman & Sussman. The idea that you should program by extending the language to fit your domain has been a big influence on the way I write code and this book is the seminal text on that style of programming. Every chapter ends with exercises, so it'll also give me plenty to do while stranded.



And finally I'll pack *The Haskell Road to Logic, Math and Programming* by Kees Doets and Jan van Eijck. After months alone on a desert island, having grown unkempt facial hair and lost the ability to converse with other people, I'm all set for career as a mathematician. This book will help close the gap by improving my mathematical thinking and proof techniques.



My choice of non-technical book is Gibbon's *The Decline and Fall of the Roman Empire* (just imagine my suitcase has one of those zipped expandable sections). I've always been interested in history, especially ancient and pre-history. Gibbon's huge work will provide ***plenty*** of reading and his writing style will make a pleasant change from the dry technical texts that fill the rest of my suitcase.

That leaves me just enough space to wedge in a couple of records. My first choice is *Is a Woman* by Lambchop: beautiful, sparse music that defies categorisation. The second record I'll take is *Bright Phoebus* by Lal and Mike Waterson. That record will remind me of England. It's a mix of folk, rock & jazz with strange pagan lyrics. The title track that closes the album is a wonderful, uplifting song.

Right, the water's up to my neck. Time to float off... Cheerio.

What's it all about?

Desert Island Disks is one of Radio 4's most popular and enduring programmes. The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml).

The format of 'Desert Island Books' is *slightly* different from the Radio 4 show. You choose about five books, one of which must be a novel, and up to two albums. Some people even throw in the odd film. Quite a few ACCUers have chosen their Desert Island Books to date and there are plenty more to go.

The rules aren't too strict but the programming books must have made a big impact on your programming life or be ones that you would take to a desert island. The inclusion of a novel and a couple of albums helps us to learn a little more about you. The ACCU has some amazing personalities and Desert Island Books has proved we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

Next issue: Chris O'Dell

ACCU Regional Meetings

This time the spotlight falls on London.

ACCU London

Chris Oldwood reviews a presention by Russel Winder at the November 2010 meeting

At the ACCU Conference back in April Russel gave one of the keynotes on the future of computer architectures in the face of the stagnation that has occurred with single processor speeds. The London branch of the ACCU welcomed him to one of the Skills Matter offices to talk further on the subject in November. There was an excellent turnout of around twenty-something people so latecomers such as myself had to stand at the back. There were once again a number of new faces – some of whom I later grilled in The Slaughtered Lamb only to discover that they had already become members.

The premise of Russel's talk was similar to that of his keynote, but whereas previously the bias seemed to be towards hardware, there was a larger software and programming language element to this presentation which made the trip equally worthwhile for us conference attendees. By the time I arrived Russel was in full swing describing the multi-core revolution and how threads are merely a distraction on the way to the true goal of independent computation units that consist of a straight pairing of CPU and memory. This was well and truly rammed home by the continual reuse of one presentation slide that showed two columns of CPU/RAM pairs connected by an 'interconnect'. It has been apparent for some time that the chasm between CPU and memory performance is widening rapidly and multi-core CPUs will only fan the flames. Russel contends that although the likes of Intel have a 48 core chip in the wings the memory contention this introduces means that its possible that 16 cores could well be the ceiling (in compute bound scenarios).

On the software side the picture painted was even less rosy as Russel pointed out that none of the major programming languages (i.e. Java, C++, Ruby, Python etc) have any natural support for the programming models of the future. Yes there are libraries designed to make the task less painful but you still have to get your feet wet to some degree. I've never quite been sure what the actual distinctions are between the Actor, CSP and DataFlow models and so I was pleased that Russel spent the time spelling this out. Of course the code we write in our high-level languages still needs to execute within some runtime environment that itself is likely hosted by an Operating System. And the picture looks no better here either as the key players are all monolithic architectures with an inherent limit that assumes all memory is globally addressable. One answer it seems may be in the form of Hypervisors and micro-kernels where each computation unit runs its own (possibly different) OS. Naturally Russel was quick to point out that none of this is new, it's just that most of us have managed to avoid it until now.

What made this presentation an improvement on the keynote was the acknowledgement of how all this affects those outside the world of Super Computing. Yes it all makes sense for the big number crunchers like meteorology and quantum physics, but how does this affect the man on the street whose PC spends the majority of its time waiting for user input? It probably won't, at least for a traditional PC set up, but a move to a Thin Client model might provide the kind of catalyst whereby small chunks of processing would need to be farmed out, e.g. spell checking paragraphs in parallel. Interesting times lie ahead, that's for sure.

Steve Love reviews a presentation by Chis Oldwood on xUnit Style Database Testing at the January 2011 meeting

The January installment of the ACCU London meeting found us in a new venue (St Alban's Centre Main Hall), with a new speaker. Chris Oldwood has been talking about this subject (informally, and on his blog: http://chrisoldwood.blogspot.com/) for some little while now, and I've been looking forward to seeing him present it. His premise is that unit testing with tools such as NUnit, jUnit and their like is popular in many modern programming circles, but SQL databases do not seem to be one of them. Never one to just accept that it's 'just the way it is', Chris has set about developing such a practice within his own team, and this presentation explored the approach taken, some of the obstacles encountered, how they were overcome, and some of the compromises which had to be made.

The talk began with a little history and revision of how the xUnit-style of testing is done in Object Oriented languages (C# was the language of the examples). The next step was to show how the same basic principles could be applied to a database environment (SQL Server being the database used, but the principles are easily applied to other environments too). The obvious candidates for this style of testing are stored procedures, but Chris expanded on the idea to explain how default constraints, triggers, and referential integrity tools (e.g. Foreign Keys) could also benefit from being 'unit' tested.

The basic approach was a library of tools which included such facilities as 'assert' in various forms, and some utility functions, which could then be composed to write unit tests in the SQL environment, and capture results easily. The structure of an individual test, as demonstrated by Chris, was recognisably similar to an NUnit test for a C# program, characterised by three sections: Arrange (set up the data/environment for the test), Action (do something interesting) and Assert (check it worked).

Chris then went on to explore more precisely *what* should be tested. He identified that the modern unit testing best-practice is to test the publicly observable behaviour of a unit, and went on to describe what the public interface of a database system should be. This included stored procedures and views, but excluded constraints, triggers and actual table schemata, the latter being characterised as implementation detail.

Lastly, Chris discussed with us the tools he's used, and some of the obstacles he has encountered in implementing this technique in his team. Perhaps not surprisingly, many of the objections he received from within the team were not different from those experienced with respect to unit testing other aspects of software development.

One important (perhaps defining) point of the entire talk was the implication of using SQL to write the tests, instead of writing some Data Access Layer in C# (or Java or whatever) to exercise the logic. The fact that the tests are in the database language itself means that those developers *maintaining* the database who might not be familiar with OO languages, can understand and therefore more easily maintain the *tests*. This has obvious consequences for longevity and long-term regression testing – one side effect of 'regular' unit testing – whereby the tests form a safety net for ongoing changes in the system.

The talk was very popular – a great turnout! – and provoked many interesting questions, some of which had to be saved until we had adjourned to a local public house (The Cittie of Yorke) for some well-earned refereshment.

Code Critique Competition 68

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.

Last issue's code

Can anybody help me to cast wide characters to an stl string? I can handle single characters successfully like this:

```
std::string str;
wchar_t wch = L'X';
str += (char)wch;
```

but I can't seem to get the syntax right for doing a whole array of them. Here's a program (Listing 1) showing what I've tried.

Listing 1

```
#include <string.h>
#include <string>
#include <iostream>
std::string castToString(wchar_t * wideStr)
{
  std::string str;
// This is what I want, but it won't compile:
//  str = (std::string)wideStr;
// This compiles, but I just get "H":
  str = (char*)wideStr;
// This compiles too, with the same output:
  std::wstring wstr;
  wstr = wideStr;
  str = (std::string)(char*)wstr.c_str();
// This is nearly there I think:
// but I now just get "H e l l o"
  str = std::string((char*)wstr.c_str(),
        wstr.size());
  delete wideStr;
  return str;
}
int main()
{
  wchar_t * source = new wchar_t[12];
  memcpy(source, L"Hello world", 24);
  std::string str = castToString(source);
  std::cout << str << std::endl;
}
```

Critiques

Peter Sommerlad <peter.sommerlad@hsr.ch>

There is an easy answer to the question and another question in return: 1. No, I cannot help, because you can't do that. 2. Why do you want to cast a wide character string to a `std::string` anyway?

Nevertheless, I want to put my reply in a historic context (as far as I remember it).

When C was invented 7bit ASCII character set ruled. When I learned C in Germany often a C program looked on the screen and printed like this (only guessing from memory):

```
£include <stdio.h>
int main ä
 printf("HalloÖn");
ü
```

very interesting, because the Umlauts used the code points of curly and angle brackets and backlash and vertical bar. With IBM DOS a great relief to European programmers came to allow for 8bit ASCII using the code points above 128 as the space to represent many of the European characters, like the Umlauts. However, for most people in the world these about 200 different characters are far from sufficient to represent their language. I do not want to go into the details of the woes of wrong code pages under DOS etc.

With the standardization of C in the later 80s and also C++ the concept of a character type supporting more then 8 bits was successfully introduced, but the standard didn't say, if the 'wider' character type would be 16 or 32 bits. Also during that time Unicode and some of its representations were defined in parallel. They weren't ready then and thus the C standard couldn't refer to it. Today Unicode is a norm and its UTF-8 8bit representation is IMHO a suitable means to represent Unicode within programs and externally, since it is the most compact form. However, for some applications a UTF-32 fixed width representation of characters also can be appropriate. That seems what you try to achieve with your usage of `wchar_t` (right?). Nevertheless I am not an expert in Unicode usage and others have problems with it also, e.g., I had the experience that regular expression libraries trying to support UTF-8 failed to do so correctly in interesting ways.

Now back to the problem. As stated above `sizeof(wchar_t)` is different from `sizeof(char)`, on my system shows:

```
sizeof(wchar_t) = 4
sizeof(char) = 1
```

On other systems `sizeof(wchar_t)` used to be 2 (e.g. early Windows versions), because that was sufficient to represent the relevant number of different characters.

Using `wchar_t` interchangeably with `char` is calling for trouble. For the simple ASCII-character set characters, this seems to work, since the numerical value of `L'X'` is the same as for `'X'`, so truncating it with a cast is not a problem. However, whenever you use a character that requires more than 8 bits to represent, your code will fail to work as you expected it.

```
wchar_t wch = L'←'; // left arrow
  // UTF8 0xE2 86 90, Unicode: 2190 (hex)
str += (char) wch;
std::cout << "wch = "<< int(wch) << std::endl;
std::cout << "(char) wch =" << char(wch) <<
  " as int: "<< int(char(wch)) << std::endl;
```

will deliver the following output (platform dependent):

```
wch = 8592
(char) wch =ê as int: -112
```

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

You see, your code that seemed to work, doesn't. The value of 8592 gets truncated (% 256) and because the highest bit is set becomes negative value when converted back to **`int`**.

Now back to your original problem and my question number 2: I suppose you want to represent string literals that use a character set beyond ASCII that doesn't fit a single byte or char variable. It depends on what you want to do with these strings on how to handle that. First option is to use **`std::wstring`** that provides the same API as **`std::string`** but uses wide characters (**`wchar_t`**) instead of **`char`** as **`std::string`** does. **`std::wstring`** and **`std::string`** are typedefs for **`std::basic_string<>`** template class that only differ in the element type (**`char`** or **`wchar_t`** respectively).

To convert between them you need to use a locale that suits your needs. All details of using locales can be found in *Standard IOStream and Locales* by Angelika Langer (a speaker at ACCU conferences) and Klaus Kreft. On p. 279 we find a solution to your problem by using the function (templates) **`narrow()`** or **`widen()`** to convert between **`wchar_t`** and **`char`** respectively. But understand, narrowing a wide character might result in loss of information, if its character cannot be represented in the 'narrow' character set. A more suitable option might be the **`codecvt`** template class. Your vendor might provide an implementation that can convert between UTF-8 and UTF-32 for example. Because of the variable-width representation of unicode code points in UTF-8 such a **`codecvt`** object might need state to represent partially converted characters. Langer/Kreft's section 6.1.3 contains an example of how to do that. I have to confess, I only learned that while writing this feedback.

Now back to your code snippet, line by line:

```
wchar_t * source = new wchar_t[12];
```

Why do you allocate that wide character array on the heap. There is no need to do so. In addition, even if you do so, I consider it very bad practice to pass the pointer to a function that in turn deletes the memory and leaves the original pointer intact. That calls out for unintended memory overwrites using the pointer referring memory already freed. In addition the parameter name of source suggest a read-only access that it definitely isn't with the delete. If you insist on allocating the memory on the heap and delete it in **`castToString`** then you should at least pass the pointer variable by reference. Another problem is, that you use the array version of **`operator new[]`** but the plain version of **`operator delete`** in the function. If these are overloaded and non standard you can be in trouble again.

```
memcpy(source, L"Hello world", 24);
```

As you learned above, **`memcpy`** call assumes **`sizeof(wchar_t)`** is 2, which it isn't on my machine. Thus it is inherently non-portable code. I would refrain from using **`memcpy`** anyway in C++ and use the corresponding (w)string class or **`std::vector`** instead for all 'stringish' data. Assignment or initialization is as effective as memcpy for simple or trivial member types.

If you remove the **`delete wideStr;`** from **`castToString`** a better means than these two lines with the same effect would be

```
wchar_t const * const source = L"Hello world";
```

Now for an alternative implementation of **`castToString`** that can somehow achieve what you are looking for with using locale and the corresponding **`ctype<wchar_t>`** facet:

```
#include <string>
#include <iostream>
#include <locale>
#include <iterator>
#include <algorithm>
#include <boost/bind.hpp>
std::string narrowWideString(
  std::wstring const &in){
  using namespace std;
  using namespace boost;
  string result;
  locale loc; // copy of the global locale
  ctype<wchar_t> const &facet =
    use_facet<ctype<wchar_t> >(loc);
  // the following selects the right overload
  // of the narrow member function of the
  // ctype<> facet
  char(std::ctype<wchar_t>::*narrow)
    (wchar_t,char)const =
    &std::ctype<wchar_t>::narrow;
  transform(in.begin(),in.end(),
    back_inserter(result),
    bind(narrow,cref(facet),_1,' '));
  return result;
}
int main() {
  wchar_t const source[] =L"Hello world";
  std::string str = narrowWideString(source);
  std::cout << str << std::endl;
}
```

In **`main`** using an array type representing the wide string literal is better than just a pointer. I pass the wide character array as a **`wstring`** parameter to the function, so I can use **`begin()`** and **`end()`** to call the transform algorithm. The decision to use transform makes the code a bit tricky, since selecting the right version of an overloaded member function requires us to 'cast' it by assigning it to a member-function pointer (narrow) of the right member function signature type. A typedef for **`ctype<wchar_t>`** might have made it a bit more readable. the blank in the bind call provides the default char value to use, whenever the narrowing wouldn't produce a valid character fitting in the string.

However, I do not recommend you to blindly narrow your **`wchar_t`** strings, so. Select a good internal and external representation, e.g. UTF-8 and use a library so convert that representation into another format, if you really need it. I made the experience (in the server domain) that just passing UTF-8 strings through a program works fine with **`std::string`**. But there the rendering of non-ASCII characters was left to a web browser, which was fine, when it was told the data comes in UTF-8.

I am out of time and steam. There remains more to be said, such as not using C-style casts or 'naked' pointers in C++, knowing what you internally to your data when casting (even with C++-style casts) and not accessing internal data of objects, when you do not need to (e.g. calling **`c_str()`** or **`data()`** member functions of string).

Huw Lewis <huw.lewis2409@googlemail.com>

Before tackling the primary subject matter (of wide character strings), we need to cover the other mistakes in this code example.

Dynamic memory allocation strategy

The **`castToString`** helper function attempts to delete the memory of the wide string data passed to it. This is a poor design choice as it places a constraint on the caller to always pass in some raw, unmanaged dynamically allocated memory that will not be deleted by any other means. For example, the caller could not pass in a **`wstring(L"blah").c_str()`** result as it would be doubly deleted leading to a crash.

Operators new/delete for arrays

The memory was allocated with the array variant of **`operator new`**. It must be deleted using the array variant of the delete operator, otherwise corruption of the free store will ensue and we walk towards crash territory once again (but much more unpredictably).

```
delete [] wideStr;
```

Size of wchar_t

The program allocates an array of 12 **`wchar_t`** elements to hold the string **`L"Hello world"`**. However, the **`memcpy`** call used to write to the array is given 24 as the length parameter. This parameter is the length in bytes to be copied, and so shows the author's assumption that **`wchar_t`** is 2 bytes in length. However, in my environment **`wchar_t`** is actually a 4 byte field so only half of the intended buffer is copied. The size of **`wchar_t`** is compiler dependent so instead use **sizeof(wchar_t)** or **<climits>** (**std::MB_LEN_MAX**) [Ed: warning: MB_LEN_MAX is **not** the same as sizeof(wchar_t)]. Note also that the wide character **strlen** counterpart is named **wcslen**.

```
const wchar_t* temp = L"Hello world";
wchar_t* wideSource = new wchar_t[64];
memcpy(wideSource, temp,
   wcslen(temp) * MB_LEN_MAX);
```

Converting the wide string

All of the attempts to convert the wide-char string have the same underlying fault. They are trying to convert an array of multi-byte characters directly to an array of single characters. Therefore the 'unused' bytes of the multi-byte characters are also copied into the single character string. The value of these bytes is zero and so they prematurely terminate the single byte string. This is why the result of all attempts so far is simply **"H"**1.

Note 1: I could not reproduce the reported output of **"H e l l o"** with my test platform. The result remained the same (**"H"**).

Fortunately, the C++ standard provides an easy way to convert a sequence of any type to a string as long as there is a sensible conversion of the type to char, in the form of this template constructor:

```
template<InputIterator>
string(InputIterator begin,
   InputIterator end);
```

This constructor populates the new string by converting the value of each item in the range [begin, end) to a char in the new sequence. In this example the wide character 'H' (0x00000048) is assigned to the single byte character 'H' (0x48), and so on.

Our example is simple with all characters in the ascii range. All characters convert to single byte characters without data loss. However, what if our wide string contains characters outside of the basic ascii range (> 127) e.g. '£' (0x00A3)? The simple solution using the template constructor would have the effect of truncating the character to 0xA3 which isn’t a printable character in my console’s utf-8 encoding. In ‘normal’ single byte strings, the '£' character is actually a multi-byte character – 0xC2A3. Confusing, I know.

A safer conversion

Anyway, what to do about characters like this in our input data? Here are some options in ascending order of complexity:

1. Ignore and simply put up with the unprintable characters in the output
2. Detect the character and reject the conversion i.e. throw an exception.
3. Detect the character and attempt a conversion to a multi-byte utf-8 character.

The choice of how to handle this situation is down to the application i.e. is it a throw away test harness or a production enterprise application.

Option 1 is trivial as shown here:

```
std::string castToStringOption1(
  const wchar_t* wStr, size_t len)
{
  // Use the template constructor taking
  // an iterator range.
  std::string output(wStr, wStr + len);
  return output;
}
```

Option 2 is also an easy implementation. Simply loop through each character checking its range. Throw the exception of your choice if any are out of range. Following the successful check, use the template constructor to create the new string.

```
class StringConverterOption2
{
public:
  class CharConversionError :
   public std::logic_error
  {
  public:
    CharConversionError(
      const std::string& msg)
    : std::logic_error(msg)
    {
    }
  };
  static std::string convert(const wchar_t* w,
    size_t len, bool check = true)
  {
    // check conversion
    if (check)
    {
      std::for_each(w, w + len,
        check_char_conversion);
    }
    // the check has passed, it is safe to use
    // the string template constructor
    return string(w, w + len);
  }
private:
  // the checking function. This throws if the
  // input character is outside of the basic
  // ascii range
  static void check_char_conversion(
    wchar_t wideC)
  {
    if (wideC < 0 || wideC > 127)
    {
      throw CharConversionError(
        "Wide character out of ascii range");
    }
  }
};
```

Option 3 would be ideal, but is not straight forward. First we’ll assume that we’re converting from an input encoding such as unicode to a single byte encoding such as utf-8. Then we’d need a function to do this conversion for us. Sometimes the result of this would be 2 (or more) characters out for one in (see '£' above) so the template constructor wouldn’t work. The best approach would be to reserve the maximum amount of memory required then build the output string one character at a time. Here is a simple example implementation limited to single and double utf-8 characters.

```
// An encoder functor to convert to utf-8
class ConvertWideCharToUtf8
{
public:
  // Convert the given wide char to a sequence
  // of utf-8 encoded characters.
  // in - the input wide char
  // begin - the beginning of the output
  //   sequence
  // end - the end of the output sequence
  // return - the next char pointer in the
  //   output buffer after this encoded
  //   character
  char* operator()(wchar_t in, char* begin,
    char* end)
  {
    if (!begin)
      throw std::runtime_error(
        "ConvertWideCharToUtf8 - "
        "invalid begin pointer");
    if (!end)
      throw std::runtime_error(
        "ConvertWideCharToUtf8 - "
        "invalid end pointer");
    if (begin >= end)
      throw std::runtime_error(
        "ConvertWideCharToUtf8 - "
        "invalid iterator range");
    if (in <= 127)
    {
      // simple straight assignment
      *begin = static_cast<char>(in);
    }
    else if (in <= 0x07FF)
    {
      if (std::distance(begin, end) < 2)
      {
        throw std::runtime_error(
          "ConvertWideCharToUtf8 - "
          "output buffer too small");
      }
      // 2 byte field
      *begin = static_cast<char>(0xC0);
      *begin |= static_cast<char>(in >> 6);
      ++begin;
      // the lsb
      *begin = static_cast<char>(0x80);
      *begin |= static_cast<char>(in & 0x3F);
    }
    else
    {
      // TBD for larger characters
      throw std::runtime_error(
        "utf-8 encoding > 2 not supported");
    }
    // return the pointer to the next char in
    // the output buffer
    return ++begin;
  }
};
template<class Encoder =
  ConvertWideCharToUtf8>
class StringConverterOption3
{
public:
  static std::string convert(
    const wchar_t* wStr, size_t len)
  {
    // reserve enough space for twice the
    // whole of the wide string so as to
    // prevent repeated allocations
    std::string output;
    output.reserve(len * sizeof(wchar_t) * 2);
    // Use a simple buffer to accept the
    // multi-char output
    char conversionOutput[sizeof(wchar_t)*2];
    // use an instance of the encoder's
    // operator() to do the conversion work
    Encoder encoder;
    for ( size_t i = 0; i < len; ++i )
    {
      // append the encoded characters
      output.append(conversionOutput,
           encoder(wStr[i], conversionOutput,
              conversionOutput + sizeof(
              conversionOutput)));
    }
    return output;
  }
};
```

Finally here is the new main function. I've used **wstring** to wrap up all of the memory allocation and size issues described earlier.

```
int main(int argc, char** argv)
{
  const wstring wideSource(
    L"Hello, world! ££");
  cout << "option 1: " <<
   castToStringOption1(wideSource.c_str(),
     wideSource.size()) << endl;
  // option 2 will throw if it can't convert
  // the string.
  try
  {
    cout << "option 2: " <<
     castToStringOption2(wideSource.c_str(),
       wideSource.size()) << endl;
  }
  catch (StringConverterOption2::
     CharConversionError& e)
  {
    cout << "option 2 converter found a"
      " character out of ascii range" << endl;
    cout << e.what() << endl;
  }
  cout << "option 3: " <<
   castToStringOption3(wideSource.c_str(),
     wideSource.size()) << endl;
  return 0;
}
```

The program output is given below. So, there's more to complexity to strings and character encodings than we thought!!

```
option 1: Hello, world! úú
option 2 converter found a character out of ascii
range
Wide character out of ascii range
option 3: Hello, world! ££
```

Commentary

This critique had a double focus: one issue is the casting and the other is the problem of changing character representation. I think the responses received do a good job of covering the second point, but I'd like to say a little more about the first.

C++ is a strongly-typed language, but unlike some such it does allow the programmer to change the type of an expression. Some of these casts are implicit (i.e. require no extra syntax) and others are explicit (i.e. require a cast operator). Unfortunately the compiler cannot, in general, tell whether the resultant coercion is valid.

It is doubly unfortunate that, because of the array to pointer conversion rules, a valid cast of a single value may become invalid when applied to an array.

So let's define a couple of classes:

```
struct base {
  int first;
  base() : first(1) {}
  virtual ~base() {}
};
struct derived : public base {
  int second;
  derived() : second(2) {}
};
```

Now we use these classes with (implicit) conversion:

```
base *baseptr1 = new derived();
base *baseptr2 = new derived[10];
```

The implicit conversion is valid, syntactically, in both cases. However problems come when we try to access the elements of the array:

```
std::cout <<  baseptr2[1].first;
```

When I tried this I got 9540004 as the output – rather than the hoped for value of 2. This problem is caused by C++ treating indexing as pointer arithmetic using the **declared** type of the variable. So the difference between the addresses given by **baseptr2[1]** and **baseptr2[0]** is

Bookcase

The latest book review.

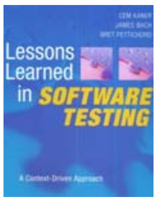
If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

Jez Higgins (jez@jezuk.co.uk)

Lessons Learned in Software Testing: A Context-Driven Approach

Cem Kaner, James Bach & Bret Pettichord, published by Wiley, ISBN: 978-0471081128

Reviewed by Alan Lenton


Lessons Learned in SOFTWARE TESTING
A Context-Driven Approach
CEM KANER
JAMES BACH
BRET PETTICHORD

I first heard about this book at a London Tester gathering which I sneaked into (I am after all a programmer, not a tester!). It's a fabulous collection of tips and hints and techniques for both the new and the experienced person working in a software test department. It covers obvious areas testing techniques, automated testing (the material about what automated testing can't do is very high grade material), documenting testing, and managing a test project.

But it also covers some less obvious issues such as thinking like a tester, bug advocacy, and how to interact with programmers. The style is to offer the advice in bite sized chunks, and, to my surprise, it works, making it easy to look up something only half remembered, in a moment.

Even more importantly, from my point of view, the book is easily useable if you aren't a professional tester. If you are a programmer, or even the CTO, in a small company that doesn't have a software testing department, you will still get a lot of new ideas out of the book. Many of the ideas are a nice fit with programmer test driven development – some of them will work for you, some won't. Happily, the book isn't dogmatic, it's much more of a 'this is what we have found can work in some of the projects we have been involved in' style. And it works very well indeed. Highly recommended

Code Critique Competition 68 (continued)

`sizeof(base)`, whereas the actual difference between elements of the array is `sizeof(derived)`.

In the critique we are using an explicit cast, which adds further complications, but it is primarily the addressing problem that means casting a single character 'works' but casting an array doesn't.

As a general rule, trying to avoid the so-called C-style casting and preferring `static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast` can catch at least some of the problematic cases. However, as the array case above demonstrates, even implicit conversions can be problematic so it is important to understand what happens when conversions occur (and what might go wrong!)

The Winner of CC 67

The two entries covered very similar ground. I liked the three options Huw provided (since it was not specified what the user *really* wanted), and he also provided just a little more explanation of *why* the original code was flawed. However I appreciated Peter's use of standard library facilities to do the conversion (whether narrow or codecvt) and the background and explanation about UTF-8 so on balance I decided to award him the prize.

Note that Peter won last time, and Huw won the two times previous to that. As Tom Lehreh puts it in his delightful *New Math* song (http://edit.mp3lyrics.org/t/tom-lehrer/new/) 'let's not always see the same hands'. I'm sure others of you could provide a critique! If you've read this far, why don't you enter the next competition?

Code Critique 68

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

What's the best way to read output from Fortran fixed-format strings using `scanf`? The example is a database file with lines such as

```
"  26  2996100   1"
```

which were written by a fixed width format (i.e. this ***should*** be interpreted as 'leading space, _26, __2, 996, 100, __1').

The problem is that automatic whitespace skipping in `scanf` means that any %d format string is almost guaranteed to get confused for one or other variant of full fields, e.g. the 'obvious' `" %3d%3d%3d%3d%3d"` format string reads 26 299 610 0 1 here.

Listing 2 is a simple program demonstrating the problem (and the asymmetry of C input/output formats!), and the results are:

```
C:\cc68>output | input
1 2 3 4 5
100 200 300 400 500
26 299 610 0 1
```

(Thanks to Robin Williams for suggesting this issue's critique.)

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

Listing 2

```
//-- output.c --
#include <stdio.h>
int process(int v1, int v2, int v3,
  int v4, int v5)
{
  printf(" %3i%3i%3i%3i%3i\n",
    v1, v2, v3, v4, v5);
}
int main()
{
  process(1, 2, 3, 4, 5);
  process(100, 200, 300, 400, 500);
  process(26, 2, 996, 100, 1);
  return 0;
}
//-- input.c --
#include <stdio.h>
int main()
{
  while (!feof(stdin))
  {
    int i1, i2, i3, i4, i5;
    scanf(" %3i%3i%3i%3i%3i\n",
      &i1, &i2, &i3, &i4, &i5);
    printf("%i %i %i %i %i\n",
      i1, i2, i3, i4, i5);
  }
}
```

View From The Chair

Hubert Matthews
chair@accu.org

You will be reading this shortly before the ACCU Conference, another blockbuster of an event that we're all looking forward to. On the last day of the conference, as is customary, is the ACCU Annual General Meeting. I want to give you all a heads-up view of the motions that the main ACCU committee is intending to propose.

There will be a motion to tidy up some legacy aspects of the constitution, primarily to make official the transition from the ACCU's origins as a technology-specific user group (C and C++) to a technology-independent organisation dedicated to software development in general. I don't expect these will be controversial in any way so I won't give more details here.

The next motion concerns the Conference Chair. The conference has grown over the years from a series of technical talks at the AGM to a full-blown conference that is now the public face of the ACCU. It requires lots of dedication and commitment from the conference team led by the Conference Chair. It therefore seems incongruent and intrinsically unsatisfying that the role of Conference Chair is neither recognised by nor accountable to the membership as a whole. The committee is therefore proposing that the Conference Chair become an elected position and a named committee member in the constitution. Because of the demands placed on the Conference Chair and the learning curve it involves, the committee is proposing that the post be for a period of two years unlike the other officers. If this motion is passed the first election for the role would be held in 2012 for the period 2012 to 2014.

The other motion regards the membership fees. Over the last two years the ACCU has been running at a loss, primarily because of the decision to move to monthly posting of magazines instead of bi-monthly. This decision has proved popular with the members but is costly. Full details of these costs and the accounts for 2009 and draft accounts for 2010 will be presented at the AGM. The committee is undertaking a number of actions to reduce costs whilst maintaining monthly magazines. However this still leaves us with a significant shortfall that can only be covered by increasing membership revenues. Therefore it will be proposed to raise the standard membership by £10/year from £35 to £45. The concessions rate, the standing order discount and corporate memberships would all be adjusted too. Without this increase the ACCU will have completely depleted its reserves within two years, reserves that have been built up over many years. Fees have been at the current levels for a number of years and they would have needed to be revised upward anyway to allow for inflation and increased suppliers' costs anyway.

Anyone wishing to propose any additional motions (or as an alternative to any of the committee motions) should inform the Secretary at least fourteen days before the AGM, as detailed in the constitution (Section 7.5, available online at http://accu.org/index.php/constitution).

The 23rd AGM

Notice is hereby given that the 23rd Annual General Meeting of The C Users' Group (UK) publicly known as ACCU will be held at 13:00 on Saturday 16th April 2011 at the Oxford Barceló Hotel, (formerly the Oxford Paramount Hotel), Godstow Road, Oxford OX2 8AL, United Kingdom.

Current agenda

1 Apologies for absence
2 Minutes of the 22nd Annual General Meeting
3 Annual reports of the officers
4 Accounts for the year ending 31st December 2010
5 Election of Auditor
6 Election of Officers and Committee
7 Other motions for which notice has been given.
8 Any other Annual General Meeting Business (To be notified to the Chair prior to the commencement of the Meeting).

The attention of attendees under a Corporate Membership is drawn to Rule 7.8 of the Constitution:

> ... Voting by Corporate bodies is limited to a maximum of four individuals from that body. The identities of Corporate voting and non-voting individuals must be made known to the Chair before commencing the business of the Meeting. All individuals present under a Corporate Membership have speaking rights.

Also, all members should note rules 7.5:

> Notices of Motion, duly proposed and seconded, must be lodged with the Secretary at least 14 days prior to the General Meeting.

and 7.6:

> Nominations for Officers and Committee members, duly proposed, seconded and accepted, shall be lodged with the Secretary at least 14 days prior to the General Meeting.

and 7.7:

> In addition to written nominations for a position, nominations may be taken from the floor at the General Meeting. In the event of there being more nominations than there are positions to fill, candidates shall be elected by simple majority of those Members present and voting. The presiding Member shall have a casting vote.

For historical and logistical reasons, the date and venue is that of the last day of the ACCU Spring Conference. Please note that you do not need to be attending the conference to attend the AGM. (For more information about the conference, please see the web page at http://accu.org/conference.)

More details, including any more motions, will be announced later. A full list of motions and electoral candidates will be supplied at the meeting itself. We currently expect constitutional motions regarding a widening of the aims of the society as stated in the constitution to cover software development as a whole rather than just C and C++, and for the creation of a new officer's post, to be elected every two years, to be responsible to the committee (and thus to the society as a whole) for the organisation of the conferences.

Please also note that both I, as Secretary, and Stewart Brodie, as Treasurer, wish to step down from our posts. We've both been doing this for far too long (in my case, for a whole decade, and Stewart isn't far behind). So if anyone considers that they could stand for either position, please let either me (as secretary@accu.org) or Hubert Matthews (chair@accu.org) know.

Alan Bellingham
Secretary, ACCU