The magazine of the ACCU

www.accu.org

{cvu}

Volume 23 • Issue 6 • January 2012 • £3

Features

How To Be Dispensable
Frances Buontempo

Coping With Complexity
Pete Goodliffe

An Introduction to CLASP
Matthew Wilson

Getting More Fiber in Your Diet
Robert Clipsham

Writing a Bazaar Plugin
Peter Hammond

Using the Windows Debugging API on Windows 64
Roger Orr

Regulars

Baron Muncharris

Code Critique

Desert Island Books

Inspirational (P)articles

{cvu}
Volume 23 Issue 6
January 2012
ISSN 1354-3164
www.accu.org

Features Editor
Steve Love
cvu@accu.org

Regulars Editor
Jez Higgins
jez@jezuk.co.uk

Contributors
Frances Buontempo, Robert Clipsham, Pete Goodliffe, Paul Grenyer, Peter Hammond, Richard Harris, Derek Jones, Roger Orr, 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
R G Pauer
treasurer@accu.org

Advertising
Seb Rose
ads@accu.org

Cover Art
Pete Goodliffe

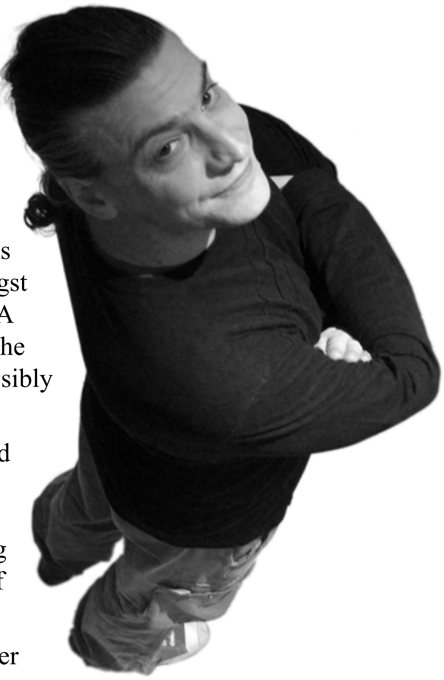
Repro/Print
Parchment (Oxford) Ltd

Distribution
Able Types (Oxford) Ltd

Design
Pete Goodliffe

accu

Resolution



As I look for the umpteenth time at the Raspberry Pi website I see I can still buy a keyboard sticker for it but as yet, no actual hardware. By the time you read this, it may well be available (and you may even have managed to get your hands on one!), promised as it is for early 2012.

For those who've not yet heard of it, the Raspberry Pi is a personal computer the size of a credit card. It plugs into a TV, has ports for network, USB, HDMI (amongst other things) and comes in two flavours – the Model A and Model B. See http://www.raspberrypi.org/faqs. The model names are – apparently – quite intentional! Possibly unsurprising with David Braben closely involved...

Personally I began with a Sinclair ZX81, and regarded those few of my friends who had a BBC Micro Computer with a strong sense of envy (Oooh! Proper keyboard!). So it was that affordable home computing grabbed my imaginationand launched me on a path of discovery and wonder. And yes, I'm still on it.

The Raspberry Pi's stated purpose is to make computer programming fun and cheap, and is being targeted at youngsters. The key to making that work will be the quality and accessibility of the APIs to the various bits of hardware. Given that it's essentially a Linux-based PC, it'll need something 'extra' to set it apart, notwithstanding the cost, of course.

I hope it becomes the platform that inspires a whole new generation of schoolkids to be the hackers and geeks of tomorrow, and to find fun and exciting things to do with it.

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 24.1: 1st February 2012
C Vu 24.2: 1st April 2012

Overload 108: 1st March 2012
Overload 109: 1st May 2012

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

Coping with Complexity

Pete Goodliffe helps us to pick our battles.

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.
~ Edsger Wybe Dijkstra

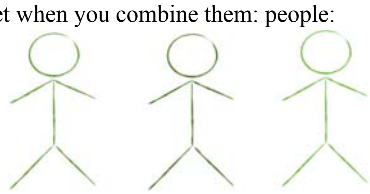
Code is complex. It's a battle that we all have to fight daily.

Of course, your code is great, isn't it? It's other people's code that is complex.

Well, no. Not always. Admit it. It's all too easy to write something complicated. It happens when you're not paying attention. It happens when you don't plan ahead sufficiently. It happens when you start working on a 'simple' problem, but soon you've discovered so many corner cases that your simple algorithm has grown to reflect a labyrinth, ready to entrap an unwary programmer.

My observation is that software complexity stems from three main sources. Blobs. And lines.

And what you get when you combine them: people:



In this article, we'll take a look at each of these and see what we can learn about writing better software.

Blobs

The first part of software complexity we should consider relates to blobs: the components we write. The size and number of those blobs determine complexity.

Some software complexity is a natural consequence of size; the larger a project becomes, the more blobs we need, the harder it is to comprehend, and the harder it is to work with. This is *necessary* complexity.

But there is plenty of unnecessary complexity that causes hassle. I've lost count of the times I have opened a C++ header file, and balked at thousands of lines in a single class declaration. How is a mere mortal supposed to be able to understand what such a beast does? This is surely *unnecessary* complexity.

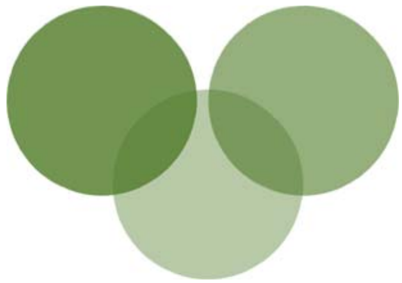
Sometimes these large monsters come auto-generated, from 'wizard' systems, for example in GUI construction. However, serious code hooligans can produce these code monsters without a second thought. (In fact, the lack of thought is often the cause of such abominations.)

So we need to manage our *necessary* complexity. And educate – or shoot – our unnecessary programmers.

It's important to realise that size itself is *not* the enemy. If you have a software system that has to do three things, then you need to code in there to do those three things. If you remove some of that code in order to reduce complexity, then you'll have different problems. (That's being *simplistic* rather than simple, and it's not a good thing.)

No, size itself is not the problem. We need a enough code to meet requirements. The problem is how we structure that code. It's how that size is distributed.

Imagine you start working on a *vast* system. And you discover the class structure of the beast is like this:



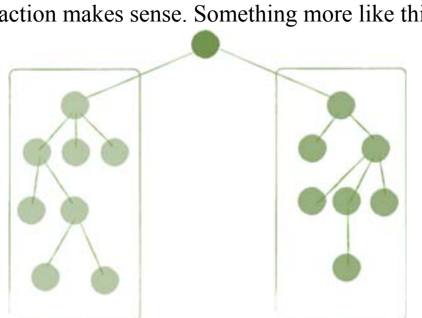
Three whole classes! Now: is that a complex system or not?

On one level, it doesn't seem complicated at all. There are only three parts! How could that be hard to understand? And the software design has the added benefit of looking like Mickey Mouse, so it must be good.

In fact, this appears to be a beautifully simple design. You could describe it to someone in seconds.

But, of course, each of those parts will be so large and dense, presumably with so much interconnection and spaghetti logic that they are likely to be practically impossible to work with. So this is almost certainly a *very* complex system, hidden behind a *simplistic* design.

Clearly, a better structure – one that is simpler to understand, and simpler to maintain – would consider those three sections as 'modules' and further sub-divide them into other parts: packages, components, classes, or whatever abstraction makes sense. Something more like this:



Immediately, this feels better. It looks like a lot of small (so understandable, and likely simpler) components connected into a larger whole. Our brains are suited to dividing problems into hierarchies like this and reasoning about the problems when thus abstracted.

The consequences of such a design are increased comprehension, and greater modifiability (you can work on a part of the system's functionality by identifying the smaller part that relates to it, rather than having to roll your sleeves up and dive into a single behemoth class).

Of course, the trick to making this work, the trick that enables a design like this to actually *be* simple rather than just *look* simple, is to ensure that each of the blobs has the correct *roles and responsibilities*. That is, a single

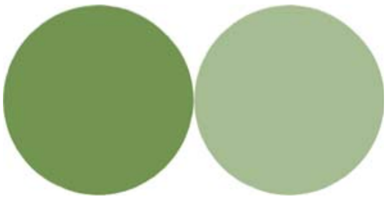
PETE GOODLIFFE

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

responsibility resides in a single part of the system rather than smeared across it.

A case study: Reducing blob complexity

One of my favourite recent reductions in software complexity was a section of code with two very large objects that we so inter-related they were practically one-and-the-same class.



I started chipping away at one of the objects, realising that it contained hundreds of unused ‘helper’ methods. I mercilessly removed them; an enjoyable experience not unlike deflating a helium balloon. And so for effect, I started speaking in an excitable high voice. This was code becoming simpler:

Now that I could see the remainder of the object, it was clear that the majority of its methods simply forwarded to the partner. So I removed those methods and made all calling code just use the object. There were just two remaining methods, one of which belonged on the partner anyway, and one which should have been a simple non-member function.

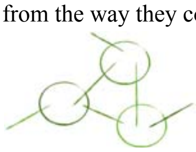
The result?

A far simpler class design, I think you’ll agree.

Of course, the next step was to decompose the remaining blob. But that’s another story. (And nowhere near as interesting.)

Lines

We’ve considered blobs: the components and objects that we create. To paraphrase John Donne: No code is an island. Complexity is not borne solely from the blobs, but from the way they connect:

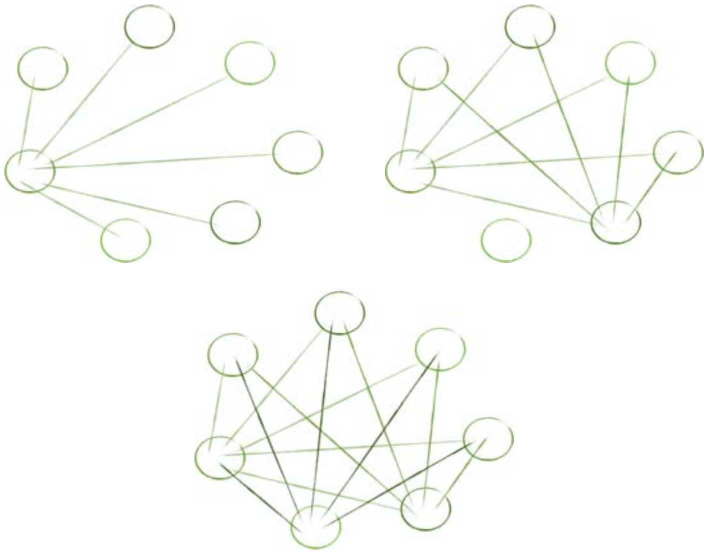


In general, software designs are simpler when there are fewer lines. The more connections between blobs (this is known as greater *coupling* if you’re talking proper grown-up talk), the more rigid a design is, and the more inter-operation you have to comprehend (and fight) as you work on a system.

At the most basic level, a system comprised of many objects, none of which are connected at all would appear the simplest. But it is not a single system at all. It’s a number of separate systems.

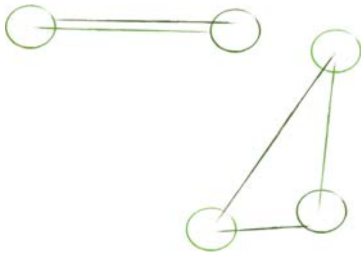
As we add connections, we create actual software systems. As we add more blobs and, crucially, lines between them, the more complex our systems become.

The structure of our software interconnections dramatically affects our ease of working with it. Consider the following structures, which are based on real examples I have been working on:



What’s your reaction to them? Which looks simpler? I’ll admit that working on the last one almost caused my head to explode.

When we map out connections, we see complexity often springs from cycles in our graph. These are generally complex relationships to consider. When objects are co-dependent their structure is rigid, not easy to change, and often very hard to work with. A change to one object usually requires a change to the other. The objects effectively become one entity; one that's harder to maintain.



These kinds of relationship can be simplified by breaking the links. Perhaps by introducing new abstract interfaces to reduce the coupling between objects:

This kind of structure enhanced composability, introduces flexibility and fosters testability (you can write testing versions of components behind those abstract interfaces). We can use well-named interfaces to make those relationships descriptive.

One of the nastiest systems I’ve had to work on in a long time looked like this:

It seems a superficially simple model: one parent object represents ‘the system’ and creates all of the child objects. However, each of those objects was given a back-reference to the parent, so they could access each other. This design effectively allowed every child-object to rely on (and become closely coupled with) every sibling, locking the entire system down into one rigid shape.

Michael Feathers described this to me as the known-anti pattern *distributed self*. I had another name for it, but it’s not polite enough to print.

And finally: People

So software complexity depends on the structure of our blobs and lines.

But it’s important to observe that blobs and lines don’t create themselves. Those structures are not intrinsically to blame. It is the *people writing the code* who are responsible (yes, that’s you, gentle reader). It is programmer

On a Game of Lucky Sevens

A student performs his analysis.

The Baron's most recent game consisted of a race to complete a trick of four sevens, with the Baron dealing cards from a pristine deck, running from Ace to King once in each suit, and Sir R----- dealing from a well shuffled deck. As soon as either player held such a trick the game concluded and a prize was taken, eleven coins for the Baron if he should have four sevens and nine for Sir R----- otherwise.

The key to reckoning the equity of the wager is to note that it is unchanged should the Baron and Sir R----- take turns dealing out the rest of their cards one by one after the prize has been taken.

If Sir R----- had lost the game then he would most assuredly deal a seven after the Baron and, since the Baron's last seven is seven cards from the end of the deck, there consequently should have been at least one seven in the last seven cards of Sir R------'s deck.

The fairness of the wager is therefore identical to one in which Sir R----- draws first and in which the first player to deal a seven from the bottom of the deck loses. Indeed, I made this observation to the Baron, but am not sure that he grasped its significance.

Now, we can say with full confidence that there are either one or more sevens in Sir R-----'s last seven cards or that there are none and that the chance of the former is therefore one minus the chance of the latter.

That probability is the product of the chance that each card drawn is any but one of the four sevens which, given that each draw reduces the number of cards in the deck by one, is given by

$$p=\frac{48}{52}\times\frac{47}{51}\times\frac{46}{50}\times\frac{45}{49}\times\frac{44}{48}\times\frac{43}{47}\times\frac{42}{46}$$

We can cancel out the 48, 47 and 46 that appear in the top and bottom of the fractions, but the calculation still appears somewhat daunting

$$p=\frac{1}{52}\times\frac{1}{51}\times\frac{1}{50}\times\frac{45}{49}\times 44\times 43\times 42$$

Fortunately, we can simplify our work still further if we factorise each term into a product of primes

$$\begin{aligned}p&=\frac{1}{2^2\times 13}\times\frac{1}{3\times 17}\times\frac{1}{2\times 5^2}\times\frac{3^2\times 5}{7^2}\times\left(2^2\times 11\right)\times 43\times\left(2\times 3\times 7\right)\\&=\frac{2^3\times 3^3\times 5\times 7\times 11\times 43}{2^3\times 3\times 5^2\times 7^2\times 13\times 17}\\&=\frac{3^2\times 11\times 43}{5\times 7\times 13\times 17}\\&=\frac{4257}{7735}\end{aligned}$$

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

$$\begin{aligned}p\times 9-(1-p)\times 11&=\frac{4257}{7735}\times 9-\left(1-\frac{4257}{7735}\right)\times 11\\&=\frac{4257}{7735}\times 20-11\\&=\frac{4257\times 4}{7735\div 5}-11\\&=\frac{17028}{1547}-11\\&=\frac{17028-1547\times 11}{1547}\\&=\frac{17028-17017}{1547}\\&=\frac{11}{1547}\end{aligned}$$

and, given that they are slightly biased in his favour, I should have happily suggested that he take up the Baron's wager. ■

Coping with Complexity (continued)

has the power to introduce incredible complexity, or to reduce a nasty problem down to a elegant and simple solution.

How often do people set out to write nasty, complex code? Despite your opinion about how your corrupt co-workers are planning to introduce more stress in your life with their machiavellian code, in general complexity is accidental, *rarely* something some adds wilfully.

It's often the product of history: programmers extend and extend and extend system, with no time allowed for refactoring, or the 'prototype to throw away' turns into a production system. By the time it's being used there no change to take it apart and start again.

Software complexity is caused by humans working in real-world situations. The only way we can reduce complexity is by taking charge of it, and trying to prevent work systems from forcing out code into unworkable structures.

Conclusion

In this little saunter through software complexity territory we've seen that complexity arises from blobs (our software components), lines (the connections between those components), but mostly from people (the muppets who construct these software disasters).

Oh, and of course, it comes from the Singleton design pattern. But no one uses that any more, do they? ■

Questions

1. Why is simplicity in code design better? Is there a difference between simplicity in design and in code implementation?
2. How do you strive for simplicity in your code? How do you know you've achieved it?
3. Do the nature of connections matter as much as the number of connections? What connections are 'better' than others?
4. If software complexity stems from social problems, how can we address it?
5. How can you tell the difference between necessary and unnecessary complexity?
6. If it's true that many programmers do know that their software designs should be simpler, how can we encourage them to craft simpler code?

Getting More Fiber In Your Diet

Robert Clipsham shows the benefits of fibers in D.

Contrary to what you may be thinking this is about, it isn't about food or dieting or anything of the sort. In this article I will introduce fibers[1], also known as coroutines, and how to start using them in D.

Doing two things at once

So, what are these mysterious fibers, and, for that matter, why would you want to use them? Like threads, fibers allow you to work on two tasks at once. Take, for example, the following pseudo-code:

```
function breathe()
{
  // We have to breathe to survive
}
function walk()
{
  // I need to get somewhere
}
```

We're going for a walk, so we'll need to run both of the above at once:

```
// Create a thread for both tasks so they can
// execute at once
breathingThread = new Thread(breathe);
walkingThread   = new Thread(walk);

// Start doing both of the tasks at the same time
breathingThread.start();
walkingThread.start();
```

This is simple enough, for each thing we want to do at the same time we define a task, create a thread for it, then run it. Of course, for those of you reasonably familiar with threading, you'll know that it is anything but simple in the real world – it's very easy to create bugs which are very difficult to track down. Let's adapt the functions from above to see this (see Listing 1).

```
variable amountOfBreathToTake;
function breathe()
{
  do
    breathe amountOfBreathToTake;
    amountOfBreathToTake = 0;
  while alive;
}
function walk()
{
  do
    for every second we walk
      increase amountOfBreathToTake;
  while alive;
}
```

Listing 1

At first glance this seems fair enough – as we walk we need to breathe more, once we've taken a breath we need don't need to breathe again until we have a reason to (for simplicity's sake, we only need to breathe while walking here, and we'll also be walking indefinitely). But there's a problem. We're breathing and walking at the same time. What if `walk()` tries to increase `amountOfBreathToTake` at the same time as `breathe()` sets it to zero? Does `amountOfBreathToTake` end up as zero? Then we won't take enough breath next time we walk. Does it end up as an increased version of what it was before? Then we'll breathe more air than our lungs can hold. For that matter, why can't `amountOfBreathToTake` be a random value which is a mixture of the current value, the increased value, and zero? They are both happening at once after all.

How do we avoid this?

The typical way to solve this problem is to introduce a lock. Each time we want to read or write to `amountOfBreathToTake`, we attempt to lock the lock. If the lock is already locked, we wait until it is unlocked. If it is not locked, we lock it and continue. When we are done reading or writing, we unlock the lock, and someone else is free to lock it. Of course, this has its own problems. What if we introduce a `run()` function, and forget about the lock? Then we have the same problem again. And how about all the time spent locking, unlocking and waiting, rather than actually walking and breathing?

So let's take a step back. We now have a way to breathe and walk without taking too much or too little breath. We're doing both of these things at the same time. But are we really? We've introduced a lock, so only one of them is actually happening at once! This is where fibers come in. We want to appear to be doing two things at once, but don't actually need to do them at the same time. So, let's modify the code (Listing 2).

There are three main differences to the original code here. First, we only call `walk()` now, no creating and starting threads. Second, `walk()` now creates a breathing fiber. Finally, two new method calls have been introduced, one in `breathe()` and one in `walk()`.

The first we'll look at is `breathingFiber.call();` in `walk()`. If we call `breathe()` directly, it would enter an infinite loop, where we breathed until we died. We wouldn't continue walking, and our multitasking would be non-existent. Of course, we could call `walk()` from within `breathe()`, but then every time we breathe we start walking!

```
variable amountOfBreathToTake;
function breathe()
{
  do
    breathe amountOfBreathToTake;
    amountOfBreathToTake = 0;
    Fiber.yield();
  while alive;
}
function walk()
{
  breathingFiber = new Fiber(breathe);
  do
    for every second we walk
      increase amountOfBreathToTake;
      breathingFiber.call();
  while alive;
}

walk();
```

Listing 2

ROBERT CLIPSHAM

Robert Clipsham began programming as teenager, and hasn't stopped since. He is now a student at the University of Glasgow, and enjoys science fiction, breaking things, and making things fast.

Listing 3

```
class Person
{
  struct Nutrients
  {
    int Fiber;
    int Calcium;
    int Iron;
  }
  int       mNumNutrients;
  Nutrients mNutrients;

  this(int numNutrients)
  {
    mNumNutrients = numNutrients;
  }

  bool satisfied() const nothrow @property
  {
    return mNutrients == Nutrients(mNumNutrients,
       mNumNutrients, mNumNutrients);
  }
}
```

By using **breathingFiber.call()**, we call the method in a fiber, so we will still be able to multitask.

The final method call, and the most important, is **Fiber.yield();** in the **breathe()** function. Calling **Fiber.yield()** causes the currently running fiber to yield to its caller. You can think of it like a return statement, but rather than returning from the function or method, you're returning to the previous state of the program. This does exactly what we want, as breathing doesn't force us to walk, it is still happening at the same time as walking, and we don't have to spend lots of time locking and unlocking.

Let's see some real code

Now for a slightly more involved example. This is done using the D programming language [2], you should be able to follow along regardless of whether you know it or not, providing you have some experience with object-oriented programming [3], and a C/Java style language [4].

In this example, I've filled a room full of malnourished people – somehow they've ended up with no iron, no calcium, and no fiber! Of course, this is going to have very serious health impacts if we don't act quickly. So much so, that if we help them one at a time they probably won't make it. So we'll have to feed them all at the same time. First let's define one of our people. (Listing 3)

The constructor should be called with the number of nutrients required before the person is healthy. **satisfied()** will return true when the **mNutrients** member is equal to a struct containing the required level of nutrients.

We now have to decide how we are going to feed them. We could use either threads or fibers. For the sake of having some pretty graphs and thus some more data to compare and contrast with later, I've implemented both (neither of which are very sophisticated). I'll show the Fiber implementation here, see the Links section for how to get the Threads implementation. In D, fibers can either be derived from the **Fiber** class [7], or composed, by calling **Fiber**'s constructor with the function or method you want it to execute. As we will have a fiber per person, we will use derivation so we can associate the fiber with its person. (Listing 4)

A **FeedFiber** will be created for each person, we will then use the **.call()** method of **Fiber**, which will call the **run()** method within that fiber. The code for this could easily be improved; you'll notice it's quite different from how we use fibers above. Rather than using **foreach** below, you could make each fiber yield to the next until they are all satisfied, however I wrote the threaded version first, and I'm lazy. You could adapt the code and see what difference it makes performance wise and how much nicer the code is. (Listing 5).

Here we create a fiber for each person, then loop over the fibers, calling each until the given fiber terminates – at which point the fiber's function

Some notes for people who don't know D

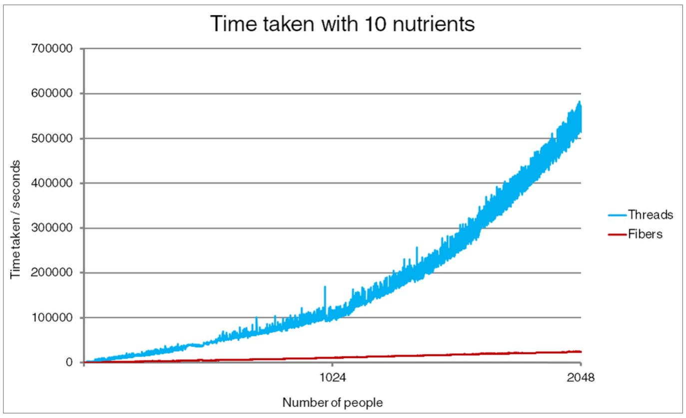
- D is a typed language, that is, you can't assign a variable a string, then decide you want to put an integer in it. You can infer types using **auto**, but if you are declaring them without assigning a value to them, you must specify the type. **int** is a 32 bit integer type, **bool** is boolean.
- A **struct** works the same way as in C, it's plain old data. For those of you coming from higher level languages, you can think of a struct as a class without inheritance, it's just a wrapper for some variables and methods.
- All variables are initialised by default to **Type.init**, see http://digitalmars.com/d/2.0/type.html [5] for a list.
- **const**, **pure**, **nothrow** and **@property** are attributes, you can find out more about them at http://digitalmars.com/d/2.0/attribute.html.[6] They can be ignored for the sake of this tutorial.
- You can initialise a **struct** using **StructName(firstValue, secondValue, etc)**.
- **this()** is the constructor function
- **this->** (or **this.** as it is in D) is not required to access member variables
- A colon, **:**, is equivalent to extends in Java.
- **super()** calls the constructor of the parent class.
- The ampersand, **&**, takes the address of a variable, method or function. It is used when you want to pass one method or function to another (among other things).
- A period, **.**, is used to access members of a class or struct, much like **->** in C++ and PHP.
- **void** means 'no type'
- **size_t** is the type used to represent the length of an array. Its length varies depending on the arcitecture of the computer you are using, it is unsigned in all cases.
- **auto** is used to infer type.
- **new MyClass[number]** results in an empty dynamic (number of elements can vary) array of **MyClass** with number number of elements.
- **foreach** is used to iterate over elements in an array, it is in form **foreach(index, element; array){}**, **index** is optional and the type of each is infered (although can be stated explicitly)
- If **ref** is placed before the element name in **foreach**, you will receive a reference to the value in the array, allowing you to mutate the array.
- All arrays have the property **length**, which returns the length of the array.

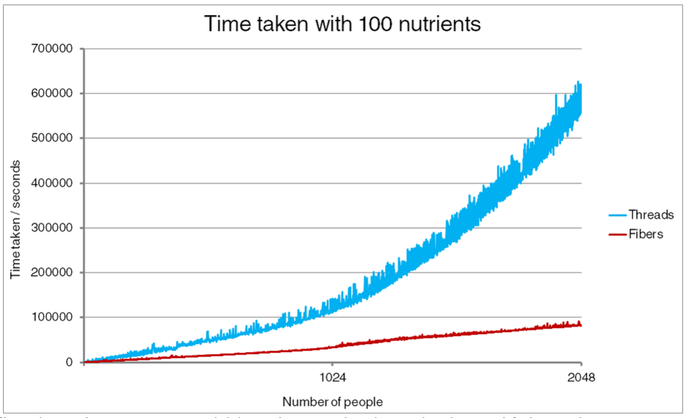
Listing 4

```
class FeedFiber : Fiber
{
  Person mPerson;
  this(Person p)
  {
    mPerson = p;
    super(&run);
  }

  void run()
  {
    while(!mPerson.satisfied)
    {
      mPerson.mNutrients.Fiber++;
      mPerson.mNutrients.Calcium++;
      mPerson.mNutrients.Iron++;
      Fiber.yield();
    }
  }
}
```

Listing 5

```
void feedWithFibers(int numPeople, int
numNutrients)
{
  size_t terminated;
  auto fibers = new FeedFiber[numPeople];
  foreach (ref f; fibers)
  {
    f = new FeedFiber(new Person(numNutrients));
  }

  while (terminated != fibers.length)
  {
    foreach (ref f; fibers)
    {
      if (f)
      {
        // The fiber has run to completion
        if (f.state == Fiber.State.TERM)
        {
           terminated++;
           f = null;
           continue;
        }
        f.call();
      }
    }
  }
}
```





returns rather than yielding. When it does, we increment a counter and set the fiber to null, then move onto the next fiber until there are no fibers left to operate on, that is, we've fed all the people and they're no longer malnourished. The final thing to do is actually call this method (Listing 6).

Like C, D uses the **main()** method for program entry, but with an array of strings as the arguments. The program will accept two arguments, the number of people, and the number of nutrients each person needs. We first check for the correct number of arguments – this is three as the first argument is always the path to the application. We then convert the strings to integers so we can use them as such – **to!()** will throw an exception if a valid integer isn't passed. Strictly speaking we should be using **size_t** throughout the application – arrays have a length of type **size_t**, and we are using these numbers to specify the length of the array. If you use a negative number as the first parameter it will cause an error, and a negative for the second will lead to an incredibly long runtime – the integer will have to overflow [8] before the correct number of nutrients is hit. Once this is

Listing 6

```
void main(string[] args)
{
  // Check for valid arguments
  enforce(args.length == 3);
  int numPeople    = to!int(args[1]);
  int numNutrients = to!int(args[2]);
  StopWatch sw;

  // Time feeding with threads
  sw.start();
  feedWithThreads(numPeople, numNutrients);
  sw.stop();
  writef("%s, ", sw.peek().usecs);
  sw.reset();

  // Time feeding with fibers
  sw.start();
  feedWithFibers(numPeople, numNutrients);
  sw.stop();
  writefln("%s", sw.peek().usecs);
}
```

fixed, a nicer error could be given – look at the beautiful stack traces you get if you don't use enough parameters or try and pass a non-integer for the arguments.

The next two chunks of code do two things – start and stop a stopwatch so we can time how long each takes, and feed people using both threads and fibers (see the 'Links' section at the end of this article for complete implementations of both). The output is written in csv format to allow output to easily be plotted.

So which is faster then?

Let's look at some graphs of the output. Each graph is created using the output of this command, using the number of nutrients given in the graph's title, and a number of people from 1 through 2048. The application was compiled as follows:

```
$ dmd -O -release -inline main.d
```

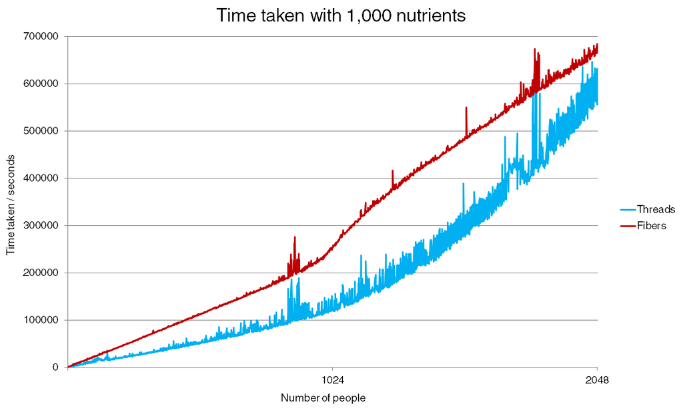
using dmd v2.052 on OS X 32bit. The machine in question has a 2.2Ghz Core 2 Duo CPU (dual core) and 2GB ram.

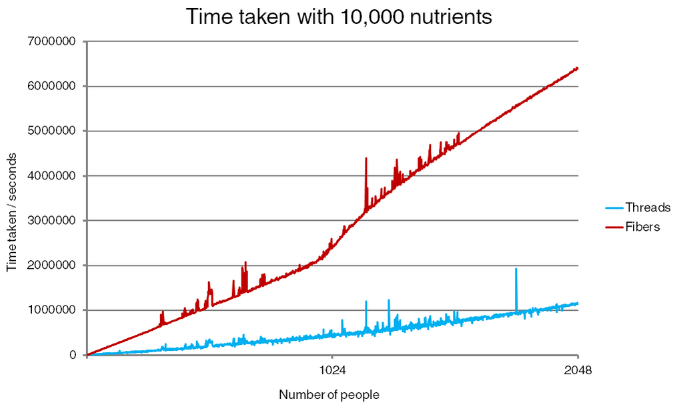
As you can see, with ten nutrients (Figure 1) there is a huge difference between threads and fibers – the time taken to feed the masses with fibers scales linearly with the number of people. When using threads it is fairly linear until about one thousand threads, where the time taken per each additional person is far greater.

If we increase the number of nutrients by an order of magnitude (Figure 2), we see a similar trend, however threads now have a lower gradient, leading to a more curve like shape – they are still far slower, however.

With another order of magnitude we see some more interesting results (Figure 3). Threads have overtaken fibers in performance. There are also some more noticable spikes in the graph at this point. This is entirely my fault, as I generated these statistics on my laptop which I was using for other things. This resulted in additional context switches being required, which had a dramatic effect on some of the numbers, particularly when using threads. I've ironed out the more anomalous results, there are still a few which need fixing though. Ideally I would rerun the benchmarks on a computer which isn't doing anything else.

The final step up in magnitude (Figure 4) leads to both threads and fibers appearing to scale linearly, but now fibers use up a lot more time.





What does this data actually mean?

Let's start with the obvious. The overhead of using fibers scales (fairly) linearly under all the tested workloads, and it'd be a fair bet to say this trend continues. This is an excellent thing – no matter what you're doing, you can keep adding tasks and scale the hardware with it. Threads on the other hand tend to be anything but linear until you have a certain workload – each time you add a task, the next task will need twice as many resources (or thereabouts) as the last task. This is definitely not a good thing.

The next thing to notice is that as the workload increases, threads become far more appealing. They become closer and closer to scaling linearly, and use less time. Fibers, on the other hand, take up a lot more time – after all, there's only so much work one processor core can do. Let's not forget however, that you can have multiple fibers per thread – you could take a hybrid approach and get the best of both worlds.

So how do I decide on the best approach?

The first thing you should look at is what you are trying to do. How many tasks will you have to do? How processor intensive are these tasks, and how does this compare to anything else your application is doing? Clearly if you're doing a few expensive tasks, threads are the way to go. If you're doing lots of cheap tasks fibers are the way to go – the overhead of creating threads will likely outweigh the tasks themselves. In the middle ground you can take the hybrid approach, both threads and fibers. Or, even better, processes and fibers. By using processes instead of threads you remove the need to worry about synchronisation, and if one process crashes, the others are still intact.

There is also the issue of deciding what is expensive and what isn't. The chances are if you're doing any kind of IO, whatever processing you're doing is negligible in comparison. Rather than using blocking, synchronous IO, you could switch to non-blocking and asynchronous IO, allowing you to process data for other IO sources while you wait. In the case of networking, this is what the fastest webservers (nginx, lighttpd, etc) do, in combination with epoll, kqueue and similar.

If you still aren't sure which you should be using (or even if you are!) try benchmarking and profiling each to see which performs better with whatever task you happen to be doing. ■

Links

Complete source code listing for the malnourished people example, including implementation for threads – https://gist.github.com/902318

Raw data and chart for 10 nutrients – https://spreadsheets.google.com/ccc?key=0AqnbEz4qka4ddEJZZnJRZFpMNGVUYUhKRXQ0Sk5YckE&hl=en&authkey=CNLR9rkC

Raw data and chart for 100 nutrients – https://spreadsheets.google.com/ccc?key=0AqnbEz4qka4ddHJzVzlmVExkbkQyTkVNcGFJSG0yNFE&hl=en&authkey=CM2GsMME

Raw data and chart for 1,000 nutrients – https://spreadsheets.google.com/ccc?key=0AqnbEz4qka4ddGtRSlFLejNpWDhvNFU5MWRYWWREa3c&hl=en&authkey=CILtgvIH

Raw data and chart for 10,000 nutrients– https://spreadsheets.google.com/ccc?key=0AqnbEz4qka4ddHRtaW9kVV9XN3Q5dWx5elpUQi1iSXc&hl=en&authkey=CL_K9Fc

References and notes

[1] http://en.wikipedia.org/wiki/Fiber_%28computer_science%29
[2] http://www.d-programming-language.org/
[3] http://en.wikipedia.org/wiki/Object_oriented_programming
[4] http://en.wikipedia.org/wiki/List_of_C-based_programming_languages
[5] http://www.d-programming-language.org/type.html
[6] http://www.d-programming-language.org/attribute.html
[7] http://www.d-programming-language.org/phobos/core_thread.html#Fiber
[8] http://en.wikipedia.org/wiki/Integer_overflow

Using the Windows Debugging API on Windows 64

Roger Orr finds smoke and mirrors inside 64-bit Windows.

In March 2010 I wrote an article in *CVu* about the Windows application debugging API and I illustrated the basic operation of the API with a simple process tracing application. This showed such events as the loading of DLLs, the starting and ending of threads and any exceptions.

This article uses the experience of porting the program to 64-bit Windows to provide a brief introduction to some of the issues of migrating applications from 32-bit Windows to 64-bits.

Note that there are two main flavours of 64-bit hardware that supports Windows. One is the Itanium architecture from Intel and the other is the AMD/64 architecture from AMD but now also supported (with minor modifications) by Intel.

However, the Itanium architecture has not been a great success and Microsoft announced last year that they are not supporting Itanium beyond Windows Server 2008 R2 and Visual Studio 2010. So I am ignoring Itanium in this article. (Interested readers can Google for articles such as 'How the Itanium Killed the Computer Industry' by John C. Dvorak.)

A very quick refresher on ProcessTracer

ProcessTracer works by creating a target process using the **`CreateProcess()`** function in conjunction with the **`DEBUG_ONLY_THIS_PROCESS`** flag. The flag results in Windows creating a channel between the parent and child process allowing the parent to receive notification of key events in the child process' lifecycle.

ProcessTracer then enters the main debugging loop which consists of:

- get the next debug event by calling **`WaitForDebugEvent()`**
- display details of the event as required
- acknowledge it with **`ContinueDebugEvent()`**.

The main loop ends when the child process exits.

Some of the debug events provide virtual addresses, such as the instruction pointer, and ProcessTracer maps these to symbolic addresses. This functionality is provided by the **`SimpleSymbolEngine`** class, which wraps the Microsoft DbgHelp API (and was originally written about in *Overload 67*: http://accu.org/index.php/journals/276)

Compiling ProcessTracer for 64-bit Windows

I found porting the code to 64-bit Windows was very easy: I set up a command prompt for x64 development using **`vcvarsall x64`** (in the `VC` subdirectory of the Visual Studio 2010 installation) and modified the makefile to write output to the `x64` directory.

I then ran **`nmake`** – everything compiled except the stack walking code in `SimpleSymbolEngine.cpp`.

The error is that the code setting up the **`StackFrame64`** structure was written for 32-bit code and referred to the **`Esp`**, **`Ebp`** and **`Eip`** fields of the **`CONTEXT`** structure. The equivalent fields for the 64-bit code are **`Rsp`**, **`Rbp`** and **`Rip`**. Additionally the machine type in the call to **`StackWalk64`** must be changed to **`IMAGE_FILE_MACHINE_AMD64`** from **`IMAGE_FILE_MACHINE_I386`**.

Listing 1

```
context.ContextFlags = CONTEXT_FULL;
GetThreadContext(hThread, &context);

stackFrame.AddrPC.Offset = context.Rip;
stackFrame.AddrPC.Mode = AddrModeFlat;

stackFrame.AddrFrame.Offset = context.Rbp;
stackFrame.AddrFrame.Mode = AddrModeFlat;

stackFrame.AddrStack.Offset = context.Rsp;
stackFrame.AddrStack.Mode = AddrModeFlat;

while (::StackWalk64(IMAGE_FILE_MACHINE_AMD64,
   hProcess, hThread, &stackFrame, pContext,
   0, ::SymFunctionTableAccess64,
      ::SymGetModuleBase64, 0))
{
  // ...
}
```

Listing 1 is a code fragment with the changes emboldened.

The rest of the code compiled (and worked) without problem, which is I think indicative that Microsoft has done a good job of making the current version of the API portable between 32-bit and 64-bit. However, this was not achieved without some changes: the older functions in the DbgHelp library were not portable, hence the introduction over the years of the various Xxxx64 structures and types which are portable between 32-bit and 64-bit Windows.

The same thing has been done with much of the Windows API: parameters and data structures are defined using typedefs to the appropriately sized underlying values. The same techniques can be used in application code that needs to compile in both environments.

There are two slightly different approaches taken with making the API portable between 32- and 64-bit windows. In general the Windows API has made use of typedefs for data types which map to the appropriate sized underlying type for each platform. So for example the **`ULONG_PTR`** type is defined in the Microsoft documentation as 'an unsigned long type used for pointer precision. It is used when casting a pointer to a long type to perform pointer arithmetic.' So the actual type is 32-bit when compiled for 32-bit Windows and 64 when compiling for 64-bit applications. However, the DbgHelp API has simply widened everything to 64-bits so the code produced for the previous article, targetting 32-bit Windows, was using types like **`STACKFRAME64`** and calling methods like **`StackTrace64`**.

Comparing 32-bit and 64-bit in action

I used a 'minimal' program for testing ProcessTracer. There are various ways to write this using MSVC, here is one way:

```
--- TrivialProgram.cpp ---
#pragma comment(linker, "/nodefaultlib")
int main() {return 0;}
int mainCRTStartup() {return 0;}
```

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

Listing 2

```
C:> x86\ProcessTracer x86\TrivialProgram.exe
CREATE PROCESS 3896 at 0x001D1010
LOAD DLL 77340000 ntdll.dll
LOAD DLL 75900000 C:\Windows\system32\kernel32.dll
LOAD DLL 75670000 C:\Windows\system32\KERNELBASE.dll
EXIT PROCESS 0
  Frame       Code address
  0x003EF994  0x773870B4 KiFastSystemCallRet
  0x003EF9A8  0x7736F652 RtlExitUserThread + 65
  0x003EF9B8  0x7594ED73 BaseThreadInitThunk + 25
  0x003EF9F8  0x773A37F5 RtlInitializeExceptionChain + 239
  0x003EFA10  0x773A37C8 RtlInitializeExceptionChain + 194
```

Listing 3

```
C:> x64\ProcessTracer x64\TrivialProgram.exe
CREATE PROCESS 4692 at 0x000000013FE91010
LOAD DLL 0000000077C90000 ntdll.dll
LOAD DLL 0000000077700000 C:\Windows\system32\kernel32.dll
LOAD DLL 000007FEFE3D0000 C:\Windows\system32\KERNELBASE.dll
EXIT PROCESS 0
  Frame       Code address
  0x00000000001DF720  0x0000000077CE15DA NtTerminateProcess + 10
  0x00000000001DF750  0x0000000077CB418B RtlExitUserProcess + 155
  0x00000000001DF790  0x0000000077CD697F RtlExitUserThread + 79
  0x00000000001DF7C0  0x0000000077716535 BaseThreadInitThunk + 21
  0x00000000001DF810  0x0000000077CBC521 RtlUserThreadStart + 33
```

Listing 4

```
C:> x86\ProcessTracer x86\TrivialProgram.exe
CREATE PROCESS 4844 at 0x00A21010
LOAD DLL 77E70000 ntdll.dll
UNLOAD DLL 77700000
UNLOAD DLL 77080000
UNLOAD DLL 77700000
UNLOAD DLL 77820000
LOAD DLL 77080000 C:\Windows\syswow64\kernel32.dll
LOAD DLL 76920000 C:\Windows\syswow64\KERNELBASE.dll
EXIT PROCESS 0
  Frame       Code address
  0x001EFC04  0x77E8FCB2 ZwTerminateProcess + 18
  0x001EFC18  0x77ECD5D9 RtlExitUserThread + 65
  0x001EFC28  0x770933A1 BaseThreadInitThunk + 25
  0x001EFC68  0x77EA9ED2 RtlInitializeExceptionChain + 99
  0x001EFC80  0x77EA9EA5 RtlInitializeExceptionChain + 54
```

When compiled and linked this produces a very small executable with no dependencies.

Listing 2 is the output from running the 32-bit version of ProcessTracer from the previous article on 32-bit Windows 7.

The obvious change we will see when running the 64-bit program is that the addresses are now 64-bit numbers rather than 32-bit ones. The output for the equivalent 64-bit program is shown in Listing 3.

There are a few changes in the call stack on process exit caused by implementation differences between the 32-bit and 64-bit Win32 subsystem.

Next step – running the 32-bit code on 64-bit

One of the main reasons why the AMD approach to a 64-bit architecture has been more successful than the Intel one is to do with running *existing* 32-bit programs. The AMD processor can be run in 64-bit and 32-bit mode and in the latter case it runs the existing 32-bit x86 instruction set. Hence, at least in theory, you should be able to run existing 32-bit code on the new chip without requiring changes and without any reduction in performance.

Of course, most programs require support from an operating system, so in order to run an existing 32-bit program the OS needs to provide an environment that is as close as possible to the 32-bit version of the OS. 64-bit Windows provides such an environment 'out of the box' and supports 32-bit applications using the 'Windows on Windows 64' subsystem, abbreviated to WOW64, which runs in user mode and maps the 32-bit calls to the operating system kernel into an equivalent 64-bit call. This is normally almost invisible to the calling program, but the debugging API exposes some of the scaffolding as shall see.

Windows provides a set of 64-bit DLLs in `%windir%\system32` and an equivalent set of 32-bit DLLs in `%windir%\syswow64`. In fact the bulk of the binary images in this directory are identical to the same files in the system32 directory on a 32-bit Windows installation. (It seems to me an unfortunate naming issue that the **64**-bit DLLs live in `system`**`32`** and the **32**-bit ones live in `syswow`**`64`**, but there it is!)

Listing 4 is the output when we run the 32-bit program on 64-bit Windows 7.

The two main differences (apart from minor changes in virtual addresses) are firstly the presence of a few **`UNLOAD DLL`** messages with no corresponding **`LOAD DLL`** (we shall return to these later on) and secondly the subdirectory used for system DLLs is `syswow64` rather than `system32`.

The WOW64 subsystem maps all attempts by 32-bit programs to access files in the `system32` subdirectory to requests for the equivalent file in the `syswow64` subdirectory. Hence when `trivialProgram.exe` loads `ntdll`, `kernel32` and `kernelbase`, the WOW64 layer transparently loads these files from the `syswow64` directory (even though the PATH does not include it).

This mapping can be turned off, per thread, using two functions designed for this specific purpose: **`Wow64DisableWow64FsRedirection`** and **`Wow64RevertWow64FsRedirection`**. Note, however, that since the redirection was designed to make existing 32-bit applications load the right DLL, use of these functions with code that loads system DLLs (implicitly or explicitly) can lead to strange failures. Alternatively the application can use the virtual subdirectory name `sysnative`, which the WOW64 subsystem maps to `system32`.

This can be somewhat confusing in practice when you use a mix of 32-bit and 64-bit programs on the same machine. As a simple example, consider this sequence of operations:

1. Start a command prompt.
2. Type notepad and press Enter.

   Windows starts the 64-bit notepad program from `C:\Windows\system32`.
3. Type `C:\Windows\SysWow64\cmd.exe` and press Enter to run a 32-bit command shell.
4. Type notepad and press Enter.

Windows now starts the **32-bit** notepad program simply because the current shell is a 32-bit process and WOW modifies the target directories used when searching the PATH for the notepad program. I defy you to identify which notepad is which: but the same sequence of key strokes in two different command shells has executed two different programs because of the special treatment of the `system32` directory by the WOW subsystem.

The other main translation that the WOW64 layer provides is for access to the Windows registry. The 32-bit specific parts of the registry are held underneath the **`Wow6432Node`** key in various places in the registry, such as **`HKEY_LOCAL_MACHINE`**, and so a request from a 32-bit program to **`HKLM\Software\Test`** is transparently mapped by WOW64 to a request for **`HKLM\Software\Wow6432Node\Test`**. Note that the

Listing 5

```
C:> x64\ProcessTracer x86\TrivialProgram.exe
CREATE PROCESS 3004 at 0x0000000000121010
LOAD DLL 0000000077C90000 ntdll.dll
LOAD DLL 0000000077E70000 ntdll32.dll
LOAD DLL 0000000073EF0000 C:\Windows\SYSTEM32\wow64.dll
LOAD DLL 0000000073E90000 C:\Windows\SYSTEM32\wow64win.dll
LOAD DLL 0000000073E80000 C:\Windows\SYSTEM32\wow64cpu.dll
LOAD DLL 0000000077700000 WOW64_IMAGE_SECTION
UNLOAD DLL 0000000077700000
LOAD DLL 0000000077080000 WOW64_IMAGE_SECTION
UNLOAD DLL 0000000077080000
LOAD DLL 0000000077700000 NOT_AN_IMAGE
UNLOAD DLL 0000000077700000
LOAD DLL 0000000077820000 NOT_AN_IMAGE
UNLOAD DLL 0000000077820000
LOAD DLL 0000000077080000 C:\Windows\syswow64\kernel32.dll
LOAD DLL 0000000076920000 C:\Windows\syswow64\KERNELBASE.dll
EXCEPTION 0x4000001f at 0x0000000077F10F3B LdrVerifyImageMatchesChecksum + 2412
  Parameters: 0
  Frame       Code address
  0x000000000046F414  0x0000000077F10F3C LdrVerifyImageMatchesChecksum + 2413
  0x000000000046F41C  0x000000007716936D WakeConditionVariable + 127125
  0x000000000046F424  0x7EFDE00000000000
  0x000000000046F42C  0x0046F41C00000000
  0x000000000046F434  0x0046F60477EE1ECD
  0x000000000046F43C  0x00B9DABD77EE1ECD
  0x000000000046F444  0x0046F5C400000000
  0x000000000046F44C  0x7EFDD00077EF1323
  0x000000000046F454  0x77F7206C7EFDE000
EXIT PROCESS 0
  Frame       Code address
  0x000000000029E190  0x0000000077CE15DA NtTerminateProcess + 10
  0x000000000029E1C0  0x0000000073F0601A Wow64EmulateAtlThunk + 34490
  0x000000000029EA80  0x0000000073EFCF87 Wow64SystemServiceEx + 215
  0x000000000029EB40  0x0000000073E82776 TurboDispatchJumpAddressEnd + 45
  0x000000000029EB90  0x0000000073EFD07E Wow64SystemServiceEx + 462
  0x000000000029F0E0  0x0000000073EFC549 Wow64LdrpInitialize + 1065
  0x000000000029F5D0  0x0000000077CD4956 RtlUniform + 1766
  0x000000000029F640  0x0000000077CD1A17 RtlCreateTagHeap + 167
  0x000000000029F670  0x0000000077CBC32E LdrInitializeThunk + 14
```

Wow6432Node key name is reserved and applications should not program against this value – the registry functions have been enhanced to allow a 32-bit process to see 64-bit registry keys (and vice versa).

I am not going to cover more of this mapping in this article. I refer interested readers to the Microsoft web site: for example 'Running 32-bit Applications' at http://msdn.microsoft.com/en-us/library/aa384249%28v=VS.85%29.aspx

Mixing it up

We have so far tried running a pair of 64-bit programs and a pair of 32-bit programs on 64-bit Windows. What happens if we mix and match?

First off, we try to execute the 32-bit ProcessTracer with a 64-bit target:

```
C:> x86\ProcessTracer x64\TrivialProgram.exe
Unexpected exception: Unable to start
x64\TrivialProgram.exe: 50
```

This failing error code (**50**) is **"ERROR_NOT_SUPPORTED"** – Windows does not allow a 32-bit program to debug a 64-bit program. This makes some sort of sense: an existing 32-bit debugging program would be unlikely to be able to interpret the 64-bit values correctly for addresses, register values, etc. As far as I know there is no way to work around this.

This is one place where an approach like that taken by the DbgHelp API,of widening everything to 64-bits, might have enabled cross boundary debugging in this direction.

We have more success if we try it the other way round (Listing 5). However, some of the output produced is a little unexpected! After all, we are debugging *exactly* the same process that we debugged earlier under 'Next step – running the 32-bit code on 64-bit' but we are getting much more output and a very different call stack on program exit. What is going on?

The first change when using the 64-bit debug API is that the debugger is notified about additional DLLs loaded into the process address space. The first DLL loaded, `ntdll.dll`, is in fact the **64-bit** version of the DLL and it is followed by the 32-bit `ntdll32.dll`. Note that the 32-bit mode debugger doesn't see the 64-bit DLL, only the 32-bit one.

The next few DLLs are the WOW64 implementation – these are 64-bit DLLs loaded into the target process. Again, the 32-bit mode debugger does not receive any notification when these DLLs are loaded into the process. Notice however that, probably through an oversight in the implementation of the 32-bit debug interface, the 32bit debugger sees the **UNLOAD DLL** messages for the special WOW64 DLLs.

We then receive an unexpected exception – code **0x4000001f** – which turns out to be a new value, **STATUS_WX86_BREAKPOINT**. This exception code is barely documented but it appears to be a 'start up' breakpoint much like the initial breakpoint that the debugger always receives. On older versions of Windows the default processing for this exception code terminates the process, on Windows 7 it is ignored.

So I enhanced the exception handling in **ProcessTracer::run()** to cater for this new breakpoint (Listing 6).

The last issue with the output from the debugger is that the call stack shown on process exit has almost nothing in common with the call stack from the 32-bit version of ProcessTracer.

This is because the 64-bit debugger is displaying the call stack in the 64-bit context of the WOW64 call to terminate the process. In the 32-bit debugger we see the other half of the picture – the call stack in the 32-bit part of the call stack **above** the WOW64 emulation layer.

Reading 32-bit context from 64-bit debugger

Windows provides a way for 64-bit debuggers to access the same data the 32-bit debug interface gives. The debugger can use **IsWow64Process()** to determine if the target process is a 32-bit process or a 64-bit one, and in the former case it can read the 32-bit context using **Wow64GetThreadContext()**. (Note: this API was added in Windows 7. The information can be obtained on earlier versions of 64-bit Windows but the mechanism is a little more complicated).

Once the 32-bit context has been obtained it can be used to populate the **StackFrame64** structure and passed into to the **StackWalk64()** function together with the **IMAGE_FILE_MACHINE_I386** machine type to print out the 32-bit call stack.

Listing 6

```
  case EXCEPTION_DEBUG_EVENT:
    if (!attached)
    {
      // First exception is special
      attached = true;
    }
#ifdef _M_X64
  else if
(DebugEvent.u.Exception.ExceptionRecord.Exception
Code == STATUS_WX86_BREAKPOINT)
  {
    std::cout << "WOW64 initialised"
              << std::endl;
  }
#endif // _M_X64
  else
  {
    OnException(DebugEvent.dwThreadId,
      DebugEvent.u.Exception.dwFirstChance,
      DebugEvent.u.Exception.ExceptionRecord);
    continueFlag =
       (DWORD)DBG_EXCEPTION_NOT_HANDLED;
  }
  break;
```

Listing 7

```
void SimpleSymbolEngine::stackTrace(
   HANDLE hThread, std::ostream & os )
{
  CONTEXT context = {0};
  PVOID pContext = &context;
  STACKFRAME64 stackFrame = {0};

#ifdef _M_IX86
  DWORD const machineType =
     IMAGE_FILE_MACHINE_I386;
  context.ContextFlags = CONTEXT_FULL;
  GetThreadContext(hThread, &context);
  stackFrame.AddrPC.Offset = context.Eip;
  stackFrame.AddrPC.Mode = AddrModeFlat;

  stackFrame.AddrFrame.Offset = context.Ebp;
  stackFrame.AddrFrame.Mode = AddrModeFlat;

  stackFrame.AddrStack.Offset = context.Esp;
  stackFrame.AddrStack.Mode = AddrModeFlat;
#elif _M_X64
  DWORD machineType;
  BOOL bWow64(false);
  WOW64_CONTEXT wow64_context = {0};
  IsWow64Process(hProcess, &bWow64);
  if (bWow64)
  {
    machineType = IMAGE_FILE_MACHINE_I386;
    wow64_context.ContextFlags =
       WOW64_CONTEXT_FULL;
    Wow64GetThreadContext(hThread,
       &wow64_context);
    pContext = &wow64_context;
    stackFrame.AddrPC.Offset = wow64_context.Eip;
    stackFrame.AddrPC.Mode = AddrModeFlat;

    stackFrame.AddrFrame.Offset =
       wow64_context.Ebp;
    stackFrame.AddrFrame.Mode = AddrModeFlat;

    stackFrame.AddrStack.Offset =
       wow64_context.Esp;
    stackFrame.AddrStack.Mode = AddrModeFlat;
  }
```

Listing 7 is the final version of the stack walking code, which compiles on both 32-bit and 64-bit builds and handles 32-bit targets in 64-bit mode.

The reading of the 32-bit context from 64-bit mode also works where the program is currently executing in 32-bit mode (such as when a breakpoint occurs).

The transition between 32-bit and 64-bit mode occurs when the 32-bit program calls into the operating system. In the case we've looked at when our 32-bit program exits it calls **`ZwTerminateProcess`** in the 32-bit world. This function sets up parameters in exactly the same way as it does on 32-bit Windows but then jumps to **`X86SwitchTo64BitMode`** in the WOW64 subsystem. This jump switches the CPU from 32-bit mode back into 64-bit mode. The target function saves away the 32-bit register state and (after some further processing) calls into the operating system in 64-bit mode. The reverse process occurs on return from 64-bit mode when the function completes.

The **`Wow64GetThreadContext()`** function retrieves the 32-bit context by reading it from the memory location where the WOW64 process has saved it, when the target thread is in 64-bit mode, or by mapping the current

Listing 7 (cont'd)

```
  else
  {
    machineType = IMAGE_FILE_MACHINE_AMD64;
    context.ContextFlags = CONTEXT_FULL;
    GetThreadContext(hThread, &context);
    stackFrame.AddrPC.Offset = context.Rip;
    stackFrame.AddrPC.Mode = AddrModeFlat;

    stackFrame.AddrFrame.Offset = context.Rbp;
    stackFrame.AddrFrame.Mode = AddrModeFlat;

    stackFrame.AddrStack.Offset = context.Rsp;
    stackFrame.AddrStack.Mode = AddrModeFlat;
  }
#else
#error Unsupported target platform
#endif // _M_IX86
  DWORD64 lastBp = 0; // Prevent loops with
                      // optimised stackframes
  os << "  Frame       Code address\n";
  while (::StackWalk64(machineType,
    hProcess, hThread,
    &stackFrame, pContext,
    0, ::SymFunctionTableAccess64,
    ::SymGetModuleBase64, 0))
  {
    if (stackFrame.AddrPC.Offset == 0)
    {
      os << "Null address\n";
      break;
    }
    PVOID frame =
      reinterpret_cast<PVOID>(
      stackFrame.AddrFrame.Offset);
    PVOID pc =
      reinterpret_cast<PVOID>(
      stackFrame.AddrPC.Offset);
    os << "  0x" << frame << "  " <<
       addressToString(pc) << "\n";
    if (lastBp >= stackFrame.AddrFrame.Offset)
    {
      os << "Stack frame out of sequence...\n";
      break;
    }
    lastBp = stackFrame.AddrFrame.Offset;
  }

  os.flush();
}
```

How To Be Dispensable

Frances Buontempo considers the virtue of being non-essential.

Suppose you make a fortune overnight and don't have to go to work tomorrow. How will they manage without you? The company will either collapse in a heap because you were a vital, indispensable member of the team or they will carry on regardless. Which will it be? Which would you prefer?

Let us first consider how to make the company fall to pieces. Each ploy depends in essence on ensuring that no-one else you work with, or who ever gets hired, can do what you do. Clearly, you can use tools you have written yourself, rather than standard tools. Standard tools mean other people know how they work. This should be avoided at all costs. Cover yourself by explaining that yours will run much quicker in this company's environment. To compound this, you can have several different versions of your home grown apps in different places, making it hard for people to figure out which version should be used for what. If they can find them in the first place. So, first top tip

1. Avoid putting your work in version control.

   This gives corollary.

2. Hack straight into production, so no one knows which version is out there, even if a bloody-minded colleague managed to get your work in version control.

Regardless of whether your team can find your code or not, you must now consider which approach to take to documentation. You have two options, though the outcome will be the same in both cases.

3a) Never, ever, document anything.

or

3b) Any documentation you write will be at least 1,000 pages long, so that no-one can face reading it all. (You can verify it is unread by placing the sentence 'Donkeys are aliens'[1] in a couple of paragraphs. This is fool-proof. Someone will speak up if they find this sentence. Other sentences can be tried. But this phrase has been shown to work.)

Now we must look at the specifics of what you are writing and how you write it. An obvious choice is utilising a language no-one else knows, or as few people as possible. A more subtle choice is something everyone knows, for example VBA: Even your cat can programme in that. History has shown how easy it is to create a strategic spreadsheet that is such a memory hog, requiring dedicated hardware to run it. This allows the possibility of writing an xll, which uses C++ extensions you can write yourself, once you've implemented your own compiler. Let us summarise this as

4. Choose the right language.

Having decided where to hide your code and what language to write it in, the last technical point concerns how to test your code.

5. Make sure no-one, including you, can test your code.

This can be achieved in several ways. For example [2] gives several hints and tips on how to write difficult-to-test code. This suggests it is theoretically possible. Though I can find no documented example of untestable code, you should aim for the best. That's why they can't do without you. Don't forget to remind them of this as frequently as possible. For example

6. Use obscure language corners and rely on specifics of your chosen compiler version or interpreter.

This allows you to show off your detailed knowledge and show up the rest of your team. Management will then realise how much cleverer you are than anyone else and how vital you are to the company.

Finally, remember management are in charge, so they must realise you are irreplaceable. An easy way to convey this is our final point.

7. Ensure that your name is on every Gantt chart and project plan, forming a major bottleneck to everything.

This is the only way to be sure that the team cannot cope without you. You are now indispensable. Congratulations.

Before considering the alternative (they manage without you), be aware of the warning, 'If a programmer is indispensable, get rid of him as quickly as possible. .' [3] Outrageous though this may seem at first sight, more worrying thoughts have been expressed, in particular, 'If you can't be replaced, you can't be promoted.' [4] Clearly, not everyone is entering into the spirit of things here. If being indispensable is bad, how can you be dispensable?

Taking an opposing approach to the steps required for indispensability is a good starting place.

FRANCES BUONTEMPO

Frances Buontempo has a BA in Maths and Philosophy, an MSc in Pure Maths and a PhD in AI and data mining. She has been a programmer for over 10 years and learnt by reading the manual for her Dad's BBC Micro. She can be reached at frances.buontempo@gmail.com

Using the Windows Debugging API on Windows 64 (continued)

context back into the 32-bit structure when the target thread is in 32-bit mode.

Conclusion

Microsoft have done a reasonably good job of allowing application to be ported to 64-bit operation and also support transparently running 32-bit applications on 64-bit Windows. For many users whether a particular application is 32-bit or 64-bit is simply not relevant, nor obvious.

However, as programmers, it is useful to have understanding of how the process operates and what issues there can be; especially when running existing 32-bit applications on 64-bit.

The debugging API provides one way of looking at some of the implementation details of supporting the Windows-on-Windows subsystem and I hope this understanding will help inform those migrating from 32-bit to 64-bit Windows. ■

Source code

The full source code for this article can be found at:
http://www.howzatt.demon.co.uk/articles/ProcessTracer.zip

Writing a Bazaar Plugin

Peter Hammond makes Bazaar do more than version control.

Bazaar is a popular open source Distributed Version Control System (DVCS). The tool itself is fairly easy to use, and well documented. As with many modern applications, it has a plugin model for extending its functionality, by intercepting published hooks, creating new commands or decorating existing commands. Unfortunately, the developer documentation leaves something to be desired. For example, all that the official plugin development guide [1] says about 'Extending an existing command' is 'TO BE DOCUMENTED'. This article describes some of my experiences in writing a rudimentary plugin to meet our needs.

Our problem was integrating with the Jira project tracking tool, and in particular how to enforce commit messages that match issue numbers. Jira is a commercial issue tracking tool, which also has a plugin architecture with a lively marketplace of plugins, both free and commercial. Its support for Bazaar is limited, but it does have good support for Subversion, and using Subversion as a central repository is well supported in Bazaar (using a plugin, naturally) [2]. We are using a Jira plugin that enforces the rule that messages on commits to the central Subversion repository must refer to an issue number, for tracking. However, since we are using Bazaar for local version control, it is easy to forget to put these messages on, leading to much frustration when a batch of changes cannot be pushed back up to the server. This was the problem that the plugin described here was intended to fix.

The plugin has three parts, which fortuitously correspond to the three extension approaches for Bazaar:

- A hook on the creation of the commit message, to put the branch name at the start of the message
- A new command to set an option for the Jira repository location

PETER HAMMOND

Peter Hammond is a senior software engineer with BAE Systems, with particular interests in open and component architectures and agile methods. Previously he developed embedded systems for non-destructive testing applications.

How to be Dispensable (continued)

1. Put your work in version control.

   This also has a corollary

2. Set up continuous integration. Consider doing this just for yourself, even at home [5]. This will obviously mean the code is testable and has tests.

Broadening out the observation on documentation, the point is to communicate with the team. If you make some form of notes, you will be able to pick up a previous project years, months, weeks or days, say after the weekend, with minimal hassle. If you can do this, anyone will be able to. How many times have you got in on Monday morning to realise you have no idea what you were in the middle of?

3. Make notes on a Wiki page. Have daily standup meetings, so everyone knows who is up to what. Then your team mates can remind you what you are doing if you ever forget.

Each of the ideas so far are rooted in communication. Tools such as version control, CI and Wiki pages facilitate communication, even for one person to themselves in the future. Be part of a team. Talk to each other. Share problems and triumphs.

4. Pair programme, or at least have code reviews.

A collaborative approach can help avoid bugs, encourages knowledge to be shared and means no one person is indispensable for a project, since at least one other team member knows about the code.

A dispensable team member is likely to enjoy working with the rest of the team, staying positive and constantly learning from those around. Even if the dispensable programmer suspects she is becoming indispensable, use she will use her awesome powers to immediately make herself dispensable. She will find and train understudies. [6]

your team mates can remind you what you are doing if you ever forget

We have discovered you should aim for the company to carry on regardless when you leave. At very least, you might be able to carry on where you left off when you return on Monday morning. If you're dispensable your team can do without you, but they'd rather not. ■

References

[1] Bill Bailey
[2] Working with legacy code
[3] Gerald Weinberg in *The Psychology Of Computer Programming*
[4] http://programmers.stackexchange.com/questions/48697/should-a-programmer-be-indispensable
[5] Advised by Jez Higgins at his Jenkins lightning talk, ACCU London Nov 2011
[6] http://c2.com/cgi/wiki?GetRidOfIndispensableProgrammerAsQuicklyAsPossible

- An extension to the branch command to enforce making a branch name correspond to an issue ID.

I will go through the steps involved in creating each one of these, describing the Bazaar APIs that are used. I will focus on the parts that were not obvious from the documentation I found, and not go into too much detail about the actual implementation of the logic.

First steps

A Bazaar plugin at its simplest is just a python module that can be found by Bazaar. A deployed plugin lives in the plugins folder under the bazaar installation, but for development and testing you can put the plugin anywhere, and point the **`BZR_PLUGIN_PATH`** environment variable at it.

It is customary to use a python module for a plugin, even if the plugin only needs one file, although a single `.py` file on the plugin path will be found and loaded correctly. In this example, I created a folder `D:\Projects\bzr_plugin\jira`, set **`BZR_PLUGIN_PATH=d:\Projects\bzr_plugin`**, and created `__init__.py` in the `jira` directory:

```
"""Hooks to help prevent common mistakes when
working with Jira"""
version_info=(0,0,1)
```

Now typing **`bzr plugins`** at the command line will show the new plugin in the list, with the version number and the description given. Note how the **`docstring`** is used to provide help text; this simple idea is reused a few times. The **`version__info`** attribute is described as 'a tuple defining the current version number of your plugin' [3] . The documentation does not specify any constraints, and empirically it appears that the tuple is simply passed through to be displayed. However, it is noted in the bzrlib source [4] that it should be 'the same format as **`sys.version_info`**'.

All plugins are loaded on every invocation of bzr. To see this in action, put a **`print`** statement in the module: the output will be seen before the actual bzr output.

Now we have a basic skeleton in place, we just need to make it do something.

Writing a hook

The easiest way to extend Bazaar's behaviour is to use one of the predefined hooks. The full list of available hooks can be found in the bazaar documentation [5], although at the time of writing some dead links makes this page a little more tricky to find than it might be. A summary is available by using the **`bzr hooks`** command. In this case, I browsed the list in [5] to find something that might meet our needs; **`commit_message_template`** in the section **`MessageEditorHooks`** looks like the one I need. The documentation states that

> The class that contains each hook is given before the hooks it supplies. For instance, BranchHooks as the class is the hooks class for bzrlib.branch.Branch.hooks.

This makes more sense when read in conjunction with the bzrlib documentation [6] to explain what the various classes and namespaces are. **`MessageEditorHooks`** refers to a class that can be found in the **`bzrlib.msgeditor`** package, after a little searching. However, the bzrlib documentation misses out a crucial piece of information: the **`msgeditor`** package has an instance of that class, called **`hooks`**, that can be used to register the new hook. The **`bzrlib.branch`** module used in the documentation's example does not have a **`hooks`** attribute, but the **`bzrlib.branch.Branch`** class does. Finding where a particular **`Hooks`** class is instantiated appears to be something of a matter of trial and error. Using **`print`** to test various candidates seems to be the easiest approach.

Having located the **`Hooks`** instance to register with, creating and registering a hook is a fairly painless task. An outline of the hook is given in listing 1. The **`install_named_hook`** method is called with three arguments: the name of the hook to install, the hook function, and a string that will appear in the output of the **`bzr hooks`** command to describe installed hooks. The implementation of the hook also introduces another interesting concept: how to get at the 'current' branch. The branch is not passed to the hook. The static method **`bzrlib.branch.Branch.open_containing`** can be used to get the branch that contains a given path. Here we are assuming that the command will be run in a directory that is part of the branch. Since the **`commit`** command does not take an argument to specify the branch to work on, this seems like a fair assumption.

Writing a command

Writing a command involves a little more code. A command in Bazaar is an instance of a class that specialises the **`Command`** class, and has certain expected properties. The important parts of the command structure are shown in listing 2. Going through this listing, we see that

- The class name is **`cmd_jira_host`**, this will automatically create a command 'jira-host' in bzr when it is registered.

Listing 1

```
from bzrlib import msgeditor
from bzrlib.branch import Branch

def commit_message_template(commit, msg):
  root = Branch.open_containing(".")[0].base
  if root[-1]=="/":
    root=root[:-1]
  branch = os.path.split(root)[1]

  if not msg:
    msg = ""
  return branch + ": " + msg

msgeditor.hooks.install_named_hook(
   "commit_message_template",
   commit_message_template,
   "Provide default issue id")
```

Listing 2

```
from bzrlib.commands import Command,
register_command
from bzrlib.config import BranchConfig
from bzrlib.ui import ui_factory

class cmd_jira_host (Command):
 """Sets the jira-host option for branch
    checking.

    The option is stored as a branch option.
    With no arguments, reports the current
    setting. The host value should be the network
    location part of the URL."""
  takes_args=['host?']

  def run (self, **kwargs):
    if 'host' in kwargs:
      host = kwargs['host']
      ...
      config = BranchConfig(
         Branch.open_containing(".")[0])
      config.set_user_option("jira-host", host)
    else:
      config = BranchConfig(
         Branch.open_containing(".")[0])
      host = config.get_user_option("jira-host")
      if(host):
        ui_factory.show_message (host)
      else:
        ui_factory.show_message (
           "Jira host is not set")
register_command (cmd_jira_host)
```

Listing 3

```
# bzr help jira-host
Purpose: Sets the jira-host option for branch
checking.
Usage:   bzr jira-host [HOST]

Options:
 --usage        Show usage message and options.
 -v, --verbose  Display more information.
 -q, --quiet    Only display errors and warnings.
 -h, --help     Show help message.

Description:
 The option is stored as a branch option.
 With no arguments, reports the current setting.
 The host value should be the network location
part of the URL.

From:     plugin "jira"
```

- The class has a docstring, which is used as help for **bzr help** commands and **bzr help jira-host**. In the former case, the first line is used for a summary. In the latter case, the whole docstring is used. Listing 3 shows the help that is generated for free.
- The class has a **takes_args** attribute, which is used to automatically handle option parsing and passing. The **?** indicates it is an optional positional parameter. This command has no options, otherwise it would also declare a **takes_options** attribute. These attributes are also used to generate help for **bzr help jira_host**.
- The **run** method is called to do the command. It receives keyword arguments including the optional "**host**" argument, corresponding to the argument declared in **takes_args**. Here we use the presence or absence of that argument to decide whether to set or display the current setting.
- Error checking on the provided host has been elided.
- The **BranchConfig** class handles configuration options for a specified branch. Configuration can also be stored per user or per system.
- **bzrlib.ui.ui_factory** provides an object that can do text-based user interaction, within the framework. Using a specialisation of **bzrlib.ui.UIFactory** is preferred over direct interaction with the terminal, because it can enforce the application's conventions. The documentation for **bzrlib.ui** does not mention this object [7], but does describe the various types that it may be. One might expect from this that it is necessary to instantiate one, perhaps using **make_ui_for_terminal**, but actually the **ui_factory** instance is provided by the framework and should be used directly.

Extending an existing command

I want to extend the branch command to enforce the new branch name being a valid Jira issue ID, so that the message template can use that name. Extending a command is similar to creating a new command, with a few extra traps, as shown in listing 4.

- I did not wish to re-create the original **docstring**, so I used the **__doc__** attribute directly to set it.
- Similarly, the **takes_options** and **takes_args** attributes are inherited and extended by the class.
- The **run** method must remove our arguments from the **kwargs** list otherwise the base command throws out what it sees as an invalid argument list.
- The **register_command** function has to be told that this is an extension, otherwise it will reject a duplicate command name.

Listing 4

```
from bzrlib import builtins
from bzrlib.option import Option

class cmd_branch (builtins.cmd_branch):
  __doc__=builtins.cmd_branch.__doc__
  takes_options=[Option("nojira",
     help="Suppress Jira checks")]+
     builtins.cmd_branch.takes_options
  takes_args=builtins.cmd_branch.takes_args

...

  def run(self, *args, **kwargs):
    to_loc = kwargs["to_location"]
    from_loc = kwargs["from_location"]
    if ("nojira" in kwargs):
      nojira = kwargs["nojira"]
      del kwargs["nojira"]
    else:
      nojira=False

    if nojira or self._branch_is_ok(from_loc,
       to_loc) or self.confirm(to_loc):
      return builtins.cmd_branch.run(self,
        *args, **kwargs)

register_command (cmd_branch, True)
```

Testing

Testing a plugin presents some issues, as its dependencies in the framework must be set up in a test environment. Fortunately Bazaar has fairly comprehensive support for testing, built on top of the standard **unittest** package. The first step is to create a function called **test_suite** within the plugin's package that exports a test suite. I created a new module in the package to hold the tests, and delegated to this module in my **test_suite** function. Listing 5 shows the outline of the **bzrlib.plugins.jira.jira_plugin_tests** module.

Bazaar provides a number of specialisations of **unittest.TestCase** that provide the necessary context for a plugin to be tested. **TestCaseWithMemoryTransport** should be used as a default.

Listing 5

```
from bzrlib.tests import (
    TestCaseWithMemoryTransport,
    TestCaseInTempDir,
    )

from unittest import TestLoader, TestSuite
from bzrlib.plugins import jira

class message_tests (
   TestCaseWithMemoryTransport):
   def test_MessagPrependsBranch(self):
   branch_dir = os.path.split(self.TEST_ROOT)[1]
  self.assertTrue(
     jira.commit_message_template(
     None, "Foo").
  startswith(branch_dir))

class jira_location_tests(TestCaseInTempDir):
...

def test_suite():
    ldr = TestLoader()
    return TestSuite([ldr.loadTestsFromTestCase(
       message_tests),
       ldr.loadTestsFromTestCase(
         jira_location_tests)])
```

Listing 6

```
class jira_location_tests(TestCaseInTempDir):
  def test_option_command_shows_setting_for_
  location(self):
    stdout = StringIOWrapper()
    ui.ui_factory = TestUIFactory(stdout=stdout)
    cmd = jira.cmd_jira_host()
    cmd.run(location="bm:foo")
    self.assertEqual (stdout.getvalue(),
                      "Jira host is not set\n")

    cmd.run (location="bm:foo",
             host="company.com:8080")
    cmd.run(location="bm:foo")
    self.assertContainsRe (stdout.getvalue(),
      "company.com\:8080\n")
    self.assertContainsRe (self.request_url,
      "http://company.com:8080/rest/api/
      2.0.alpha1")
```

`TestCaseInTempDir` should be used where an actual tree has to be created on disk. It is not clear from the documentation why the latter is required for a test that uses branch configuration, but it worked where the former did not. A plugin exercised in the context of one of these will see a repository that is created for the duration of the test. The test case can see the name of the repository through the **`TEST_ROOT`** attribute on the test case instance. Listing 6 shows how I used that facility to test that the commit message template function uses the repository name.

Testing a plugin's interaction with the console demonstrates why one should use **`ui_factory`** for output, as described above. Replacing **`ui.ui_factory`** with a **`TestUIFactory`** instance, based on a **`StringIO`**, allows the test to see what was output. This of course relies on the code under test always using **`ui.ui_factory`** for output, and not importing **`ui_factory`** into its own namespace. Listing 6 shows a test that makes use of this facility. Points to note here are:

- **`TestUIFactory`** does not work with **`StringIO`** directly, you must use **`bzrlib.tests.StringIOWrapper`**.
- **`Bzrlib.test.TestCase`** comes with some extra assertions, including **`assertContainsRe`** to find a string within a target string.

Closing remarks

The documentation of the process for creating a Bazaar plugin poses a higher barrier to entry than might be expected, given how easy the tool is from a user's perspective and how pleasant Python usually is to work in. This article should hopefully save some of the early frustrations to getting a basic plugin working.

There are many aspects that are not covered here, mainly because the plugin I wrote worked perfectly well without. For example, the plugin does everything in its **`__init__.py`**, which is not recommended as it can pose performance issues. Bazaar provides a lazy loading facility to avoid the performance penalty of loading unused modules on every invocation, but I found performance to be acceptable without having to go into those complexities. ■

References

[1] http://doc.bazaar.canonical.com/plugins/en/plugin-development.html

[2] http://doc.bazaar.canonical.com/migration/en/foreign/bzr-on-svn-projects.html

[3] http://doc.bazaar.canonical.com/latest/en/user-guide/writing_a_plugin.html

[4] http://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev/view/head:/bzrlib/__init__.py

[5] http://doc.bazaar.canonical.com/development/en/user-reference/hooks-help.html

[6] http://people.canonical.com/~mwh/bzrlibapi/bzrlib.html

[7] http://people.canonical.com/~mwh/bzrlibapi/bzrlib.ui.html

Effect of Risk Attitudes on Recall of Assignment Statements (Part 1)

Derek Jones reveals the results of his ACCU 2011 Conference developer experiments.

One of the first major discoveries in experimental psychology was a feature of human memory that has become generally known as short term memory. People are able to temporarily retain a small amount of information in memory whose accuracy quickly degrades unless an effort is made to 'refresh' it, and the information is easily overwritten by new information.

The capacity of short term memory (STM) has been found to correspond to approximately two seconds worth of sound, with some people have less capacity and some more.

There have been a huge number of experiments investigating the characteristics of STM and its impact on human cognitive performance. Since 2004 I have been trying to experimentally [7, 8] measure the impact of STM on developer performance when recalling information about previously seen source code (usually a sequence of assignment statements). In these experiments subjects have always been given the option to answer 'I would refer back', i.e., if they have to recall this information in a work environment they would refer back to the previously read code rather than use whatever information they currently recall. In all experiments there have been subjects giving a much higher percentage of 'would refer back' answers than average.

The immediate explanation that comes to mind for a subject giving a high percentage of 'would refer back' answers is that they have a lower capacity STM than other subjects; an alternative explanation is that these high 'would refer back' subjects are risk averse (they may or may not also have a lower capacity STM).

This is the first of a two part article that reports on an experiment carried out during the 2011 ACCU conference investigating and analysing their performance on a memory task and measuring their risk attitude.

This first article provides general background on the experiment and discusses the 'risk' related results, while part two discusses subject performance on recall of recently seem assignment statements.

The hypothesis

When people recall information from memory they get a feeling for the confidence level associated with the recalled information. A person who is comfortable taking risks is more likely to make use of information for which they have a low confidence level than a person who is risk averse.

Risk attitude is hypothesized to effect subject performance in a memory recall task in the following three ways:

1. a risk averse subject works more slowly through the experiment questions than a subject who is less risk averse,
2. a risk averse subject works through questions at a similar rate to other subjects but gives a higher percentage of 'would refer back' answers than less risk averse subjects,
3. a risk taking subject will work through the questions at a rate that is faster than their cognitive abilities can reliably support; such behaviour would be expected to generate a higher percentage of incorrect answers than somebody working within the bounds of their cognitive abilities.

Risk

Living in an uncertain world we are all used to taking risks and human risk behaviour has been found to be influenced by many different factors. People vary in their willingness to take risks and an individual's approach to risk may vary across different domains (e.g., play vs. work). [11] People's willingness to take risks within a given domain may depend on their interaction with that domain (e.g., athletes taking greater risks during recreational activities, gamblers more gambling risks, smokers more health risks, etc). [6]

Risks might be taken for the thrill of it, because the risk taker believes the outcome will produce a benefit rather than a cost or because a person is unaware that the outcome of their actions is uncertain.

When making cost/benefit decisions people have been found to give answers that do not agree with the mathematically optimum answer. For instance, people are risk adverse for gains (e.g., given an 85% chance of winning £1,000 or unconditionally winning £800, the majority of subjects have opted for the unconditional option) and risk seeking for losses (e.g., given an 85% chance of loosing £1,000 or unconditionally losing £800, the majority of subjects have opted for the 85% option). [9]

When working on source code what perception of risk [10] do developers have and is any risk analysis they perform correct or misconceived [2]? Your author is not aware of any research investigating risk taking by developers while they are working on source code and so this experiment is something of a jump in the dark.

Measuring risk attitude

In 2002 Weber, Blais and Betz [11] created a questionnaire intended to measure people's risk attitude in six domains: Investing (e.g., money), Health/safety, Recreational, Gambling, Ethical and Social decisions; this set of statements has become widely used and was updated in 2006. [3] The questionnaire consists of various statements each specifying some action; subjects are asked to rate the likelihood they would perform the action, if they found themselves in that situation, on a scale of 1 to 7 (extremely unlikely to extremely likely). The answers are combined to create a measure of risk attitudes.

The 30 statements are:

- Approaching your boss for a raise (S)
- Swimming far out from shore on an unguarded lake or ocean (R)
- Betting a day's income at the horse races (G)
- Investing 10% of your annual income in a moderate growth mutual fund (I)
- Drinking heavily at a social function (H)
- Taking some questionable deductions on your income tax return (E)
- Disagreeing with an authority figure on a major issue (S)
- Betting a day's income at a highstake poker game (G)

DEREK JONES

Derek used to write compilers that translated what people wrote. These days he analyses code to try to work out what they intended to write. Derek can be contacted at derek@knosof.co.uk

- Having an affair with a married man/woman (E)
- Passing off somebody else's work as your own (E)
- Going down a ski run that is beyond your ability (R)
- Investing 5% of your annual income in a very speculative stock (I)
- Going whitewater rafting at high water in the spring (R)
- Betting a day's income on the outcome of a sporting event (G)
- Engaging in unprotected sex (H)
- Revealing a friend's secret to someone else (E)
- Driving a car without wearing a seat belt (H)
- Investing 10% of your annual income in a new business venture (I)
- Taking a skydiving class (R)
- Choosing a career that you truly enjoy over a more secure one (S)
- Riding a motorcycle without a helmet (H)
- Speaking your mind about an unpopular issue in a meeting at work (S)
- Driving while taking medication that may make you drowsy (H)
- Bungee jumping off a tall bridge (R)
- Piloting a small plane (R)
- Walking home alone at night in an unsafe area of town (H)
- Moving to a city far away from your extended family (S)
- Starting a new career in your midthirties (S)
- Leaving your young children alone at home while running an errand (E)
- Keeping a wallet you found that contains £150 (E)

Risk domains are not limited to the six domains addressed by the DOSPERT questionnaire. People have been shown to exhibit other recognizable risk attitudes when operating in different domains, e.g., driving a car. [1]

The DOSPERT questionnaire has been used in a variety of domains (see www.dospert.org), continues to be used and researched and provides a starting point for the empirical investigation of developer risk attitude during software development.

Experimental setup

The experiment was run by your author during a 40 minute lunch time session at the 2011 ACCU conference (www.accu.org) held in Oxford, UK. Approximately 370 people attended the conference, 30 (8.1%; 3 joined 7 minutes after the experiment started) of whom took part in the experiment. Subjects were given a brief introduction to the experiment, during which they filled in background information about themselves, and then spent 20 minutes answering problems. All subjects volunteered their time and were anonymous.

The problem to be solved

The problem to be solved followed the same format as an experiment performed at a previous ACCU conference and the details can be found elsewhere. [7]

Figure 1 is an excerpt of the text instructions given to subjects.

Figures 2 and 3 are an example of one of the problems seen by subjects. One side of a sheet of paper (Figure 2) contained three assignment statements while the second side of the same sheet (Figure 3) contained the five expressions and a table to hold the recalled information. A series of X's were written on the second side to ensure that subjects could not see through to identifiers and values appearing on the other side of the sheet. Each subject received a stapled set of sheets containing the instructions and 40 problems (one per sheet of paper).

The task consists of remembering the value of four different variables and recalling these values later. The variables and their corresponding values appear on one side of the sheet of paper and your response needs to be given on the other side of the same sheet of paper.

1. Read the variables and the values assigned to them as you might when carefully reading lines of code in a function definition.
2. Turn the sheet of paper over. Please do NOT look at the assignment statements you have just read again, i.e., once a page has been turned it stays turned.
3. For each of the following two statements, please indicate the likelihood that you would engage in the described activity or behaviour if you were to find yourself in that situation.
4. You are now asked to recall the value of the variables read on the previous page. There is an additional variable listed that did not appear in the original list.
   - if you remember the value of a variable write the value down next to the corresponding variable,
   - if you feel that, in a real life code comprehension situation, you would reread the original assignment, tick the 'would refer back' column of the corresponding variable,
   - if you don't recall having seen the variable in the list appearing on the previous page, tick the 'not seen' column of the corresponding variable.

Figure 1

```
p = 7 ;
q = 4 ;
r = 9 ;
t = 8 ;
```

Figure 2

For each of the following two statements, please indicate the likelihood that you would engage in the described activity or behaviour if you were to find yourself in that situation.

Provide a rating from Extremely Unlikely to Extremely Likely, using the following scale:

1. Extremely Unlikely
2. Moderately Unlikely
3. Somewhat Unlikely
4. Not Sure
5. Somewhat Likely
6. Moderately Likely
7. Extremely Likely

Swimming far out from shore on an unguarded lake or ocean :

Approaching your boss for a raise :

	remember	would refer back	not seen
q =			
t =			
p =			
s =			
r =			

Figure 3

The question answering task acts as both a time filler for the assignment remember/recall problem and as a method of gathering as much information as possible in the limited time available.

Results

All subjects answered questions for the same amount of time (20 minutes) and were requested to perform at the rate they would use during normal work. In the past some subjects ignored this request and attempted to answer all questions in the booklet they were given. To try to prevent this behaviour occurring the booklet contained many more questions than it was thought subjects could complete and they were told during the introduction about this rationale.

The 30 subjects had a mean of 14.3 years (sd 8.2) experience writing software professionally and answered 582 complete questions (average of 38.8 risk ratings and 96.7 individual assignment answers).

Subject performance will depend on a spectrum of cognitive abilities, one of which is risk attitude. The Spearman rank correlation coefficient was the statistical test used to measure the correlation between the six risk domains in the DOSPERT questionnaire and the various performance measurements described below.

The source code of R program written to analyse the data is available on the web page www.knosof.co.uk/devexperiment/accu11.html along with the (anonymous) data extracted from subject answers.

There are 30 statements in the risk questionnaire and two were randomly chosen, without replacement, to appear in each complete experimental question. Subjects who answered fewer than 15 complete experimental questions will not have given ratings to all risk statements, while subjects answering more than 15 will have rated some risk statements twice (the last answer given was used). Each subject's risk rating answers were averaged within each of the six risk domains.

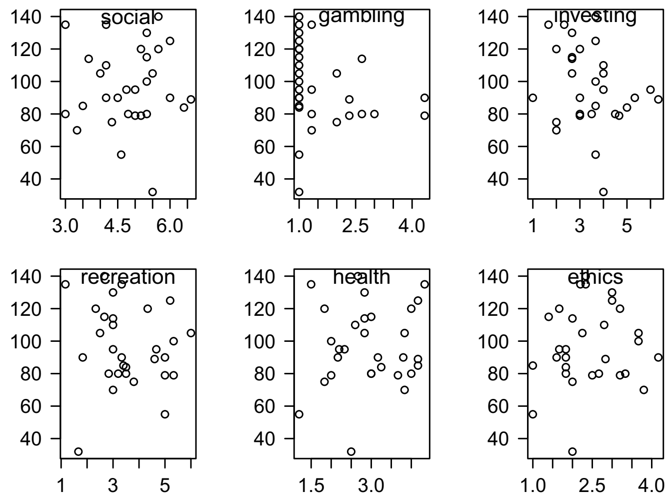


Figure 4

Working more slowly

Figure 4 is a scatter plot of the total number of answers given for the assignment/recall component of the experiment (i.e., all correct, incorrect, 'would refer back' and 'not seen' answers) against the six risk attitudes (each dot represents one subject).

There is no obvious pattern to the subject responses and the Spearman correlation coefficients don't stray too far from zero and have a p-value that is not statistically significant (values not given here to save space and can be obtained by running the R source available on the experiments web page).

Higher percentage of 'would refer back'

Figure 5 is a scatter plot of the percentage of 'would refer back' answers against the six risk attitudes.

There is no obvious pattern to the subject responses and the Spearman correlation coefficients don't stray too far from zero and have a p-value that is not statistically significant. Treating the three subjects having a significantly higher percentage of 'would refer back' answers than other subjects as outliers and not including them in the correlation analysis does not change the analysis (values not given here to save space and can be obtained by running the R source available on the experiments web page).

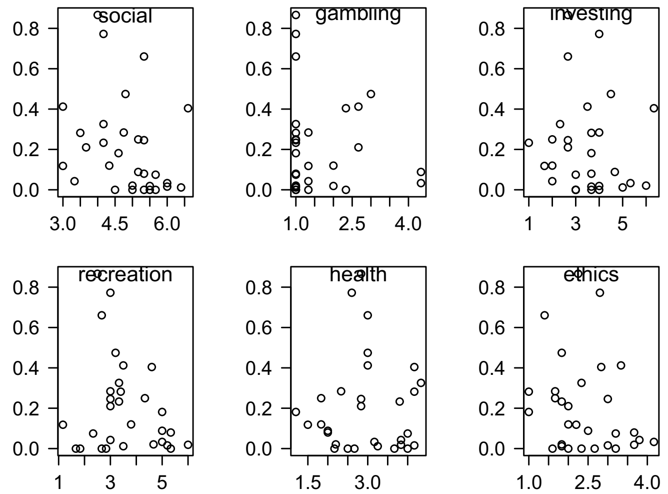


Figure 5

Higher percentage of incorrect answers

Figure 6 is a scatter plot of the percentage of incorrect answers given for the assignment/recall component of the experiment against the six risk attitudes.

There is no obvious pattern to the subject responses and the Spearman correlation coefficients don't stray too far from zero and have a p-value that is not statistically significant (values not given here to save space and can be obtained by running the R source available on the experiments web page).

The gambling risk correlation p-value (0.039) was less than the often used significance level of 0.05. However, if enough correlation tests are performed one will eventually be found that has a p-value below 0.05. The Bonferroni correction adjusts for multiple tests by dividing the significance level by the number of tests, in this case 0.05/6 gives 0.003 as the level below which a p-value will be considered significant. The gambling risk correlation is not significant at the level adjusted for the number of tests.

Summary of risk attitude

Table 1 gives the mean and standard deviation of risk attitudes, over all subjects, for each of the six domains, along with Cronbach's reliability coefficient alpha. Also included are values obtained by Blais[4] from 382 subjects from a variety of backgrounds.

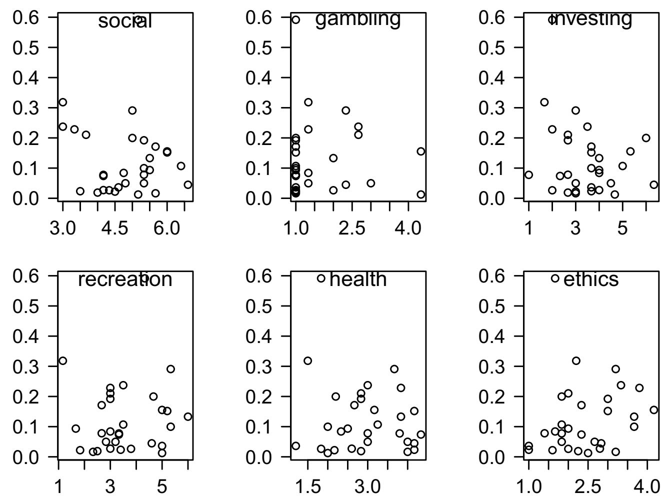


Figure 6

The mean value in all risk domains for the ACCU subjects is less than the corresponding Internet subject means, but still within one standard deviation of the Internet mean values. For six cases there is a 1 in 64 chance of this pattern occurring through random selection.

The standard deviation of the mean in all risk domains for the ACCU subjects is less than the corresponding Internet standard deviations.

For the gambling domain the ACCU subject mean and standard deviation is a lot less than for the Internet subjects.

Cronbach's reliability coefficient alpha is a measure of the internal reliability (or intercorrelation among items) of a set of test scores that are combined to create a single score. The third colum of values in Table .1 is the Cronbach alpha for the ACCU subjects. In those cases where a subject did not rate all of the DOSPERT statements missing ratings were treated as having a value that was the mean of the rating given for that domain by the subject.

A Cronbach alpha less than 0.5 is considered unacceptable (by statisticians), between 0.5 and 0.6 poor, between 0.6 and 0.7 questionable and between 0.7 and 0.8 acceptable. Only one risk domain had an acceptable value, one questionable and two poor, with two being unacceptable. The domain values derived from the Internet subjects all had a Cronbach alpha above 0.77.

It is possible that having to remember assignment information affected subject risk ratings. Like the ACCU subjects the Internet subjects spanned a wide range of ages.

Table 1

Mean and standard deviation for subject responses in each risk domain and Cronbach's alpha for the mean score. Last two columns are reported by Blais [4] based on responses from 382 subjects (after filtering to meet various criteria) recruited via the internet and paid for their time

Domain	Mean	Standard deviation	Cronbach alpha	Internet mean	Internet SD
Social	4.84	0.95	0.58	5.27	1.09
Recreation	3.57	1.22	0.60	3.78	1.57
Gambling	1.6	0.96	0.73	2.85	1.92
Investing	3.44	1.24	0.58	4.12	1.53
Health	2.96	0.89	0.29	3.65	1.41
Ethical	2.42	0.82	0.34	3.14	1.31

Threats to validity

The structure of the experiment is such that giving a 'would refer back' answer has a much lower cost than would have to be paid in real life, i.e., ticking the appropriate answer row vs. spending time searching back through code. A study by Fu and Gray[5] showed that this difference can be significant. They asked subjects to copy a pattern of colored blocks (on a computer-generated display). To carry out the task subjects had to remember the color of the block to be copied and its position in the target pattern, a memory effort. An effort cost was introduced by graying out the various areas of the display where the colored blocks were visible.

These grayed out areas could be made temporarily visible using various combinations of keystrokes and mouse movements. When performing the task, subjects had the choice of expending memory effort (learning the locations of different colored blocks) or perceptual-motor effort (using keystrokes and mouse movements to uncover different areas of the display).

The subjects were split into three groups: one group had to expend a low effort to uncover the grayed out areas, the second acted as a control, and the third had to expend a high effort to uncover the grayed out areas.

The results showed that the subjects who had to expend a high perceptual-motor effort, uncovered grayed out areas fewer times than the other two groups. These subjects also spent longer looking at the areas uncovered, and moved more colored blocks between uncoverings. The subjects faced with a high perceptual-motor effort reduced their total effort by investing in memory effort.

DOSPERT was originally developed and tested using university students as subjects. Based on an average of 14.3 years of professional software development the average age of the ACCU subjects is likely to be in the mid-thirties. Answers to some of the questions may change as people get older (e.g., older people are more likely to be in a stable relationship and having unprotected sex); the effect of age on the reliability of the measurement is not known. The average age of the Internet subjects was probably late twenties (82 were 21 years or younger, 223 between 22 and 35, and 71 were 36 or older).

Conclusion

This experiment failed to find any statistically significant correlation between subject risk attitude in six domains, as measured using the DOSPERT questionnaire, and subject performance in recalling information about a previously seen sequence of assignment statements. Either risk attitude is not a significant factor in recalling information about assignment statements or the attitudes measured by the DOSPERT questionnaire are not applicable.

The ACCU subject risk attitudes were more risk averse than those from the Internet survey, the ACCU subjects were also more self consistent (i.e., the standard deviation in their scores was lower). However, Cronbach alpha values suggest that in some risk domains the single value obtained from combining all subjects ratings is not reliable. ■

Further reading

Statistics Explained by Perry R. Hinton provides a very good introduction to statistics.

The Art of R Programming by Norman Matloff teaches the R language as a language rather than as a tool to use for statistical analysis.

Acknowledgments

The author wishes to thank everybody who volunteered their time to take part in the experiment and the ACCU for making a conference slot available in which to run it.

References

[1] J. Adams. *Risk and Freedom: The record of road safety regulation*. Transport Publishing Projects, 1985.

[2] T. Aven. *Misconceptions of Risk*. Wiley, 2010.

[3] A.-R. Blais and E. U. Weber. 'A domain-specific risk-taking (DOSPERT) scale for adult populations.' *Judgment and Decision Making*, 1(1):33–47, Apr 2006.

[4] A.-R. Blais and E. U. Weber. 'The domain-specific risk taking scale for adult populations: Item selection and preliminary psychometric properties'. *Technical Report TR 2009-203*, Defence R&D Canada, Dec 2009.

[5] W.-T. Fu and W. D. Gray. 'Memory versus perceptual-motor tradeoffs in a blocks world task.' In *Proceedings of the Twenty-second Annual Conference of the Cognitive Science Society*, pages 154–159, Hillsdale, NJ, 2000. Erlbaum.

[6] Y. Hanoch, J. G. Johnson, and A. Wilke. 'Domain specificity in experimental measures and participant recruitment.' *Psychological Science*, 17(4):300–304, Apr 2006.

[7] D. M. Jones. 'Experimental data and scripts for short sequence of assignment statements study.' http://www.knosof.co.uk/cbook/accu04.html, 2004.

[8] D. M. Jones. 'Developer beliefs about binary operator precedence.' *C Vu*, 18(4):14–21, Aug 2006.

[9] D. Kahneman and A. Tversky. 'Choices, values, and frames.' In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 1, pages 1–16. Cambridge University Press, 1999.

[10] P. Slovic. *The Perception of Risk*. Earcthscan Publications Ltd, 2000.

[11] E. U. Weber, A.-R. Blais, and N. E. Betz. 'A domain-specific risk attitude scale: Measuring risk perceptions and risk behaviors.' *Journal of Behavior and Decision Making*, 15(4):263–290, Apr 2002.

An Introduction to CLASP, Part 1: C

Matthew Wilson presents a cure for his command line blues.

This article introduces the **CLASP** command-line handling library. It covers the major requirements and design parameters, along with brief discussion of important implementation features of the variant for C, CLASP/C. Also presented are example usages, and discussion of possible future enhancements. Implementations for other languages will be covered in future articles.

Introduction

CLASP stands for **C**ommand-**L**ine **A**rgument **S**orting and **P**arsing. It is a library to assist with the handling of command-line arguments. (Actually CLASP is a suite of libraries, covering different languages and different command-line-related tasks, of which more, later; the one I'm going to discuss in this article is for use by (and written in) C. In future articles I'll discuss other facilities and languages.)

CLASP exists in part to provide facilities missing in the C standard library, which offers the busy programmer nothing more than an array of string pointers (**`argv`**) of a given length (**`argc`**) as parameters to **`main()`**. Furthermore, it also attempts (and largely succeeds in) the homogenisation of the stark differences in utility between the UNIX and Windows shells. Simply, the UNIX shell(s) expand wildcards passed on the command-line before invoking a given program; the Windows shell does not.

History

Like anyone who has written command-line programs, I have had to write far too many hand-coded command-line argument processing loops over the years. And, like many, I presume, I've written the occasional command-line argument processing library to ameliorate the tedium, and the subtle complexities, of command-line processing. None of these efforts have left me with much satisfaction, until now.

A few years ago I started yet-another-open-source-project, **systemtools** (http://www.sourceforge.net/projects/systemtools). I was (and still am, though more for academic than commercial reasons) interested in exploring the differences between programming techniques when solving the same problems in different languages (e.g. C++ vs C# vs Ruby), or with different libraries (e.g. Boost vs. FastFormat vs. POCO vs. …). Since I was at the time still spending most of my time on Windows I thought I could kill two birds with one stone by (re-)writing many missing standard (i.e. present on UNIX) and several proprietary tools along these lines. I also thought there was a book (or several) in it, and still hope to explore that in the future.

The first project tackled was cat, in C. For the first version I wrote absolutely everything from scratch – nothing other than the C standard library – as a pedagogical exercise. I then started about refining and refactoring, and pushing it towards a standard I would be happy to write commercially. With each subsequent project – tee, ws, etc. – I allowed more use of third-party libraries, and explored different languages. The dominant common theme to all was that the handling of command-line arguments was a giant, verbose, tedious, repetitive, pain in the neck. Thus, CLASP was born.

Since that time, I have been too busy with commercial things to advance my systemtools project (and its associated book idea(s)). However, I have needed to write a multitude of non-trivial command-line applications (in C, C++, C#, and Ruby), and so the nascent CLASP has had quite the workout. Although it's not yet at a 1.x stage – part of the reason for writing this article is to solicit opinion and help to take it to that level – it is now at a level where I am willing to let it be seen and discussed and (hopefully) used by others.

Example program

In order to talk meaningfully about the issues involved in command-line handling, I will posit a hypothetical program **`prg`**, which behaves like a classic UNIX filter, with the following usage that has just enough complexity to shed light on most of the issues of concern (Listing 1).

```
Synopsis:
 prg [... options ...] [<in-file> | -]
     [<out-file> | -]

Options:

 filtering:

 -a
 --all
   equivalent to -bclt

 -b => --strip-blanks=all
 -B => --strip-blanks=no
 --strip-blanks=<value> as one of
   {all|multiple|no}; default value=multiple
   causes blank lines in the input to be stripped

 -c => --strip-comments=yes
 -C => --strip-comments=no
 --strip-comments=<value> as one of {yes|no};
   default value=yes
   causes comments in the input to be stripped

 -l => --trim-leading-whitespace=yes
 -L => --trim-leading-whitespace=no
 --trim-leading-whitespace=<value> as one of
   {yes|no}; default value=yes
   causes leading whitespace to be trimmed

 -t => --trim-trailing-whitespace=yes
 -T => --trim-trailing-whitespace=no
 --trim-trailing-whitespace=<value> as one of
   {yes|no}; default value=yes
   causes trailing whitespace to be trimmed

 history:

 -h <value>
 --history-file=<value>
   specifies a file into which will be written
   the history of each modification made to the
   input stream that will cause the output to
   differ

 -e
 --relative
   use relative paths in history
```

Listing 1

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.

Standard UNIX flags `--help`, `--version`, and `--` and the standard flag/value `-` are also to be supported. (See following discussions for `-` and `--`.)

Some example usages might be:

```
$ prg --version
$ prg src.cpp stripped.cpp
$ prg --all src.cpp stripped.cpp
$ prg -aL -h actions.log src.cpp stripped.cpp
$ prg -aL --history-file=actions.log src.cpp
stripped.cpp
$ prg --trim-leading-whitespace:yes
--strip-blanks:yes src.cpp stripped.cpp
```

Manually writing command-line handling even for a relatively-simple program such as this involves a number of non-trivial challenges, including:

- processing arguments array argv in range [1 .. `argc`);
- detecting arguments with one or two hyphens;
- detecting `-a`, `--all`, `-b`, `-B`, `-c`, `-C`, `-l`, `-L`, `-t`, `-T`, `-e`, `--relative`, `--help`, and `--version` arguments, and adjusting program behaviour-controlling state accordingly;
- detecting two-hyphen arguments `--strip-blanks`, `--strip-comments`, `--trim-leading-spaces`, `--trim-trailing-spaces`, and `--history-file`, by search for the equals sign (or a colon, if that is preferred/allowed) coupled with `strncmp()`, and then ascertaining following value, if present, and recording;
- looking ahead to next argument, if present, when processing `-h`;
- detecting `--`, and changing remaining parsing behaviour accordingly;
- detecting `-`;
- detecting arguments with zero hyphens (for `<in-file>` and `<out-file>`);
- dealing with unrecognised one/two-hyphen-prefixed arguments;
- dealing with insufficient/surplus zero-hyphen-prefixed arguments;
- writing out usage (in form similar to above), keeping consistent with parsing.

Most likely you'll have a `for`-loop with nested `switch`-statements, along with several parsing state variables indicating whether the next argument will be a value (for the `-h` option) and whether the `--` argument has been specified (and so all subsequent arguments are to be treated as if they have no hyphen prefixes). There may be similar/identical sections in separate switch statement for dealing with one- and two-hyphen-prefixed arguments that have the same/similar roles.

As well as the considerable challenge in getting and keeping the logic correct, you will also have the onerous task of keeping the usage information printing code consistent with the evolving set of arguments. I am sure you can imagine the boring and error-prone nature of this kind of coding. Something better is required.

CLASP is intended to help you permanently eschew this kind of effort, as well as to offer other important features and characteristics as outlined in the following sections.

Design principles

There are a number of design principles that inform the characteristics of CLASP and why (I believe) it is worth creating another such library (rather than, say, using `getopt` and Windows-ports thereof). In no particular order:

- Support Multiple Languages.
- Support Multiple Platforms, to the degree that is possible.
- 100% Pure Language Implementations. The variant for each language should be written entirely in that language, except where there's a persuasive reason to the contrary.
- Expand Wildcards on Windows, and do so implicitly.
- Never Fail to Parse a Command-line, excepting memory exhaustion or programmer error.
- Distinguish between *Flags*, *Options*, and *Values*.
- Define *Flags* and *Options* in Declarative Form, within the bounds of the given language.
- Support Short-form *Flags* and *Options*.
- Support Long-form *Flags* and *Options*.
- Support (short-form) *Flags* as aliases for (long-form) *Flags*.
- Support *Flags* as aliases for value'd *Options*.
- Support Grouped (Single-letter) *Flags*.
- Handle the `-` (Single-hyphen) Special Argument.
- Handle the `--` (Double-hyphen) Special Argument.
- Support default *Values* for *Options*.
- Group and order the *Flags*, *Options*, and *Values*.
- Render Argument Order (Almost) Irrelevant.
- Retain Original Command-Line Context Information.
- 'Reuse' aliases for Generating Usage Information.
- Validate Arguments Under Programmatic Control.
- Detect Whether *Flags* and *Options* are 'Recognised'.
- Detect Whether *Flags*, *Options*, and *Values* are 'Used'.
- Combine *Flags* into Programmatic Bitmasks.
- Follow library good practice, in implementation, including being highly testable.

Support multiple languages

As I mentioned above, the original systemtools research/writing project was intended to cover multiple languages, so CLASP was always required to do so. Furthermore, since using CLASP, heavily, in C, C++, and C# in recent years, each time I go to Python or Ruby I miss it tremendously.

While not getting beyond myself – I do not suggest that CLASP is the ultimate in command-line processing – I do now find that some of its facilities are invaluable in simplifying both the writing of a command-line program and, by dint of its flexibility in detecting and assigning flags, options, and values, how flexible, and therefore how usable, resulting programs can be. This is pretty subjective, however, and I look forward to feedback (especially the negative) from you, gentle readers.

some of its facilities are invaluable in simplifying both the writing of a command-line program

Support multiple platforms.

This is pretty much a given, except to say that if you're writing C or C++ exclusively on UNIX it may be preferable to stick with `getopt` (or its slightly more powerful derivative library `getopt_long`), despite its limitations (discussed below).

Furthermore, several other languages (including D, Perl, Python, and Ruby) have their own cross-platform `getopt`/`getopt_long`-analogues and other (non-`getopt`-like) facilities, so if you're comfortable with the `getopt` paradigm (and I don't persuade you otherwise in this series of articles) you may be sufficiently well-served already.

100% pure language implementations

The principle is simple. For a variety of reasons, if it's possible to write the implementation of a library in the language that it's going to express, then we should do so. Reasons include: willingness to use it by others; simplifying build; simplifying change; reduced dependencies on other third-party libraries; reduced need to ship binaries. Only in the minority of cases is it preferable to have the library implementation in a language different to that it is expressing.

So, the CLASP/C variant is written in C. The CLASP/.NET variant is written in C#. The only exception so far is that the CLASP/C++ variant is a thin (header-only) wrapper over C.

Expand wildcards on Windows.

The biggest difference between command-line processing on UNIX and Windows is that, by default, the UNIX shell expands any wildcard arguments (in terms of the relative and absolute current extant file-system contents) before invoking the given command, whereas the Windows shell does no expansion. In general, UNIX applications need not concern themselves about wildcard arguments. Conversely, every Windows program that receives file paths as command-line arguments *must do its own expansion*. Obviously, this is a huge impost on Windows program functionality.

Consider a directory containing the files `list.c`, `vector.c`, `containers.h`, `program.c`. If you type **`"cc *.c"`** in, say, bash on UNIX, **`cc`** will be invoked with the command-line **`"cc list.c program.c vector.c"`**. On Windows you'll just get **`"cc *.c"`**.

It seemed almost pointless to write a cross-platform command-line processing library without tackling this biggest of failings (for Windows). So, when building CLASP on Windows, the **recls** library (http://www.recls.org/) facilities are used inside the library to expand any wildcard arguments. If this is not required, there are two mechanisms.

You can define the preprocessor symbol **`CLASP_CMDLINE_ARGS_NO_RECLS_ON_WINDOWS`** during the library compilation, in which case a variant of CLASP is built entirely independent of **recls**, and will never expand wildcards. Alternatively, you can specify the runtime parsing flag **`CLASP_F_DONT_EXPAND_WILDCARDS_ON_WINDOWS`** to suppress expansion for a given program; in this case, you will still have to link to the **recls** library, even though it's not used.

On UNIX, wildcard expansion is suppressed by enclosing a wildcard argument within single quotes, as in **`"cc '*.c'"`**, in which case the command will be **`"cc *.c"`**. Thankfully, the Windows shell passes single-quoted arguments through to the program without stripping the single-quotes, so CLASP detects this and emulates UNIX by not expanding any such quoted arguments. Again, this can be altered (at runtime) if required by specifying the parsing flag **`CLASP_F_DO_EXPAND_WILDCARDS_IN_APOSQUOTES_ON_WINDOWS`**, in which case the single quotes will be ignored and any wildcards expanded regardless.

Never fail to parse a command-line.

In another article in a loosely-related series – of which this is the first – in the coming instalments of *CVu* I will be looking at anatomies of programs, including command-line programs. I will be considering the 'levels' at which it's appropriate to invoke different infrastructure services common to programs. Command-line parsing is one of the more fundamental, and, as such, it should have as few failure modes as possible.

Consequently, CLASP (the C variant, at least) can fail only as a result of (i) memory exhaustion, or (ii) programmer error in the specification of aliases. In every other circumstance a meaningful form of the program's command-line will be obtained (albeit the partition of flags, options, and values may be a surprise if you've mis-specified the aliases).

Considering our **`prg`** example, if the short-form alias for option --history-file was **`-L`** rather than **`-h`** (as it was during its development while preparing this article!), then the call to parse the command-line would fail, and a message reporting the programmer error would be issued.

Distinguish between Flags, Options and Values

In my reckoning, there are three types of command-line arguments, which I define as follows:

- ***Flags***, which are arguments that act like Boolean switches. Flags begin with one or more hyphens, and are always comprised of only a single argument. For example: **`--help`**, **`-e`**, **`-a`**.
- ***Options***, which are arguments that can represent a range of options. Options begin with one or more hyphens, and can take three forms. Two of these forms are comprised of only a single argument, as in **`--history-file=abc`** or **`--history-file:abc`** (or **`-h=abc`** or **`-h:abc`**), and **`-habc`**. The third form is comprised of an argument pair, as in **`-h abc`** or **`--history-file abc`**.
- ***Values***, which are arguments that do not begin with hyphens, such as the input file path and output file path values to be specified to **prg**.

(For the remainder of the article I will capitalise and italicise them, to avoid ambiguity with other uses of the words.)

Flags have only a name. *Values* have only a value. *Options* have both a name and a value (although the value may be defaulted).

Note that CLASP treats the prefixing hyphen(s) as part of the name of a *Flag* or *Option*.

Currently, CLASP support *Flags* and *Values* completely, and supports the first (**`--history-file=abc`**) and last (**`-h abc`**) form of *Options*; it does not currently support the second (**`-habc`**) form.

Depending on requirements and sophistication of implementation, a given program may have any combination of *Flags* and/or *Options* and/or *Values*, including none at all.

Furthermore: some of these may be optional, others mandatory; some may have aliases, others not; some may be specified only once, others multiple times; some may have a specific ordering, others not. These permutations are all supported by CLASP, as discussed in the following sections.

With respect to **prg**, we would delineate its arguments as follows:

- Flags: **`-a`**, **`--all`**, **`-b`**, **`-B`**, **`-c`**, **`-C`**, **`-l`**, **`-L`**, **`-t`**, **`-T`**, **`-e`**, **`--relative`**, **`--help`**, and **`--version`**;
- Options: **`--strip-blanks`**, **`--strip-comments`**, **`--trim-leading-spaces`**, **`--trim-trailing-spaces`**, and **`--history-file`**;
- Values: **`<in-file>`** (or **`-`**), and **`<out-file>`** (or **`-`**)

Define Flags and Options in declarative form

In all languages, the intention is to have the *Flags* and *Options* be defined as declaratively as possible. Listing 2 shows a complete sample implementation of the **prg** program, including the declaration of the 'aliases array' **`ALIASES`** in which its *Flags* and *Options* are defined.

Listing 2

```
#include <systemtools/clasp/clasp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static clasp_alias_t const ALIASES[] =
{
  /* Filtering behaviour flags/options */
  CLASP_GAP_SECTION("filtering:"),

  CLASP_FLAG("-a", "--all",
     "equivalent to -bclt"),

  CLASP_OPTION(NULL, "--strip-blanks=multiple",
     "causes blank lines in the input to be
     stripped", "|all|multiple|no"),
  CLASP_OPTION_ALIAS("-b", "--strip-blanks=all"),
  CLASP_OPTION_ALIAS("-B", "--strip-blanks=no"),
```

Listing 2 (cont'd)

```
  CLASP_OPTION(NULL, "--strip-comments=yes",
     "causes comments in the input to be
     stripped", "|yes|no"),
  CLASP_OPTION_ALIAS("-c",
     "--strip-comments=yes"),
  CLASP_OPTION_ALIAS("-C",
     "--strip-comments=no"),

  CLASP_OPTION(NULL,
     "--trim-leading-whitespace=yes",
     "causes leading whitespace to be trimmed",
     "|yes|no"),
  CLASP_OPTION_ALIAS("-l",
     "--trim-leading-whitespace=yes"),
  CLASP_OPTION_ALIAS("-L",
     "--trim-leading-whitespace=no"),

  CLASP_OPTION(NULL,
     "--trim-trailing-whitespace=yes",
     "causes trailing whitespace to be trimmed",
     "|yes|no"),
  CLASP_OPTION_ALIAS("-t",
     "--trim-trailing-whitespace=yes"),
  CLASP_OPTION_ALIAS("-T",
     "--trim-trailing-whitespace=no"),

  /* History flags/options */

  CLASP_GAP_SECTION("history:"),

  CLASP_OPTION("-h", "--history-file", "specifies
     a file into which will be written the
     history of each modification made to the
     input stream that will cause the output to
     differ", ""),
  CLASP_FLAG("-e", "--relative",
     "use relative paths in history"),

  /* Standard flags */

  CLASP_GAP_SECTION("standard flags:"),

  CLASP_FLAG(NULL, "--help",
     "show this help and quit"),
  CLASP_FLAG(NULL, "--version",
     "show version and quit"),

  CLASP_ALIAS_ARRAY_TERMINATOR
};
int main(int argc, char** argv)
{
  clasp_arguments_t const* args;
  unsigned const cflags = 0;
  int const cr = clasp_parseArguments(cflags,
     argc, argv, ALIASES, NULL, &args);

  if(cr != 0)
  {
    fprintf(stderr,
       "failed to initialise : %s (%d)\n",
       strerror(cr), cr);
    return EXIT_FAILURE;
  }
  else
  {
    if(clasp_flagIsSpecified(args, "--help"))
    {
      clasp_show_usage(
        NULL
      , ALIASES
```

Listing 2 (cont'd)

```
      , "prg"
      , "SystemTools
        (http://systemtools.sourceforge.net/)"
      , "Copyright Matt Wilson"
      , "Exercises CLASP for CVu"
      , "prg [... options ...] [<infile> | -]
        [<outfile> | -]"
      , 1, 0, 1 /* version: maj, min, rev */
      , clasp_show_header_by_FILE,
        clasp_show_body_by_FILE, stdout
      , 0
      , 76 /* console-width */
      , -2 /* indent size */
      , 1  /* blank line between args? */
      );
    }
    else
    {
      /* Dump out the arguments, in groups */
      puts("");
      printf("flags:\t%lu\n", args->numFlags);
      { size_t i; for(i = 0; i != args->numFlags;
        ++i)
      {
        clasp_argument_t const* const flag =
           args->flags + i;

        /* Treat strings as slices {len+ptr} */
        printf("flag-%02d:\t%.*s\t%.*s\n", i,
           (int)flag->givenName.len,
           flag->givenName.ptr,
           (int)flag->resolvedName.len,
           flag->resolvedName.ptr);
      }}

      puts("");
      printf("options:\t%lu\n",
         args->numOptions);
      { size_t i; for(i = 0;
         i != args->numOptions; ++i)
      {
        clasp_argument_t const* const option =
           args->options + i;

        /* Treat strings as C-style strings */
        printf("option-%02d:\t%s\t%s\t=\t%s\n",
           i, option->givenName.ptr,
           option->resolvedName.ptr,
           option->value.ptr);
      }}

      puts("");
      printf("values:\t%lu\n", args->numValues);
      { size_t i;
           for(i = 0; i != args->numValues; ++i)
      {
        clasp_argument_t const* const value =
           args->values + i;

        /* Treat strings as slices {len+ptr} */
        printf("value-%02d:\t%.*s\n", i,
           (int)value->value.len,
           value->value.ptr);
      }}
    }
    clasp_releaseArguments(args);

    return EXIT_SUCCESS;
  }
}
```

Listing 3

```
static void Main(string[] args)
{
  Alias[] aliases =
  {
    new Alias(ArgumentType.Flag, null,
        "--help", "displays this help"),
    new Alias(ArgumentType.Flag, null,
        "--version", "show version and quit"),
    . . .
  };
  Arguments.InvokeMain(args, aliases, ToolMain);
}
```

With the use of macros such as **`CLASP_FLAG()`**, **`CLASP_OPTION()`**, and so on, it is a straightforward matter to define a constant static non-local (file/namespace-scope) alias array, in which the short- and/or long-form names, possible values, default values, and help-text can be specified for all *Flags* and *Options*.

The general intention of focusing on a declarative form is twofold: to make as clear as is possible (within the bounds of a given language) the *Flags* and *Options* while minimising the risks of programming mistakes; to be able to reuse this information in the generation of usage (**`--help`**) information. I'll discuss the latter more below.

Currently, CLASP/C++ uses the same declarative form as CLASP/C. The CLASP/C# library currently supports syntax such as Listing 3.

Note that this is subject to change before I release the C# library and write an article about it early next year: in particular, I hope to use C# attributes to effect a significant reduction in 'code', maximising the declarative information.

Support short-form Flags and Options

Some program(mer)s prefer short-form *Flags* and *Options* – such as **`-B`**, **`-e`**, **`-h`** – which have only single letter names and, by convention, a single prefixing hyphen. CLASP supports this form.

Support long-form Flags and Options

Sophisticated programs can have many *Flags* and *Options*, in which case 26 (or 52, if mixing case) names may be insufficient. Consequently, some program(mer)s prefer long-form *Flags* and *Options* – such as **`--relative`**, **`--history-file`** – which have one or more (usually non-contracted) word names and, by convention, two prefixing hyphens. CLASP supports this form.

Support (short-form) *Flags* as aliases for (long-form) *Flags*

It's very common for programs to support both short- and long-form *Flags*, and for some/all of the short-form *Flags* to act as aliases for some/all of the more frequently-used long-form *Flags* as a convenience to 'power users', as in **prg**'s **`-e`** alias for **`--relative`**. CLASP supports *Flag* →*Flag* aliasing.

Support *Flags* as aliases for value'd Options

Though less common, it's still widely seen practice to use a (usually short-form) *Flag* as an alias for a value'd *Option*, as in **prg**'s **`-b`** and **`-B`** *Flags*, which are aliases for **`--strip-blanks=all`** and **`--strip-blanks=no`**, respectively. CLASP supports *Flag* →*Value*'d-*Option* aliasing.

Support grouped (single-letter) Flags

As a further convenience to lazy power users, some program(mer)s allow grouping of single-letter *Flags* (and *Options*, in rare cases), in a single argument, e.g. **`"tar -cvf …"`**. CLASP partially supports this: it supports the grouping of single-letter *Flags*; it does not support grouping of *Options*.

Handle the - (single-hyphen) special argument

Many filters read from an input stream and write to an output stream, and these often support reading from and write to files, as well as reading from standard input and writing to standard output, the latter behaviour often being selected implicitly in the case where input and output paths are not given as arguments. Consider the following three commands:

```
$ prg src.cpp stripped.cpp
$ prg src.cpp
$ prg
```

Sensible behaviour is available in all three cases, respectively: read from `src.cpp` and write to `stripped.cpp`; read from `src.cpp` and write to standard output; read from standard input and write to standard output. However, the case that is not easily catered for by this scheme is to read from standard input and write to a named file. Thus, by convention, an argument consisting of a solitary hyphen is treated as an instruction to read from standard input, as in:

```
$ prg - stripped.cpp
```

which is a request to read from standard input and write to `stripped.cpp`.

Sometimes, a second occurrence of the solitary hyphen argument is treated as an explicit instruction to write to standard output, as in:

```
$ prg - -
```

CLASP supports the special single-hyphen argument, and also supports the occurrence of multiple instances. By default, it is treated as a flag (with the name **`"-"`**), but will be treated as a value (and uniquely distinguishable as such by having both the name **`"-"`** and the value **`"-"`**) if the parsing flag **`CLASP_F_TREAT_SINGLEHYPHEN_AS_VALUE`** is specified at runtime.

Handle the -- (double-hyphen) special argument

Sometimes filenames (and other names) that are to be submitted as *Values* may begin with a hyphen, and so would, by default, be interpreted as *Flags* (or known *Options*), with unwanted consequences. By convention a solitary double-hyphen argument **`"--"`** is interpreted as a directive to treat all following arguments as *Values* regardless of any prefixing hyphens. CLASP supports this by default (and 'swallows' the argument in the process); specifying the parsing flag **`CLASP_F_DONT_RECOGNISE_DOUBLEHYPHEN_TO_ START_VALUES`** at runtime will suppress this behaviour, in which case it will be interpreted and available to the program as a flag with the name **`"--"`**. (In most cases, it's better to allow the default treatment, and have the user specify the double-hyphen sequence twice – the first to change interpretation mode, the second as the actual value **`"--"`** – since this is the convention on UNIX.)

Support default values for Options

Another common practice is to support default values for *Options*, as seen above with four of **prg**'s *Options*, such as **`--strip-blanks`**, whose default values is **`yes`**.

In many instances of the application of this technique, it is used for controlling a Boolean behaviour, and is simply an alternative to having a default behaviour coupled with a *Flag* to select the non-default behaviour: **prg**'s **`--relative`** *Flag* is an example of this, where (we may reasonably presume) the application uses absolute paths by default. I'm not keen on this form, although I note in my own work that I do tend to use it in more complex applications, usually those with many *Flags*/*Options*.

In cases where there are more than two options, it is less common to see, but arguably makes sense. Consider a program that issues progress information (whether by diagnostic logging or contingent reporting), according to a 'verbosity' setting. Let us assume four settings: verbose, chatty, terse, silent. Let us further assume that the application default is terse.

One option is to have a **`--verbosity`** *Option*, without defaults. Unless you specify **`--verbosity=<some-level-other-than-terse>`**

you will get the application default terse. Getting something other than what you want requires quite a long argument.

Another alternative is to have a **`--verbose`** *Option*, which defaults to **`verbose`**. Thus, you can specify **`--verbose`** to get verbose, or **`--verbose=silent`** to get silent, and so forth.

I'm not sure there's a whole lot to choose between them. Being the pedantic soul I am, I would probably use the **`--verbosity`** *Option*, and have a *Flag* **`--verbose`** that acts as an alias for the *Value*'d-*Option* **`--verbosity=verbose`**.

The good news is that I don't have to decide which way you should do it. CLASP supports all the variations just discussed.

Group and order the Flags, Options and Values

Even with command-line parsing facilities that don't allow interleaving of *Values* with *Flags* and *Options*, programs must, in general, be able to work with arbitrary ordering of *Flags* and *Options*. But it is often convenient to process separately the *Flags*, the *Options*, and the *Values*. This is where the Sorting part of CLASP comes in: after all the arguments have been parsed and placed in an array, the array is sorted according to argument type.

Thus, the CLASP/C **`clasp_arguments_t`** (see Listing 6) type has field pairs {**`numFlags`** and **`flags`**}, {**`numOptions`** and **`options`**}, and {**`numValues`** and **`values`**}, representing the slices of the overall arguments array in which the *Flags*, *Options*, and *Values* reside. As can be seen in Listing 2, the three argument groups – *Flags*, *Options*, and *Values* – can be easily processed separately.

you can process the requisite aspects of interest of your command-line without having to concern yourself with the precise ordering presented on the command-line

Furthermore, since it's occasionally convenient to treat *Flags* and *Options* together, the two arrays are adjacent, so there is also the field pair {**`numFlagAndOptions`** and **`flagsAndOptions`**}. And to round it out, there is a field pair {**`numArguments`** and **`arguments`**} that represents all parsed arguments.

Thus, as we'll see later in this article – and in subsequent articles covering other languages – you can process the requisite aspects of interest of your command-line without having to concern yourself with the precise ordering presented on the command-line. Unless, that is, you deem it of interest.

Render argument order (almost) irrelevant

In some programs, having a prescribed argument order is meaningful. Both Subversion's svn and Bazaar's bzr programs take a first argument to specify what 'service' is being invoked, as in **`"bzr log --forward"`**. In such cases, the service-specific argument(s) cannot occur before the service name – **`"bzr --forward log "`** – for what are clearly sound reasons.

However, in other cases, argument order can be pointlessly restrictive. Certainly, now that I understand some of the complexities involved in command-line argument processing, I see why implementers have opted to place that restriction. But it makes some programs harder to use than they need be.

Furthermore, I have occasionally found that the ability to use *Flags* in the midst of *Values* can be very useful, as in the following example featuring one of my source-analysis tools:

```
$ vdd -e src '*.h' '*.c' -X src/3pty '*.c'
```

Listing 4

```
/* file: systemtools/clasp/clasp.h */
CLASP_CALL(size_t)
clasp_reportUnrecognisedFlagsAndOptions(
  clasp_arguments_t const* args
, clasp_alias_t const* aliases
, clasp_argument_t const** nextUnrecognisedArg
, unsigned nSkip /* = 0 */
);
```

Here, **`vdd`** is instructed to report (according to **`-e`**, a mode flag that means ignore comments and space) source statistics on every `.h` and `.c` file under the directory `src`, excluding (**`-X`**) all `.c` files under `src/3pty`.

CLASP provides access to the original ordering in two ways. The simple way is that each parsed argument structure includes an integer field representing the index of the original, unparsed argument in the program command-line. Since the arguments structure includes a copy of the original **`argc`** and **`argv`** it is possible to access the precise original argument corresponding to any translated argument. The most common use of this facility is for providing targeted contingent reports in the case of invalid (combinations of) grouped flags. We can test for (the first of) any unrecognised arguments, using the API function **`clasp_reportUnrecognisedFlagsAndOptions()`** (Listing 4).

The code in Listing 5 determines whether any *Flags* or *Options* are unrecognised, and displays the original command-line argument corresponding to the first unrecognised argument.

The following command-line:

```
$ prg -bcd
```

would produce the following

```
invalid arg: -bcd
```

The complex way is to specify the parsing flag **`CLASP_F_PRESERVE_ORIGINAL_ARGUMENT_ORDER`**, which prevents the sorting, thereby preserving the original argument order. (Note that two-argument options are still presented a single translated *Option*). In this case, the values of the field pairs {**`numFlags`** and **`flags`**}, {**`numOptions`** and **`options`**}, {**`numValues`** and **`values`**}, and {**`numFlagAndOptions`** and **`flagsAndOptions`**} are all {**`0`**, **`NULL`**}; only {**`numArguments`** and **`arguments`**} are useful.

Retain original command-line context information

Any given parsed argument can differ from its source argument in several respects, depending on the language. First, in C (and C++) the zeroth string always contains the program path; in C# and other languages it does not. Consider the command-line:

```
$ prg src.cpp stripped.cpp
```

In C and C++, the indexes of the arguments **`src.cpp`** and **`stripped.cpp`** will be **`1`** and **`2`**, respectively.

Second, due to aliasing, the parsed name of an argument may differ from that specified on the command-line. For example, in:

```
$ prg -e src.cpp stripped.cpp
```

the flag **`-e`** will actually be parsed into the **`--relative`** name.

Third, with grouped *Flags* and two-argument *Options*, the actual source argument will be shared by several translated arguments. For example, in:

Listing 5

```
clasp_argument_t const* arg;
size_t const n =
clasp_reportUnrecognisedFlagsAndOptions(args,
   ALIASES, &arg, 0);
if(0 != n)
{
  fprintf(stderr, "invalid arg: %s\n",
     args->argv[arg->cmdLineIndex]);
}
```

Listing 6

```
struct clasp_slice_t
{
  size_t              len;
  clasp_char_t const* ptr;
};
enum clasp_argtype_t
{
    CLASP_ARGTYPE_INVALID = 0
  , CLASP_ARGTYPE_FLAG
  , CLASP_ARGTYPE_OPTION
  , CLASP_ARGTYPE_VALUE
};
struct clasp_argument_t
{
  clasp_slice_t   resolvedName;
  clasp_slice_t   givenName;
  clasp_slice_t   value;
  clasp_argtype_t type;
  int             cmdLineIndex;
  int             numGivenHyphens;
  int             aliasIndex;
  int             reserved0;
};
struct clasp_arguments_t
{
  size_t                      numArguments;
  clasp_argument_t const*     arguments;
  size_t                      numFlagsAndOptions;
  clasp_argument_t const*     flagsAndOptions;
  size_t                      numFlags;
  clasp_argument_t const*     flags;
  size_t                      numOptions;
  clasp_argument_t const*     options;
  size_t                      numValues;
  clasp_argument_t const*     values;
  int                         argc;
  clasp_char_t const* const*  argv;
};
struct clasp_alias_t
{
  clasp_argtype_t     type;
  clasp_char_t const* name;
  clasp_char_t const* mappedArgument;
  clasp_char_t const* help;
  clasp_char_t const* valueSet;
  int                 bitFlags;
};
#define
CLASP_F_DONT_RECOGNISE_DOUBLEHYPHEN_TO_START_VALU
ES
#define CLASP_F_TREAT_SINGLEHYPHEN_AS_VALUE
#define CLASP_F_DONT_EXPAND_WILDCARDS_ON_WINDOWS
#define
CLASP_F_DO_EXPAND_WILDCARDS_IN_APOSQUOTES_ON_WIND
OWS
#define CLASP_F_PRESERVE_ORIGINAL_ARGUMENT_ORDER
```

```
$ prg -eb src.cpp stripped.cpp
```

the composite flag **`-lt`** will be parsed into the *Flag* **`--relative`** and the *Option* **`--strip-blanks=all`**.

Information about the transformations carried out by CLASP's parsing is captured in the members of the **`clasp_argument_t`** structure (see Listing 6). Consider the third example given above. After parsing, there will be four arguments, with fields as shown in Table 1.

The information provided allows the programmer to reconstruct part/all of the original command-line, should that be necessary, e.g. for displaying precise contingent reports to the user in the case of unrecognised/inappropriate command-line options.

'Reuse' aliases for generating usage information

Probably the most tedious aspect of dealing with command-line processing is creating 'usage' information – usually in response to the **`--help`** flag. In particular, the customary lack-of-DRY-SPOT redundancy is hair-tearingly onerous to get right and keep right.

One of the major aims with CLASP is to ensure that the information used to specify what *Flags* and *Options* are recognised by a program is also used, in a largely automatic fashion, in producing usage information.

Table 1

Field	arguments[0]	arguments[1]	arguments[2]	arguments[3]
type	FLAG	OPTION	VALUE	VALUE
givenName	-eb	-eb		
resolvedName	--relative	--strip-blanks		
value		all	src.cpp	stripped.cpp
cmdLineIndex	1	1	2	3
numGivenHyphens	1	1	0	0
aliasIndex	16	3	-1	-1

Although this aspect of the library is less polished, and probably more amenable to refinement in response to your feedback, there already exist powerful facilities for doing this, as shown in the call to **`clasp_show_usage()`** in Listing 2. This call produces a usage list similar to that shown in the section introducing **prg**.

Validate arguments under programmatic control

There are several ways in which a program's command-line arguments can be mismatched with the program's expectations:

1. One or more **Missing Required Arguments**;
2. One or more **Unrecognised Arguments**;
3. One or more **Duplicate Recognised Arguments**;
4. **Invalid Combinations of Arguments**; and
5. **Invalid Details of Arguments**.

In my opinion, it is too hard and/or too restrictive – at least in C/C++ – to have a command-line parsing library detect and police all these invalid command-line arguments. While it may be easy to consider how the arguments for any single program might be validated, a general solution is too hard (for me at least) to contemplate.

Consequently, I have focused on having CLASP make it simple for each program to detect and police them under programmatic control.

Missing Required Arguments may be detected either by directly examining the contents of the various field pairs discussed above, or by invoking API functions, such as **`clasp_findFlagOrOption()`**.

Listing 7

```
clasp_argument_t const* arg;
size_t nSkip = 0;
size_t const n =
clasp_reportUnrecognisedFlagsAndOptions(args,
   ALIASES, &arg, nSkip);
if(0 != n)
{
  fprintf(stderr,
     "%lu unrecognised argument(s):\n", n);
  do
  {
    fprintf(stderr,
       "\tunrecognised argument: %s\n",
       args->argv[arg->cmdLineIndex]);
  }
  while(0 !=
        clasp_reportUnrecognisedFlagsAndOptions(
        args, ALIASES, &arg, ++nSkip));
}
```

Listing 8

```
clasp_argument_t const* arg;
size_t nSkip = 0;
size_t const n = clasp_reportUnusedValues(args,
   &arg, nSkip);
if(0 != n)
{
  fprintf(stderr,
     "%lu unused argument(s):\n", n);
  do
  {
    fprintf(stderr, "\tunused argument: %s\n",
       args->argv[arg->cmdLineIndex]);
  } while(0 != clasp_reportUnusedValues(args,
    &arg, ++nSkip));
}
```

Invalid Combinations of Arguments and **Invalid Details of Arguments** are so highly application-specific that they seem self-evidently (to me, at least) to be best handled by the program code.

The other two ways are examined in the following two sections, levering specific API functions for these purposes.

Detect whether Flags and Options are 'recognised'

Since *Flags* and *Options* may be declared in an aliases array (see Listing 2), it is useful to be able to determine whether a *Flag*/*Option* argument is not one of those declared. This is achieved using the API function `clasp_reportUnrecognisedFlagsAndOptions()`. The following code shows how to report all unrecognised *Flag*/*Option* arguments (Listing 7).

Detect whether Flags, Options and Values are 'used'

CLASP/C has a notion of whether an argument has been 'used' by the application. When the command-line is parsed, all arguments are marked not-used.

When any of the following functions are called, one or more arguments are marked as used:

- `clasp_checkAllFlags()` – checks all given *Flag* arguments and combines their bitmasks into a caller-supplied variable, and marks them all as used. See following section for explanation;
- `clasp_checkFlag()` – if the *Flag* is in the command-line, add its bitmask to a caller-supplied variable and mark it as used;
- `clasp_findFlagOrOption()` – looks for the named *Flag* or *Option*, and mark it used;
- `clasp_flagIsSpecified()` – determines whether the given *Flag* is in the command-line, and mark it used.

Furthermore, the programmer can explicitly mark an argument as used by calling `clasp_useArgument()`.

The program may then issue a warning and continue, or issue a contingent report and fail, if one or more arguments are unused, to detect if the user specified invalid (combinations of) arguments.

There are several functions for reporting unused *Flags*, *Options*, *Flags* and *Options*, *Values*, and all *Arguments*. Listing 8 shows how to report all unused *Value* arguments.

In CLASP/C++, there are several inline functions that call the requisite C-API functions to determine whether any (of a particular class of) arguments are unused and, if so, elicit the first argument and throw an exception with this information, as in:

```
int tool_main(clasp::arguments_t const* args)
{
  . . . arguments processing …
  clasp::verify_all_flags_and_options_used(args);
}
```

Listing 9

```
static int flags;

static struct options const OPTIONS[] =
{
  { "relative", no_argument, &flags, 0x01 },
  { "recursive", no_argument, &flags, 0x02 },

  . . .

  { 0, 0, 0, 0 }
};
```

Combine Flags into programmatic bitmasks

The `getopt_long` library provides the facility for associating a *Flag* with an integer variable, such that the variable is automatically set (to a programmer-supplied value) to reflect the presence/absence of the *Flag* on the command-line.

This is useful, but it doesn't suit all programming styles. There are two problems with the way `getopt_long` assigns to flag variables. First, the variable is assigned, rather than OR'd. Therefore, it is impossible to use the library to build up a flags bit-pattern into an integer flags variable (see Listing 9).

With respect to the above `getopt_long` options, if the user specifies the flag `--relative`, then flags will be set to `0x01`. If the user specifies the flag `--recursive`, then flags will be set to `0x02`. If the user specifies both flags, then flags will be set to `0x01` or to `0x02` (depending on the order of the command-line flags), but it will not be set to `0x03`, as intended.

The second problem is that the variable must exist in order that its address may be specified in the array. If the array is declared in non-local scope, then the variable must also be non-local (or the address of a local variable poked into the non-local struct option array prior to calling `getopt_long()`, which is horrible). If the variable must be local, then the array must be local (and mutable), which means it's all but impossible to reuse it for writing out usage information.

Consequently, CLASP/C provides the ability to specify, for each *Flag*, an associated bitmask that will be OR'd under programmatic control, via the function `clasp_checkAllFlags()`, into a caller-supplied flags variable.

Consider that **prg** has an additional flag `--truncated`, whose meaning we don't have to care too much about. We can then declare the two flags declaratively, in terms of bit-flag enumerators, as shown in Listing 10. We may then used this in combination with the `clasp_checkAllFlags()` function, as follows:

Listing 10

```
enum PRG_FLAGS
{
  PRG_F_RELATIVE  = 0x0001,
  PRG_F_TRUNCATED = 0x0002,
};

static clasp_alias_t const PRG_ALIASES[] =
{
  . . . // as before

  /* History flags/options */

  CLASP_OPTION("-h", "--history-file", . . .,
  CLASP_BIT_FLAG("-e", "--relative",
PRG_R_RELATIVE, "use relative paths in history"),
  CLASP_BIT_FLAG("-u", "--truncated",
PRG_R_TRUNCATED, ". . ."),

  /* Standard flags */

  . . . // as before
```

```
int flags = 0;
clasp_checkAllFlags(args, PRG_ALIASES, &flags);
. . . more work on flags
```

or:

```
int const flags = clasp_checkAllFlags(args,
   PRG_ALIASES, NULL);
```

This function marks all *Flag* arguments (that match entries within the alias array and that have not already been used) as used.

Follow library good practice

I have attempted to follow good programming principles in the implementation of the library, in particular:

Wide-string compatibility

The typedef `clasp_char_t` is used throughout CLASP/C. By default it resolves to `char`. If you explicitly define the preprocessor symbol `CLASP_USE_WIDE_STRINGS`, or if it is defined implicitly on Windows by the presence of definitions of Windows' preprocessor symbols `UNICODE` and `_UNICODE`, then the typedef resolves to `wchar_t`.

At the moment, there is one small part of the CLASP/C++ library that has an issue with wide-string compilation, but that'll be addressed very soon. CLASP/C# uses `System.String` throughout, so encoding is not an issue per se.

Everything is immutable

There is copious use of `const` throughout, including the copied `argv` pointer (which is of type `clasp_char_t const* const*`), to prevent accidental overwrites. When an argument is to be marked 'used', it must be passed to the API function `clasp_useArgument()` (which overrides const internally).

No globals

One of the problems with `getopt`/`getopt_long` is the use of global variables. Now, it's certainly unlikely that one would wish to use a command-line handling library multiple times (possible in multiple threads) in a real application, but having no global state simplifies automated testing tremendously. (As well as just fitting in with all good programmers' sensibilities.)

Clean (un)initialisation

A command-line is parsed into an arguments structure via `clasp_parseArguments()`, and the state is destroyed via a call to `clasp_releaseArguments()`. Again, this aids testing, is in keeping with good practice, and is amenable to encapsulation within RAII types in C++.

Diagnostics

CLASP/C supports the specification and use of custom diagnostic facilities for memory (de)allocation and diagnostic logging, via the fifth parameter to `clasp_parseArguments()`, which is a (non-mutating) pointer to an instance of `clasp_diagnostic_context_t`. For brevity, I will leave discussion of these facilities to a future article in which CLASP/C is used in building programs.

The CLASP/C API

The CLASP/C API consists of numerous types (typedefs, structures, enumerations), functions, constants, and preprocessor symbols. I will now briefly list the ones you need to be aware of in order to use CLASP/C.

Types

There are five main types that you must be aware of to use CLASP/C, as shown in Listing 6. `clasp_slice_t` is a string slice structure, comprising of a length of the string and a (non-mutating) pointer to the first character. Furthermore, in CLASP/C all the string slices actually point to nul-terminated C-style strings. Consequently, you can use them as either `len+ptr` or as C-style strings, depending on your preference, as illustrated in the various calls to `printf()`-statements for *Flags* and *Options* in Listing 2.

`clasp_argtype_t` is an enumeration, defining the range of argument types.

`clasp_argument_t` is a structure representing a parsed argument, capturing the name (`givenName` as specified on the command-line; `resolvedName` representing the translated form), the `value`, the `type`, the original command-line index, the number of hyphens in the name, and the index of the alias to which it's matched.

`clasp_arguments_t` is a structure representing all parsed arguments, partitioned into the groups previously discussed. The program accesses the parsed argument information via a non-mutating pointer to this structure.

`clasp_alias_t` is a structure used to define an alias, representing the `type`, the alias name, the mapped argument, a help string, a set of accepted values for *Options*, and a bit-mask for use by *Flags*.

Functions

The API functions are split into three groups: parsing, validation, and usage. Several of these feature in Listing 2, and in previous code snippets.

Parsing is done via `clasp_parseArguments()` and `clasp_releaseArguments()`.

Validation is done via a group of functions including:

- `clasp_reportUnrecognisedFlagsAndOptions()`
- `clasp_reportUnused[Flags|Options|FlagsAndOptions|Values|Arguments]()`
- `clasp_useArgument()`
- `clasp_flagIsSpecified()`
- `clasp_checkFlag()`
- `clasp_checkAllFlags()`
- `clasp_findFlagOrOption()`

These functions are quite mature, but thegroup will be expanded in future.

Usage is performed via a group of functions including:

- `clasp_show_usage()`
- `clasp_show_header()`
- `clasp_show_body()`
- `clasp_show_version()`

all of which take are implemented in terms of other output functions for output to a specific stream. CLASP/C currently comes with stock specific stream output functions `clasp_show_version_by_FILE()`, `clasp_show_header_by_FILE()`, and `clasp_show_body_by_FILE()`. This group of functions is likely to be refactored and improved in the near future.

Constants

Parsing behaviour may be changed at runtime by passing flags to `clasp_parseArguments()`. Each of the five `CLASP_F_*` constants currently defined has been discussed earlier in the text.

CLASP::Main/C

There's an additional facility, ostensibly a separate library `CLASP::Main`, that is able to significantly reduce the amount of CLASP boilerplate that you have to write, as shown in Listing 11. The calls to `clasp_parseArguments()` and `clasp_releaseArguments()`, along with the requisite contingent reporting if parsing fails, are encapsulated within this thin, header-only, library, in the guise of the function `clasp_main_invoke()`.

Listing 11

```
#include <systemtools/clasp/main.h>

. . . // other includes as before

static clasp_alias_t const ALIASES[] =
{
  . . . // aliases as before
};

static
int main1(clasp_arguments_t const* args)
{
  if(clasp_flagIsSpecified(args, "--help"))
  {
    . . . // clasp_show_usage() as before
  }
  else
  {
    /* Dump out the arguments, in groups */

    . . . // for-loops as before
  }

  return EXIT_SUCCESS;
}

int main(int argc, char** argv)
{
  unsigned const cflags = 0;

  return clasp_main_invoke(argc, argv, main1,
     "prg.main", ALIASES, cflags, NULL);
}
```

Summary

What CLASP does not (yet) do

There are four limitations of CLASP (C) that stand out to me. (You may have more, of course).

First, the one functional gap is the inability to work with the second form of *Options*, as in **`-habc`**. My guess is that it would be feasible to make this work, requiring the following logic:

1. Attempt to match the argument name (in this case **`'oabc'`**) against known flags (aliases or full-names) declared in the aliases. If that fails, then
2. attempt to split the argument name into letters and match all (in this case **`'o'`**, **`'a'`**, **`'b'`**, **`'c'`**) against known flags (aliases or full-names) declared in the aliases. If that fails, then
3. attempt to match the first letter (in this case **`'o'`**) against a known option (alias or full-name). If that succeeds, interpret the argument
4. interpret the argument as an unknown flag (in this case **`'-habc'`**)

I have not done this mainly because I don't like, and don't use, that form of *Options*. I have the feeling that there's just too much possibility for inscrutability in the output. However, that might be my (willing) ignorance on the matter. I'm keen to hear contrasting opinions.

Second, the (Windows-only) dependency on **recls** (and its dependency on **STLSoft**) makes CLASP/C that bit more of a hassle than a standalone library, and will inevitably cause the library to be that bit less desirable. A possible improvement would be to write the globbing facility in terms of the Windows API, but that's a lot of work (and it's harder to get right than you might imagine). For me, as the author of **recls** (and **STLSoft**), there is no motivation to spend the time required. But if some suitably motivated future user wishes to do the necessary work then we can perhaps remove the dependency at such time.

Third, there is nothing 'automatic' in terms of using arguments. In CLASP/C (and in the other language variants written thus far) the programmer must always issue calls to check for all arguments they wish to use. In my opinion, implicit use of arguments could only ever be done in a subset of all cases, and it's subtle. The opportunity for confusion seemed a greater evil than having to explicitly use arguments. (And CLASP/C++ does a reasonably good job of having very succinct use-statements.) But others may demur, and even have clear and compelling ideas on how it can be done.

Finally, the 'print usage' side of the library has had much less focus than the parsing side. Although it works, it's verbose and inelegant to use, and it's very much tailored to how I have come to format/display program usage. Since I use a lot of program-generating wizards, the verbosity tends not to hit me, so I have as yet been unmotivated to make it more succinct. Again, I'm keen to hear from others on this.

There are, of course, limitations in both the C++ layer and the C# implementation, but there's a reasonable chance that I'll address them before writing the next article on those, so I will hold my fire on them. Just be aware if you use them that they're a work in progress.

Obtaining CLASP

For the foreseeable future, CLASP will be available as a package from the systemtools project on SourceForge (at http://www.sourceforge.net/projects/systemtools). The first publicly distributed version will likely contain basic makefile(s) for GCC for building on UNIX, and project files for various Visual C++ versions for building on Windows. Other compilers and platforms will be available as the project matures.

Next steps

In terms of the libraries, I need to update the C++ layer in line with latest changes to the core C library, make some updates to the C# layer to keep it conceptually identical with the C/C++ ones before releasing it too, and then sit back for the constructive criticism.

I also want to do a CLASP for Ruby, but I need to work out if/how I can override **`ARGF`** to ignore *Flags* and *Options* and only work with the remaining *Values*: if any Ruby gurus want to help me out (or disabuse me of my ambitions), please get in contact.

In terms of writing, the next few articles I hope to submit to *CVu* will tackle the subject of program anatomies, starting with C (and C++) programs, referring to this article on CLASP/C where necessary. As my explorations of program anatomies move to other languages, I may write further CLASP articles in preparation for them. ■

Acknowledgements

Customarily, I want to thank Chris Oldwood and Garth Lancaster for helpful review comments – what blither remains is all mine – and the inhumanly patient Steve Love, whose deadlines I have stretched to bursting yet again.

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?

Desert Island Books

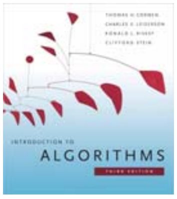
In this issue, Roger Orr kindly introduces Ola Mierzejewska on my behalf.

I first met Ola briefly at an ACCU conference but I've got to know her better since then as last year she joined the team I'm working in (and also started coming along to some of the monthly Canary Wharf lunches). She is an active attender of the London Region meetings so some of you will know her from there.

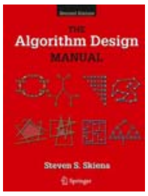
We were both at the 2011 conference and, on our return, Ola had the impossible task of reporting back on the whole conference to interested members of the wider team – all in about an hour and a half. Wisely she chose to give a top level summary of the topics to give a feel for the conference as a whole and then talked in more detail about a very small selection of the presentations she attended.

One of Ola's hobbies is mountain climbing so I hope the desert island she is stranded on contains a volcano or something similar to keep her in condition!

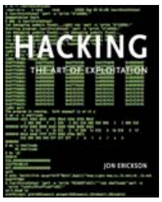
Ola Mierzejewska



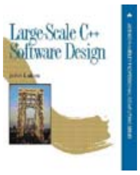
There are three books that have been sitting on my bookshelf already for quite a while waiting for the unlikely event of me having a bit of time to go through them. The first one is the *Introduction to Algorithms* by Thomas H. Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein. It covers lots of basics, but I have the feeling that the big volume still contains quite a few topics new to me, or topics that I have heard about, but wished to understand in more detail.



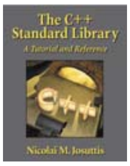
The second one is on a very similar subject: *The Algorithm Design Manual* by Steven S. Skiena. Got it recommended at some point and after reading the first few chapters I think it's written in a nice, light style. So the two algorithms books could well complement each other – the *Introduction to Algorithms* is a very solid classic, but gets a bit dry at times.

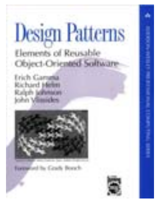


I found the third one randomly in a charity shop with second-hand books. And think it was one of my best buys of a book I haven't heard of before. It's *Hacking, The art of Exploitation* by Jon Erickson. The contents look very interesting (if you're interested in application security, of course). The first chapter explains programming in C. The explanation is quite good, but as you can imagine, extremely dense. Assuming that the pace will be kept – and as I don't feel that familiar with the following topics, in spite of a lot of enthusiasm I put down the book. Felt like it requires a bit more focus and time to check out some of the examples or to look for some additional references. Which, I realise, could be not doable on the Desert Island...

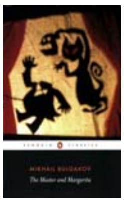


The fourth book is a bit harder choice. one volume I have queued up to read is *Large-Scale C++ Software Design* by John Lakos, but think I can manage this one from home, and planning to do it shortly (will see if I'm wrong on that...). Another great book, which is a bit of a lengthy read is *The C++ Standard Library: A Tutorial and Reference* by Nicolai M. Josuttis. I wouldn't bring it just because I have read it all, so now would rather have it as reference.





I think I might take the *Design patterns: elements of reusable object-oriented software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. The last time I tried to read it I thought it was rather dry, but as it is a classic, and a topic I'd like to read on – would give it another go.



For a novel it's a tough choice. Probably choosing by author it would be Fyodor Dostoevsky. But I wouldn't be able to choose which of his works to pick. So probably would bring another Russian novel: *The Master And Margarita* by Mikhail Bulgakov. I have enjoyed reading it a lot, but only after we discussed fragments of the novel on one of my Russian classes I realised how much content I have missed due to lack of background and historical knowledge, it's full of second and hidden meanings! So, no access to references could be a problem again. But hopefully I could get an edition with lots of comments and background information.

As for music, I'd be very cautious about bringing any of my favorite albums, as it seems easy to get bored of whatever you listened to over and over again. I think I would take some of Wladimir Wysotsky's albums. I do like the music, and also could try to improve on my Russian :).

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: Derek Jones

Inspirational (P)articles

Doctor Love finds inspiration in the simple things.

It was a delight to read Daniel Higgins' book review of *Invent your own computer games with python, 2nd edition* in the last *CVu*. Daniel's delight and enthusiasm was contagious. This made me start reading the book, which is available online at http://inventwithpython.com/chapters. I haven't finished yet, but intend to. I got diverted by extending the games and recalling the excitement at writing simple things, such as guess the number, when I first learned to programme. Starting with a simple task and finding ways to extend it is a great way to practise and learn the basics of which ever language you have chosen. Finding an excuse to use Python for a bit has been great. I am looking forward to the chapter on Reversi. I tried to write a Reversi game years ago and never got anywhere. This time I might achieve something. Thanks again to Daniel (and his Dad).

Write for us!

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. 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

ACCU London – November 2011

Frances Buontempo reports on a recent meeting.

Since we'd run out of speakers and ideas we held a session of lightning talks in November. Instead of one person talking for an hour or so, several people talked for five or ten minutes each. I suspect this encouraged some unusual suspects to volunteer (myself included). A selection of the slides (self-selected by the speakers) have been uploaded to http://accu.org/index.php/accu_branches/accu_london accu_london_nov2011.

The topics included a Hudson CI with an almost live demo ('If this were working you'd see a button here' etc – catastrophic 3g #fail but only when stood by the projector), a retrospective of the magic that happens and continues to happen in the ACCU, various spontaneous common themes about monte-carlo simulation in Python and R and multi-processing, an endorsement of **`make_shared`** which several of us had forgotten about, a brief description of what counts as a model with particular reference to what certainly doesn't, a light-hearted view of how to be dispensable. and a decision assistant, which among other things solves the perennial tabs versus spaces debate.

It was great to see common themes cropping up, such as Python and multi-processing, and I do wonder if Herb Sutter listened to the smart pointers talk, since he recently blogged about make_unique on http://herbsutter.com/2011/12/02/gotw-102-exception-safe-function-calls-difficulty-710/.

Many thanks to everyone who attended and spoke. We hope to have another go in the not too distant future.

Code Critique Competition 73

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

I've written a function that escapes a string of UTF-8 characters using the html entity format and I'm trying to use it with different compilers. One compiler fails to find `std::runtime_error` – don't know why – and another compiles it but produces unexpected output. Please help!

I wrote a test program using the four example UTF-8 sequences from http://en.wikipedia.org/wiki/UTF-8#Description

Here is what I want:

```
> test_escape_utf8
U+0024 \x24 = $
U+00A2 \xc2\xa2 = &#xa2;
U+20AC \xe2\x82\xac = &#x20ac;
U+024B62 \xF0\xA4\xAD\xA2 = &#x24b62;
```

Listing 1

```
#include <iostream>
#include <string>
#include "escape_utf8.h"
//U+0000 to U+007F
//Example: code point U+0024 ("Dollar sign")
//UTF-8 hex: 24
char test1[] = "U+0024 \\x24 = \x24";
//U+0080 to U+07FF
//Example: code point U+00A2 ("Cent sign")
//UTF-8 hex: C2 A2
char test2[] = "U+00A2 \\xc2\\xa2 = \xc2\xa2";
//U+0800 to U+FFFF
//Example: code point U+20AC ("Euro sign")
//UTF-8 hex: E2 82 AC
char test3[] = "U+20AC \\xe2\\x82\\xac"
  " = \xe2\x82\xac";
//U+010000 to U+10FFFF
//Example: code point U+024B62 (A CJK Unified
//Ideograph)
//UTF-8 hex: F0 A4 AD A2
char test4[] = "U+024B62 \\xF0\\xA4\\xAD\\xA2"
  " = \xF0\xA4\xAD\xA2";
int main()
{
  try
  {
    std::cout
      << escape_utf8(test1) << std::endl
      << escape_utf8(test2) << std::endl
      << escape_utf8(test3) << std::endl
      << escape_utf8(test4) << std::endl;
  }
  catch (std::exception const & ex)
  {
    std::cerr << "Exception: " << ex.what();
  }
}
```

Here is the unexpected output:

```
> test_escape_utf8
U+0024 \x24 = $
U+00A2 \xc2\xa2 = &#xffffffa2;
U+20AC \xe2\x82\xac = &#xffffdfac;
U+024B62 \xF0\xA4\xAD\xA2 = &#xfff20a62;
```

(See Listing 1 for test_escape_utf8.cpp, Listing 2 for escape_utf8.h and Listing 3 for escape_utf8.cpp)

Listing 2

```
#ifndef escape_utf8
#define escape_uft8
std::string escape_utf8(char const * utf8);
#endif
```

Listing 3

```
#include <string>
#include "escape_utf8.h"
std::string escape_utf8(char const * utf8)
{
  std::string result;
  long value;
  int multibyte(0);
  while (char const ch = *utf8++)
  {
    if (multibyte-- > 0)
    {
      if ((ch & 0xc0) != 0x80)
        throw std::runtime_error(
          "Bad multibyte continuation");
      value <<= 6;
      value += ch - 0x80;
      if (!multibyte)
      {
        result += "&#x";
        char buff[7];
        sprintf(buff, "%lx", value);
        result += buff;
        result += ';';
      }
    }
    else if ((ch & 0xC0) == 0xc0)
    {
      value = ch & 0x1f;
      multibyte = 1;
      if (ch & 0x20)
      {
        multibyte++;
        if (ch & 0x10)
        {
          value -= 0x10;
          multibyte++;
```


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

Listing 3 (cont'd)

```
        if (ch & 0x8)
          throw std::runtime_error(
            "Bad multibyte start");
        }
      }
    }
    else if (ch & 0x80)
      throw std::runtime_error(
        "Bad multibyte start");
    else
    {
      result += ch;
    }
  }
  return result;
}
```

Critiques

Peter Sommerlad <peter.sommerlad@hsr.ch>

This time, I will give two answers, one quick and dirty and one elaborate.

1. Quick and Dirty Fix

One of the most obvious things in the program is that bit-operations are used on (potentially) signed values and they are mixed with regular arithmetic, e.g.,

```
value <<= 6;
value += ch - 0x80;
```

While left shifting a signed variable is not really a problem, using subtraction from a **char** variable to clear a bit is a doubtful practice. Since the literal **0x80** is not of type **char**, the value of the variable **ch** is automatically promoted to **int**, before the subtraction is done. On many current processors a **char** is 8 bits large and often it is signed and represented in twos-complement. The if-check before guarantees that **ch** has its high bit set and thus is negative. As a result we add an even more negative number to **value** (e.g., **ch=0x80=-128**, **ch-0x80=-256**), eventually making it negative itself. Since **value** as a 4 byte representation can become negative, when printed as hex, then it can result in the many f's in the output.

The quick fix is to make **ch** an **unsigned char**:

```
while (unsigned char const ch = *utf8++)
```

In addition to make `escape_utf8.cpp` compile on most systems one needs to

```
#include <stdexcept>
```

to obtain **std::runtime_error**'s definition.

With those minimal changes the code compiles and delivers the following output:

```
U+0024 \x24 = $
U+00A2 \xc2\xa2 = &#xa2;
U+20AC \xe2\x82\xac = &#x20ac;
U+024B62 \xF0\xA4\xAD\xA2 = &#x24b62;
```

If we would be a non-caring programmer that would be sufficient, but that is not a good feedback to our student, since the code has many more problems to deal with.

2. Elaborate additional feedback and refactoring

First a subtle observation, while Linticator recognizes the left shift of the signed variable **value**, it doesn't tell us the problem of using bitmasks and arithmetic on the potentially signed **char** variable **ch** (Figure 1).

Now, first things first and let us start with the simplest file in the problem:

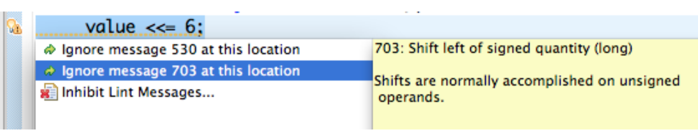


Figure 1

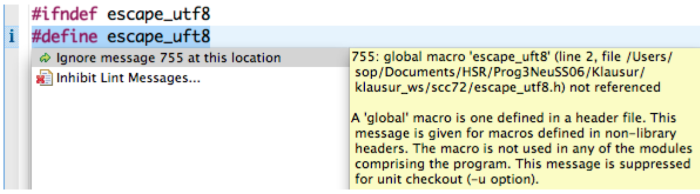


Figure 2

```
#ifndef escape_utf8
#define escape_uft8
std::string escape_utf8(char const * utf8);
#endif
```

A brief look shows, that the include guard macro is spelled lower case. That is bad practice, especially when the guard is spelled exactly like the function it exports. However, fortunately or by design the **#define** uses a different spelling, did you see that. Fixing that first, results in interesting compile errors of the header. Even without that fix the header is still non-problematic, since it only declares a function and defines nothing. So multiple inclusions will not violate C++'s ODR (one-definition-rule) and thus create compile errors.

Even Linticator indicates there might be a slight problem (Figure 2).

One could argue that the **escape_utf8** function could also use a **std::string const &** as input instead of the C legacy **char const ***, but that is not really a problem. A bigger problem is that the header file itself is not self-contained. An easy test is to always use the own header file as the first **#include** in its implementation file. Changing the code accordingly will show the missing definition of **std::string**:

```
In file included from ../escape_utf8.cpp:1:0:
 ../escape_utf8.h:4:1: error: 'string' in namespace
'std' does not name a type
```

So we move the **#include <string>** into `escape_utf8.h`. That makes the fixed header file look like:

```
#ifndef ESCAPE_UTF8_H_
#define ESCAPE_UTF8_H_
#include <string>
std::string escape_utf8(char const * utf8);
#endif
```

OK, now let us look for the beef in `escape_utf8.cpp`. The file now starts as follows:

```
#include "escape_utf8.h"
#include <stdexcept>
std::string escape_utf8( char const * utf8)
{
```

that is OK so far. In best C-manner our code defines (most of) the local variables at the beginning of the function. Two of the variables get initialized (**result** and **multibyte**), however, close inspection and Linticator shows that **value** is uninitialized. While the control flow seems to guarantee that it is never used before it is initialized, it is bad practice to not initialize a variable, especially one that is local (globals will be zero initialized, at least). In addition we adapt **value** to be of **unsigned** type

In C++11, we should use the new universal initialization syntax:

```
unsigned long value{};
int multibyte{0};
```

This new syntax will help to avoid problems with unintentionally declaring a function instead of a local variable, as would have been the case when we would have written

```
unsigned long value();
```

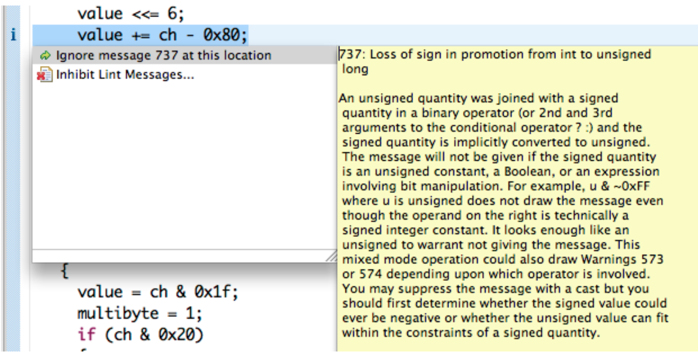
With those changes we get another warning from Linticator, indicating the underlying problem (see Figure 3).

Now applying our previous quick fix we remove that problem but resurrect a similar one, on initializing **ch**:

```
while (unsigned char const ch = *utf8++)
```

but at least the code seems to work now. But the complexity of the function is overwhelming and the control flow non-intuitive. There is a need to simplify. Before that, we should write some unit tests to actually execute

Figure 3



all of the branches (I refrain from that this time, because I also have other things to do, but it would be a better practice!)

Now let us start with making all bit-mask operations using really the bit operations intended for it:

```
value += ch - 0x80;
```

becomes:

```
value |= ch & 0x3f; // lower 6 bits are used
```

Another issue is to use the C-ish **sprintf** for hex conversion. I would suggest to use an **std::ostringstream** to collect the result and return its underlying **std::string** this allows to use hex conversion of the stream instead of a fixed size buffer that in the end might be too small for a long value (8 hex characters + '\0' termination character), even though with the corrected code this shouldn't happen. So instead of using **+=** on a string, we use **<<** on an **ostringstream** and return the underlying result string in the end with **.str()**.

We replace the code that is wrong in respect to errors

```
char buff[7];
sprintf(buff, "%lx", value);
result += buff;
result += ';';
```

with

```
result << "&#x" << hex << value << dec << ';';
```

instead.

A big thing is the complicated logic of the **while** loop. And the only reason seems to be to have a single place to obtain the next character from the input string. Also selecting the number of bytes to read is a bit obfuscated with respect to the rules given on Wikipedia. I read it the following way: If the high nibble is hex C or D one additional character must be consumed and 5 low bits of the current character are used, if it is E, two more characters follow and 4 low bits of the current character are used, and if it is F, three more characters follow and the 3 low bits are used. However, there is room for 'optimization', since in the last case the 4th bit must be zero. That is something we should check in addition to the bad continuation if we want to ensure correct input. On the other hand, we could also remove all sanity checks and assume the input is correct. Also it should be checked if **multibyte** is zero on leaving the loop, otherwise the encoding prematurely got to the end.

Another problem is the C-ish parameter definition. Changing that from **char const *** to use **std::string** will open a multitude of interesting options. One thing is that it will allow us to use a standard algorithm: **transform**. Nevertheless this will require a functor with some memory, since transform will provide individual chars like the overall loop is doing now.

But first let us deal with the unintuitive logic. Close observation of the UTF-8 definition shows that instead of using bit-masking to figure out how to dissect a lead-in byte of a multi-byte code point representation one could use a range discrimination to distinguish between the 1 to 3 following bytes. If the byte under consideration is greater than **0xf0** we know 3 bytes must follow and only the lower 3 bits are significant. If it is greater than **0xf8** the byte is invalid. Otherwise, if it is greater than **0xe0**, 2 continuation bytes follow, if it is greater than **0xc0**, 1 continuation byte is needed. A continuation byte must be between **0x80** and **0xbf**. I hope with that insight the code can become simpler, let us see:

```
else if (ch >=0xf8) {
  throw std::runtime_error(
    "Bad multibyte lead in");
} else if (ch >= 0xf0) {
  multibyte = 3;
  value = ch & 0x07;
} else if (ch >= 0xe0) {
  multibyte = 2;
  value = ch & 0x0f;
} else if (ch >= 0xc0) {
  multibyte = 1;
  value = ch & 0x3f;
} else if (ch >= 0x80) {
   throw std::runtime_error(
   "Bad multibyte lead in");
} else {
  result << ch;
}
```

Ok, looks a bit more symmetrical. Now how can we deal with the **multibyte** continuations in a similar way:

```
if (multibyte-- > 0)
{
  if (ch > 0xbf || ch < 0x80)
    throw std::runtime_error(
      "Bad multibyte continuation");
  value <<= 6;
  value |= ch & 0x3f; // lower 6 bits are used
  if (!multibyte)
  {
    result << "&#x" <<
      std::hex << value << std::dec << ';';
    }
  }
```

still ugly, but at least we now cover all values. Before we return, we check that **multibyte** is actually zero:

```
  if (multibyte) throw std::runtime_error(
    "Missing multibyte continuation at end");
  return result.str();
}
```

Now for the transformation to transform resulting in the following code in `escape_utf8.cpp`:

```
#include "escape_utf8.h"
#include <sstream>
#include <stdexcept>
#include <iterator>
#include <algorithm>
struct utf8_to_html {
  utf8_to_html() :
      value(0), multibyte(0) {
  }
  std::string operator()(unsigned char ch) {
    if (multibyte-- > 0) {
      if (ch > 0xbf || ch < 0x80)
        throw std::runtime_error(
          "Bad multibyte continuation");
      value <<= 6;
      // lower 6 bits are used
      value |= ch & 0x3f;
      if (!multibyte) {
        std::ostringstream result;
        result << "&#x" << std::hex
          << value << std::dec << ';';
        return result.str();
      }
    } else if (ch >= 0xf8) {
      throw std::runtime_error(
```

```
        "Bad multibyte lead in");
    } else if (ch >= 0xf0) {
      multibyte = 3;
      value = ch & 0x07;
    } else if (ch >= 0xe0) {
      multibyte = 2;
      value = ch & 0x0f;
    } else if (ch >= 0xc0) {
      multibyte = 1;
      value = ch & 0x3f;
    } else if (ch >= 0x80) {
      throw std::runtime_error(
        "Bad multibyte lead in");
    } else {
      return std::string(size_t(1), char(ch));
    }
    return std::string();
  }
  unsigned long value;
  int multibyte;
};
std::string escape_utf8(
  std::string const & utf8) {
  std::ostringstream result;
  utf8_to_html converter;
  transform(utf8.begin(), utf8.end(),
      std::ostream_iterator
      <std::string>(result), converter);
  if (converter.multibyte)
    throw std::runtime_error(
     "Missing multibyte continuation at end");
  return result.str();
}
```

One can see, no more loop needed. OK, the functor is still a bit ugly, but it works.

I forgot to provide the corresponding changed header:

```
#ifndef ESCAPE_UTF8_H_
#define ESCAPE_UTF8_H_
#include <string>
std::string escape_utf8(
  std::string const& utf8);
#endif
```

We are left with the test program with its main function. I do not want to spend more time on that, since my wife already got impatient. However, I suggest defining the test data arrays to be

```
char const test1[]=....
```

A minor glitch that might result not getting any output at all in case of an exception is that the output to `std::cerr` is not flushed. Adding a `<< std::endl` should help:

```
int main()
{
  try
  {
    std::cout
      << escape_utf8(test1) << std::endl
      << escape_utf8(test2) << std::endl
      << escape_utf8(test3) << std::endl
      << escape_utf8(test4) << std::endl;
  }
  catch (std::exception const & ex)
  {
     std::cerr << "Exception: " << ex.what()
       << std::endl;
  }
}
```

A bad thing is that all calls to `escpe_utf8` occur in a single statement, this might be hard to diagnose which call actually threw an exception. But I refrain from fixing that. My time is over. Nevertheless, I hope you learned something.

Huw Lewis <huw.lewis2409@gmail.com>

Wow! Unreadable code for a reviewer not familiar with UTF-8 encoding. I needed a little revision to understand this one.

To address the question's comment about some compilers failing to find `std::runtime_error` – `<stdexcept>` is the correct standard library header to include from `escape_utf8.cpp`. While we're on this subject, the header file should also include `<string>` as it is required as the `escape_utf8` function's return type.

The results of the test harness show that test 1 comes out ok (the single byte character), but the other values are prepended with one or more `0xff` bytes. It looks like there is a signing problem where the resultant integer has become negative. This is down to the line:

```
value += ch - 0x80;
```

The `value (int)` variable adds to itself (`ch - 0x80`) which seems reasonable given that we know from the previous check that `(ch & 0x80)` is true. The problem is that `ch` is a variable (const) of type `char` which is a signed char. For test 2, this is going to be (-94 - 128). This makes `value` negative and explains the presence of the `0xff`.

I have a choice of easy fixes. Either make `ch` an `unsigned char`:

```
while (const unsigned char ch =
  static_cast<unsigned char>(*(utf8++) ) )
```

or change the method of obtaining the relevant 6 bits from the subtract operation into a bitwise 'and':

```
value += (ch & 0x3f);
```

So, now the results are as expected. Job done? Yes, but being a pedantic so-and-so I will carry on picking;-)

The array `buff` of characters has a suspicious length: 7. This buffer is used with `sprintf` to accept the converted hex string. For 64 bit systems this could be 8 characters in length – buffer overflow!

As we're working with C++, I would prefer to use `ostringstream` to perform the conversion. I admit that `sprintf` is probably more performant, but this eliminates the need for the fixed length buff array and makes for less verbose code. I have still kept a `std::string` variable for the return value so as to encourage the 'Named Return Value Optimisation' in the compiler which eliminates the construction (and destruction) of the return value, placing it directly in the client's object. I'm not sure, but I've a feeling that the presence of exceptions in this function might ruin the chances of the optimisation being applied (some experimentation required).

The code that detects the start of the multi-byte sequence is quite complex and could quite easily hide a bug or two (although I don't think it does). I have re-written this section for clarity – hopefully less bugs will be introduced by well meaning maintainers in future. The final version is given below:

```
std::string escape_utf8(const char* utf8)
{
  // the result string to be returned
  std::ostringstream result;
  long value;
  int multibyte(0);
  // loop through each byte in the string
  while (const char ch = *(utf8++) )
  {
    if (multibyte-- > 0)
    {
      // A multibyte continuation
      // This must start with 10xx xxxx
      if ((ch & 0xc0) != 0x80)
        throw std::runtime_error(
          "Bad multibyte continuation");
      value <<= 6;
```

```
      value += (ch & 0x3f); // 6 bits
      if (!multibyte)
      {
        result << "&#x" << std::hex
          << value << ';';
      }
    }
    else if ((ch & 0xE0) == 0xc0) // 110x xxxx
    {
      // A multi-byte start
      value = (ch & 0x1f); // 5 least sig bits
      multibyte = 1; // one byte continuation
    }
    else if ((ch & 0xF0) == 0xE0) // 1110 xxxx
    {
      // a multi-byte start
      value = (ch & 0x0F); // 4 least sig bits
      multibyte = 2; // 2 byte continuation
    }
    else if ((ch & 0xF8) == 0xF0) // 1111 0xxx
    {
      // a multi-byte start
      value = (ch & 0x07); // 3 least sig bits
      multibyte = 3; // 3 byte continuation
    }
    else if (ch & 0x80)
    {
      // invalid continuation byte
      throw std::runtime_error(
        "Bad multibyte continuation");
    }
    else
    {
      // not a multibyte start
      // or continuation.
     result << ch;
    }
  } // end loop through input string
  // Declare the return variable to encourage
  // the named-value return optimisation
  std::string resultString(result.str());
  return resultString;
}
```

Commentary

The original code had one obvious bug – the leading f's caused by the use of **char** which can be either signed or unsigned depending on the implementation. The distinction is not important for ASCII characters but becomes very important for everything else. As it happens I had another problem today in completely unrelated code where the handling of a character value greater than 128 as negative caused a program to abort.

Many compilers – including both g++ and MSVC – provide command line options to allow the programmer to specify the type of **char**. This can be useful when porting existing programs from one environment to another but it is better to write the code correctly in the first place!

One note about **char buff[7];** This fixed size buffer is completely correct (as Peter mentioned) *provided* the rest of the code is correct. In the problem case above more than 7 characters were actually written to **buff** and we are perhaps fortunate that we didn't have some harder to solve symptoms such as memory corruption or an access violation. I think in general it is good to avoid such brittle solutions because of the potential difficulty when debugging. One safer solution might be to use **snprintf** instead of **sprintf** but unfortunately, despite having been standardized in C99, implementations still seem to provide a variety of semantics so it must be used with care!

I don't think I've otherwise got much to add to Peter's and Huw's critiques above which between them seem to cover pretty well all the bases

The winner of CC 72

Both entries identified the root cause of the problem and then carried on to refactor the code to make it clearer. In Peter's case this was done using comparisons to split into the correct range, in Huw's case this was done by **and**ing with various bit patterns. I do wonder slightly whether Peter's clever solution with **std::transform** might be a little too much for the likely ability of the original writer: I'm sure opinions over this will differ! It was hard to decide to whom to give the prize, but I have eventually decided that this time Huw was the winner.

Those who follow this column regularly will notice that the same names keep turning up. I'd like to remind you, dear reader, that there's nothing to stop you also sending in a critique!

Code Critique 73

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

I've tried to write a simply program to split up a single text file into separate files. The idea is each line starting with `"--- "` and ending with `" ---"` contains the filename for the lines following it. It doesn't work on my old machine: gives me a '**bad allocation**' error. It works on my new machine – although it seems a little slow – but the filenames don't get the trailing minus signs removed. Can you help me find my bug?

The example code (unpack.cpp) is in Listing 4 and a sample input file is:

```
--- file1.txt ---
This is file 1
--- file2.txt ---
This is file 2
Line 2
Line 3
```

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 4

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
  std::ofstream ofs;
  ofs.exceptions(std::ios::failbit);
  try
  {
    std::string lbufr;
    while (std::getline(std::cin, lbufr))
    {
      if (lbufr.find("--- ") == 0 &&
          lbufr.find(" ---") > 0)
      {
        unsigned len(lbufr.find(4, ' ') - 4);
        lbufr.erase(0, 4);
        lbufr.resize(len);
        if (ofs.is_open())
        {
          ofs.close();
        }
        ofs.open(lbufr.c_str());
        continue;
      }
      ofs << lbufr << std::endl;
    }
  }
  catch (std::exception const & ex)
  {
    std::cerr << "Error: " << ex.what();
  }
}
```

View From The Chair

Hubert Matthews
chair@accu.org Members.fm

I have been doing a lot of C++ training work recently and, as ever, I have been explaining the ACCU and its value to delegates. What surprises me is that so few of them seem to want to find out more about the world of programming outside of their own work environments. Perhaps ACCU members are a special and select few. Perhaps we are all collectively mad in our own quiet kind of a way. (Modesty insists that I skip over the possibility that I'm a poor salesman.) It seems to me that until people have experienced what it's like to be part of a broader programming community and interact with other individuals who are knowledgeable, interested (and often interesting) and keen they don't seem to 'get it' at all. Their own programming community is purely their peers and for them programming is an interesting job but not more. They haven't been bitten by the bug (well, not that particular bug, anyway).

What is it that makes a programmer change from accepting the status quo and a small-world view to being someone who wants to improve, who wants to learn and share? I think it comes down to examples, role models and leadership. In my local Oxford ACCU group there are a number of non-members who have come to the meetings because some high-ranking officer of their company has encouraged the developers to come along. I can only hope that these new visitors find what we do appealing and interesting and will therefore continue to attend under their own steam. Some of them have seen our magazines left lying around in prominent places and have gently perused them. This is where paper copies are still a far better alternative than electronic versions – it is hard to flick through an electronic version in the corporate kitchen whilst waiting for the kettle to boil.

So, what does this all mean for ACCU members? We are a self-selecting bunch; not everyone wants to be a member and maybe they shouldn't be. Ours is a fragmented and young industry with little centralised control or leadership. In my travels and interactions with developers I am continually reminded of how insular are the worlds in which they work. Some of them do want to explore and find out more but there are a good number that are happy to bumble along without ever grasping the self-improvement nettle by the horns. Should we try to show them another way or should we leave them to their own devices? Should we evangelise more or be content to offer support and camaraderie to those of a similar ilk?

Committing to improving one's own programming and helping others to do so too is an important jump as it requires effort, the desire to change well-ingrained habits and to choose the right way and not just the easy way. It also takes time and that's one thing that ACCU members often seem to run short of (probably because work migrates to the competent and the willing). As chairman, I am, as ever, grateful to those members that do make time to help the organisation and I hope that others will join in and 'do their bit' over time, in whatever way they feel able to contribute.

The 24th ACCU AGM

Notice is hereby given that the 24th Annual General Meeting of ACCU will be held at 13:00 on Saturday 28th April 2012 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 23rd Annual General Meeting
3. Annual reports of the officers
4. Accounts for the year ending 31st December 2011
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.

Please also note we are looking to appoint a new Secretary, and if anyone considers that they could stand for this position, please let either me (as secretary@accu.org) or Hubert Matthews (chair@accu.org) know.

Roger Orr
Acting Secretary, ACCU