overload 83

FEBRUARY 2008 £3

Generics Without Templates
Emulating STL collections in the absence of templates and exceptions

The PfA Papers
Simpletons and The Borg invade Parameterise From Above

The Model Student
We do some more modelling (of mathematical problems)

Watersheds and Waterfalls
An introductory look at segmenting images into regions using a landscape analogy

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 83

February 2008

ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Guest Editor

Roger Orr
rogero@howzatt.demon.co.uk

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 84 should be submitted to the editor by 1st March 2008 and for Overload 85 by 1st May 2008.

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.

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

Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website:
www.accu.org

Copyrights and Trade Marks

When Things Go Wrong

Can we reduce the pain of computer problems?

Hello and welcome to Overload 83. I am guest editor for this issue, giving Alan Griffiths a break from his duties and me a chance to see what's involved in the editorial role. Alan will be back for the next issue.

Computer problems

It has been a trying month for the Orr household computers.

Firstly we installed IE7 on Windows Server 2003 SP2 after a number of reminders from various web sites, with comments about the benefits over IE6, made us decide that the time was now right to update. However, this change has proved a rather mixed blessing; it seems to be so secure that many web sites won't work at all... however as it has persuaded my wife to try out (and switch to) Firefox perhaps it wasn't altogether a bad thing.

While trying to resolve some of the problems with the sites that weren't working I decided to verify that we had installed all the latest Windows hot fixes by running Microsoft Update interactively. However the check for available updates failed with an error code.

Unfortunately a quick google for the error code didn't provide a great deal of help – a couple of suggestions were provided but neither resolved the problem. Further investigation in the event log revealed that Windows Update wasn't running properly, and in fact hadn't been working for over a month.

Fortunately faults in Microsoft Update can be reported for free, so I fired off a request for help to Microsoft. I was asked for some additional information including the contents of `WindowsUpdate.log`, a file that I hadn't previously known about. Sadly the log file, while it did contain a bit more information about the error ('WARNING: DownloadFileInternal failed for http://download.windowsupdate.com/v7/windowsupdate/redir/wuredir.cab'), didn't immediately help me to solve my problem.

I carried on with the original problem, trying to get IE7 to work. At this point I was running a packet sniffer (Packetyzer, free from Network Chemistry) and mixed up with the HTTP traffic I was expecting I noticed a single request to my ISP's web proxy. 'That's odd', I thought, 'I removed the proxy cache from my configuration back in October as my ISP announced they were going to decommission the facility. Why is it still being tried?'

After more googling on Web proxies I discovered a pointer to a tool, **proxycfg**, which displays and edits the WinHTTP proxy configuration. Using this tool I found that WinHTTP was still using the obsolete proxy cache and so I reconfigured it to use direct access. Having made this change Microsoft Update started working, so at least that problem was resolved.

Oh yes, I did get a reply from Microsoft to my request for help, which did give me number of additional things to try to resolve the problem; however this list didn't include checking the proxy settings. Perhaps it would have been suggested had I continued to need assistance.

Then my daughter arrived home from University, complaining that her new laptop had suddenly stopped displaying DVDs. Sure enough, when you put one in and pressed 'play' the screen simply went black. No error was shown, and nor could I find a log file that seemed to shed any more light on the problem.

I checked the obvious things – looking for errors in the event log from around the time of failure, looking for newly installed software, running a spyware scan, but to no avail. Data DVDs were readable, it was only videos that seemed to fail. Once more Google came to the rescue, and I found a thread reporting the same behaviour, and explaining that the problem is caused by the CODECs expiring. There was a link to a patch on the manufacturer's site containing updated CODECs so I downloaded and installed this patch, but sadly it made no difference. It seems that the updated CODECs had also expired. A further search pointed me to a further set of (free) CODECs, which resolved the problem, and my daughter was soon happily catching up with her unwatched DVDs into the small hours.

So why have I told these stories? I'm not trying to get at any company or operating system in particular – the previous month I tried and failed (on two different machines) to get successful network connections under Linux – but there seem to me to be couple of issues that these problems highlight.

Why are there so many problems?

I am an experienced programmer, although not a trained support engineer, and even so I can spend hours trying to fix such problems. I expect many of you have also spent more time than you'd like on fixing systems that don't work, or programs that won't communicate. For the majority of users, without in-depth technical experience, fixing these problems is even harder.

A large part of the problem is the 'execution environment' of the programs. No two PCs seem to be exactly alike, and the differences can include any or all of: the hardware configuration, the type (and version) of the operating system, the configuration settings, the amount of disk space, the network topology and the selection of other programs installed on the machine. I'm sure you, like me, can think of application or installation failures that each one of these factors can cause. Hence a log of what software and hardware was installed, when, and what options we selected can provide valuable help when experiencing software failures.

One of the adages of programming is 'separate out the things that change from those that stay the same'. I consider that two of the problems I had

Roger Orr has been programming for over 20 years, most recently in C++, Java and C# for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and runs the Code Critique column in CVu. He can be contacted at rogero@howzatt.demon.co.uk

last month could have been resolved more easily if the software writers had borne this rule in mind with reference to the execution environment.

Network configurations are not static but are subject to change; this may be frequently for a laptop but will also occur if you move house or change ISP. Given this, silently caching the Internet Explorer proxy settings elsewhere in the machine seems to me to be a poor design choice. The DVD player problem was also caused by something changing, in this case the date. The software is checking the expiry date of the CODECs but insufficient thought seems to have been given to the action taken when the check failed.

It is easy to make mistakes over which things may change and which are constant when considering installation versus runtime checks. A program might, for example, check at installation time that a specific version of some other component is available. Later on, this component can be updated causing hard-to-disagnose faults in the application. The sad thing is that the code to diagnose the incompatability has already been written – but a decision was implicitly taken that this only executed during installation. Some programs sensibly have a separate 'self test' function that verifies some of the environmental dependencies, which can be executed during installation and also re-run automatically on startup or manually as part of fault diagnosis.

Where do we get help?

The second reflection on the stories is how much the Web has changed the way I resolve support issues. I typically start by googling to find information from other people who have experienced and, hopefully, resolved the same problem. For this to work well three things are needed: specific error data, good questions and documented solutions.

One trouble with the Web is that it is so big and contains so much information that a good search query is vital to locate relevant information easily. The more specific the error data provided by the failing program is the better the hit rate of a search will be. Numeric error codes can be useful – in the Windows Update case above adding the Hex error code into a simple Google query reduces 3,230,000 possible pages to 419. On the other hand searching for 'DVD player displays blank screen' gives 638,000 possible pages but, alas, the DVD player provided no other details to refine the search.

Some of us are writers of software that we do not support in person; how much thought do we give to ensuring that our users are not only given notification of errors but also that any such error messages are easily identifiable? How specific is the message text, and are there log files containing further details (and if so is the location of these files known)? More specific errors are useful anyway ('File not found' begs the question: 'Which file?'), but when searching the Web a generic error message can make the chance of success vanishingly small.

Often the initial search doesn't find a solution, but does find a news group or Web site that has answers to similar queries. The next challenge is how to make my request. Asking a good question significantly improves the chance that someone will be willing and able to assist. Eric S Raymond, probably best known at the author of *The Cathedral and the Bazaar*, maintains a Web page entitled 'How to Ask Questions the Smart Way' (http://catb.org/~esr/faqs/smart-questions.html) that contains a number of examples of ways to ask good questions.

Finally when the problem is solved we can make sure the solution is published to help other people who have the same problem later. If we posted a question to a support site and got a couple of replies, then post a final message saying thank you and stating which piece of advice fixed the problem; if none of the proffered advice helped then post the method that was finally successful. We may be providing support to users ourselves, and if so we may be able to make problem resolutions searchable via the Internet. Again, to help with searching for answers, be specific in any reply.

It is often remarked that computers are an increasing part of all aspects of modern life. Ten or fifteen years ago people may have played with a computer as a hobby, now it is likely to be a tool they rely on for a variety of tasks. Unfortunately these tools seem to lack reliability; and I think something is wrong when these types of failures are so common. However, such users are often very motivated to sort out their problem, and if sufficient information is provided in the public domain they may well be able to resolve their own troubles.

Preventing problems

As the proverb puts it: 'An ounce of prevention is worth a pound of cure'. I hope that, in line with our 'professionalism in programming' motto, the articles in this issue will help us write programs that are part of the solution to such computer woes.

Letter to the Editor

In 'The Essence of Success' (Overload 82), Alan Griffiths says that, when enhancing an open source project such as Mozilla, it would deliver the same value to a business to use the enhanced version internally as to take the additional cost of getting a submission approved. While I certainly don't want to criticise the overall argument of the article, I'd like to question this particular example, as it would be nice if businesses were encouraged to continue to submit their changes, and I can think of some business benefits of doing so. For one thing, once your code is accepted into the main branch you won't have to keep re-integrating it against newer versions, and if it's part of the whole project then you may later benefit from fixes and enhancements to it that are contributed by others. Also, it results in a tighter quality check on your work, and you might learn something in the process. Finally, there's a bit of publicity to be gained. So I don't think it's true that getting a submission approved will deliver no more value to a business than just using it internally, but this is not a criticism of the article as a whole.

Silas S Brown
ssb22@cam.ac.uk

Watersheds and Waterfalls

An introductory look at segmenting images into regions using a landscape analogy.

Introduction

In my last two articles [Golodetz], I talked about template metaprogramming, something most people would regard as a relatively 'pure' programming subject. For that reason, I thought it would make a nice change to look at something more applied this time: image segmentation.

Computing is one of those fields that encompasses so many diverse topics that it's impossible to know about all of them in depth. As programmers, we know more or less how to code (although there's always room for improvement!), but beyond a certain point, merely knowing how to write code in a certain language isn't enough. Before we can sit down and start hacking, we need to know what to code: we need, in short, domain knowledge.

Over the course of your programming career, there will inevitably be moments when you are forced outside your comfort zone, when you are asked to do things for which you initially lack the technical knowledge to do a good job. When that happens, you have three choices:

1. **Give up.** This could involve panicking and trying to get the job allocated to someone else, quitting, or otherwise finding creative ways to avoid making a substantive attempt at the problem. Winners don't do this. (Well, unless it's also a really boring and pointless job, in which case they probably quit before it got to this point, anyway.)
2. **Fudge it.** This is the 'it doesn't really matter if we do an inferior job as long as we get it out of the way' approach. It works quite well for problem sheets, but it's not the optimal approach to writing life-support code for the space shuttle. It's even more unprofessional than giving up and admitting that you don't know what you're doing.
3. **Learn.** This involves looking upon your lack of knowledge as an opportunity. You can't possibly be expected to know everything about every possible subject. Being asked to do something unfamiliar is an excuse to expand what you know, giving you another tool in your armoury for the future. (Of course, some subjects are just genuinely hard, and may be beyond your mental capacity to comprehend them: if so, you may eventually have to reluctantly go to 1, but it's worth keeping at it for as long as possible first. You'd be surprised at what you can understand if you really try.)

During the course of my doctoral research at Oxford, I've found myself making this very choice numerous times. Whilst learning your way out of a problem is clearly the right way forward, it's not always easy: my most recent stumbling block, image processing, is only the latest in a long line of things which I've initially found it difficult to get to grips with. If you're faced with a difficult learning problem, then, the following tips might come in handy:

- Focus in on exactly what it is you're trying to find out – you can't learn about the whole field all at once.
- Don't get bogged down by reading papers which you don't understand – there's a reason you don't understand them. Either you don't have the prerequisite knowledge to understand them (most of the time) or they're badly written (some of the time), but either way, stubbornness will get you nowhere. Spend your time more productively by finding a different paper which explains things more clearly.
- Try and work out exactly what it is about an algorithm you don't understand – it's rarely the case that you understand 'absolutely nothing' about something. Identify the gaps in your knowledge and read as many papers (or other sources) as it takes to try and plug them. (This doesn't always have to be large numbers of papers – a few good papers will do just fine.)
- Start by trying to understand an algorithm at a high-level, then gradually focus in on the details. The lowest-level stage generally involves a pen and paper. (Incidentally, a pen and paper really does tend to be more effective for this stage than a computer. Don't make the mistake of substituting typing for thinking.)
- Once you think you've got something, try writing it out clearly in your own words or explaining it to someone who doesn't know anything about it. This is a good way to identify the bits you still don't quite get.

Ultimately, the efficacy of these methods is best demonstrated by the fact that until recently, the amount I knew about image processing could have been written on the back of a postage stamp! There's rarely a bad time to learn about something new...

The segmentation problem

Image processing is itself a large field, so we're going to focus on one problem in particular, that of segmenting images. The idea is to classify

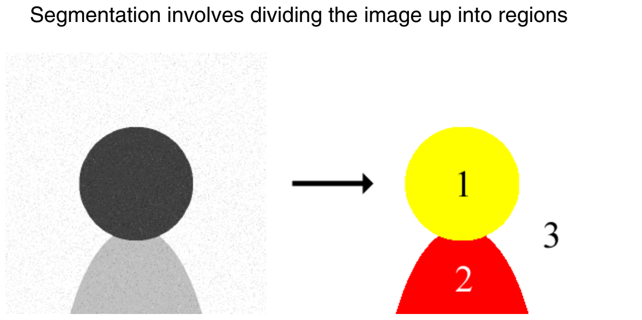
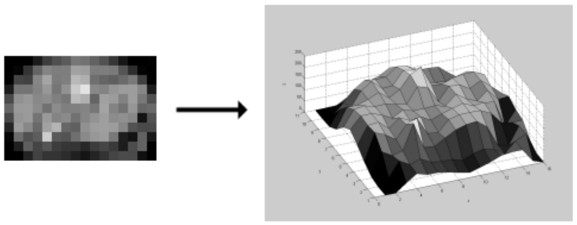


Figure 1

Stuart Golodetz has been programming for 13 years and is studying for a computing doctorate at Oxford University. His current work is on the automatic segmentation of abdominal CT scans. He can be contacted at stuart.golodetz@comlab.ox.ac.uk

Imagine **poking holes in the landscape** at each of its regional minima, then **lowering the landscape** slowly into a lake



Viewing an image as a landscape

Figure 2

the pixels of the image into different regions (see Figure 1 for a very simple example). This is important for the medical imaging work I'm doing because it allows relevant organs to be identified on the images; the results can then be used for further processing, e.g. for building 3D visualizations of the organs.

There are numerous different approaches to the problem [Varshney], but the one I want to look at in particular is a method from the field of mathematical morphology called the watershed transform.

The landscape analogy

Anyone who's ever tried writing a terrain renderer will be familiar with the concept of using a heightmap to represent a landscape. Essentially, you have a 2D array of values, which can be viewed as an evenly-spaced finite grid located in the (*x*,*y*) plane. Each value represents the *z* height of the landscape at that point. (More formally, we could say that we have a rectangular domain $\Omega \subset Z^2$ and a function $f: \Omega \longrightarrow Z$ which gives the height of the landscape for every point in Ω.)

The insight behind the watershed transform is that a grey-scale image is nothing but such a 2D array of values, so it can be viewed as a landscape, where the heights are given by the grey levels in the image (see Figure 2).

4-connected pixels are only connected to their horizontal and vertical neighbours, whereas 8-connected ones are connected to their diagonal neighbours as well

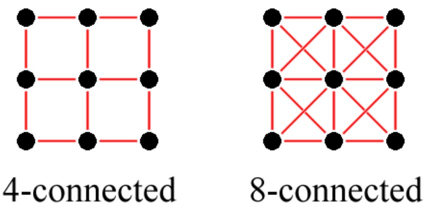


Figure 3

We now define a few terms. A pixel $p \in \Omega$ has height $f(p)$ and *neighbour set* $N(p)$, according to some implementation-specific definition of neighbourhood (usually pixels are considered to be either 4-connected or 8-connected: see Figure 3).

A *singular minimum* of the image is a point whose neighbours are all strictly higher than it. (More formally, *p* is a singular minimum if for all $p' \in N(p)$, $f(p') > f(p)$.) A *plateau* of the image is a maximal set of (two or more) connected pixels of equal altitude. A *minimal plateau* is a plateau from which it is impossible to descend, and a *non-minimal* plateau is the opposite. Together, the *singular minima* and *minimal plateaux* of the image form the *regional minima* of the image.

Flooding the image landscape

Imagine poking holes in the landscape at each of its regional minima, then lowering the landscape slowly into a lake. As the water begins to rise, pools of water will gradually form at each of the minima (see Figure 4). For reasons which will become obvious later, we'll call each pool of water the *catchment basin* of its associated minimum. If the water keeps rising, eventually some of the catchment basins will meet (see Figure 5): at this point, we imagine constructing a dam, or *watershed*, to keep them apart

Viewing an image as a landscape

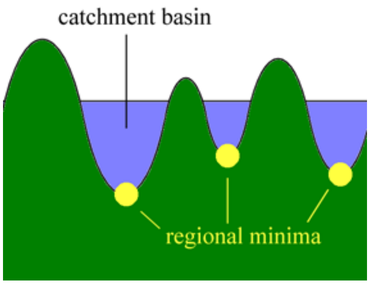


Figure 4

Two catchment basins meet

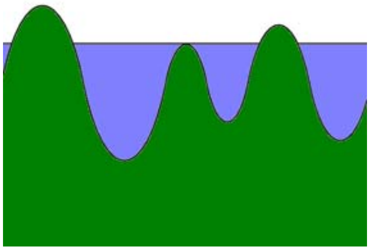


Figure 5

> **Water is notorious for taking the path of least resistance, and in this case that means running downhill to a regional minimum via a path of steepest descent**

Building a dam at the join point

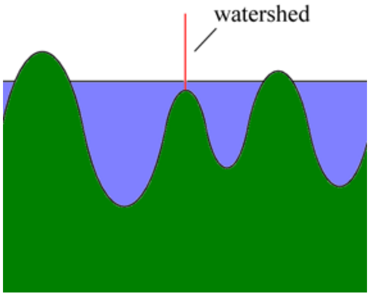


Figure 6

The final division into regions

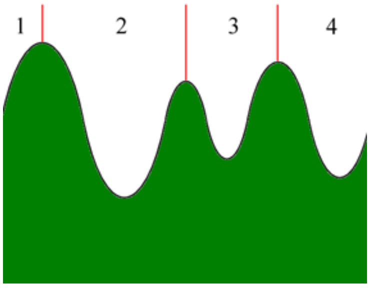


Figure 7

In the gradient image, homogeneous features of interest end up in the valleys

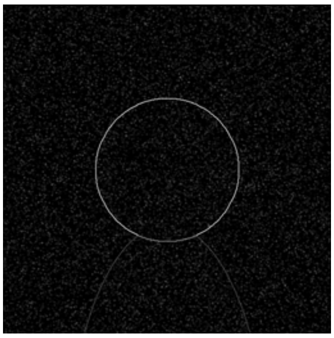


Figure 8

(see Figure 6) and continue the flooding. When the landscape has been fully flooded, the dams we've created will separate the different regional minima from each other at the points where their catchment basins would have met, thus segmenting the image into a number of regions, each associated with a different regional minimum (see Figure 7).

This all sounds fine in theory, but there are a number of problems to be overcome:

1. There's no reason to suppose that the things we want to segment (e.g. organs in a medical image) will have low grey levels and thus be in a 'valley'. If they are on top of a 'hill', we're apparently stuffed.
2. The idea of the algorithm is one thing, but implementing it from this definition is far from straightforward.
3. Images can have large numbers of regional minima, especially in the presence of noise. Most of them are irrelevant, but the end result is that the image will end up being greatly oversegmented.

Using the gradient image

The first problem is definitely image-specific. However, for medical images, we're helped greatly by the fact that the organs we're segmenting (e.g. the kidneys and liver) tend to be relatively homogeneous, i.e. the grey levels are relatively similar throughout the organ. This implies that the image landscape is relatively flat over each organ, or in other words that the gradient is small there. By contrast, the gradient at the edges of organs will, we hope, be quite large. By using the gradient of the original image, then, instead of the image itself, we engineer a situation where the things we're trying to segment are (by and large) in the valleys of the image, and are (more or less) surrounded by hills (see Figure 8).

Rainfall simulation

Implementing the algorithm using a flooding method is only one way of approaching the problem. It's certainly possible to implement it that way (e.g. [Rambabu]), but it's sometimes helpful to think about things from a different perspective. Instead of thinking of the watershed process as one of flooding, we can now imagine rain falling on each point of the landscape from above.

Water is notorious for taking the path of least resistance, and in this case that means running downhill to a regional minimum via a path of steepest descent (i.e. a path where at each stage we choose to move to a lowest neighbour of the current point). This gives us an idea for an alternative approach to the watershed transform: we can think of the catchment basin of a regional minimum as including all points whose unique path of steepest descent leads to the minimum in question. Where a point has more than one path of steepest descent, it can be allocated to any of the resulting minima according to programmer preference (see Figure 9).

Dealing with non-minimal plateaux

A problem occurs when we think about which way water should run off a non-minimal plateau. There are various different approaches to dealing with this [e.g. Bieniek; Osma-Ruiz; Stoev]; for the purposes of this article, we're going to use the approach in [Meijster] and transform the image to remove all non-minimal plateaux at the outset, thus making it what is called *lower-complete*.

The flow direction at a point is ambiguous if it has **more than one path** of steepest descent

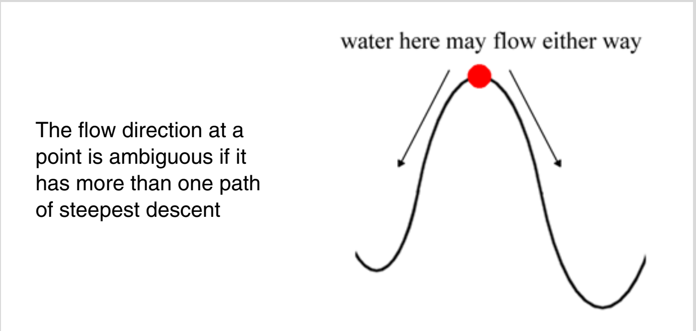

The flow direction at a point is ambiguous if it has more than one path of steepest descent
water here may flow either way

Figure 9

In essence, the idea is to raise all plateau pixels up by their distance from the plateau edge (see Figure 10a). Of course, doing this naively doesn't work, since we might end up changing the ordering of the plateau pixels with respect to the other pixels in the image (see Figure 10b). The solution, then, is to find the maximum amount by which we're going to raise a plateau pixel and multiply the base image by that before raising any pixels on the plateau: this has the effect of 'spreading the landscape out' to accommodate the new altitudes in the middle (see Figure 10c).

Implementing this is relatively straightforward using a queue (see Listing 1, building a lower-complete function). The basic idea is to add all pixels with a lower neighbour to the queue at the start, then gradually flood out from them a level at a time (essentially a breadth-first search), incrementing the distance counter after each level.

Fletching

Having constructed a lower-complete image (see Figure 11b), the rest is all downhill (excuse the pun). Our next step is to construct an arrow on each node (see Figure 11c). In the case of a regional minimum, the arrow is a self-loop back to the node itself (for a minimal plateau, one of the nodes is chosen as a canonical element of the plateau and all the other nodes point to it); for all other nodes, the arrow points to a lowest neighbouring node (i.e. it points in the direction of a path of steepest descent). We also take the opportunity to numerically label all the canonical elements during this phase of the process.

The implementation (see Listing 2, constructing arrows on the nodes) uses an interesting disjoint-set forest data structure which I'll talk about in more detail next time. The idea is to combine all the minimum points into their respective regional minima using this data structure, and make all the other non-minimal points point to one of their lowest neighbours.

Labelling the image

The final step of the basic watershed algorithm is to label the pixels (see Listing 3, labelling all the pixels by following the arrow chains). This involves following the arrows for each pixel to find which minimum it's associated with, and giving it the same label as that minimum. To speed things up, we use path compression when following a path to a minimum (i.e. we make all the arrows on the path point to the minimum once we've found it). Interestingly, this bears many similarities to the implementation of the disjoint-set data structure I just mentioned: we'll see more of this next time.

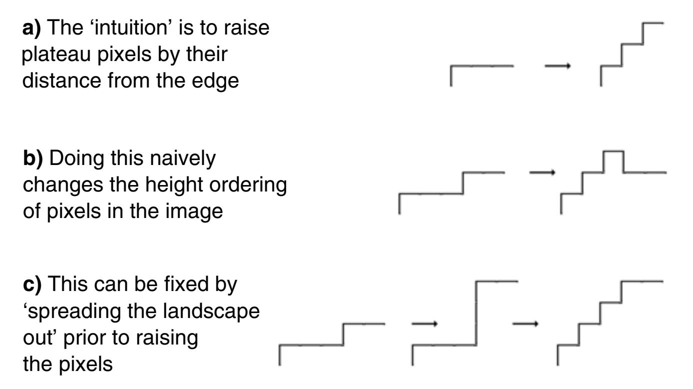
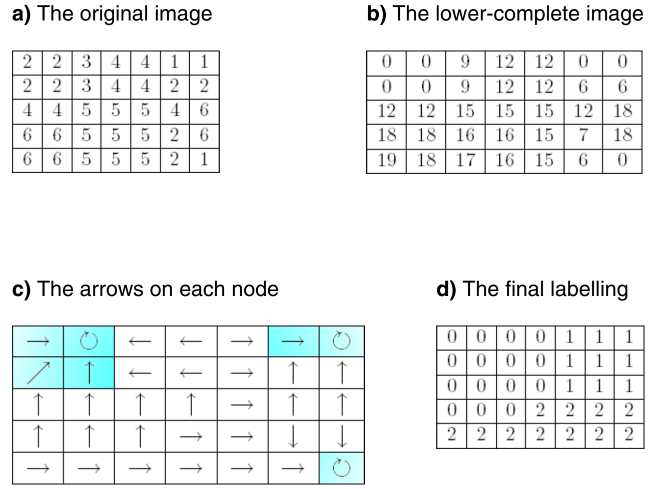

a) The 'intuition' is to raise plateau pixels by their distance from the edge
b) Doing this naively changes the height ordering of pixels in the image
c) This can be fixed by 'spreading the landscape out' prior to raising the pixels

Figure 10

The result (see Figure 11d) is, in general, an oversegmented image on which further processing is then required.

Pride comes before a 'fall

At this stage, we can be tolerably pleased with our efforts. We've managed to segment the image into a number of regions, each associated with a regional minimum of the image, but we haven't yet got what we need. In



a) The original image

2	2	3	4	4	1	1
2	2	3	4	4	2	2
4	4	5	5	5	4	6
6	6	5	5	5	2	6
6	6	5	5	5	2	1

b) The lower-complete image

0	0	9	12	12	0	0
0	0	9	12	12	6	6
12	12	15	15	15	12	18
18	18	16	16	15	7	18
19	18	17	16	15	6	0

c) The arrows on each node

→	↻	←	←	→	→	↻
↗	↑	←	←	→	↑	↑
↑	↑	↑	↑	→	↑	↑
↑	↑	↑	→	→	↓	↓
→	→	→	→	→	→	↻

d) The final labelling

0	0	0	0	1	1	1
0	0	0	0	1	1	1
0	0	0	0	1	1	1
0	0	0	2	2	2	2
2	2	2	2	2	2	2

Figure 11

particular, our image is greatly over-segmented, because most of the regional minima aren't 'relevant': they're not associated with the objects of interest in our image. A general method to solving this problem involves trying to merge some of the regions together to reduce the overall number of regions in our image and obtain a better segmentation. One algorithm which takes this approach is the waterfall algorithm described in [Marcotegui], which I'll talk about next time. In and of itself, this still won't give us what we need for medical images, but it will take us a little closer to where we need to be. It also produces reasonable results on some non-medical images (e.g. the ones used in the paper). To get acceptable results for medical images, we have to make use of anatomical knowledge to process the results of application-independent algorithms like the waterfall, but that's something that is very much still a work in progress! Till next time... ■

```
Build-Lower-Complete(Function<Ω,Z> image)

Function<Ω,Z> lc;
Queue<PixelCoords> queue;
// A marker indicating when we need to increase
// the distance value.
PixelCoords marker(-1,-1);
// Initialise the queue with pixels that have a
// lower neighbour.
foreach(PixelCoords p ∈ Ω) {
  lc(p) = 0;
  foreach(PixelCoords neighbour ∈ N(p)) {
    if(image(neighbour) < image(p)) {
      queue.push(p);
      // To prevent it being queued twice.
      lc(p) = -1;
      break;
    }
  }
}
// Compute a function which indirectly indicates
// the amount by which we need to raise the
// plateau pixels (see the referenced paper for
// more details).
int dist = 1;
queue.push(marker);
while(!queue.empty()) {
  p = queue.pop();
  if(p == marker) {
    if(!queue.empty()) {
      queue.push(marker);
      ++dist;
    }
  }
```

Listing 1

```
  else
  {
    lc(p) = dist;
    foreach(PixelCoords neighbour   N(p)) {
      // If the neighbouring pixel is at the
      // same altitude and has not yet been
      // processed.
      if(image(neighbour) == image(p) &&
         lc(neighbour) == 0)
      {
        queue.push(neighbour);
        // To prevent it being queued twice.
        lc(neighbour) = -1;
      }
    }
  }
}
// Compute the final lower-complete function.
// Note that at this point, dist holds the
// amount by which we want to multiply the base
// image.
foreach(PixelCoords p ∈ Ω) {
  if(lc(p) != 0) {
    lc(p) = dist * image(p) + lc(p) - 1;
  }
}
return lc;
```

Listing 1 (cont'd)

References

[Bieniek] 'An efficient watershed algorithm based on connected components', Andreas Bieniek and Alina Moga; Albert-Ludwigs-Universitt Freiburg; *Pattern Recognition 33* (2000) pp. 907-916

[Golodetz] 'Functional Programming Using C++ Templates (Parts 1 and 2)'; *Overload 81* and *Overload 82*

[Marcotegui] *Fast Implementation of Waterfall Based on Graphs*, B. Marcotegui and S. Beucher; Ecole des Mines de Paris

[Meijster] *A Disjoint Set Algorithm for the Watershed Transform*, Arnold Meijster and Jos B. T. M. Roerdink, University of Groningen

[Osma-Ruiz] 'An improved watershed algorithm based on efficient computation of shortest paths', Víctor Osma-Ruiz et al., Universidad Politecnica de Madrid; *Pattern Recognition 40* (2007) pp. 1078-1090

[Rambabu] 'An efficient immersion-based watershed transform method and its prototype architecture', C. Rambabu and Indrajit Chakrabarti; *Journal of Systems Architecture 53* (2007) pp. 210-226

[Stoev] *RaFSi – A Fast Watershed Algorithm Based on Rainfalling Simulation*, Stanislav L. Stoev, University of Tübingen

[Varshney] *Abdominal Organ Segmentation in CT Scan Images: A Survey*, Lav R. Varshney, Cornell University

```
Construct-Arrows(Function<Ω,Z> lc)

Function<Ω,PixelCoords> arrows;
Function<Ω,PixelCoords> labels;

// Add all the minimum points to a disjoint set
// forest.
int labelCount = 0;
DisjointSetForest<PixelCoords> minima;

foreach(PixelCoords p ∈ Ω) {
  if(lc(p) == 0) {
    labels(p) = labelCount++;
    minima.add_node(p);
  }
}

foreach(PixelCoords p ∈ Ω) {
  if(lc(p) == 0) {
    // Union any neighbouring minimum points
    // into the same regional minimum.
    foreach(PixelCoords neighbour ∈ N(p)) {
      if(lc(neighbour) == 0) {
        minima.union_nodes(labels(p),
                           labels(neighbour));
      }
    }
  }
  else {
    // Find a lowest neighbour and make this
    // point's arrow point to it.
    PixelCoords lowestNeighbour(-1,-1);
    int lowestNeighbourValue = INT_MAX;

    foreach(PixelCoords neighbour ∈ N(p)) {
      if(lc(neighbour) < lowestNeighbourValue) {
        lowestNeighbour = neighbour;
        lowestNeighbourValue = lc(neighbour);
      }
    }

    // There will always be a lowest neighbour
    // here since the function's lower-complete.
    arrows(p) = lowestNeighbour;
  }
}
```

Listing 2

```
// Assign new labels to the canonical points of
// the regional minima and make the arrows of
// the non-canonical points point to them.
labelCount = 1;
foreach(PixelCoords p ∈ Ω) {
  if(lc(p) != 0) continue;
  int root = minima.find_set(labels(p));
  if(root == labels(p)) {
    // This is a canonical point.
    arrows(p) = p;
    labels(p) = labelCount++;
  }
  else {
    arrows(p) = minima.value_of(root);
  }
}

return (arrows, labels);
```

Listing 2 (cont'd)

```
Resolve-All(Function<Ω,PixelCoords> arrows,
            Function<Ω,Z> labels)
foreach(PixelCoords p ∈ Ω) {
  Resolve-Pixel(p, arrows, labels);
}

Resolve-Pixel(PixelCoords p,
              Function<Ω,PixelCoords> arrows,
              Function<Ω,Z> labels)

PixelCoords parent = arrows(p);
if(parent != p) {
  parent = Resolve-Pixel(parent);
  labels(p) = labels(parent);
}
return parent;
```

Listing 3

The PfA Papers: Deglobalisation

More history of Parameterise from Above as Kevlin Henney looks at Simpletons and the Borg.

Whenever it is mentioned, the PARAMETERIZE FROM ABOVE (PFA) pattern is often discussed in connection with – or rather, as a counterpoint or antidote to – the SINGLETON pattern. Perhaps the other most recurrent feature of any PFA discussion is to note its lack of a proper written description [Deigh2007]:

> Much has been written about the pattern identified by Kevlin Henney as PARAMETERIZE FROM ABOVE. Indeed, much has been written about it (just search the Web for 'Parameterize from Above' and 'Parameterise from Above'), but as a pattern it has never been written up. Much has also been written on accu-general about how Kevlin should get around to writing it up properly!

Well, I can reveal that a short write-up of the pattern, forces and all, was due to appear in the German magazine *JavaSPEKTRUM* nearly five years ago. The 'Patterns in Java' column was in essence a continuation of the identically named column in the defunct *Java Report*, but in German thanks to Martina Buschmann's translating skills. The article containing the missing link and unwritten pattern was to have followed the article entitled 'One or Many?' [Henney2003]. Indeed, the raison d'être for 'One or Many?' was to set up the follow-on article by focusing on some of SINGLETON's problems. The follow-on article would then have presented two pattern descriptions: PARAMETERIZE FROM ABOVE and then SINGLETON revisited in the light of PARAMETERIZE FROM ABOVE. 2003, however, was quite a lean year for *JavaSPEKTRUM*, which ultimately survived by slimming down and reducing overheads. Alas one such overhead was the translation of the 'Patterns in Java' column. There were two notable consequences of the cost-cutting exercise: (1) a short but nonetheless documented form of PARAMETERIZE FROM ABOVE never saw the light of day; (2) *JavaSPEKTRUM* survived, unlike its sister magazine *Java Report* two years before it.

This fourth instalment of 'The PfA Papers' takes time to explore some of the SINGLETON-related territory that in part motivates PARAMETERIZE FROM ABOVE, including revisiting 'One or Many?' and some extracts from its unpublished and unfinished successor.

Simpleton

To set the scene, let's kick off with the draft opening of the unpublished article:

> For many years I have used SINGLETON as an indicator of pattern and design maturity. Programmers who rarely use it either do not know about it by name – and hence are probably unfamiliar with design patterns – or they understand it fully – and hence understand the relationship between design and patterns. A few questions and some conversation usually differentiate one end of the scale from the other. However, there is a significant group of programmers that falls between these two extremes, and they are the principal users of SINGLETON. Without meaning to, many programmers are creating code that is difficult to evolve, hard to comprehend, awkward to test and resistant to change in the belief that they are following good practice. Why is the design assumed to be good? Because it has been documented in a book, *Design Patterns*, that is widely recognised as a purveyor of good practice. Trust in a design pattern is an important quality, but it should not be unquestioning. A critical eye is needed when evaluating any design.

Superficially, the SINGLETON pattern comes across as a simple idea [Gamma+1995]:

> Ensure a class only has one instance, and provide a global point of access to it.

The associated class diagram for the pattern also looks simple enough: a single class. What could be simpler? Sadly, this apparent simplicity belies the accidental complexity introduced by the pattern. (It is both interesting and in some ways cautionary that many etymologies of the word singleton state that it is derived from single and patterned after simpleton.) The emphasis of the pattern's intent is often misread [Henney2003]:

> The race to embrace the apparent convenience offered by second half of the sentence – '... global point of access...' – often eclipses the necessity of the first half and the classification of the pattern as a creational rather than a structural pattern.
>
> The notion of providing a global point of access is seen by many as the main motivation for the pattern, whereas first and foremost SINGLETON is a factory pattern: it concerns the encapsulated creation of objects. What is it encapsulating? Instance control. In this case, to be precise, the existence of no more than a single instance – an exceedingly rare constraint in practice.
>
> A quick examination of the majority of so-called SINGLETONs in code reveals that they are global variables and not SINGLETONs: they just happen to share some of the same solution structure, but not the same motivating problem, forces and consequences, all of which are required to correctly characterise a pattern. Most of these misapplications either do not enforce instance control or the uniqueness of the instance that they control happens to be a coincidence rather than a genuine constraint.

Part of the misreading of the pattern is down to the individual reader, and much is down to the cultural interpretation of the pattern, but some credit (or, indeed, debit) must also go to the original Gang of Four write-up. When compared to the other pattern descriptions in *Design Patterns*, the entry on SINGLETON seems surprisingly weak. It lacks the much of the detail and considered discussion that characterises the other patterns in the catalogue. For example, only benefits and no liabilities are listed for it, which seems not only surprising to the modern reader but is also out of step with the more even-handed appraisal of the other patterns in the catalogue. Similarly, no worked example is presented to motivate the pattern, only the following [Gamma+1995]:

Kevlin Henney is a long-standing member of ACCU, joining before it actually was ACCU and contributing to Overload when it was numbered in single digits. He recently co-authored two volumes in the Pattern-Oriented Software Architecture series, *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*. Kevlin can be contacted at kevlin@curbralan.com.

beyond the initial sugar rush of **apparent coding convenience and cleverness**, its subsequent inconvenience can manifest itself in a number of ways

> It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

No satisfactory explanation is offered for the opening sentence, which is also not quite right: the pattern constrains the number of instances to at most one, not exactly one. And no justification is offered for any of the examples, which are essentially incorrect for one reason or another. Much of the remaining pattern description is devoted to the mechanics of the pattern's solution rather than the understanding of the problem.

With this background – and in spite of its notable and intentional absence from the Gang of Four's list of 'simplest and most common patterns' – it is perhaps unsurprising that SINGLETON is most commonly applied as a souped-up global. The result is typically a design that has all the issues associated with globals, but without the relative simplicity. This situation inspired Kent Beck's full and candid (if somewhat flippant) write-up of SINGLETON [Beck2003]:

> How do you provide global variables in languages without global variables? Don't. Your programs will thank you for taking the time to think about design instead.

The problems that SINGLETON can introduce into a design are not always immediately apparent, and nor is its misapplication. However, beyond the initial sugar rush of apparent coding convenience and cleverness, its subsequent inconvenience can manifest itself in a number of ways: it complicates testing, safe and simple threading, architectural configurability and pluggability, adaptation and evolution of code, design reasoning and code readability, application start-up and shutdown, and so on. Many applications and implications of SINGLETON's design lead programmers to come up with increasingly 'clever' solutions – books, magazines and a multitude of web pages offer a dizzying variety of cure-alls. These may sometimes reveal coding virtuosity, but they serve mostly to highlight a missed trick: if these workarounds seem to recur in the context SINGLETON, why not address the root cause rather than attempt to repeatedly and ingeniously mollify its effects?

Program to an interface, not an instance

One of the most useful guidelines and enduring sound bites from the *Design Patterns* is 'program to an interface, not an implementation' [Gamma+1995]. This encourages a style of design that is strongly encapsulated. Rather than working with glorified C structs dressed as classes with assembleresque getters and setters, develop classes that have rich, intentional public interfaces. Rather than working with class hierarchies rooted in classes that are mostly (or completely) concrete toolkits of default functionality, favour hierarchies rooted in IDL-style interfaces – interface in Java and C#, fully abstract classes in C++. In situations where duck typing is used, as in C++'s template system or the type systems of dynamic languages such as Ruby and Python, program according to the concept or protocol in question without mention or assumption of a particular concrete type. The resulting design style is loosely coupled, testable and refreshingly clear. But the consequences of this guideline do not stop there [Henney2003]:

> We can extend this with a further principle:
>
> Program to an interface, not an instance.
>
> This second principle can be considered a deeper reading and consequence of the first.

This second principle was also the working title for the unpublished article, the draft of which included the following observation:

> Knowledge of multiplicity should be encapsulated rather than shouted from the rooftops, hence PFA rather than SINGLETON, which litters the code with the assumption. A more complete failure of encapsulation it would be hard to find.

And also the following:

> Which brings us to the question of the multiplicity constraint itself. One or many. SINGLETON focuses on the particular multiplicity constraint of 1, but it could in principle be any number N. The idea is that in a given situation the total number of instances of a particular class is constrained to a single instance. OK, but who is doing the constraining? The class itself or the situation that it finds itself in? In the SINGLETON pattern the constraint is expressed and enforced in the class. However, what determines the multiplicity of a given type of object in a particular scenario is just that: the details of the scenario. The coupling in SINGLETON is often back to front: the application that uses the class should constrain its instance count, not the class itself. So the general rule should be that if a given infrastructure or design situation demands an instance limit of N, the enforcement should be at that level and not within the instance type. The property belongs to the application, not the class.

The necessity of the instance constraint is one of the reasons I often refer to SINGLETON as the HIGHLANDER pattern – 'There can be only one'. Most abuses of the pattern fall into the obvious category of being nothing more than global variables by another name, but a great many abuses relate to coincidence: only a single object happens to be needed. For an example of this kind of misuse we need look no further than *Design Patterns* itself.

The Gang of Four's description of the STATE pattern suggests that SINGLETON can be used to implement classes that represent individual state behaviour, but which are otherwise stateless (a terminology collision that gives rise to the intriguing concept of stateless states). If an object has no associated state, and its behaviour but not its identity is all that matters, it makes little difference to code whether there is one instance or many. Each instance can be substituted transparently for any other, hence why we might favour having just a single instance for all uses. However, note that this is not SINGLETON: there is no requirement on the type that 'there can be only one', and hence no need to restructure the type so that it prevents public constructability, offers a global point of access, and so on. Instead, we have a possibility to reduce the number of instances based on 'there may be only one'. In other words, the code that wants to share the instance can simply declare a static variable of the stateless type, leaving the stateless type untouched and free of all the unnecessary SINGLETON coding clutter.

In the interests of fairness it is worth saying that I fell into precisely the same trap a number of years ago. Following a crisp and prescient description of SINGLETON's actual applicability, along with a spirited denouncement of its widespread misapplication, I then wrote the following [Henney1997]:

> Another suitable application of SINGLETON is as an optimisation in cases where the identity of an object is not an issue, and there is no variation across instances of a class.

Publish and be damned! Redemption can be found in the draft for the unpublished article:

> The important point here is that SINGLETON is really a creational pattern, which means that the main characteristic being constrained and controlled is creation, not access. Most of the few good uses of SINGLETON should, therefore, be invisible.

Such invisibility is achieved by decoupling usage of the SINGLETON instance from its access and creation. Instead of using the sole instance globally via the SINGLETON's class name, the class should implement an interface and the instance should be passed around according to that interface. In other words, not according to its concrete implementation type or the absolute path to the actual instance. In this sense, the locality and use of any Singleton becomes just like the locality and use of any other loosely coupled use of a factory. The incomplete, unpublished description of PARAMETERIZE FROM ABOVE includes the following paragraph:

> Create the object at the highest level it is needed and known and pass it down from there. This stresses the importance of having a clear layered structure. On close inspection most global concepts turn out to be regional rather than global.

In essence what has been described here is little more than a classic separation of concerns: separate how an object is used from how it is created. Such advice is both common and unsurprising and, put in such simple terms, it is also advice that is quite easy to follow. Furthermore, whether or not an object is actually a SINGLETON pretty much ceases to be an issue because you are programming to an interface, not an instance.

Resistance is useful

The view that SINGLETON is more of a problem than a solution has become increasingly widespread to the point that steering clear of it is considered to be common knowledge by many development communities. Where once the conspicuous use of SINGLETON was considered a sign of patterns know how and object-oriented design expertise, nearly a decade and a half after its publication it is increasingly seen as an obsession of the larval stage of pattern learning.

However, simply making SINGLETON a pariah without exploring the problem at hand or showing reasoned alternatives does not constitute constructive advice. The absence of a reasoned guideline can lead to the adoption of alternatives that solve superficial rather than deep issues. One such approach is the MONOSTATE pattern [Henney2005]:

> This pattern can be considered a salve for programmers who dislike the guilt-by-association of employing SINGLETON. It also plays the role of syntax sugaring for those who want a less cumbersome usage syntax than SINGLETON's. A MONOSTATE [Ball+1997] object looks like an ordinary object but shares its state static-ally with all other instances of the class, leading to 'spooky action at a distance' and aliasing problems when the state changes. If SINGLETON is the problem, MONOSTATE as a treatment can be worse than the problem, although some developers mistake it for a cure. MONOSTATE is also known affectionately and revealingly as the BORG pattern.

It is true that with SINGLETON 'the programmer is obliged to use counter-intuitive syntax to access objects' [Ball+1997], but that ugliness should be taken as a hint. It is not the problem to be solved; it is the signpost to a deeper problem. Likewise, the use of the name BORG [Martelli2001] for MONOSTATE in the Python community seems more like an early warning signal rather than an invitation.

A MONOSTATE object is not always a drop-in replacement for a SINGLETON, but in many of its applications it is seen as comparable or equivalent (with the added bonus of absolving guilt and syntax). From a developmental point of view it also has some significant drawbacks [Ball+1997]:

1. The sharing that is occurring may be overly subtle since all instances of a MONOSTATE class may appear to be unique.
2. The subtlety of sharing can lead to aliasing problems, e.g., calling mutators on one instance of a MONOSTATE object will update all instances. This can cause subtle bugs if programmers don't understand that all instances are aliases.

And I would contend that even though this list is briefer than it should be, it is damning enough, especially when the implications are considered more deeply – testing, code evolution, threading, etc. It is difficult to recommend a technique that is subtle and surprising, indiscreetly messing with the fundamental notion that different objects represent different objects, when the simpler alternative is to pass objects around as arguments – in plain sight and without the need for covert semantics.

One of the things about patterns is that they are recurring (and, of course, not all that recurs is necessarily good). This doesn't just mean that you read about patterns in books, hear about them at conferences or see them in other people's code. It also means that they are reinvented and rediscovered by individuals on a regular basis. A long time ago, in the land before GoF, I ended up solving an instance management issue by creating a design that I later recognised as MONOSTATE. Without going into details, what I can say with the benefit of hindsight is that I wish I had known about SINGLETON: I may not be particularly fond of it, but it would have been a major improvement. To paraphrase Dorothy Parker, MONOSTATE is not a pattern to be tossed aside lightly; it should be thrown with great force. ■

References

[Ball+1997] Steve Ball and John Crawford, 'Monostate Classes', *C++ Report 9(5)*, SIGS, May 1997

[Beck2003] Kent Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2003

[Deigh2007] Teedy Deigh, 'A Practical form of OO Layering', *Overload 78*, April 2007, http://accu.org/index.php/journals/1327

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison-Wesley, 1995

[Henney1997] Kevlin Henney, 'Java Patterns and Implementations', BCS OOPS Patterns Day, October 1997, http://www.two-sdg.demon.co.uk/curbralan/papers/JavaPatternsAndImplementations.html

[Henney2003] Kevlin Henney, 'One or Many?' is the English original used for translation and publication in German as 'Eins oder Viele?', *JavaSpektrum*, September 2003, http://www.two-sdg.demon.co.uk/curbralan/papers/javaspektrum/OneOrMany.pdf

[Henney2005] Kevlin Henney, 'Context Encapsulation', EuroPLoP 2005, July 2005, http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ContextEncapsulation.pdf

[Henney2007a] Kevlin Henney, 'The PfA Papers: From the Top', *Overload 80*, August 2007, http://accu.org/index.php/journals/1411

[Henney2007b] Kevlin Henney, 'The PfA Papers: The Clean Dozen', *Overload 81*, October 2007, http://accu.org/index.php/journals/1420

[Henney2007c] Kevlin Henney, 'The PfA Papers: Context Matters', *Overload 82*, December 2007, http://accu.org/index.php/journals/1432

[Martelli2001] Alex Martelli, 'Singleton? We don't need no stinkin' singleton: the Borg design pattern', http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/66531

The Regular Travelling Salesman, Part 2

Richard Harris explores more of the mathematics of modelling problems with computers.

Last time I described the regular travelling salesman problem and we discovered that whilst the shortest tour was trivial to determine, the distribution of tour lengths was a little more difficult. Specifically, the factorial growth of the number of tours as the number of cities increased limited us to tours of no more than 14 cities.

So how should we go about reducing the computational expense? Well, if we can spot any more symmetries we might be able to exploit them. Taking a look at every 5 city tour, fixing the first city as usual, might give a hint as to whether any more symmetries exist.

Figure 1 shows the complete set of tours for 5-city fixed-start regular TSP. Clearly there's a symmetry we've not yet taken into account since only 4 of the 24 possible tours are distinct from one another!

So where is it?

Well, perhaps surprisingly, it's the most obvious of them all. The fixed starting city and tour direction symmetries that we have already addressed exist for all TSPs. This final symmetry results from our tour being around a regular polygon. Specifically, it results from the fact that we can rotate and reflect the city labels on the polygon.

Trivially, reversing the city labels is equivalent to reversing the direction of the tour. More interestingly, rotating the city labels is not necessarily equivalent to rotating the starting city.

This is easily demonstrated by taking a tour that does not have rotational symmetry, say the second in Figure 1, rotating the labels and then checking whether rotating the starting point results in the same tour.

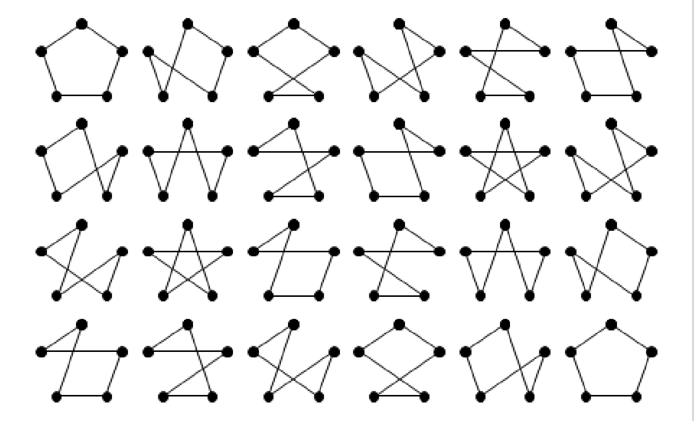


Figure 1

Rotating labels for a 5-city regular TSP

Initial tour:	0-1-2-4-3
Rotate labels:	1-2-3-0-4
Rotate starting point:	0-4-1-2-3

Figure 2

Figure 2 clearly shows that rotating the labels results in a tour that cannot be created by rotating the starting point.

Before we embark on constructing an algorithm to efficiently generate the minimal set of symmetrically distinct tours, it's probably worth figuring out how many of them there are. The analysis is easiest for tours with a prime number of cities, *p*.

First of all, we should count the number of tours for which any rotation of the labels is equivalent to changing the starting city. Trivially, these tours must move the same number of vertices around the perimeter of the polygon at each step since if two consecutive steps were of different lengths, rotating the labels would mean that one of the cities would be followed by a different step, as illustrated in Figure 3.

For odd, and hence prime, regular tours there are

$$c_1 = \frac{p-1}{2}$$

such tours (the factor of ½ resulting from the reflectional symmetry).

For prime regular TSPs, all remaining distinct tours must have a layout such that no rotation of the labels is equivalent to a rotation of the starting city.

To see why, assume that rotating the labels *k* times, where *k* is not equal to either 1 or *p*, is equivalent to the initial tour with a different starting city. Rotating it another *k* times must also be equivalent, as must rotating it any multiple of *k* times, since we return to an equivalent of the starting tour every time. We should also note that rotating the labels more than *p* times is equivalent to rotating them that number modulo *p*.

For each label, *l*, and any multiple of the *k* rotations, *m*, *l* will be mapped to

$$l \to (l + mk)(\bmod p)$$

Now, it is a property of prime numbers that repeatedly applying this mapping must result in every number between 0 and *p*-1. For *p* equal to 5 and *k* equal to 2 we can demonstrate this by enumerating every step

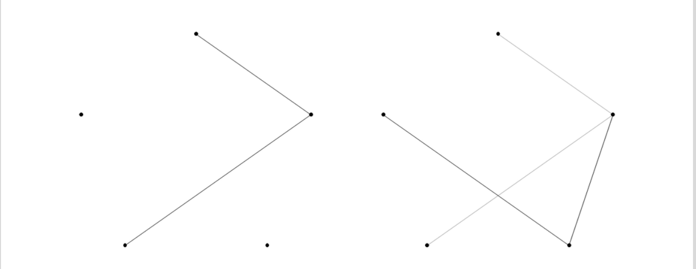


Figure 3

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

if we're willing to sacrifice a little accuracy, we can simply generate a random subset of the tours

$$0 \to 0+2 = 2 \to 2$$
$$2 \to 2+2 = 4 \to 4$$
$$4 \to 4+2 = 6 \to 1$$
$$1 \to 1+2 = 3 \to 3$$
$$3 \to 3+2 = 5 \to 0$$

Whilst this is a reasonable illustration of this fact, it is not remotely akin to a proof. To prove it, we first look for a multiple of the k rotations that maps every label to itself.

$$l = (l + mk)(\bmod p)$$

We can subtract the label value from both sides of the equation giving

$$0 = mk(\bmod p)$$

Since p is prime, mk can only be a multiple of p if either m or k are a multiple of p.

This demonstrates that if k is not equal to a multiple of p, repeatedly applying k rotations of the labels must generate all other rotations of the labels before returning to the initial layout. Therefore if k label rotations lead to a tour which is equivalent to the first, we simply keep repeating them to find that every possible rotation must also be equivalent.

So the remaining distinct tours must generate $2p^2$, rather than $2p$, tours since they have the extra rotational symmetry of the labels. The total number of tours must be equal to the sum of them both, giving

$$2p^2 c_p + 2pc_1 = p!$$
$$c_\mathrm{p} = \frac{p! - 2pc_1}{2p^2}$$
$$= \frac{p! - p(p-1)}{2p^2}$$
$$= \frac{(p-1)! - (p-1)}{2p}$$

Hence the total number of distinct tours is given by

$$c = c_p + c_1$$
$$= \frac{(p-1)! - (p-1)}{2p} + \frac{p-1}{2}$$

Whilst this does save us an extra order of magnitude, it's still factorial complexity so it doesn't really help us all that much.

For odd non-prime regular TSPs, the situation is even worse. This is because there will be some distinct tours for which there *is* a partial rotation of the labels that is equivalent to a rotation of the starting city. Since these will generate fewer tours, there must be more distinct tours.

For even regular TSPs, it is only the tour around the perimeter of the polygon for which label and starting city rotation are equivalent. This leads, by a similar argument, to a lower bound for the number of distinct tours being

```
void
tsp::sample_tour(tour_histogram &histogram,
                 size_t samples)
{
  distances dists(histogram.vertices());
  tour t(histogram.vertices());
  generate_tour(t.begin(), t.end());
  while(samples--)
  {
    std::random_shuffle(t.begin()+1, t.end());
    histogram.add(tour_length(t, dists));
  }
}
```

Listing 1

$$c = \frac{(n-1)! - 2}{2n} + 1$$

The reason that this is only a lower bound is that, as for odd non-prime regular TSPs, there exist partial label rotations that are equivalent to starting city rotations which will each generate fewer tours.

I rather suspect that it's not therefore worth the effort it would require to develop an efficient algorithm for enumerating the symmetrically distinct tours.

So how should we proceed?

Well, if we're willing to sacrifice a little accuracy, we can simply generate a random subset of the tours. If the subset is large enough the resulting distribution of tour lengths should be approximately equal to that of the complete set of tours.

Fortunately for us, the standard library also includes a function for generating random permutations of sequences that we can use to generate our random tours; `std::random_shuffle`. Once again, we will ignore the reflectional symmetry for the sake of simplicity. We will still, however, exploit the rotational symmetry, although this time it's to distribute the samples as evenly as possible amongst the full set of tours. Listing 1 shows sampling the tour histogram.

Since we're no longer bound by the number of cities, but by the number of samples we might as well take a look at histograms for large numbers of cities.

Figure 4 and Figure 5 record the results of 1,000 and 10,000 city regular TSPs with 10,000,000 and 100,000,000 samples respectively. Table 1 shows the approximate average tour lengths for these histograms.

It seems reasonable that the limit of the average tour length is going to be approximately $1.27n$. The question that remains is why? Can we deduce a formula for the limit of the distribution of tour lengths for very large numbers of cities?

For extremely large numbers of cities, most steps in a regular TSP tour are more or less independent to those that have already been taken. It is only

The **central limit theorem** states, for a very wide class of distributions, that the sum of a set of **independently drawn random numbers** is normally distributed

when the majority of cities have been visited that the choice of steps will be restricted to limited regions on the circumference of the polygon.

There is a statistical theorem called the law of large numbers which states that as *n* tends to infinity, the sum of *n* random numbers independently drawn from any single given distribution tends to *n* times the average of that distribution. If our assertion that the steps are more or less independent to each other is valid we should be able to approximate the average tour length with *n* times the average step length. For very large *n*, the average step length will be approximately equal to the average distance between two randomly selected points on the circumference of a circle of unit radius. In the same way that we can add up a finite set of step lengths and divide by the number of them to get the average, we can integrate the lengths of steps to cities separated by an angle of θ around the circumference and divide by 2π.

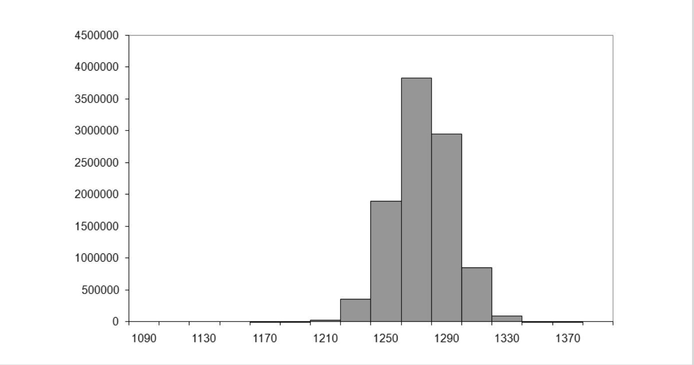


Figure 4

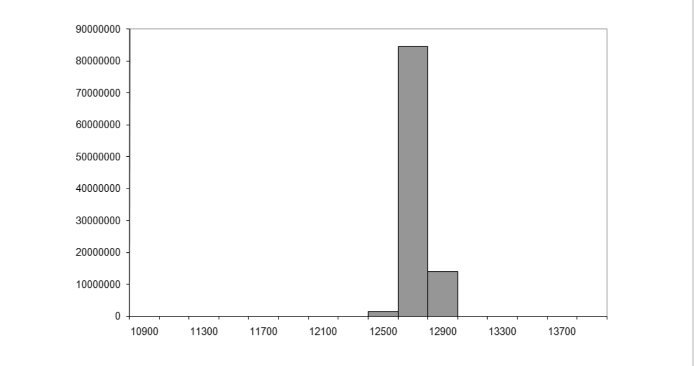


Figure 5

n	mean	mean/n
1,000	1,274.5	1.27
10,000	12,725.1	1.27

Table 1

$$
\begin{aligned}
\mu &= \frac{1}{2\pi}\int_0^{2\pi} 2\sin\frac{\theta}{2}d\theta \\
&= \frac{1}{\pi}\int_0^{2\pi} \sin\frac{\theta}{2}d\theta \\
&= \frac{1}{\pi}\left[-2\cos\frac{\theta}{2}\right]_0^{2\pi} \\
&= \frac{1}{\pi}((-2\times -1)-(-2\times 1)) \\
&= \frac{4}{\pi}\approx 1.27
\end{aligned}
$$

This clearly confirms that our expectation of the average tour length was correct, but is not enough for us to completely determine how the tour lengths are distributed.

There is another statistical theorem we can use to help us; the central limit theorem. The central limit theorem states, for a very wide class of distributions, that the sum of a set of independently drawn random numbers is normally distributed. Because of this property, it shows up in a vast number of places.

The normal distribution is defined in terms of both the average, μ, and the standard deviation, σ, of the numbers drawn from it. The standard deviation is a measure of how different on average the numbers in a set are from their mean and it is calculated as follows

$$
\begin{aligned}
E(x) &= \mu = \frac{1}{n}\sum_i x_i \\
E((x-\mu)^2) &= \sigma^2 \\
&= \frac{1}{n}\sum_i (x_i-\mu)^2 \\
&= \frac{1}{n}\sum_i (x_i^2 - 2\mu x_i + \mu^2) \\
&= \frac{1}{n}\sum_i x_i^2 - 2\mu\frac{1}{n}\sum_i x_i^2 + \mu^2\frac{1}{n}\sum_i 1 \\
&= \frac{1}{n}\sum_i x_i^2 - 2\mu^2 + \mu^2 \\
&= \frac{1}{n}\sum_i x_i^2 - \mu^2
\end{aligned}
$$

Note that in this context *E* means the expected, or average, value.

Given these values the normal distribution is defined by its cumulative density function, or cdf, which is the function in *x* that gives the probability that a random number will be less than *x*.

those of you for whom the word ‘trigonometry’ conjures images of sinister maths teachers intent on ruining your life … might want to skip ahead and just trust me

$$F(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{x} e^{\frac{-(t-\mu)^2}{2\sigma^2}}\,dt$$

Unfortunately this integral does not have a closed form, meaning a simple formulaic, solution. The derivative, known as the probability density function, or pdf, is simple to calculate however and its graph is shown in Figure 6 (the normal distribution pdf).

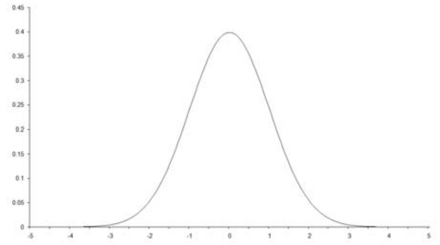


Figure 6

So the final piece of the puzzle is to calculate the average squared distance between two cities in a regular TSP, which we can use to determine which normal distribution is applicable. We could approximate it with an integral over the circle again, but there is an approximate formula for regular TSPs with a number of cities equal to a multiple of 4, so we may as well use it.

$$\begin{aligned} \mathrm{E}(x^2) &= \frac{1}{n}\sum_i l_i \\ &= \frac{1}{n-1}\sum_i^{n-1} 4\sin^2\frac{k\pi}{n} \\ &\approx \frac{1}{n}\sum_i^{n} 4\sin^2\frac{k\pi}{n} \end{aligned}$$

This may not look very easy to solve, but appearances can be deceptive. The trick is to exploit some trigonometric identities. It does get a little bit fiddly though, so those of you for whom the word ‘trigonometry’ conjures images of sinister maths teachers intent on ruining your life (or at least that double period after lunch on Thursdays) might want to skip ahead and just trust me.

Now, the identities in question are

$$\begin{aligned} \sin\theta &= \sin(\pi-\theta) \\ \cos\theta &= \sin\left(\theta+\frac{\pi}{2}\right) = \sin\left(\frac{\pi}{2}-\theta\right) \\ \sin^2\theta+\cos^2\theta &= 1 \end{aligned}$$

We can use these by splitting the sum into four parts (Equation 1).

$$\begin{aligned} E(x^2) &\approx \frac{4}{n}\sum_{k=1}^{n}\sin^2\frac{k\pi}{n} \\ &= \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=\frac{n}{4}+1}^{\frac{n}{2}}\sin^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=\frac{n}{2}+1}^{\frac{3n}{4}}\sin^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=\frac{3n}{4}+1}^{n}\sin^2\frac{k\pi}{n} \end{aligned}$$

Equation 1

Now since the last three terms are sums over ¼*n* steps offset by a constant factor, we can simply shift the constant factor from the index into the sum itself (Equation 2).

$$E(x^2) \approx \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{4}\right) + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{2}\right) + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{3\pi}{4}\right)$$

Equation 2

The next point to note is that we can perform the second and fourth sums backwards by subtracting from the last angle in each sum (Equation 3).

$$\begin{aligned} E(x^2) &\approx \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=0}^{\frac{n}{4}-1}\sin^2\left(\frac{\pi}{2}+\frac{k\pi}{n}\right) + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{2}\right) + \frac{4}{n}\sum_{k=0}^{\frac{n}{4}-1}\sin^2\left(\pi-\frac{k\pi}{n}\right) \\ &\approx \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{\pi}{2}+\frac{k\pi}{n}\right) + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\frac{k\pi}{n}+\frac{\pi}{2}\right) + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\left(\pi-\frac{k\pi}{n}\right) \end{aligned}$$

Equation 3

Now we exploit the identity that equates the sine of the angle added to or subtracted from ½π to the cosine of the angle (Equation 4).

$$\begin{aligned} E(x^2) &\approx \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\cos^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\cos^2\frac{k\pi}{n} + \frac{4}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n} \\ &= \frac{8}{n}\sum_{k=1}^{\frac{n}{4}}\sin^2\frac{k\pi}{n}+\cos^2\frac{k\pi}{n} \end{aligned}$$

Equation 4

picking the location of the next city in a TSP is equivalent to picking the next city in a tour

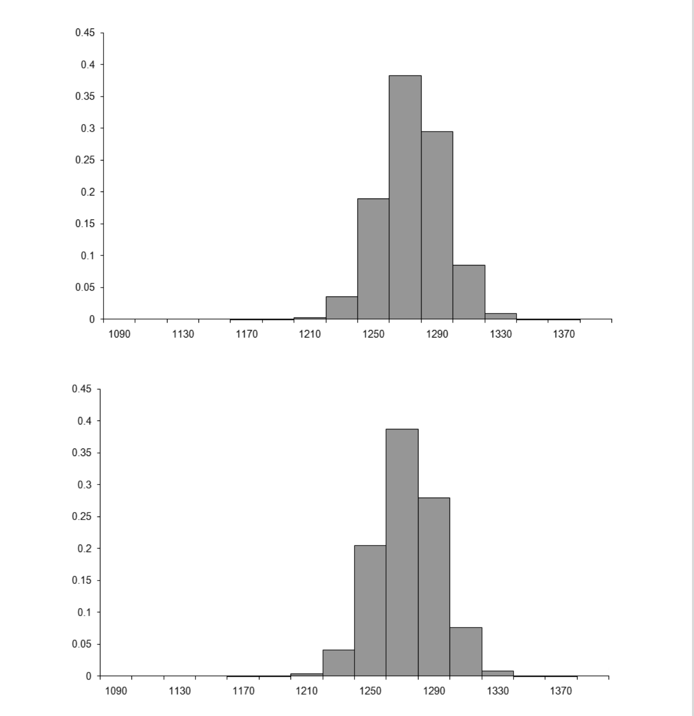


Figure 7

Finally, we exploit the identity that equates the sum of the squares of the sine and cosine of an angle to 1 to yield the result.

$$E(x^2) \approx \frac{8}{n}\sum_{k=1}^{\frac{n}{4}} 1 = \frac{8}{n} \times \frac{n}{4} = 2$$

Therefore, the standard deviation of the step length is given by

$$\sigma^2 = 2 - \frac{16}{\pi^2}$$

$$\sigma = \sqrt{2 - \frac{16}{\pi^2}}$$

In addition to stating that the sums of random numbers are normally distributed, the central limit theorem states that the specific normal distribution will have an average equal to *n* times that of their distribution and a standard deviation equal to the square root of *n* times that of their distribution.

This means that the distribution of tour length of a regular TSP with *n* cities should tend, for large *n*, towards

$$N\left(\frac{4n}{n}, \sqrt{2n - \frac{16n}{\pi^2}}\right)$$

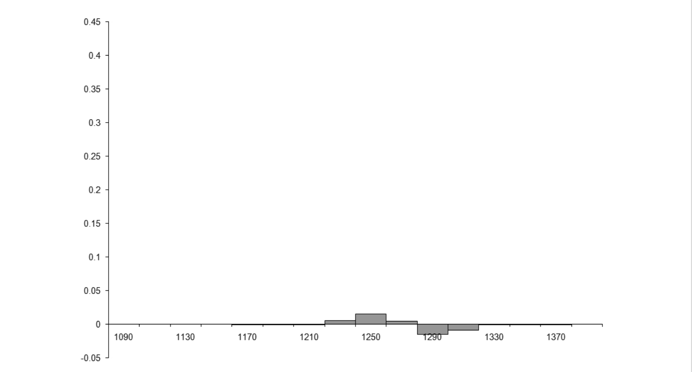


Figure 8

Figure 7 compares the histogram we'd expect from the normal distribution (at the bottom) to that we generated by sampling the 1,000 city tour (at the top). Under the assumption of normality a bucket with mid point *x* and width *w* should contain the proportion of the samples given by

$$F\left(x + \frac{w}{2}; \frac{4n}{\pi}, \sqrt{2n - \frac{16n}{\pi^2}}\right) - F\left(\left(x - \frac{w}{2}\right); \frac{4n}{\pi}, \sqrt{2n - \frac{16n}{\pi^2}}\right)$$

Well, despite the fact that the assumption that the tour steps are independent is demonstrably false these look remarkably similar, a fact borne out by the histogram of the difference between them, plotted on the same scale in Figure 8.

In fact, there exists a mathematical technique for determining the likelihood that a sample histogram is consistent with a particular distribution. I strongly suspect that it would indicate that the sample histogram is not consistent with the normal distribution, but since we have already acknowledged that our assumptions are false we shouldn't find that surprising. Nevertheless, given that the maximum difference is of the order of 0.015, or 1½%, it's not too bad an approximation.

So can we perform a similar analysis on the usual type of TSP?

Well, let's assume that the cities are evenly randomly distributed on the unit square. If we're interested in the average tour length of all possible tours we should firstly note that we can take a tour of a random TSP by simply visiting each city in order. Furthermore, every possible tour can be generated by changing the labels and using the same scheme, since we can view the labels as instructions as to the order in which we should visit them. This means that picking the location of the next city in a TSP is equivalent to picking the next city in a tour. Since the former is independent of the cities already chosen, the latter must be independent the steps already taken, satisfying the independence requirement of the law of large numbers.

However, the distribution of step lengths is dependent on where in the square we are currently located, and this breaks the requirement that the step lengths are identically distributed. However, there is another version of the law of large numbers which states that the sum of independent random numbers from different distributions will tend to the sum of the averages of those distributions. Known as the strong law of large numbers, it requires that the standard deviations of those distributions have a particular property which happens to be satisfied if they do not grow without limit, or in other words are all less than some finite number. For cities in the unit square, this will be true for any reasonable definition of distance and so this approximation is actually *more* reasonable for normal TSPs than it is for regular TSPs.

Unfortunately, the expression for the average step length is a little bit more complicated this time. If we represent a pair of points by their coordinates on the unit square, (x, y) and (a, b), we have

$$
\begin{aligned}
E(l) &= \frac{\int_0^1\int_0^1\int_0^1\int_0^1 dist((x, y), (a, b))dx\ dy\ da\ db}{\int_0^1\int_0^1\int_0^1\int_0^1 1\ dx\ dy\ da\ db} \\
&= \int_0^1\int_0^1\int_0^1\int_0^1 dist((x, y), (a, b))dx\ dy\ da\ db
\end{aligned}
$$

Once again, this is because the integral is the continuous limit of a sum. The fraction is the limit of the sum of the distances between all pairs of points in the unit square divided by the number of such pairs.

For the usual definition of distance, the integral becomes

$$
E(l) = \int_0^1\int_0^1\int_0^1\int_0^1 ((x-a)^2 + (y-b)^2)^{\frac{1}{2}}\ dx\ dy\ da\ db
$$

Whilst I'm not willing to assert that this does not have a closed form solution, it's too complicated for me to attempt. If we change the cost of travelling between cities to the square of the distance it becomes a little easier however.

$$
\begin{aligned}
E(l) &= \int_0^1\int_0^1\int_0^1\int_0^1 (x-a)^2 + (y-b)^2\ dx\ dy\ da\ db \\
&= \int_0^1\int_0^1\int_0^1\int_0^1 x^2 - 2ax + a^2 + y^2 - 2by + b^2\ dx\ dy\ da\ db \\
&= \int_0^1\int_0^1\int_0^1 \left[bx^2 - 2abx + a^2b + by^2 - b^2y + \frac{1}{3}b^3\right]_0^1\ dx\ dy\ da \\
&= \int_0^1\int_0^1\int_0^1 x^2 - 2ax + a^2 + y^2 - y + \frac{1}{3}\ dx\ dy\ da
\end{aligned}
$$

Continuing in the same vein leads to the result

$$
\begin{aligned}
E(l) &= \frac{1}{3} - \frac{1}{2} + \frac{1}{3} + \frac{1}{3} - \frac{1}{2} + \frac{1}{3} \\
&= \frac{4}{3} - 1 \\
&= \frac{1}{3}
\end{aligned}
$$

The average cost of a tour should therefore be approximately equal to $^1/_3n$.

If you are interested, I invite you to investigate the accuracy of this approximation for different numbers of cities. You may be surprised as to just how accurate it actually is.

So is there anything more that can be said about the statistical properties of tours through TSPs? Well certainly, but not by me as I am afraid I have exhausted my mathematical toolbox. But this is an active area of research and a great many results have been found, of which just a few are described below.

Beardwood, Halton and Hammersley [Beardwood59] proved that the expected length of the shortest path through a random TSP tends to a value proportional to the square root of the number of cities.

Jaillet [Jaillet93] examined the probabilistic TSP in which each city has a probability that it may be skipped during the tour and provided bounds on the expected length of the shortest tour.

Agnihothri [Agnihothri98] examined the travelling repairman problem in which a repairman must travel to fix machines when they break down and developed a mathematical model with which expected travelling time, amongst other things, can be calculated.

And you, dear reader, may be able to shed further light on the properties of either the regular or normal TSP, and if you do please let me know. ■

Acknowledgements

With thanks to Larisa Khodarinova for a lively discussion on group theory that led to the correct count of distinct tours and to Astrid Osborn and John Paul Barjaktarevic for proof reading this article.

References and further reading

[Agnihothri98]Agnihothri, 'A Mean Value Analysis of the Travelling Repairman Problem', *IEE Transactions*, vol. 20, pp. 223-229, 1998.

[Beardwood59]Beardwood, Halton and Hammersley, 'The Shortest Path Through Many Points', *Proceedings of the Cambridge Philosophical Society*, vol. 55, pp. 299-327, 1959.

[Jaillet93] Jaillet, 'Analysis of Probabalistic Combinatorial Optimization Problems in Euclidean Spaces', *Mathematics of Operations Research,* vol. 18, pp. 51-71, 1993.

Archimedes, *On the Measurement of the Circle*, c. 250-212BC.

Basel and Willemain, 'Random Tours in the Travelling Salesman Problem: Analysis and Application', *Computational Optimization and Applications*, vol. 20, pp. 211-217, 2001.

Clay Mathematics Institute 'Millennium Problems', http://www.claymath.org/millennium.

Hoffman and Padberg, 'Travelling Salesman Problem', *Encyclopedia of Operations Research and Management Science*, Gass and Harris (Eds.), Kluwer Academic, Norwell, MA, 1996.

Apology

A number of errors crept into the first of this series of articles. The first of which was the title – while trying to clean up the layout of the article header, I forgot that the second article (the one in this issue) also covered the travelling salesman problem (and that 'part one' referred to this, not to the series as a whole).

We also managed to confuse 2θ with 2π on page 8 and to replace the Greek character μ with a question mark in Table 2 (page 12).

Apologies to our readers and to Richard Harris for these errors.

Alan (ed).

Testing Visiting Files and Directories in C#

Testing code that accesses the file system is not straightforward. Paul Grenyer looks at what is involved.

In my previous article, ‘Visiting Files and Directories in C#’ [VFDC#], I looked at how to use C# to remove a source tree and developed the code into an enumeration method [EnumMethod] and visitor [Visitor] compound that can be used for general purpose file and directory traversal (Listing 1).

The article did not discuss any form of automated testing. This not only makes me very uncomfortable, it also means the classes cannot be modified or refactored safely. In this article I am going to look at how to write automated tests for **`DirectoryTraverser`** and discuss the differences between unit and integration testing and when to use them.

Unit and integration testing

The message about automated testing is finally getting through. However, many organisations are still not doing it. The majority of those that are using automated tests cannot see past unit testing or their unit tests are a mixture of unit tests and integration tests. I’m going to use an example to demonstrate the difference between a unit test and an integration test and explain when each should be used and more importantly when unit tests should not. Despite what some fanatics may refuse to concede, it is not always appropriate or sensible to use a unit test and can often introduce unnecessary complexity or volume code for little or no gain, especially in noticeable performance.

```
public interface IDirectoryVisitor
{
  void EnterDirectory(DirectoryInfo dirInfo);
  void VisitFile(FileInfo fileInfo);
  void LeaveDirectory(DirectoryInfo dirInfo);
}
public class DirectoryTraverser
{
  private IDirectoryVisitor visitor;
  public DirectoryTraverser(
  IDirectoryVisitor visitor)
  {
    this.visitor = visitor;
  }
  public void Traverse(string path)
  {
    Traverse(new DirectoryInfo(path));
  }
  private void Traverse(DirectoryInfo dirInfo)
  {
    visitor.EnterDirectory(dirInfo);
    foreach (
     DirectoryInfo subDir in
     dirInfo.GetDirectories())
    {
      Traverse(subDir);
    }
    foreach (FileInfo file in dirInfo.GetFiles())
    {
      visitor.VisitFile(file);
    }
    visitor.LeaveDirectory(dirInfo);
  }
}
```

Listing 1

Imagine you have a class, called **`HistoricPrices`**, that is used to retrieve historic stock prices from a database. The class constructor takes an **`IDBConnection`** interface that is used to make direct calls to a database to retrieve the prices. A test is run every time the project containing **`HistoricPrices`** is compiled by the developer on their local machine. The developer needs the test to run quickly and give accurate repeatable results, so instead of passing in a real **`DBConnection`** (the class that implements **`IDBConnection`** in production) object they pass in a fake [Fake]. The fake object has a number of hard-coded recordsets that are mapped to a set of predetermined SQL strings. This means that every time a particular SQL string is executed via the **`IDBConnection`** interface to the fake object, the same recordset is always returned. As the fake object does not actually talk to the database the recordset is returned in (almost) zero time. This makes the tests very fast and easy to run. This is a *unit* test.

The unit test only tests a very small part of the system, in this case just one class. Every system should be tested as fully as possible. In the case of **`HistoricPrices`** the interaction with a real database is vital and should also be tested. The developer still requires the tests to give accurate repeatable results, so instead of using a live database a test database is constructed, tested against using the production **`DBConnection`** object and torn down every time the test is run. The test takes some time to run and tests the interaction between a **`HistoricPrices`** object and a real database. This is an *integration* test.

In my experience, creating and dropping a SQL Server database on a developer spec machine can take anything up to 20 seconds. That is a long time to wait, so the developer is likely to be less keen to run the test every time they compile. Therefore the integration test should be run, at the very least, prior to a release and ideally prior to checkin, as part of a nightly build of the entire system and/or as part of continuous integration [CI].

So, to recap, unit testing is about removing dependencies and writing tests that run in (almost) zero time, so that they can be run every time the compiler is invoked. Integration tests test the interaction between at least two things. For example between two objects or between an object and a database or an object and a file. This means that integration tests potentially

Paul Grenyer has been a member of ACCU since 2000. He founded the ACCU Mentored Developers and serves on the committee. Paul now contracts at an investment bank in Canary Wharf. He can be contacted at paul.grenyer@gmail.com

all that needs to be done to write a unit test is to mock these out and create a suitable factory to create the mocks

take a while to run and traditionally this puts developers off running them every time they invoke the compiler.

Unit testing DirectoryTraverser

The only parts of the .Net library that intrude into `DirectoryTraverser` are `DirectoryInfo` and `FileInfo`. So all that needs to be done to write a unit test is to mock these out and create a suitable factory to create the mocks when testing and the real objects when in production. The problem is that `DirectoryInfo` and `FileInfo` do not already have suitable interfaces, so new interfaces must be written and the original classes wrapped. That in itself is not too much trouble. Unfortunately `GetDirectories` and `GetFiles` methods return a `DirectoryInfo[]` and `FileInfo[]` respectively and therefore their return values must be mapped onto arrays of the new interface types. Suddenly you have much more test code than code being tested and it is far more complex, so a unit test is not appropriate or worthwhile in this case.

Integration testing DirectoryTraverser

As previously stated, integration testing is the testing of how one or more units or modules work together. In the case of `DirectoryTraverser` we need to test how it integrates with the file system. This involves creating a known set of directories and files, traversing them, checking the results and, of course, cleaning up afterwards.

Creating directories and files

Before we consider how to create directories and files we must consider *where* to create them. It needs to be a place where the following test code can find them and where they won't interfere with anything else in the file system. We could just pick a path, such as `c:\temp`, but that would only work on Windows machines. .Net has the ideal solution:

```
Path.GetTempPath()
```

The `GetTempPath` method returns the path to a directory that can be used to store temporary files and directories. The path is specific to the operating system in use. So on Windows it's something along the lines of:

```
C:\Documents and Settings\user\Local Settings\Temp
```

And on Linux it's along the lines of:

```
/home/user/tmp
```

Now we're ready to create the directories and files. .Net makes this very easy as both the `Directory` and `File` classes have create methods that take a string:

```
// Creating a directory.
Directory.CreateDirectory("…");
// Creating a file
FileStream str = File.Create("…");
str.Close();
```

The only thing to remember is that the `File.Create` method returns an open `FileStream` and must be closed so the file can be accessed in the test, as you cannot rely on `Dispose` being called in time.

As well as using different temporary paths, different operating systems also use different directory separators. The `Path.Combine` method will concatenate *two* strings together with the correct separator for the operating system. So, for the path

```
test\dir1\dir2
```

you would need to write:

```
string fullPath = "test";
fullPath = Path.Combine(fullPath, "dir1" );
fullPath = Path.Combine(fullPath, "dir2" );
```

This is more than a little tedious, especially for long or multiple paths, and is not especially clear. It would be much nicer to be able to write:

```
string fullPath = MakePath("test", "dir1",
   "dir2");
```

The C# `params` keyword allows methods to take a varying number of arguments and access them as an array, which in turn allows us to write:

```
static private string MakePath(
   params string[] tokens)
{
  string fullpath = "";
  foreach (string token in tokens)
  {
    fullpath = Path.Combine(fullpath, token);
  }
  return fullpath;
}
```

All that's left is to define the directories and file we want to create. This is easily and clearly done using arrays (Listing 2).

```
string testFolderPath = Path.GetTempPath();
string[] testDirs = {
  MakePath(testFolderPath, "Test"),
  MakePath(testFolderPath, "Test","dir1"),
  MakePath(testFolderPath, "Test","dir1", "dir2"
),
  MakePath(testFolderPath, "Test","dir1", "dir2",
   "dir3") };
string[] testFiles = {
  MakePath(testFolderPath, "Test", "dir1",
   "file1.txt"),
  MakePath(testFolderPath, "Test", "dir1",
   "file2.txt"),
  MakePath(testFolderPath, "Test", "dir1", "dir2",
   "file3.txt"),
  MakePath(testFolderPath, "Test", "dir1", "dir2",
   "file4.txt") };
```

Listing 2

I have chosen a very simple directory structure: A root directory called `Test` with two nested subdirectories, `dir1` and `dir2`, each containing two text files `file1.txt`, `file2.txt`, `file3.txt` and `file4.txt` and a further empty subdirectory, `dir3`. This allows us to test that **`DirectoryTraverser`**:

1. Enters and leaves directories in sequence.
2. Visits all files.
3. Visits all subdirectories.
4. Empty directories are handled correctly.

Creating the files and directories is easily accomplished using a couple of foreach's:

```
// Create directories
foreach (string dir in testDirs)
{
  Directory.CreateDirectory(dir);
}
// Create files
foreach (string file in testFiles)
{
  FileStream str = File.Create(file);
  str.Close();
}
```

The directories and files should be removed after the test. This can be achieved using the **`Directory.Delete`** method and setting the recursive flag (see Visiting Files and Directories in C#):

```
Directory.Delete(testFolderPath + "Test", true);
```

Finally the create and delete code needs to be put into the **`SetUp`** and **`TearDown`** methods of an NUnit [NUnit] test fixture (Listing 3).

This is the best of many options I considered for creating the directories and files. Other options included:

- Traversing XML to get the structure.
- Storing the structure in a zip file that would be extracted each time the test was run.
- Writing the structure to an output file and using an external tool for test verification.

The advantage of the final solution is that it is simple and all in the code with no need for an external XML file, zip file or external tool.

Test Visitor

`DirectoryTraverser` won't do anything without a visitor. Of the four tests listed in the previous section, 1 is the easiest to implement. All that is needed is a stack. When **`EnterDirectory`** is called the directory path is pushed onto the stack. When **`LeaveDirectory`** is called, a path is popped from the stack and compared to the path of the directory just left. As long as they are the same the test passes (Listing 4).

```
[TestFixture]
public class DirectoryTraverserTest
{
  private readonly string testFolderPath =
   Path.GetTempPath();
  static private string MakePath(
   params string[] tokens)
  {
    string fullpath = "";
    foreach (string token in tokens)
    {
      fullpath = Path.Combine(fullpath, token);
    }
    return fullpath;
  }
  [SetUp]
  public void Setup()
  {
    Directory.CreateDirectory(testFolderPath);
    string[] testDirs = {
      MakePath(testFolderPath, "Test"),
      MakePath(testFolderPath, "Test","dir1"),
      MakePath(testFolderPath, "Test","dir1",
       "dir2",
      MakePath(testFolderPath, "Test","dir1",
       "dir2", "dir3")  ) };
    foreach (string dir in testDirs)
    {
      Directory.CreateDirectory(dir);
    }
    string[] testFiles = {
      MakePath(testFolderPath, "Test", "dir1",
       "file1.txt"),
      MakePath(testFolderPath, "Test", "dir1",
       "file2.txt"),
      MakePath(testFolderPath, "Test", "dir1",
       "dir2", "file3.txt"),
      MakePath(testFolderPath, "Test", "dir1",
       "dir2", "file4.txt") };
    foreach (string file in testFiles)
    {
      FileStream str = File.Create(file);
      str.Close();
    }
  }
  [TearDown]
  public void TearDown()
  {
    Directory.Delete(testFolderPath,true);
  }
}
```

Listing 3

```
class DirRecorder : IDirectoryVisitor
{
  private Stack<string> lastDir   =
   new Stack<string>();
  public void EnterDirectory(
     DirectoryInfo dirInfo)
  {
    lastDir.Push(dirInfo.FullName);
  }
  public void VisitFile(FileInfo fileInfo)
  {
  }
  public void LeaveDirectory(
     DirectoryInfo dirInfo)
  {
    Assert.AreEqual(
     lastDir.Pop(), dirInfo.FullName);
  }
};
```

Listing 4

To run the test, an instance of the visitor must be created and passed to an instance of **DirectoryTraverser**. Then the **DirectoryTraverser** instance must be passed the path to traverse:

```
[Test]
public void TraverseDirectory()
{
  string testPath = Path.Combine(
     testFolderPath, "Test");
  DirRecorder dirRecorder = new DirRecorder();
  DirectoryTraverser trav =
     new DirectoryTraverser(dirRecorder);
  trav.Traverse(testPath);
}
```

The easiest way to ensure that all file and directories are entered and all files are visited is to create a list of both and compare them to lists of expected directories and files. The order in which directories are entered and files are visited is not guaranteed, so all lists must be sorted. The expected lists can be generated at the same time as the physical directories and files are created (the highlighted code in Listing 5 shows the modifications).

```
[TestFixture]
public class DirectoryTraverserTest
{
  private List<string> expectedDirs =
   new List<string>();
  private List<string> expectedFiles =
   new List<string>();
  ...
  [SetUp]
  public void Setup()
  {
    ...
    foreach (string dir in testDirs)
    {
      expectedDirs.Add(dir);
      Directory.CreateDirectory(dir);
    }
    expectedDirs.Sort();
    ...
    foreach (string file in testFiles)
    {
      expectedFiles.Add(file);
      FileStream str = File.Create(file);
      str.Close();
    }
    expectedFiles.Sort();
  }
  ...
}
```

Listing 5

The visitor can be modified to keep a list of entered directories and visited files, and accessors provided to retrieve the lists. Again, highlighted code in Listing 6 shows the modifications.

Then the **TraverseDirectory** test can be modified to compare the lists of visited directories and files with the expected lists (Listing 7).

This completes the implementation of the integration test for DirectoryTraverser. Running the test with the NUnit console gives the output shown in Figure 1.

The NUnit GUI gives the satisfying green bar. I successfully ran this test on both Windows XP and SuSE [SuSE] Linux under Mono [Mono]. ■

Acknowledgments

Thank you to Kevlin Henney for guidance and sanity checking and the members of accu-general for healthy discussion on testing techniques. Thank you to Caroline Hargreaves, Roger Orr and Adrian Fagg for review.

```
class DirRecorder : IDirectoryVisitor
{
  private List<string> dirs = new List<string>();
  private List<string> files = new List<string>();
  private Stack<string> lastDir =
   new Stack<string>();
  public List<string> Dirs
  {
    get
    {
      dirs.Sort();
      return dirs;
    }
  }
  public List<string> Files
  {
    get
    {
      files.Sort();
      return files;
    }
  }
  public void EnterDirectory(
     DirectoryInfo dirInfo)
  {
    dirs.Add(dirInfo.FullName);
    lastDir.Push(dirInfo.FullName);
  }
  public void VisitFile(FileInfo fileInfo)
  {
    files.Add(fileInfo.FullName);
  }
  public void LeaveDirectory(
     DirectoryInfo dirInfo)
  {
    Assert.AreEqual(
     lastDir.Pop(), dirInfo.FullName);
  }
};
```

Listing 6

```
[TestFixture]
public class DirectoryTraverserTest
{
  private readonly string testFolderPath =
   Path.GetTempPath();
  private List<string> expectedDirs  =
   new List<string>();
  private List<string> expectedFiles =
   new List<string>();

  static private string MakePath(
   params string[] tokens)
  {
    string fullpath = "";
    foreach (string token in tokens)
    {
      fullpath = Path.Combine(fullpath, token);
    }
    return fullpath;
  }

  [SetUp]
  public void Setup()
  {
    Directory.CreateDirectory(testFolderPath);

    string[] testDirs = {
      MakePath(testFolderPath, "Test"),
      MakePath(testFolderPath, "Test","dir1"),
      MakePath(testFolderPath, "Test","dir1",
       "dir2"),
      MakePath(testFolderPath, "Test","dir1",
       "dir2", "dir3") };

   foreach (string dir in testDirs)
    {
      expectedDirs.Add(dir);
      Directory.CreateDirectory(dir);
    }
    expectedDirs.Sort();

    string[] testFiles = {
      MakePath(testFolderPath, "Test", "dir1",
       "file1.txt"),
      MakePath(testFolderPath, "Test", "dir1",
       "file2.txt"),
      MakePath(testFolderPath, "Test", "dir1",
       "dir2", "file3.txt"),
      MakePath(testFolderPath, "Test", "dir1",
       "dir2", "file4.txt") };
```

Listing 7

```
    foreach (string file in testFiles)
    {
      expectedFiles.Add(file);
      FileStream str = File.Create(file);
      str.Close();
    }
    expectedFiles.Sort();
  }
  [Test]
  public void TraverseDirectory()
  {
    string testPath = Path.Combine(
     testFolderPath, "Test");

    DirRecorder dirRecorder = new DirRecorder();
    DirectoryTraverser trav =
     new DirectoryTraverser(dirRecorder);
    trav.Traverse(testPath);

    Assert.AreEqual(
     expectedDirs, dirRecorder.Dirs);
    Assert.AreEqual(
     expectedFiles, dirRecorder.Files);
  }
  [TearDown]
  public void TearDown()
  {
    Directory.Delete(
       testFolderPath + "Test",true);
  }
}
```

Listing 7 (cont'd)

References

[CI] Continuous Integration: http://en.wikipedia.org/wiki/Continuous_Integration

[EnumMethod] http://www.two-sdg.demon.co.uk/curbralan/papers/ATaleOfThreePatterns.pdf

[Fake] http://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs

[Mono] http://www.mono-project.com/

[NUnit] http://www.nunit.org/

[SuSE] http://www.novell.com/linux/

[Visitor] Design patterns: elements of reusable object-oriented software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. ISBN-10: 0201633612 ISBN-13: 978-0201633610

[VFDC#] Visiting Files and Directories in C#. http://www.marauder-consulting.co.uk/articles.php

```
NUnit version 2.4.3
Copyright (C) 2002-2007 Charlie Poole.
Copyright (C) 2002-2004 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov.
Copyright (C) 2000-2002 Philip Craig.
All Rights Reserved.

Runtime Environment -
   OS Version: Microsoft Windows NT 5.1.2600 Service Pack 2
  CLR Version: 2.0.50727.832 ( Net 2.0.50727.832 )

.
Tests run: 1, Failures: 0, Not run: 0, Time: 0.188 seconds
```

Figure 1

Generics Without Templates

Robert Jones presents an alternative implementation of C++'s std::vector that can be used the absence of templates and exceptions.

Introduction

Templates and the STL are aspects of modern C++ that I'd imagine few of us would want to be without. However that was exactly the environment I encountered in a recent role at an embedded software house.

The company I worked for produced a re-useable software platform, which was required to compile under numerous compilers (some known and some not yet known), and so the company had taken the route of writing their software in a subset of C++ that was known by experience to be supported reliably by all the compilers under consideration. This subset, broadly modeled on Embedded C++, did not include templates, and by implication the STL, nor did it include exceptions.

Lack of access to the STL was a source of frustration to some of the more progressive programmers in the company, most notably the lack of comprehensive and predictable generic containers.

This article looks at providing a substitute for the STL's vector container, which behaves as much like the STL version as possible. Some the techniques used could equally be applied to other containers and other template classes.

Syntax

The syntax of declaring and using the vector should be as close to the STL syntax as possible, and in particular not impose unexpected requirements and dependencies on the build system (e.g., not demanding that scripts generate code). Vectors should be declarable in any scope, for any type (including built-ins) without special syntax or conditions. This is rather harder than it first appears, since to mimic the STL containers the vector must exhibit copy-in/copy-out and copy-internally semantics, but the built-in types do not have copy constructors. Listing 1 shows examples of vector usage.

```
#include "vector.hpp"

vector(int) myIntVec;
typedef vector( int ) IntVec;
vector(IntVec) myIntVecVec;

namespace MySpace {
   typedef vector(char) myCharVec; }

class X { /* ... */ };
vector(X) myXVec;
class Y { vector(X) myNestedXVec; };
```

Listing 1

Design choices

Back in the old days (which I can't recall, of course), the standard way to handle anything generically was **`void*`** pointers, and this is the approach used here. A core implementation is written using **`void*`** pointers, which is then wrapped in a presentation layer to respect type. To faithfully implement STL vector semantics, the **`void*`** implementation must be provided with the means to create and destroy instances of the client type, and this is provided by the presentation layer through callback functions. As an implementation detail, the **`void*`** implementation is written to use the Pimpl idiom, both to more fully hide implementation details and to facilitate an efficient no-throw **`swap()`** method. The presentation layer is finally implemented by a large macro of single line inline methods which cast type and then call the corresponding method in the core implementation.

```
class VectorCore
{
public:
    typedef void * value_type;
    typedef const void * const_value_type;

    typedef void * iterator;
    typedef const void * const_iterator;
    // ...
    VectorCore( Traits const & );
    VectorCore( VectorCore const & );
    ~VectorCore( );
    VectorCore & operator = (
       VectorCore const & );

    unsigned size( ) const;    // ... and empty
                               // capacity, etc
    value_type operator[]( unsigned ) const;
    value_type front( ) const; // ... and back.
    iterator begin( );      // ... and end (const
                            // and non-const)
    iterator insert( iterator, const_value_type );
    // ...various other function signatures.
    void swap( VectorCore & );

private:
    struct Impl;
    Impl * pimpl;
};
```

Listing 2

Robert Jones has been programming in C++ for many years, since the early days when C++ was only available as a cross-compiler. His experience has been primarily in embedded environments, especially telecoms. Last year he attended his first ACCU conference and found the experience utterly engaging. He can be contacted at robertgbjones@gmail.com

the only information needed by the generic code is the copy constructor, destructor and client type size

Embedded development and compiler features

C++ compilers for embedded environments often lack support for some of the relatively recent additions to the C++ language, most notably templates and exceptions. There can be a number of reasons for this.

Some embedded environments are now very mature, and may now have only a small active tool development community. As a result it may not be commercially viable to update their compilers to support these features. Supporting templates and exceptions adds significantly to the complexity of the compiler, so the commercial case to make the investment may not be attractive.

However, probably the most common reason for not supporting these features is the obscurity they introduce to the runtime performance. Even with a basic C++ implementation, the apparently simple act of exiting a scope can cause a great deal of activity as a result of stack unwinding and all the destructor calls that may be made. By including exceptions and templates this unseen activity penalty is exacerbated. In the kind of environments where the system must respond within a handful of microseconds this kind of 'under-the-hood' activity can be problematic.

Implementation

The core implementation is mainly predicable and straight-forward (Listing 2: VectorCore interface).

All the methods and method signatures in this implementation are easily deducible from an examination of any STL reference, such as Josuttis, with the exception of the constructor signature:

```
VectorCore::VectorCore( Traits const & )
```

This is where the characteristics of the client type are made available to the generic implementation, and it turns out that the only information needed by the generic code is the copy constructor, destructor and client type size. To implement the STL's **resize(unsigned)** method signature would also require that a default constructor be provided, which seems too onerous a requirement to demand of all types when only some types will require the **resize()**. Using templates, the default constructor and resize signature can be provided or not on a per type basis, but using this implementation technique it is for all types or none. It is of course only that specific signature which is affected, the full **resize(unsigned, const_value_type)** signature is supported.

```
class VectorCore
{
public:
  // ...
  struct Traits
  {
    typedef void ( * Ctor   )
        ( value_type, const_value_type );
    typedef void ( * Dtor   )
        ( value_type );

    Traits( Ctor ctor, Dtor dtor, unsigned size );
    Ctor ctor;
    Dtor dtor;
    unsigned size;
  };
  // ...
};
```

Listing 3

```
bool VectorCore :: Impl :: reserve(
 unsigned capacity )
{
  if ( capacity > m_capacity )
  {
    void * data = :: operator new (
     capacity * m_traits.size );
    for ( unsigned i = 0; i < m_size; ++ i )
    {
      value_type to   = addressOf( data, i );
      value_type from = addressOf( m_data, i );
      m_traits.ctor( to, from );
      m_traits.dtor( from );
    }
    delete reinterpret_cast<char *>( m_data );
    m_data     = data;
    m_capacity = capacity;
  }
  return true;
}
```

Listing 4

The **Traits** type packages the necessary functions and presents them with **void*** interfaces for the implementation (Listing 3: Declaration of type traits).

The method signatures of the **Impl** pimpl implementation class closely mirror signatures of the **VectorCore** class, including respecting constness down to the lowest possible level. The final implementation is done with no knowledge of the ultimate client type beyond that passed in generically through the **Traits** structure. Any copying or deletion of vector elements must manually invoke the appropriate functions to maintain correct copy semantics. In accordance with the requirements of STL vector implementations, this implementation ultimately uses a C-style array, although since the implementation is unaware of type it is an array of **char** appropriately sized using the **size** parameter passed in the type **traits**. It is unnecessary to present the full implementation here, but most of the interesting characteristics of the implementation are illustrated by the **reserve()** method (Listing 4: Example of a VectorCore method implementation).

taking in to account human factors, such as willingness of uptake, this seemed the better alternative

```
#define vector( TYPE ) \
class VectorOf##TYPE                  \
{                                     \
    .                                 \
    .                                 \
    .                                 \
}
```

Listing 5

Wrapping the implementation

With the implementation in place, attention now turns to how to present the functionality to the user in a useful and useable way. In the absence of templates, it is the old and much derided mechanism of the trusty macro to which we now turn (Listing 5: Presenting the container using a macro).

The core implementation is the only data member of the macro generated class, and made private. An obvious alternative wrapping is to inherit privately from the implementation class, and this was in fact the first approach considered. However, the number of identifiers that it was convenient to reuse from the base class produced lots of name clashes, which could easily be resolved, but which made the macro textually longer and rather more intimidating. On balance, and taking in to account human factors, such as willingness of uptake, this seemed the better alternative. (Listing 6: Introducing the embedded **`VectorCore`**.)

This macro generated class must also capture the type's traits and pass them to the implementation. There is a subtlety here in that the built in types do not have a named constructor. To overcome this difficulty, a nested type is declared which has only one data member, which is of the client type. The underlying vector thus becomes a vector of this nested **`Element`** type, which always has a named constructor and destructor. (Listing 7: Fulfilling the traits requirements.)

The interesting parts of this structure are the parts we can't see! The default generated **`copy`** constructor and destructor provide access to the corresponding methods of the client type, but allow the compiler to handle the special cases of the built-in types. Using this wrapping technique does not impose any penalty of additional copy operations on the client type.

```
#define vector ( TYPE )                                   \
class VectorOf##TYPE                                      \
{                                                         \
    // ...                                                \
private:                                                  \
  struct Element                                          \
  {                                                       \
    static void ctor(                                     \
       core_type :: value_type to,                        \
       core_type :: const_value_type from                 \
       )                                                  \
      { new ( to ) Element(                               \
         * static_cast<Element const *>( from )           \
         );                                               \
      }                                                   \
    static void dtor( core_type :: value_type p )\
      { static_cast<Element *>( p ) ->                    \
          Element :: ~Element( ); }                       \
                                                          \
    value_type client;                                    \
  };                                                      \
  // ...                                                  \
}
```

Listing 7

The named methods provided by this wrapping can then be packaged into static methods, with generic (**`void*`**) signatures. This repackaging makes implicit assumptions, which will also be made by the iterator type, that a pointer to **`void`**, client type or **`Element`** type can be freely cast between the three types with no corruption introduced. This is broadly equivalent to assuming the wrapping and typing do not impose additional padding or alignment restrictions.

```
#include "VectorCore.hpp"

#define vector ( TYPE )           \
Class VectorOf##TYPE              \
{                                 \
Public:                           \
  typedef VectorCore core_type;   \
      .                           \
      .                           \
      .                           \
Private:                          \
  core_type core;                 \
}
```

Listing 6

```
#define vector ( TYPE )                                    \
class VectorOf##TYPE                                       \
{                                                          \
public:                                                    \
  // ...                                                   \
  VectorOf##TYPE ( ) : core( core_type :: Traits(          \
     Element :: ctor,                                      \
     Element :: dtor,                                      \
     Element :: assign,                                    \
     sizeof( Element )                                     \
     ) ) { }                                               \
  // ...                                                   \
}
```

Listing 8

we should not be misled into believing it is more than it is

We now have all the basic mechanisms necessary to complete the **vector** class. To construct a default vector, the constructor must package the necessary traits and pass them down to the core implementation. (Listing 8: Passing traits to the implementation.)

And provide a full complement of method signatures consistent with the STL vector implementation (Listing 9: Populating the macro class).

Limitations

So what have we achieved, and is it fit for purpose? Well, we have a 'template' for a generic **vector**, that looks and feels much like the STL's **vector**, however we should not be misled into believing it is more than it is. This is only a generic container, not a poor man's version of templates. There are many specific limitations.

- The vector may only be instantiated with a type represented by a single identifier. This is only true because the type is used to form the name of the vector type, and if an alternative naming strategy were used, it would possible to instantiate the vector with derived types.
- The vector may only be instantiated once for a given type in a given scope. Again this is because of the vector naming strategy. If multiple instances of an instantiated vector type are required, a typedef may be employed.
- Each instantiation of the vector is a distinct type, unrelated to any other instantiation.

Conclusion

In this article a techniques has been presented to fabricate a template-like generic container in circumstances where templates are not available. Although being far from a perfect solution, the resulting container permits code to be written in a more familiar and contemporary style, and is an advance on the previous home-grown container that it superseded. ■

```
#define vector ( TYPE )                                          \
class VectorOf##TYPE                                             \
{                                                                \
public:                                                          \
  // ...                                                         \
  typedef value_type * iterator;                                 \
  typedef value_type const * const_iterator;                     \
  // ...                                                         \
  size_type size( ) const { return core.size( );} // etc,...     \
                                                                 \
  value_type & operator[]( unsigned idx )                        \
       { return * static_cast<value_type *>(                     \
             core[ idx ] ); }                                    \
  value_type const & operator[]( unsigned idx ) const            \
    { return * static_cast<value_type const *>(                  \
       core[ idx ] ); }                                          \
  value_type front( ) const                                      \
    { return * static_cast<value_type *>( core.front( ) ); }     \
  value_type back( ) const                                       \
    { return * static_cast<value_type *>( core.back( ) ); }      \
                                                                 \
  iterator begin( ) // ... + end, const & non-const              \
        { return static_cast<iterator>( core.begin( ) ); }       \
  void insert( iterator pos,                                     \
          const_iterator begin, const_iterator end )             \
        { core.insert( pos, begin, end ); }                      \
    // ...various other function signatures.                     \
    void swap(VectorOf##TYPE & other )                           \
        { core.swap( other.core ); }                             \
    // ...                                                       \
}
```

Listing 9

Future Workers (Prototype)

What does it mean for IT workers to be prototype knowledge workers?

> *Knowledge workers have high degrees of expertise, education, or experience, and the primary purpose of their jobs involves the creation, distribution, or application of knowledge.*
>
> Davenport, *Thinking for a Living*, 1995

Writers and experts on the knowledge economy and knowledge workers frequently cite software developers as examples of knowledge workers. Yet it is rare for those in IT, or writers about IT, to discuss the software developers as knowledge workers. But then: why would they? What difference does it make?

When we view software developers as knowledge workers and consider development activities as knowledge creation, we gain many useful insights into the process by which software is developed and deployed. By recognizing IT staff as knowledge workers a rich field of literature and experience opens up that we may learn from to help improve our own practice.

From the same book quoted above we can distil a list of knowledge work characteristics:

- Knowledge workers like autonomy; they don't like being told what to do.
- Specifying detailed steps to follow is less valuable than in other types of work.
- Knowledge workers find it difficult to describe what they do in detail; if you want to know you are better off watching.
- Not only do knowledge workers find it difficult to describe what they do but they are aware of the value of knowledge and don't share it without a motivation.
- Even though they may not be able to describe what they do these workers often have good reason for doing what they do and have often thought about the way they work in advance.
- Commitment matters and makes a huge difference in productivity.

Two things stand out from this list: it is a list of developer characteristics; any doubt that developers are knowledge workers should be dispelled. Secondly, an individual with these characteristics is unlikely to relish routine, factory-like, work. The traditional view of management is not applicable to these workers.

Recognizing IT workers are knowledge workers also recognizes that they are not unique. They share the same characteristics as other knowledge workers. Neither are the problems they encounter unique. The opportunities and problems faced by IT staff and their managers are quite legitimate and are shared by other modern knowledge workers. Consequently it is wrong to think of the 'IT-geek' as a class apart.

Development activities – specifying, designing, coding and testing new software – are knowledge activities. Such activities are completely different from traditional factory lines processes where a worker's individual knowledge makes little immediate difference to the end product. Having recognized this critical difference it becomes meaningless to characterize software development as a factory process.

Many previous attempts to change the way IT staff worked were misplaced because they failed to recognize the roles of knowledge and the characteristics of knowledge workers.

Recognising IT workers as knowledge workers allows us to learn from the existing body of knowledge on the subject. IT workers are not alone, they are knowledge workers and there is much to learn from other knowledge workers and from research and literature about knowledge work in general. There is no need for IT managers (and writers) to reinvent the wheel.

Yet, in another way the existing literature, research and experience cannot help IT workers and their managers. This is because software developers, in particular, are at the cutting edge of knowledge work. They are in many ways the prototype of the future knowledge worker; they are pushing the boundaries of twenty first century knowledge work.

This occurs because, to paraphrase Karl Marx, software developers control the means of production. Modern knowledge work is enabled by and dependent on information technology: e-mail for communication, websites for distribution, databases for storage, word processors for writing reports, spreadsheets for analysis – the list is endless! These technologies are created by software developers and used by legions of knowledge workers worldwide. The key difference between software knowledge workers and the others is that other knowledge workers can only use the tools that exist. If a tool does not exist they cannot use it. Conversely, software developers have the means to create any tool they can imagine.

Consequently it was a programmer, Ward Cunningham who invented the Wiki. Programmers Dan Bricklin and Bob Frankston invented the electronic spreadsheet. Even earlier another programmer, Ray Tomlinson, invented inter-machine e-mail. This does not mean that non-programmers cannot invent electronic tools. Others can invent tools but for programmers the barriers between imagining a tool and creating the tool are far lower.

Consequently programmers create many more tools than other types of worker. Some tools fail, others are very specific to a specific problem, organisation or task in hand but when tools do work it is programmers who get to use them first. In addition because programmers have had internet access for far longer than any other group the propensity to use it to find tools and share new tools is far greater. Tools like Cunningham's Wiki were in common use by software developers years before they were used by other knowledge workers.

Early internet access has had other effects too: IT workers were early adopters of remote working, either as individual home workers or as remote development teams; IT people are far more likely to turn to the web for assistance with problems and more likely to find it because IT information has been stored on the web since the very beginning.

The net effect of these factors and others means that software developers are often the first to adopt new tools and techniques in their knowledge work. They are also the first to find problems with such tools and techniques. Consequently, these workers are at the cutting edge of twenty-first century knowledge work; they are the prototype for other knowledge workers. Other knowledge workers, and their managers, can learn from the way IT people work today provided we recognize these workers as knowledge workers.

Allan Kelly served his apprenticeship developing software for financial, communication and utility systems. He is now a consultant and interim manager who specialises in advising and helping the most challenged development teams deliver and improve. Allan can be reached at allan@allankelly.net.

This piece is an excerpt from Allan's new book *Changing Software Development* (John Wiley and Sons, 2008) available from all good bookshops – and a few less good ones too.