overload 93

OCTOBER 2009 £3

Quality Matters
We consider the correctness, robustness, and reliability of our code

The Model Student
We take another look at the primal skyline

Generation, Handling and Management of Errors
A pattern language for error management, to ensure that your code is robust

Multi-threading in C++0x
An expert introduction to the facilities and capabilities added to the forthcoming C++ standard enabling your code to take advantage of multiple threads

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 93

October 2009

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Simon Farnsworth
simon@farnz.co.uk

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 94 should be submitted by 1st November 2009 and for Overload 95 by 1st January 2010.

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

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.

All together now.

Can you do several things at once? Ric Parkin tries multi-tasking.

Whats going on here?

Time is time, like what prevents an ever-rolling everything stream happening bears at all its once sons away.

Clear? Perhaps we should try a different example.

He draweth words form out the thread, the thread of his on which verbosity we string finer than the stable experiences of his argument.

Hmm, not any better. Perhaps if we try enough times, it'll eventually make some sense. Prepare to wait a long time though – there are so many possible sentences that will turn up, and the above examples are ones where I tried to make them still make some vague semblance of sense.

So where did they come from? Perhaps you have guessed: both are made from two quotations interleaved with each other:

Time, like an ever-rolling stream, bears all its sons away
– Issac Watts

Time is what prevents everything happening at once
– John Wheeler

and

Words form the thread on which we string our experiences
– Aldous Huxley

He draweth out the thread of his verbosity finer than the staple of his argument– Shakespeare (Loves Labours Lost)

Those of a mathematical bent could calculate how many possible combinations of the sentences there are. Extrapolating further, how many possible ways can an arbitrary number of extremely long strings be interleaved? It gets to be a remarkably large number very quickly.

Now check them all.

Sounds ridiculous to even think of doing so, doesn't it?

If you haven't guessed yet from my choice of quotes, I've recent been thinking about concurrency and multithreading. Some of the reasons should be obvious – despite C++0x coming a little later than originally hoped, one of the areas which is comparatively mature is the memory model, the threading primitives and the higher level library built upon it. We're quite lucky in the ACCU as many members have contributed directly and indirectly to the design of this area, and in this issue Anthony Williams introduces us to some of the new facilities.

But mainly because I've realised that I've had to deal with concurrency quite a lot in the last few years, and this is a change. In much of my career, I've not actually had to deal with multithreading at all – for most of that time, the computers were single-cored, and the operating systems would often use a cooperative form of multitasking, so your application itself would just carry on as if it was the only thing around, occasionally letting other programs have a chance to run. Even when preemptive tasking appeared, most programs would be written as if they were single threaded, and let the operating system sort out which process would be running.

But now true mutithreaded machines are common, and the only way of speeding up our programs now that processor speed has plateaued is to find suitable ways of processing in parallel. Unfortunately this can be really difficult to do in practice, for several reasons.

Sharing mutable data is difficult to get right, and can be expensive in many ways – the locking adds expensive calls as an overhead; over-zealous locking can stall other threads, reducing any benefits and potentially re-serialising execution so that one one thread can run at a time! Also if the data is shared across processors, the versions in their caches have to be synchronised which can even lead to the odd situation where adding processors slows things down!

This last is an example of the problems of predicting which areas can be split into separate threads. It is difficult, and prone to counter-intuitive results. Part of it comes from the overheads associated with threads and locking tend to be quite expensive, and so too-fine-grained threading, locks and cross-thread communication can slow things down.

Herb Sutter has pointed out that locks and other synchronisation techniques are not composeable [DDJ]. That means that combining two parts that work on their own may no longer work, and you have to consider their interactions, and often the interactions between their internals. This means that you potentially get a combinatorial explosion, and building a large system becomes very difficult.

Testing is also much harder. I have already pointed out the huge number of possible execution paths, depending on the whims of the scheduler. This means that software is no longer easily deterministic – the same program on the same input can have a slightly different interleaving, and obtain locks in a different order. One will appear to work the other can deadlock or, more subtly change some timings and things start to time out. Even worse, future hardware can completely change the execution environment. Consider the difference between a single core machine interleaving threads verses a true multi-core machine where the threads are truly running at the same time. In the former the threads sees a single version of memory; in the latter they see two slightly different versions depending on whether a write by one thread has propagated to the other processor's memory cache. The can be very hard to reason about – we're used to thinking though code as if the changes happen instantaneously.

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

And the number of possible execution orders is extremely high, which was the point of my interleaved sentences example. So checking a threaded program by running it empirically is a pretty poor way of testing as you'll only cover a vanishingly small set of the possible combinations, and yet sometimes were are reduced to doing so. It can be useful so long as you understand the limitations: if you find a problem, great you know there's a bug, and yet it might be hard to reproduce, and even if you have logs it can be extremely hard to analyse what exactly happened. And if you don't find a problem, you haven't proved one doesn't exist as it might be that you haven't found it yet.

Unit testing can become much harder too. Trying to coordinate threads to get to known places so you can query their state is non trivial. I've seen naive programmers start a thread and sleep for a few milliseconds before checking some variable has been set, which worked fine on their machine, but on the heavily loaded build machine the thread would run too slowly and the test would fail. A solution was to add synchronising locks and semaphores on either side, and let the two sides let each only other run once they had done their work, but this was an awful lot of infrastructure to add that confused the code considerably (I've a gut feeling that there are ways of encapsulating such a technique – I'll leave it as an exercise for the reader, but would be interested in seeing any ideas).

There are ways to deal with these problems though. Making a lock to only allow one operation at a time works trivially, but would kill performance, so the real challenge is to loosen such constraints such that independent actions can occur simultaneously, but not to loosen any further. A good approach is to separate the threads of execution as much as possible, perhaps even into separate processes, and only communicate via asynchronous message passing. This helps avoid the temptation to share too much data, and gets the programmer into the habit of not expecting the results of a cross-thread request to be ready immediately, and instead go off and do something more useful in the meantime, waiting for notification that it has completed. This also reduces the number of locks that may be being used, which will reduce the locking overheads, and also help to avoid deadlocks.

Having shared state be immutable also avoids the need for many locks (which is one reason why Java and .Net have immutable string types), and which has led to an upsurge of interest in functional programming techniques. A good tip is to try and write code that has no side effects – in other words functions that rely only on their input values, and avoid aliases to mutable data. This means that there will be no possible interactions with other threads changing things under your feet. This means that you don't need any locks, and that function is simpler to reason about. It also has an effect that it's much easier to test.

Other interesting ideas include things like the MapReduce technique, used by many people such as Google to implement distributed processing [MapReduce].

But concurrency is not just about multithreading – in a more general sense it is about multiple 'actors' running independently yet interacting with each other and shared resources. A simple example: two processes using the file system will be having their file system calls being serialised via some sort of locking mechanism. If they are reading and writing to the same files you also now have a consistency problem, unless the processes have some way of making their changes atomically with respect to the other process. Also there can be deadlocks if multiple files with read/write locking is involved, although a well designed API should time out for you and return a failure.

Databases also provide a rich source of shared resources that can be accessed by multiple machines, and so they provide lots of locking mechanisms, such as the whole DB, individual tables or down to single rows. Designing a database that provides proper consistency and yet scales to a high number of users is non-trivial.

And the most obviously distributed system, is a distributed system built out of separate machines passing messages to each other to trigger manipulation of some resources, which need careful coordination to keep consistency without sacrificing performance. What we need to come to terms with are that these sorts of systems are already to be found in a single box, and even within a single chip. We're moving from a simplistic single-threaded world, and have to deal with the complexities of things happening all at once.

References

[DDJ] http://www.ddj.com/architect/202802983

[MapReduce] http://en.wikipedia.org/wiki/MapReduce

The Model Student: A Primal Skyline (Part 2)

How do you measure the length of a curve? Richard Harris gets his ruler out.

In the previous article I described some of the important questions regarding the lone wolves of the number line, the prime numbers; those integers greater than 1 not wholly divisible by any positive integer other than themselves and 1.

We discussed Euclid's elegant proof that there are infinitely many of them and described the prime number theorem which gives an approximate estimate of π; the function that counts the number of primes less than or equal to any given integer.

$$\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1$$

Recall that the lim term denotes the limit that the following term converges to 1 as *n* grows ever larger.

Given that greater minds than mine have tried and failed to find the precise pattern by which the prime numbers are found within the integers, I suggested that we look at the prime factorisations of the integers instead. Every positive integer can be uniquely represented by a prime factorisation, the collection of prime numbers that when multiplied together yield the integer in question.

The number 42, for example, has the prime factors 2, 3, and 7 since $42 = 2 \times 3 \times 7$.

The number 1 is, as you will no doubt recall, the special case of multiplying no primes together.

Rather than completely analyse the prime factorisations of every integer, we instead took a look at one of π's relatives, the function that counts the number of prime factors (including repeated factors) of a given integer, Ω, shown in Figure 1.

n	Ω(n)	n	Ω(n)	n	Ω(n)
0	-∞	7	1	14	2
1	0	8	3	15	2
2	1	9	2	16	4
3	1	10	2	17	1
4	2	11	1	18	3
5	1	12	3	19	1
6	2	13	1	20	3

Figure 1

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.

Noting that 0 can be considered the result of dividing 1 by an infinite number of prime factors, figure 1 gives the values of Ω for the integers between 0 and 20.

Figure 2 reproduces the graphs of Ω for the integers from 1 to 20 (top) and 1 to 100 (bottom).

In pursuit of a pattern in these graphs, we introduced our own function $\daleth_n$, defined by

$$\daleth_n(x) = \frac{2^{\Omega(\lfloor 2^n x \rfloor)}}{2^n} \qquad x \in [0, 1]$$

Recall that the odd brackets surrounding the $2^n x$ term mean the largest integer less than or equal to the value between them and that the expression on the right with the rounded E mean that *x* lies between 0 and 1.

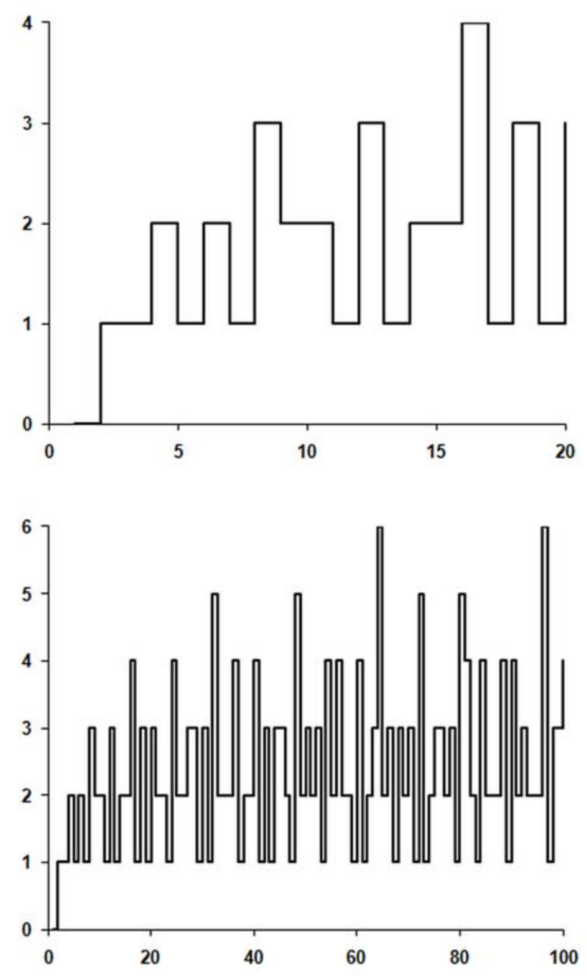

Figure 2

a fractal is a curve, area, volume or higher dimensional analogue that has a non-integer, or in other words fractional, dimension

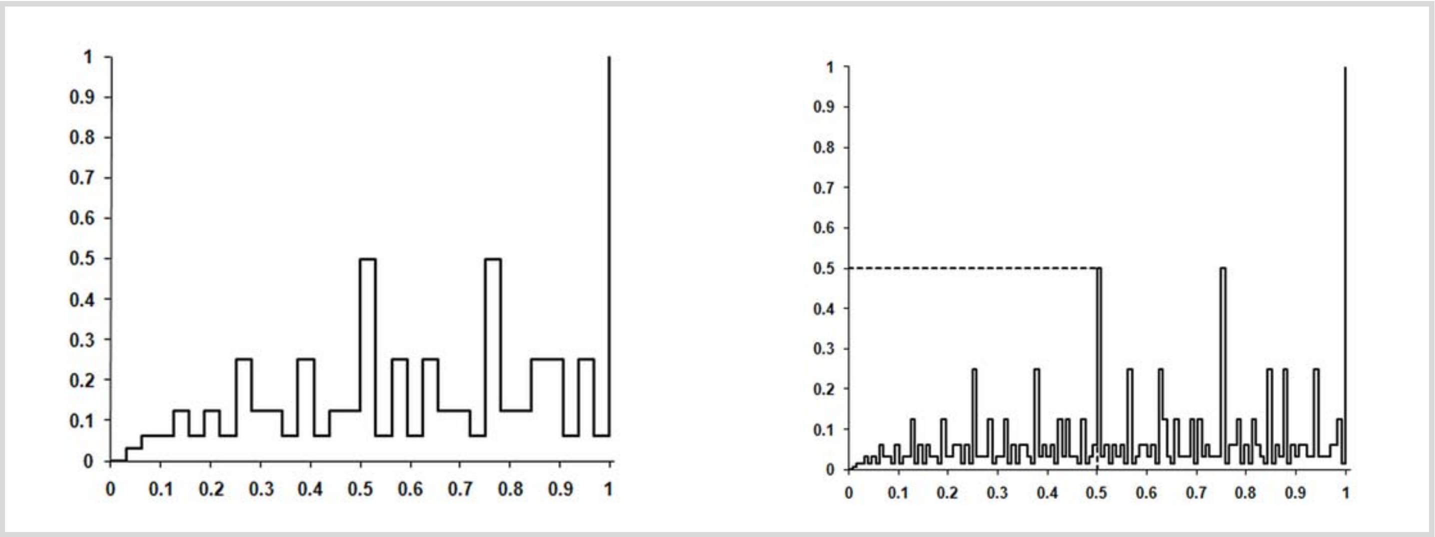
Figure 3

Two example graphs of $\daleth_n$ ($\daleth_5$ on the left and $\daleth_7$ on the right) are given in figure 3.

Finally, we demonstrated that successive versions of $\daleth_n$ are coincident for half the points in the range 0 to 1 and that the infinite limit, $\daleth_\infty$, sort of exhibits the property of self similarity since it can be entirely recovered from an arbitrarily small range of arguments.

Both of these properties bear a striking resemblance to those of fractals and we closed on the question of whether $\daleth_\infty$ is itself a fractal. Before attempting to answer that question, I think it would be prudent to define precisely what a fractal is.

OK, so what precisely is a fractal?

Beloved of undergraduate mathematicians the world over, fractals hit mainstream popular culture in the late 1980s. In fact, I still have a T-shirt with a black and white print of a fractal. Made from hemp. As is tragically illustrated in figure 4: Dear God, what was I thinking?[Asti].

Whilst these beautiful images may have adorned the torsos of many of my fellow unwashed hippies, I rather suspect that their exact definitions were not so widely disseminated.

Strictly speaking, a fractal is a curve, area, volume or higher dimensional analogue that has a non-integer, or in other words fractional, dimension. A somewhat counter-intuitive description, it relies upon a very particular definition of dimension; that of fractal dimension. There are many different types of fractal dimension and we shall focus on one of the simplest to calculate, the Minkowski, or box-counting dimension.

For a curve, this is a measure of how quickly its length grows as the ruler you use to measure it shrinks.

Specifically, if a ruler of length ε can be lain end to end $N(\varepsilon)$ times to cover a curve, then its box-counting dimension is defined as

$$d = \lim_{\varepsilon \to 0} \frac{\ln N(\varepsilon)}{\ln \frac{1}{\varepsilon}}$$

Note that the term to the left of the fraction means that we should consider the limit of the term to its right as ε tends to 0.

The Koch curve

To demonstrate the idea, we introduce the Koch curve. This curve is iteratively constructed from the element illustrated in figure 5.

The first iteration in the construction of the Koch curve is to replace each of the 4 straight line segments with copies of the basic element shrunk in length by 1/3, as illustrated in figure 6.

Figure 4

as we increase the number of sides of the inscribed polygon we both move closer and closer to the circle and make smaller and smaller corrections

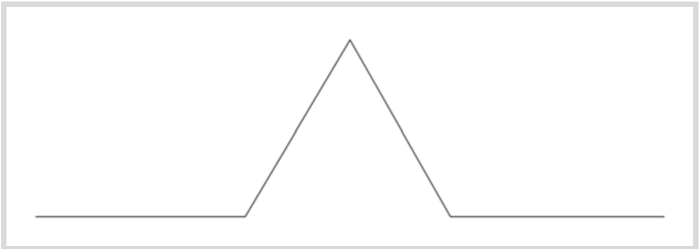
Figure 5

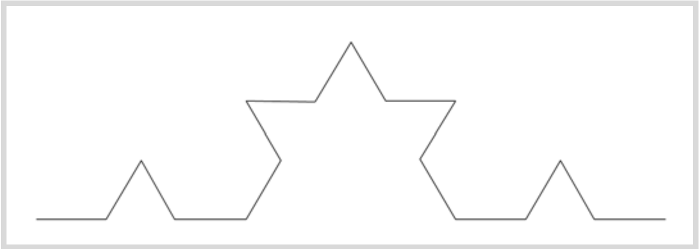
Figure 6

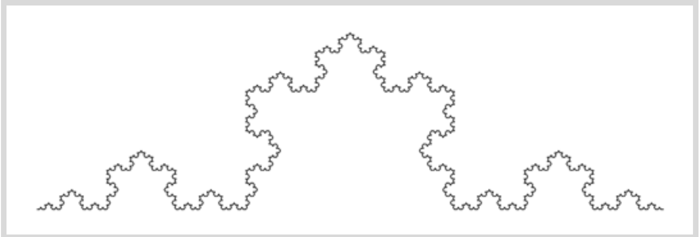
Figure 7

We continue in the same vein for further iterations, ever replacing the straight line segments with further and further scaled down copies of the basic element.

Figure 7 illustrates the result of a few more iterations in the construction of the curve.

Assuming that the curve ranges from 0 to 1 on the *x* axis, a ruler of length 1/3 can be laid end to end on the curve 4 times; at this scale it appears identical to the basic element. Similarly, to a ruler of length 1/9, the curve appears identical to the first iteration and can be covered with 16 of them.

Generalising this, to a ruler of length $1/3^n$ the curve appears identical to the $n-1$th iteration of its construction and, since at each iteration we replace each line segment with 4 lines each 1/3 its length, will therefore be covered by 4^n of them.

The dimension of the Koch curve under this definition is therefore approximately 1.2619 as shown in derivation 1.

$$\begin{aligned}\lim_{n\to\infty}\frac{\ln 4^n}{\ln\frac{1}{1/3^n}} &= \lim_{n\to\infty}\frac{\ln 4^n}{\ln 3^n}\\ &= \lim_{n\to\infty}\frac{n\ln 4}{n\ln 3}\\ &= \frac{\ln 4}{\ln 3}\approx 1.2619\end{aligned}$$

Note that we take the limit as *n* tends to infinity, which since our ruler is of length $1/3^n$ is equivalent to the ruler length tending to 0 and the length of the curve tending to infinity.

Derivation 1

The fractal dimension of a straight line

One thing we should check is that this measure of dimension yields the correct value of 1 for simple curves such as, for example, the straight line from (0,0) to (1,1). The length of this line is $\sqrt{2}$ and so, for a ruler of length ε, we will need $\sqrt{2}/\varepsilon$ of them to cover the line.

The dimension of this line using this measure is indeed 1, as demonstrated in derivation 2.

The fractal dimension of the circumference of a circle

For a more complex example, consider the circumference of a circle with radius 1 centred at the origin. If we choose our ruler lengths carefully, we can join points on the circle to construct inscribed regular polygons of increasing numbers of sides, as illustrated in figure 8.

We can see from this diagram that as we increase the number of sides of the inscribed polygon we both move closer and closer to the circle and make smaller and smaller corrections to the previous polygonal approximation. This is highly suggestive that the fractal dimension is 1, although proving it is a little more tricky, as illustrated in derivation 3.

Note that for areas, volumes and higher dimensional objects a similar approach is used in which we cover the object of interest in 2 dimensional

$$\begin{aligned}\lim_{\varepsilon\to 0}\frac{\ln\frac{\sqrt{2}}{\varepsilon}}{\ln\frac{1}{\varepsilon}} &= \lim_{\varepsilon\to 0}\frac{\ln\sqrt{2}-\ln\varepsilon}{-\ln\varepsilon}\\ &= \lim_{x\to\infty}\frac{\ln\sqrt{2}+x}{x}\\ &= 1+\lim_{x\to\infty}\frac{\ln\sqrt{2}}{x}\\ &= 1\end{aligned}$$

Derivation 2

the **box-counting measure** of fractal dimension is **supremely simple** to implement as a **numerical algorithm**

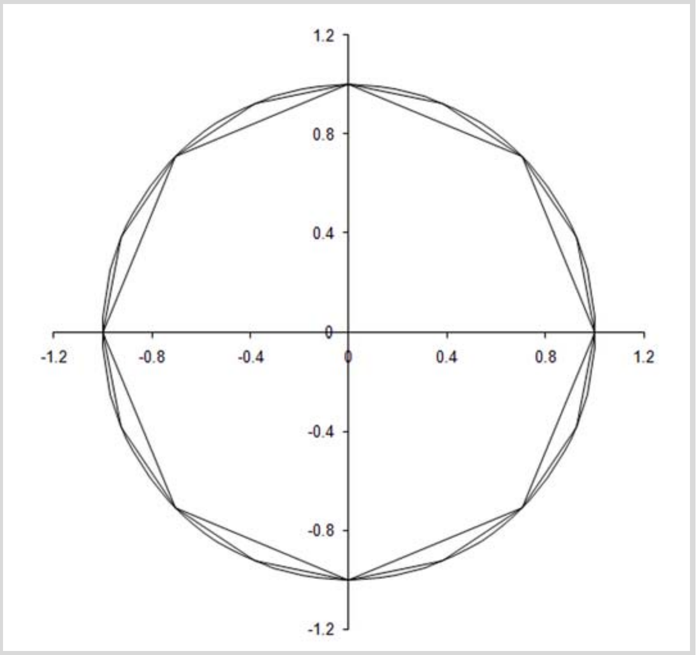

Figure 8

patches of ever decreasing area, 3 dimensional blocks of ever decreasing volume and so on.

The fractal dimension of $\daleth_\infty$

One of the principal advantages of the box-counting measure of fractal dimension is that it is supremely simple to implement as a numerical algorithm. And since I wouldn't even know how to start calculating the fractal dimension of $\daleth_\infty$ mathematically, this is going to prove rather handy.

We have a natural choice for the ruler length for $\daleth_n$, a new definition of ε_n given by

$$\varepsilon_n = \frac{1}{2^n}$$

since each horizontal line in the graph is exactly this long, and each vertical line has a length equal to an integer multiple of this, namely

$$2^{\Omega(\lfloor 2^n x \rfloor)} - 2^{\Omega(\lfloor 2^n x \rfloor - 1)} \times \varepsilon_n$$

The straight line braces in this formula denote the absolute value of the expression between them; the positive number with the same magnitude, or size, as the value of the expression, or equivalently, the strictly positive distance between it and 0.

Note that the admissible values of x will be precisely those for which the graph has a vertical line; those that are equal to an integer multiplied by ε_n.

The circumference of a polygon with n sides is trivially n times the length of each side, which for our measure of the circumference is equal to the length of the ruler, ε_n, and which we can recover with a little trigonometry.

$$\varepsilon_n = 2 \times \sin\left(\frac{1}{2}\frac{2\pi}{n}\right) = 2 \times \sin\left(\frac{\pi}{n}\right)$$

As n tends to infinity, so the length of our ruler, ε_n, tends to 0, enabling us to express the box-counting dimension as

$$\lim_{n\to\infty} \frac{\ln n}{\ln \frac{1}{\varepsilon_n}} = \lim_{n\to\infty} -\frac{\ln n}{\ln \varepsilon_n} = \lim_{n\to\infty} -\frac{\ln n}{\ln 2 + \ln \sin\left(\frac{\pi}{n}\right)}$$

As n grows ever larger so π/n grows ever smaller and we can approximate $\sin(\pi/n)$ with ever increasing accuracy using a few terms of a Taylor's series expansion in which a function is represented by a polynomial with coefficients calculated from its derivatives

$$f(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \ldots$$
$$+ \frac{1}{n!}f^{(n)}(a)(x-a)^n + \ldots$$

Here we use 1 dash to mean the first derivative, 2 dashes to mean the second and (n) to mean the n'th, and the exclamation mark stands for the factorial of the number preceding it, which is equal to the product of every number from 1 up to and including that number.

Using just the first 2 terms with a equal to 0 to approximate the sine function we have

$$\sin\left(\frac{\pi}{n}\right) \approx \sin(0) + \frac{\pi}{n}\sin'(0) = \sin(0) + \frac{\pi}{n}\cos(0) = \frac{\pi}{n}$$

The limit of our box-counting measure of the dimension of the circumference of the unit circle now becomes

$$\lim_{n\to\infty} -\frac{\ln n}{\ln 2 + \ln\frac{\pi}{n}} = \lim_{n\to\infty} -\frac{\ln n}{\ln 2 + \ln \pi - \ln n} = \lim_{x\to\infty} -\frac{x}{c-x} = 1$$

Derivation 3

An upper bound for $N(\varepsilon_n)$

Before we start, however, we shall need an upper bound on the number of rulers of length ε_n required to cover the graph of $\daleth_n$, since we need to ensure that the type we use to count them is large enough.

A simple curve that is guaranteed to be no shorter than $\daleth_n$ is one in which for every number greater than or equal to $2^i\,\varepsilon_n$ and less than $2^{i+1}\,\varepsilon_n$ is associated with a vertical line of length $2^i\,\varepsilon_n$. The reason that this is an upper bound is that above 0, there is no number that has negative infinity factors or as many factors as the smallest power of 2 greater than or equal to it. Hence at every point at which both curves have a vertical line that of the upper bound curve must be the longer.

a gross overestimate of the actual number of rulers we'll need to cover the curve

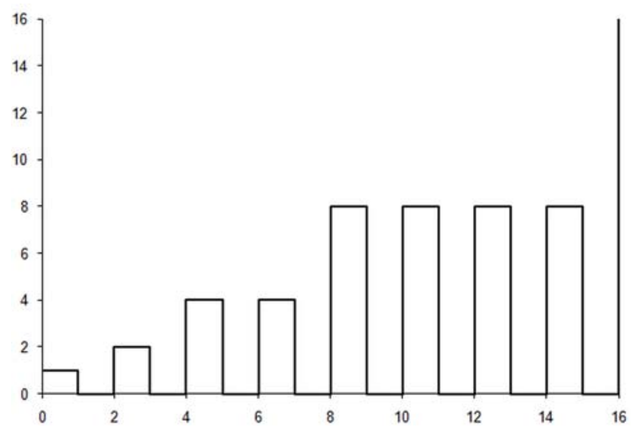

Figure 9

We construct this curve by taking a value of 1 at 0 and then dropping to 0 and rising to the relevant power of 2 of ε_n at alternate steps of ε_n.

Figure 9 illustrates this curve for $\daleth_4$ in units of ε_4.

The number of ε_n length rulers required to cover the horizontal lines in this graph is trivially 2_n. For the vertical lines, the calculation is a little more complicated and results in approximately 2^{2n-1} as shown in derivation 4.

Despite this being a gross overestimate of the actual number of rulers we'll need to cover the curve, we can be fairly sure that if we use inbuilt types we shall overflow them in short order.

It seems reasonable, therefore, to create an accumulator type of our own that won't suffer from this problem, one that I shall imaginatively call **`accumulator`** and whose definition is provided in listing 1.

The constructor is pretty trivial; we simply initialise the **`sum_`** with a single 0 as shown below:

```
  accumulator::accumulator() : sum_(1, 0) {}
```

```
  class accumulator
  {
  public:
    accumulator();
    accumulator & operator+=(unsigned long n);
    operator double() const;

  private:
    std::vector<unsigned long> sum_;
  };
```

Listing 1

Considering the $\daleth_4$ case will give us a hint as to how we might go about counting the number of rulers.

The length of the vertical lines in this curve in units of $\daleth_4$ is given by

$$\begin{aligned} & 1+2+2+4+4+4+4+8+8+8+8+8+8+8+8+16 \\ = & \ 1+2\times 2+4\times 4+8\times 8+16 \\ = & \ 16+\sum_{i=0}^{3} 2^i \times 2^i \\ = & \ 16+\sum_{i=0}^{3} 4^i \end{aligned}$$

In general, the length of the vertical lines in the upper bound curve for $\daleth_n$ is given by

$$2^n + \sum_{i=0}^{n-1} 4^i$$

and hence the total length of the upper bound curve is

$$2^n + 2^n + \sum_{i=0}^{n-1} 4^i = 2^{n+1} + \sum_{i=0}^{n-1} 4^i = 2^{n+1} + \frac{1}{3}(4^n - 1) \approx 2^{2n-1}$$

since the sum is another example of a geometric series.

Derivation 4

The **`operator+=`** member function requires a little more thought however.

Recall that the C++ standard declares that arithmetic with unsigned integer types cannot overflow and that arithmetic using n bit unsigned integer types is performed modulo 2^n [ANSI]. Any bits in the result of an arithmetic expression that don't fit into the unsigned integer type are simply discarded and the result wraps around into the type's valid range.

This is actually quite useful for us, since we'll be keeping track of the discarded bits ourselves and the correct value for our lowest **`unsigned long`** digit is exactly the wrapped around result of the addition.

Noting that when an addition wraps around the result must be less than the value that we are adding to our **`accumulator`**, listing 2 illustrates the implementation of the in place addition operator.

Since an addition can only ever add 1 to the next most significant digit, the loop we enter upon wrapping around simply checks whether incrementing subsequent digits by 1 also wrap around, pushing a 1 onto the end in the event that they all do.

The conversion to double is much simpler, as shown in listing 3. Here we simply add up the elements of our **`accumulator`** as doubles of increasing magnitude. It's not as efficient as it might be since later digits may cause the earlier ones to round off due to the limited precision of the **`double`** type, meaning that some of the earlier calculations may be wasted.

```
accumulator &
accumulator::operator+=(unsigned long n)
{
  assert(!sum_.empty());
  sum_.front() += n;
  if(sum_.front()<n)
  {
    std::vector<unsigned long>::iterator first =
       sum_.begin();
    std::vector<unsigned long>::iterator last  =
       sum_.end();

    while(++first!=last && ++*first==0);
    if(first==last)  sum_.push_back(1);
  }
  return *this;
}
```

Listing 2

We shall rarely use the conversion though, so it's really not worth making the code more complicated to improve its efficiency.

So now we have all the pieces in place to calculate the box-counting dimension of $\urcorner_n$. We will reuse the **count_factors** from the last article as illustrated again in listing 4.

```
accumulator::operator double() const
{
  static const int dig =
     std::numeric_limits<unsigned long>::digits;
  static const double base = pow(
     2.0, double(dig));
  std::vector<unsigned long>::
     const_iterator first = sum_.begin();
  std::vector<unsigned long>::
     const_iterator last  = sum_.end();
  double result = 0.0;
  double mult   = 1.0;
  while(first!=last)
  {
    result += mult * double(*first++);
    mult   *= base;
  }
  return result;
}
```

Listing 3

```
template<class FwdIt>
unsigned long
count_factors(unsigned long x, FwdIt first_prime,
   FwdIt last_prime)
{
  unsigned long count = 0;
  while(first_prime!=last_prime &&
      (*first_prime)*(*first_prime)<=x)
  {
    while(x%*first_prime==0)
    {
      ++count;
      x /= *first_prime;
    }
    ++first_prime;
  }
  if(x>1)  ++count;
  return count;
}
```

Listing 4

Recall that if a number has precisely 1 factor it is, by definition, prime and we shall use this fact to update the sequence of primes whose first and last iterators we'll pass to subsequent calls to the **count_factors** function.

This time, however, we shall wish to push on to the very limits of **unsigned long** and so shall need to take care that we do not fall foul of the wrap around problems we identified in the previous part of this article.

We shall do this by adding primes to our list when they are less than or equal to the square root of the upper bound, or as it turns out the square root of 1 less than it, rather than when their squares are less than or equal to the upper bound itself.

Newton's method for the integer square root

We shall calculate the integer square root using Newton's method for finding arguments at which functions return a value of 0, technically known as roots.

Newton's method is an iterative algorithm in which an initial estimate for a root is repeatedly updated using the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

where the dash after the *f* in the denominator indicates the derivative with respect to its argument.

At each step, Newton's method finds the root of a linear approximation of the function in question, as demonstrated in derivation 5.

Newton's method can fail to converge under certain conditions, but thankfully these only arise at 0 when using it to calculate square roots.

The equation whose roots are the square roots of a given number *a* is simply

$$f(x) = x^2 - a$$

The first derivative of this function is $2x$, meaning that Newton's method for finding the square root uses the iterative formula

$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} = x_i - \frac{x_i}{2} + \frac{a}{2x_i} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right)$$

I'm sure that some of you will have come across this algorithm for computing square roots before, although you may not have seen its derivation. I first learnt it from a book of mathematical tricks for the primitive calculators of my youth, although I'm afraid I cannot remember its name.

We implement this with the **isqrt** function, given in listing 5.

Note that we treat 0 as a special case and that rather than maintain the estimate of the square root, we maintain the two terms that are averaged to retrieve it, and that the iteration stops when these two terms differ by no more than 1. At this point, therefore, one of them must be no larger than the square root plus 1 and the other no smaller than the square root minus 1. Taking one final step ensures that we round down and can thus guarantee that we choose the largest integer less than or equal to the root.

Using the first two terms of Taylor's expansion as an approximation of *f* we have

$$f(x_{i+1}) \approx f(x_i) + f'(x_i)(x_{i+1} - x_i)$$

Solving for $f(x_{i+1})$ equal to 0 yields

$$0 \approx f(x_i) + f'(x_i)(x_{i+1} - x_i)$$

$$x_{i+1} - x_i \approx \frac{f(x_i)}{f'(x_i)}$$

$$x_{i+1} \approx x_i - \frac{f(x_i)}{f'(x_i)}$$

Derivation 5

```
unsigned long
isqrt(unsigned long n)
{
  if(n==0)  return 0;
  unsigned long a0 = 2;
  unsigned long a1 = n/2;
  do
  {
    a0 = (a0+a1)/2;
    a1 = n/a0;
  }
  while((a0<a1) ? (a1-a0>1) : (a0-a1>1));
  return (a0+a1)/2;
}
```

Listing 5

We can be sure that we won't round down from the correct value for the square root when it happens to be an integer, since the second term is simply the argument divided by the first and so if one term is equal to the root, the other must be also.

That I have bothered to implement an integer square root function rather than casting to floating point and using the inbuilt square root function might come as something of a surprise. In my defence I can assure you that, at least on my machine, it is significantly faster. I cannot unfortunately adequately explain *why*; I rather suspect it has something to do with stalling the processor pipeline.

Counting the number of rulers

This time rather than print out the values of $\daleth_n$ as we iterate over the integers, we shall accumulate the number of rulers of length ε_n required to cover their graphs and, if we have truly eliminated any possible overflow, we shall be able to do so for every n up to the number of bits in an **unsigned long**.

Recasting our formula for $\daleth_n$ in terms of our ruler's length, ε_n, we have

$$\daleth_n(x) = \varepsilon_n 2^{\Omega\left(\left\lfloor \frac{x}{\varepsilon_n} \right\rfloor\right)} \qquad x \in [0, 1]$$

since ε_n is just 1 divided by 2^n. The vertical lines in its graph must lie at points where x is an integer multiple of ε_n and will have length equal to the positive difference in the function's value at that point and its value at the previous integer multiple of ε_n.

Noting that at 0, $\daleth_n$ has a value of 0, and that every subsequent vertical line is preceded by a horizontal line of length ε_n, the total length of the graph must be equal to

$$\sum_{i=1}^{2^n} \varepsilon_n + \varepsilon_n\left|2^{\Omega(i)} - 2^{\Omega(i-1)}\right|$$

The number of rulers of length ε_n required to cover the graph is therefore this expression divided by ε_n:

$$N(\varepsilon_n) = \sum_{i=1}^{2^n} 1 + \left|2^{\Omega(i)} - 2^{\Omega(i-1)}\right|$$

A C++ implementation of this calculation, named **count_rulers**, is presented in listing 6.

```
double
count_rulers(unsigned long n)
{
  static const int dig =
     std::numeric_limits<unsigned long>::digits;
  if(n>dig)  throw std::invalid_argument("");
  const unsigned long upper_bound =
    ((n==dig) ? 0UL : (1UL<<n))-1UL;
  const unsigned long max_prime   =
    isqrt(upper_bound);
  accumulator acc;
  std::vector<unsigned long> primes;
  unsigned long prev = 0;
  unsigned long i = 0;

  while(i!=upper_bound)
  {
    ++i;
    const unsigned long f = count_factors(i,
       primes.begin(), primes.end());
    const unsigned long curr = 1UL<<f;
    const unsigned long len  =
       (curr>prev) ? curr-prev : prev-curr;
    if(i<=max_prime && f==1) primes.push_back(i);
    acc += 1;
    acc += len;
    prev = curr;
  }
  acc += 1;
  acc += upper_bound+1-prev;
  return double(acc);
}
```

Listing 6

Once again, we've had to jump through several hoops to ensure that we can never encounter integer wrap around, as enumerated below.

Firstly, we bail out with an exception if n exceeds the number of bits in an **unsigned long**, since this is the maximum value for which we will be able to represent every non-negative integer less than 2^n.

Secondly, we set the upper bound of the iteration to 2^n-1, rather than 2^n, since the latter might be equal to 0. Note that we must treat n equal to the number of bits in an **unsigned long** as a special case since the shift operator irritatingly results in undefined behaviour if we shift by that many bits. I had failed to account for this in my original treatment and my compiler exacerbated my error by both failing to warn me and by doing exactly what I expected. Fortunately the ACCU conference delegates were far more standards compliant and pointed out my mistake. It just goes to show how fiddly dealing with numeric types can be.

Thirdly, we use an integer square root rather than the square of the current integer to determine whether an integer with just 1 factor is small enough to be added to our list of primes. Once again 2^n might be equal to 0 so we're forced to use the square root of 2^n-1 instead. For n greater than 2, this always yields the correct upper bound since in these cases the square root of 2^n is a compound number; specifically a power of 2. Fortunately, for n equal to 2 or less, every number less than 2^n must be 0, 1 or prime, so we won't actually need the list of primes during the calculation.

Finally, since we have not included the final term of the sum during the iteration, we add it after the iteration has completed. For any n greater than 0, we can be sure that the penultimate term of $\daleth_n$ is not equal to 0, so the final addition to our **accumulator** must involve a number strictly less than 2^n.

Phew!

So are we *finally* ready to calculate the fractal dimension of $\daleth_\infty$?

Well yes, dear reader, at long last we are. However, I have unfortunately run out of space and so the final analysis shall have to wait until next time. ■

Acknowledgements

With thanks to Stewart Brodie for pointing out my undefined use of the shift operator and to John Paul Barjaktarevic and Lee Jackson for proof reading this article.

References and further reading

[ANSI] *The C++ Standard*, American National Standards Institute, 1998.

[Asti] Thanks Asti. I think.

Multi-threading in C++0x

Threading support is being added to C++. Anthony Williams introduces us to the new facilities.

Concurrency and multithreading is all about running multiple pieces of code in parallel. If you have the hardware for it in the form of a nice shiny multi-core CPU or a multi-processor system then this code can run truly in parallel, otherwise it is interleaved by the operating system – a bit of one task, then a bit of another. This is all very well, but somehow you have to specify *what* code to run on all these threads.

High level constructs such as the parallel algorithms in Intel's Threading Building Blocks [Intel] manage the division of code between threads for you, but we don't have any of these in C++0x. Instead, we have to manage the threads ourselves. The tool for this is **`std::thread`**. (For full documentation of this and the rest of the library, see my implementation at [JustThread]).

Running a simple function on another thread

Let's start by running a simple function on another thread, which we do by constructing a new **`std::thread`** object, and passing in the function to the constructor. **`std::thread`** lives in the **`<thread>`** header, so we'd better include that first (Listing 1).

```
#include <thread>
void my_thread_func()
{}
int main()
{
  std::thread t(my_thread_func);
}
```

Listing 1

If you compile and run this little app, it won't do a lot: though it starts a new thread, the thread function is empty. What it *will* do is terminate with an unclean shutdown because we started a thread and then destroyed the **`std::thread`** object without waiting. Leaving that aside for a moment, let's make it do something, such as print **`"hello"`** (Listing 2).

If you compile and run this little application, what happens? Does it print hello like we wanted? Well, actually there's no telling. It might do or it might not. I ran this simple application several times on my machine, and the output was unreliable: sometimes it output **`"hello"`**, with a newline; sometimes it output **`"hello"`** *without* a newline, and sometimes **it didn't output anything**. What's up with that? Surely a simple app like this ought to behave predictably?

```
#include <thread>
#include <iostream>
void my_thread_func()
{
    std::cout<<"hello"<<std::endl;
}
int main()
{
    std::thread t(my_thread_func);
}
```

Listing 2

```
#include <thread>
#include <iostream>

void my_thread_func()
{
  std::cout<<"hello"<<std::endl;
}

int main()
{
  std::thread t(my_thread_func);
  t.join();
}
```

Listing 3

Waiting for threads to finish

Well, actually, no, this app does *not* have predictable behaviour. The problem is we're not waiting for our thread to finish. When the execution reaches the end of **`main()`** the program is terminated, whatever the other threads are doing. Since thread scheduling is unpredictable, we cannot know how far the other thread has got. It might have finished, it might have output the **`"hello"`**, but not processed the **`std::endl`** yet, or it might not have even started. In any case it will be abruptly stopped as the application exits.

If we want to *reliably* print our message, we have to ensure that our thread has finished. We do that by *joining* with the thread by calling the **`join()`** member function of our thread object (Listing 3).

Now, **`main()`** will wait for the thread to finish before exiting, and the code will output **`"hello"`** followed by a newline every time. This highlights a general point: **if you want a thread to have finished by a certain point in your code you have to wait for it**. As well as ensuring that threads have finished by the time the program exits, this is also important if a thread has access to local variables: we want the thread to have finished before the local variables go out of scope. It's also necessary to avoid the unclean shutdown – if you haven't called **`join()`** or explicitly declared that you're *not* going to wait for the thread by calling **`detach()`**, then the **`std::thread`** destructor calls **`std::terminate()`**.

Anthony Williams is the Managing Director of Just Software Solutions Ltd, where he mainly develops custom software for clients. He maintains the Boost Thread Library, and has considerable experience with writing high-quality multi-threaded software in C++. He can be contacted at anthony@justsoftwaresolutions.co.uk

this is one way of passing data to the thread function: pass it in as a constructor argument and store it as a data member

Running a function object on another thread

It would be quite limiting if new threads were constrained to run plain functions without any arguments – all the information needed would have to be passed via global variables, which would be incredibly messy. Thankfully, this is not the case.

In keeping with the rest of the C++ standard library, you're not limited to plain functions when starting threads – the `std::thread` constructor can also be called with instances of classes that implement the function-call operator. Let's say `"hello"` from our new thread using a function object (Listing 4).

If you're wondering about the extra parentheses around the `SayHello` constructor call, this is to avoid what's known as *C++'s most vexing parse*: without the parentheses, the declaration is taken to be a declaration of a *function called* `t` *which takes a pointer-to-a-function-with-no-parameters-returning-an-instance-of-*`SayHello`*, and which returns a* `std::thread` *object*, rather than an object called `t` of type `std::thread`. There are a few other ways to avoid the problem. Firstly, you could create a named variable of type `SayHello` and pass that to the `std::thread` constructor:

```
int main()
{
  SayHello hello;
  std::thread t(hello);
  t.join();
}
```

Alternatively, you could use copy initialization:

```
int main()
{
  std::thread t=std::thread(SayHello());
  t.join();
}
```

And finally, if you're using a full C++0x compiler then you can use the new initialization syntax with braces instead of parentheses:

```
int main()
{
  std::thread t{SayHello()};
  t.join();
}
```

In this case, this is exactly equivalent to our first example with the double parentheses.

Anyway, enough about initialization. Whichever option you use, the idea is the same: your function object is copied into internal storage accessible to the new thread, and the new thread invokes your `operator()`. Your class can of course have data members and other member functions too, and this is one way of passing data to the thread function: pass it in as a constructor argument and store it as a data member (Listing 5).

In this example, our message is stored as a data member in the class, so when the `Greeting` instance is copied into the thread the message is copied too, and this example will print `"goodbye"` rather than `"hello"`.

This example also demonstrates one way of passing information in to the new thread aside from the function to call – include it as data members of the function object. If this makes sense in terms of the function object then it's ideal, otherwise we need an alternate technique.

Passing arguments to a thread function

As we've just seen, one way to pass arguments in to the thread function is to package them in a class with a function call operator. Well, there's no

```
#include <thread>
#include <iostream>

class SayHello
{
public:
  void operator()() cons
  {
    std::cout<<"hello"<<std::endl;
  }
};

int main()
{
  std::thread t((SayHello()));
  t.join();
}
```

Listing 4

```
#include <thread>
#include <iostream>
#include <string>

class Greeting
{
  std::string message;
public:
  explicit Greeting(std::string const& message_):
    message(message_)
    {}
    void operator()() const
    {
      std::cout<<message<<std::endl;
    }
};

int main()
{
  std::thread t(Greeting("goodbye"));
  t.join();
}
```

Listing 5

if you need to pass **more than a couple of parameters** to your thread function then you might like to **rethink your design**

```
#include <thread>
#include <iostream>
#include <string>
#include <functional>
void greeting(std::string const& message)
{
  std::cout<<message<<std::endl;
}
int main()
{
  std::thread t(std::bind(greeting,"hi!"));
  t.join();
}
```

Listing 6

need to write a special class every time; the standard library provides an easy way to do this in the form of **`std::bind`**. The **`std::bind`** function template takes a variable number of parameters. The first is always the function or callable object which needs the parameters, and the remainder are the parameters to pass when calling the function. The result is a function object that stores copies of the supplied arguments, with a function call operator that invokes the bound function. We could therefore use this to pass the message to write to our new thread (Listing 6).

This works well, but we can actually do better than that – we can pass the arguments directly to the **`std::thread`** constructor and they will be copied into the internal storage for the new thread and supplied to the thread function. We can thus write the preceding example more simply as in Listing 7.

Not only is this code simpler, it's also likely to be more efficient as the supplied arguments can be copied directly into the internal storage for the thread rather than first into the object generated by **`std::bind`**, which is then in turn copied into the internal storage for the thread.

Multiple arguments can be supplied just by passing further arguments to the **`std::thread`** constructor (Listing 8).

The **`std::thread`** constructor is a variadic template, so it can take any number of arguments up to the compiler's internal limit, but if you need to pass more than a couple of parameters to your thread function then you might like to rethink your design.

```
#include <thread>
#include <iostream>
#include <string>
void greeting(std::string const& message)
{
  std::cout<<message<<std::endl;
}
int main()
{
  std::thread t(greeting,"hi!");
  t.join();
}
```

Listing 7

```
#include <thread>
#include <iostream>
void write_sum(int x,int y)
{
  std::cout<<x<<" + "<<y<<" =
     "<<(x+y)<<std::endl;
}

int main()
{
  std::thread t(write_sum,123,456);
  t.join();
}
```

Listing 8

Invoking a member function on a new thread

What if you wish to run a member function other than the function call operator?

To start a new thread which runs a member function of an existing object, you just pass a pointer to the member function and a value to use as the this pointer for the object in to the **`std::thread`** constructor. (Listing 9)

You can of course pass additional arguments to the member function too (Listing 10).

Now, the preceding examples both use a plain pointer to a local object for the this argument; if you're going to do that, you need to ensure that the object outlives the thread, otherwise there will be trouble. An alternative is to use a heap-allocated object and a reference-counted pointer such as

```
#include <thread>
#include <iostream>
class SayHello
{
public:
  void greeting() const
  {
    std::cout<<"hello"<<std::endl;
  }
};
int main()
{
  SayHello x;
  std::thread t(&SayHello::greeting,&x);
  t.join();
}
```

Listing 9

What if you want to **pass in a reference** to an existing object, and a **pointer just won't do**?

```
#include <thread>
#include <iostream>
class SayHello
{
public:
  void greeting(std::string const& message) const
  {
    std::cout<<message<<std::endl;
  }
};
int main()
{
  SayHello x;
  std::thread t(
     &SayHello::greeting,&x,"goodbye");
  t.join();
}
```

Listing 10

`std::shared_ptr<SayHello>` to ensure that the object stays around as long as the thread does:

```
#include <thread>
int main()
{
  std::shared_ptr<SayHello> p(new SayHello);
  std::thread t(&SayHello::greeting,p,"goodbye");
  t.join();
}
```

So far, everything we've looked at has involved copying the arguments and thread functions into the internal storage of a thread even if those arguments are pointers, as in the **`this`** pointers for the member functions. What if you want to pass in a *reference* to an existing object, and a pointer just won't do? That is the task of **`std::ref`**.

Passing function objects and arguments to a thread by reference

Suppose you have an object that implements the function call operator, and you wish to invoke it on a new thread. The thing is you want to invoke the function call operator *on this particular object* rather than copying it. You could use the member function support to call **`operator()`** explicitly, but that seems a bit of a mess given that it is callable already. This is the first instance in which **`std::ref`** can help – if **`x`** is a callable object, then **`std::ref(x)`** is too, so we can pass **`std::ref(x)`** as our function when we start the thread, as Listing 11.

In this case, the function call operator just prints the address of the object. The exact form and values of the output will vary, but the principle is the same: this little program should output three lines. The first two should be the same, whilst the third is different, as it invokes the function call operator on a *copy* of **`x`**. For one run on my system it printed the following:

```
#include <thread>
#include <iostream>
#include <functional> // for std::ref

class PrintThis
{
public:
  void operator()() const
  {
    std::cout<<"this="<<this<<std::endl;
  }
};
int main()
{
  PrintThis x;
  x();
  std::thread t(std::ref(x));
  t.join();
  std::thread t2(x);
  t2.join();
}
```

Listing 11

```
this=0x7fffb08bf7ef
this=0x7fffb08bf7ef
this=0x42674098
```

Of course, **`std::ref`** can be used for other arguments too – the code in Listing 12 will print **`"x=43"`**.

When passing in references like this (or pointers for that matter), you need to be careful not only that the referenced object outlives the thread, but also that appropriate synchronization is used. In this case it is fine, because we only access **`x`** before we start the thread and after it is done, but concurrent access would need protection with a mutex.

```
#include <thread>
#include <iostream>
#include <functional>

void increment(int& i)
{
  ++i;
}

int main()
{
  int x=42;
  std::thread t(increment,std::ref(x));
  t.join();
  std::cout<<"x="<<x<<std::endl;
}
```

Listing 12

There are several **consequences** to being able to **transfer ownership of a mutex lock**

Protecting shared data with std::mutex

We have seen how to start threads to perform tasks ‘in the background’, and wait for them to finish. You can accomplish a lot of useful work like this, passing in the data to be accessed as parameters to the thread function, and then retrieving the result when the thread has completed. However, this won’t do if you need to communicate between the threads whilst they are running – accessing shared memory concurrently from multiple threads causes undefined behaviour if either thread modifies the data. What you need here is some way of ensuring that the accesses are *mutually exclusive*, so only one thread can access the shared data at a time.

Mutexes are conceptually simple. A mutex is either ‘locked’ or ‘unlocked’, and threads try and lock the mutex when they wish to access some protected data. If the mutex is already locked then any other threads that try and lock the mutex will have to wait. Once the thread is done with the protected data it unlocks the mutex, and another thread can lock the mutex. If you make sure that threads always lock a particular mutex before accessing a particular piece of shared data then other threads are excluded from accessing the data until as long as another thread has locked the mutex. This prevents concurrent access from multiple threads, and avoids the undefined behaviour of data races. The simplest mutex provided by C++0x is **`std::mutex`**, which lives in the **`<mutex>`** header along with the other mutex types and the lock classes.

Now, whilst **`std::mutex`** has member functions for explicitly locking and unlocking, by far the most common use case in C++ is where the mutex needs to be locked for a specific region of code. This is where the **`std::lock_guard<>`** template comes in handy by providing for exactly this scenario. The constructor locks the mutex, and the destructor unlocks the mutex, so to lock a mutex for the duration of a block of code, just construct a **`std::lock_guard<>`** object as a local variable at the start of the block. For example, to protect a shared counter you can use **`std::lock_guard<>`** to ensure that the mutex is locked for either an increment or a query operation, as in Listing 13.

This ensures that access to counter is serialized – if more than one thread calls **`query()`** concurrently then all but one will block until the first has exited the function, and the remaining threads will then have to take turns. Likewise, if more than one thread calls **`increment()`** concurrently then all but one will block. Since both functions lock the same mutex, if one thread calls **`query()`** and another calls **`increment()`** at the same time then one or other will have to block. This *mutual exclusion* is the whole point of a mutex.

```
#include <mutex>
std::mutex m;
unsigned counter=0;
unsigned increment()
{
  std::lock_guard<std::mutex> lk(m);
  return ++counter;
}
unsigned query()
{
  std::lock_guard<std::mutex> lk(m);
  return counter;
}
```

Listing 13

```
#include <mutex>
#include <string>
std::mutex m;
std::string s;
void append_with_lock_guard(
   std::string const& extra)
{
  std::lock_guard<std::mutex> lk(m);
  s+=extra;
}
void append_with_manual_lock(
   std::string const& extra)
{
  m.lock();
  try
  {
    s+=extra;
    m.unlock();
  }
  catch(...)
  {
    m.unlock();
    throw;
  }
}
```

Listing 14

Exception safety and mutexes

Using **`std::lock_guard<>`** to lock the mutex has additional benefits over manually locking and unlocking when it comes to exception safety. With manual locking, you have to ensure that the mutex is unlocked correctly on every exit path from the region where you need the mutex locked, *including when the region exits due to an exception*. Suppose for a moment that instead of protecting access to a simple integer counter we were protecting access to a **`std::string`**, and appending parts on the end. Appending to a string might have to allocate memory, and thus might throw an exception if the memory cannot be allocated. With **`std::lock_guard<>`** this still isn’t a problem – if an exception is thrown, the mutex is still unlocked. To get the same behaviour with manual locking we have to use a catch block, as shown in Listing 14.

If you had to do this for every function which might throw an exception it would quickly get unwieldy. Of course, you still need to ensure that the code is exception-safe in general – it’s no use automatically unlocking the mutex if the protected data is left in a state of disarray.

The ability to transfer lock ownership between instances also provides an easy way to write classes that are themselves movable

Flexible locking with std::unique_lock<>

Whilst `std::lock_guard<>` is basic and rigid in its usage, its companion class template – `std::unique_lock<>` – is more flexible. At the most basic level you use it like `std::lock_guard<>` – pass a mutex to the constructor to acquire a lock, and the mutex is unlocked in the destructor – but if that's all you're doing then you really ought to use `std::lock_guard<>` instead. There are two primary benefits to using `std::unique_lock<>` over `std::lock_guard<>`:

1. you can transfer ownership of the lock between instances, and
2. the `std::unique_lock<>` object does not have to own the lock on the mutex it is associated with.

Let's take a look at each of these in turn, starting with transferring ownership.

Transferring ownership of a mutex lock between std::unique_lock<> instances

There are several consequences to being able to transfer ownership of a mutex lock between `std::unique_lock<>` instances: you can return a lock from a function, you can store locks in standard containers, and so forth.

For example, you can write a simple function that acquires a lock on an internal mutex:

```
std::unique_lock<std::mutex> acquire_lock()
{
  static std::mutex m;
  return std::unique_lock<std::mutex>(m);
}
```

The ability to transfer lock ownership between instances also provides an easy way to write classes that are themselves movable, but hold a lock internally, such as Listing 15.:

In this case, the function `lock_data()` acquires a lock on the mutex used to protect the data, and then transfers that along with a pointer to the data into the `data_handle`. This lock is then held by the `data_handle` until the handle is destroyed, allowing multiple operations to be done on the data without the lock being released. Because the `std::unique_lock<>` is

```
#include <mutex>
#include <utility>
class data_to_protect
{
public:
  void some_operation();
  void other_operation();
};
class data_handle
{
private:
  data_to_protect* ptr;
```

Listing 15

```
  std::unique_lock<std::mutex> lk;
  friend data_handle lock_data();
  data_handle(data_to_protect* ptr_,
     std::unique_lock<std::mutex> lk_):
     ptr(ptr_),lk(lk_)
  {}

public:
  data_handle(data_handle && other):
     ptr(other.ptr),lk(std::move(other.lk))
  {
    other.ptr=0;
  }
  data_handle& operator=(data_handle && other)
  {
    if(&other != this)
    {
      ptr=other.ptr;
      lk=std::move(other.lk);
      other.ptr=0;
    }
    return *this;
  }
  void do_op()
  {
    ptr->some_operation();
  }
  void do_other_op()
  {
    ptr->other_operation();
  }
};

data_handle lock_data()
{
  static std::mutex m;
  static data_to_protect the_data;
  std::unique_lock<std::mutex> lk(m);
  return data_handle(&the_data,std::move(lk));
}

int main()
{
  data_handle dh=lock_data(); // lock acquired
  dh.do_op();                 // lock still held
  dh.do_other_op();           // lock still held
  data_handle dh2;
  dh2=std::move(dh);          // transfer lock to
                              //other handle
    dh2.do_op();              // lock still held
}                             // lock released
```

Listing 15 (cont'd)

movable, it is easy to make **`data_handle`** movable too, which is necessary to return it from **`lock_data`**.

Though the ability to transfer ownership between instances is useful, it is by no means as useful as the simple ability to be able to manage the ownership of the lock separately from the lifetime of the **`std::unique_lock<>`** instance.

Explicit locking and unlocking a mutex with a std::unique_lock<>

As we saw in earlier, **`std::lock_guard<>`** is very strict on lock ownership – it owns the lock from construction to destruction, with no room for manoeuvre. **`std::unique_lock<>`** is rather lax in comparison. As well as acquiring a lock in the constructor as for **`std::lock_guard<>`**, you can:

- construct an instance without an associated mutex at all (with the default constructor);
- construct an instance with an associated mutex, but leave the mutex unlocked (with the deferred-locking constructor);
- construct an instance that tries to lock a mutex, but leaves it unlocked if the lock failed (with the try-lock constructor);
- if you have a mutex that supports locking with a timeout (such as **`std::timed_mutex`**) then you can construct an instance that tries to acquire a lock for either a specified time period or until a specified point in time, and leaves the mutex unlocked if the timeout is reached;
- lock the associated mutex if the **`std::unique_lock<>`** instance doesn't currently own the lock (with the **`lock()`** member function);
- try and acquire lock the associated mutex if the **`std::unique_lock<>`** instance doesn't currently own the lock (possibly with a timeout, if the mutex supports it) (with the **`try_lock()`**, **`try_lock_for()`** and **`try_lock_until()`** member functions);
- unlock the associated mutex if the **`std::unique_lock<>`** does currently own the lock (with the **`unlock()`** member function);
- check whether the instance owns the lock (by calling the **`owns_lock()`** member function;
- release the association of the instance with the mutex, leaving the mutex in whatever state it is currently (locked or unlocked) (with the **`release()`** member function); and
- transfer ownership between instances, as described above.

As you can see, **`std::unique_lock<>`** is quite flexible: it gives you complete control over the underlying mutex, and actually meets all the requirements for a *Lockable* object itself. You can thus have a **`std::unique_lock<std::unique_lock<std::mutex>>`** if you really want to! However, even with all this flexibility it still gives you exception safety: if the lock is held when the object is destroyed, it is released in the destructor.

std::unique_lock<> and condition variables

One place where the flexibility of **`std::unique_lock<>`** is used is with **`std::condition_variable`**. **`std::condition_variable`** provides an implementation of a condition variable, which allows a thread to wait until it has been notified that a certain condition is true. When waiting you must pass in a **`std::unique_lock<>`** instance that owns a lock on the mutex protecting the data related to the condition. The condition variable uses the flexibility of **`std::unique_lock<>`** to unlock the mutex whilst it is waiting, and then lock it again before returning to the caller. This enables other threads to access the protected data whilst the thread is blocked. A full discussion of condition variables is a complete article in itself, so we'll leave it for now.

Other uses for flexible locking

The key benefit of the flexible locking is that the lifetime of the lock object is independent from the time over which the lock is held. This means that you can unlock the mutex before the end of a function is reached if certain conditions are met, or unlock it whilst a time-consuming operation is performed (such as waiting on a condition variable as described above) and then lock the mutex again once the time-consuming operation is complete. Both these choices are embodiments of the common advice to hold a lock for the minimum length of time possible without sacrificing exception safety when the lock is held, and without having to write convoluted code to get the lifetime of the lock object to match the time for which the lock is required.

For example, in the following code snippet the mutex is unlocked across the time-consuming **`load_strings()`** operation, even though it must be held either side to access the **`strings_to_process`** variable (Listing 16).

Note that here we are relying on **`update_strings()`** being the only function that can add strings to the list, and that it is only run on one thread – if it may be called from multiple threads concurrently then we need to ensure that **`load_strings()`** is itself thread-safe, and that the behaviour is as desired. For example, if you only want one thread to call **`load_strings()`** then additional checks may be required. In general, if you unlock a mutex then you need to assume that the protected data has changed when you acquire the lock again.

Summary

In C++0x, you manage threads with the **`std::thread`** class. There are a variety of ways of starting a thread, but only one way to wait for it to finish – the **`join()`** member function. If you forget to join your threads then the runtime library will remind you by forcibly terminating your application.

Once you've got your threads up and running, you need to ensure that any shared data is correctly synchronized, and the most basic way to do that is with a mutex such as **`std::mutex`**. The safest way to lock a mutex is with an instance of **`std::lock_guard<>`**, though occasionally the flexibility of **`std::unique_lock<>`** might be needed.

Mutexes aren't the only way to synchronize data in C++0x, and there are other ways of acquiring locks than just **`std::lock_guard<>`** and **`std::unique_lock<>`**, but those will have to wait for another time. ■

References

[Intel] http://www.threadingbuildingblocks.org/
[JustThread] http://www.stdthread.co.uk/doc/

```
std::mutex m;
std::vector<std::string> strings_to_process;
void update_strings()
{
  std::unique_lock<std::mutex> lk(m);
  if(strings_to_process.empty())
  {
    lk.unlock();
    std::vector<std::string>
       local_strings=load_strings();
    lk.lock();
    strings_to_process.insert(
       strings_to_process.end(),
       local_strings.begin(),
       local_strings.end());
  }
}
```

Listing 16

Quality Matters: Correctness, Robustness and Reliability

What do we mean by quality? Matthew Wilson considers some definitions.

In the previous instalment I defined correctness as 'the degree to which a software entity's behaviour matches its specification' [QM-1], but didn't offer definitions of robustness or reliability. This time I'm going to take the plunge and attempt definitions of them. I embark on a (possibly deranged) attempt to equate computing with the worlds of Newtonian and Quantum Physics, along with the somewhat more obvious parallel drawn between the behaviour of software systems and chaos theory.

I'll do my best to keep my feet on planet Earth by using examples from real-world experience, illustrating how some software entities can be established to be correct, but the best we can hope for with most is to ensure adequate levels of robustness. I'll also comment on why correctness may be of no interest to non-programmers, and reliability is not of much interest to programmers.

Weaving together a cogent narrative for this instalment has been exceedingly difficult, and you may well feel that it's escaped me. If so, all I can say is watch out for the instalment on contract programming!

Extant definitions

Before I can begin to pontificate about robustness and reliability, I need to consider the definitions that currently exist in the canon.

The Shorter Oxford English Dictionary (SOED) [SOED] gives the following definitions:

- **Correct**: Free from error; accurate; in accordance with fact, truth, or reason. Conforming to acknowledged standards of style, manners, or behaviour; proper.
- **Robust**: Strong and sturdy in physique or construction. Involving or requiring bodily or mental strength or hardiness. Strong, vigorous, healthy; not readily damaged or weakened.
- **Reliable**: That which may be relied on; in which reliance or confidence may be put; trustworthy, safe, sure.

Steve McConnell [CC] gives these definitions:

- **Correctness**: The degree to which a system is free from [defects] in its specification, design, and implementation.
- **Robustness**: The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions.
- **Reliability**: The ability of a system to perform its requested functions under stated conditions whenever required – having a long mean time between failures.

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

Bertrand Meyer [OOSC] gives these definitions:

- **Correctness**: The ability of software products to perform their exact tasks, as defined by their specification.
- **Robustness**: The ability of software systems to react appropriately to abnormal conditions.
- **Reliability**: A concern encompassing correctness and robustness.

As is probably quite obvious, my definition of correctness is informed by these definitions, which I've examined many times previously. The important aspect taken from Meyer's definition [OOSC] is that correctness is relative to a specification. Indeed, Meyer states this most clearly in his Software Correctness Property [OOSC]:

Software Correctness Property: Correctness is a relative notion

Without a specification against which to compare behaviour, the notion of correctness is meaningless.

The important aspect taken from McConnell [CC] is that correctness is a variable notion, and that a software entity's behaviour may correspond to a specification to a certain degree. At first blush this may seem a bizarre idea. Certainly, a software entity that is known to fail to meet its specification is defective (aka incorrect), plain and simple. From the perspective of a potential user of a software entity, that its creator (or any other agent) may volunteer that it is 50% correct or 90% correct is of no use, because such figures, even if obtained by repeatable quantitative measurements, e.g. unit-tests, cannot be meaningfully used in the calculation of quantitative failure probabilities of a software system built from the offending entity. We'll discuss why in the next section.

Beyond this, there are other, serious, objections to attempting to make use of a software entity that is known to be defective – something known as the Principle of Irrecoverability – but those discussions will have to wait until another time.

So what is the purpose of considering correctness as a quantitative concept (in addition to its being a relative one)? Well, there are the practical benefits to the producers of a software entity in being able to quantify its degree of divergence from a specification. Of course, we all know that any given defect can, upon cursory examination, appear to be of the same magnitude (of corrective development effort) as another, and yet take two, five, ten, sometimes hundreds of times longer to correct. But averaged out over the course of a project, team, career, there is a usefulness to being able to quantify. Certainly, when I'm developing software libraries, I can take the temperature for the project velocity (if you'll pardon the atrocious mixing of my metaphors) via the defect fix rate in a new (or regressive) group of tests.

But all of the foregoing paragraph, while having some utility to our consideration of the subject, is pretty pedestrian stuff, infused with more than a whiff of equivocation. It could even be taken as an invitation to debates I'm not much interested in having. In point of fact, we needn't care about this stuff, because there's a point of far greater significance in eschewing the speciously attractive binary notion of correctness. Simply, a given software entity can exist in three apparent states of correctness:

- **correct**. It has been established correct against its specification
- **defective**. It has been established incorrect against its specification
- **unknown**. Its correctness has not been established against a specification

The third state is somewhat like poor old Schrödinger's cat [GRIBBIN], who is neither dead nor alive until examined. So too, software can be correct, or defective, or neither (known to be) correct nor defective. The latter state collapses into exactly one of the former when it is evaluated against a specification.

In this instalment I'm going to consider the notion that most, perhaps all, software systems are built up from layers of abstraction most of which are in the disconcerting third state of uncertain correctness. Furthermore, I'm going to argue that software has to be like this, and that's what makes it challenging, fun, and not a little frightening.

(Note: I'm still not going to discuss the definitions of what a specification is in this instalment. What a tease …)

Exactitude, non-linearity, Newtonian software, quantum execution environments, and why Software Development is not an engineering discipline.

A perennial debate within (and without) the software community is whether software development is an engineering discipline, and, if not, why not. Well, despite plentiful (mis)use of the term 'software engineer' in my past, I'm increasingly moving over to the camp of those whose opinion is that it is not an engineering discipline. To illustrate why I'm going to draw from three of my favourite branches aspects of science: Newtonian physics, Chaos theory, and Quantum physics, with a modicum of logic thrown in for good measure.

The Unpredictable Exactitude Paradox

As my career has progressed – both as practitioner (programmer, consultant) and as (imperfect) philosopher (author, columnist) – the issues around software quality have grown in importance to me. The one that confounds and drives me more than all others is (what I believe to be) the central dichotomy of software system behaviour:

The Unpredictable Exactitude Paradox: Software entities are exact and act precisely as they are programmed to do, yet the behaviour of (non-trivial) computer systems cannot be precisely understood, predicted, nor relied upon to refrain from exhibiting deleterious behaviour.

Note that I say programmed to do, not designed to do, because a design and its reification into program form are often, perhaps mostly, perhaps always, divergent. Hence the purpose of this column, and, to a large extent, the purposes of our careers. (The issue of the commonly defective transcription of requirements to design to code will have to wait for another time.)

Consider the behaviour of the computer on which I'm writing this epistle. Assuming perfect hardware, it's still the case that the sequence of actions – involving processor, memory, disk, network – carried out on this machine during the time I've written this paragraph have never been performed before, and that it is impossible to rely on the consequences of those actions. And that is despite the fact that the software is doing exactly what it's been programmed to do.

I mentioned earlier that the relationship between the size/number of defects and the effort involved to remedy them is not linear. This non-linearity is also to be seen in the relationship between the size/number of defects and their effects. Essentially, this is because software operates on the certain, strict interpretation of binary states, and there are no natural attenuating mechanisms in software at the scale of these states. If one Iron atom in a bridge is replaced by, say, a Molybdenum atom, the bridge will not collapse, nor exhibit any measurable difference in its ability to be a bridge. Conversely, an errant bit in a given process may have no effect whatsoever, or may manifest benignly (e.g. a slightly different hue in one pixel in a picture), or may have major consequences (e.g. sending confidential information to the wrong customer).

```
bool invert(register bool value)
{
  register bool result;
  if(0 != value)
  {
    result = 0;
  }
  else
  {
    result = 1;
  }
  return result;
}
```

Listing 1

We, as software developers, need language to support our reasoning and communication about software, and it must address this paradox, otherwise we'll be stuck in fruitless exchanges, often between programmers and non-programmers (clients, users, project managers), each of whom, I believe, tend to think and see the software world at different scales. I will continue the established use of the term correctness to represent exactitude. And I will, influenced somewhat by Meyer and McConnell, use the terms robustness and reliability in addressing the inexact, unpredictable, real behaviour of software entities.

Bet-Your-Life?: review

Let's look at some code. Remember the first of the Bet-Your-Life? Test cases from the last instalment [QM-1]:

```
bool invert(bool value);
```

We can implement this easily, as follows:

```
bool invert(bool value)
{
  return !value;
}
```

In fact, it'd be pretty hard to write any implementation other than this. Certainly there are plenty of (possibly apocryphal) screeds of long-winded alternative implementations available on the web (such as on www.thedailywtf.com), but pretty much any functionally correct implementation that does not involve fatuous complexity/dependencies – such as converting value to string and them using `strcmp()` against `"0"` or `"1"` – will evaluate to the following pseudo-machine code in Listing 1.

With languages that have a bona-fide Boolean type, such as Java and C#, the value may not need to be compared against `0`, and may well be implemented as equal to `true` (or to `false`). Other languages such as C and C++ represent (for historical and performance reasons [IC++]) a notional Boolean false value as being 0, and a notional Boolean true value as being all non-0 values. In those, comparison against zero is necessary, even for their built-in `bool` types! In either case, it's almost impossible to implement this function incorrectly.

If we permit ourselves the luxury of assuming a correctly functioning execution environment, then without recourse to any automated techniques, or even to a detailed written specification, we may reasonably assert the correctness of this function by visual inspection.

Now consider the definition of `strcmp()`, the second Bet-Your-Life? Test case:

```
int strcmp(char const* lhs, char const* rhs);
```

The Sign of Shame

Even with a function as logically straightforward as **strcmp()** there are different ways to implement it. Rather than this implementation shown, we could instead take the difference between ***s1** and ***s2** for each iteration and, when **!= 0**, return that value. What was interesting to me was that I had forgotten the nuances resolved in previous implementations, and I initially wrote the last line as:

```
  return *s1 - *s2;
```

Then (being as how I'm writing a column about software quality, and my Spidey-sense is set to **max**) I stopped and wondered how this would work between compilation contexts where **char** is signed or unsigned. Clearly, for certain ranges of values, the negation result will be different between the two. I then immediately set about writing a test program, and building for both signed and unsigned modes, and saw the suspected different behaviour for certain strings (with values in the range 0x80 – 0xff).

My next instinct was to include stlsoft/stlsoft.h – STLSoft has limited C-compatibility – and write the implementation in an **STLSOFT_CF_CHAR_IS_UNSIGNED**-dependent manner. Serendipitously, it was at this point that I thought to check with the C-99 standard, in order to verify my recollection that the return values could be any negative value for less-than (rather than strictly -1) and any positive value for greater-than (rather than strictly +1). What I was wrily amused to learn (and not a little appalled to have forgotten/not known) was that clause 7.21.4(1) stipulates that the return value '*is determined by the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the objects being compared*'. I've been programming C for, ulp!, 21 years, and have written several implementations of **strcmp()** (which, by happy coincidence, have done the right thing), but either I've forgotten that comparison was to be done unsigned, or I never knew it in the first place. Either way, it's quite sobering.

What this illustrates all too well is that software is exact, humans operate on assumption and expectation, and the two are not good bedfellows. (When I spoke to my good friend and regular reviewer, Garth Lancaster, about this, he too was ignorant of the unsigned comparison aspect of **strcmp()**. He was equally abashed by this omission/presumption, but callously stated that I bear more shame than he because I've written more articles/libraries/books. Bah!)

Here's an implementation I knocked up during the preparation of this instalment, without recourse to any I've written in the past (or to various open-source and commercial implementations).

```
int strcmp(char const* s1, char const* s2)
{
  for(; '\0' != *s1 && *s1 == *s2; ++s1, ++s2)
  {}
  return (int)(unsigned char)*s1 - (int)(
     unsigned char)*s2; /* C99: 7.21.4(1) */
}
```

Notwithstanding an issue I had with the signedness of the comparison (see sidebar), I intended to use the example of **strcmp()** as a (modest) stepping-stone in complexity – up from **invert()**, and down from **b64_encode()** – which relies on more assumptions about the execution environment:

- That **s1** points to a region of memory that is read-accessible over the range **[s1, s1 + N1]**, where **N1** is a non-negative integer representing the number of **char** elements in the memory starting at **s1** that do not have the value **'\0'**
- That **s2** points to a region of memory that is read-accessible over the range **[s2, s2 + N2]**, where **N2** is a non-negative integer representing the number of **char** elements in the memory starting at **s2** that do not have the value **'\0'**

If this smells suspiciously like a contract pre-condition [OOSC, IC++], well, that's something we'll examine in a later instalment.

This additional reliance on external factors is a significant part of the increased complexity over **invert()**. In languages such as C#(/.NET) and Java, it is reasonable to assume that an object reference is valid (or is the sentinel value, **null**), but in C (and C++) where pointers have free range, it is possible for **strcmp()** to receive pointers that:

- are intentionally valid, and point into a C-style string (which might be the empty string **""**)
- are null, having the value **NULL**, representing "no string". For **strcmp()**, this is unequivocally a defect, and in many execution environments will precipitate a hardware event that should be interpreted as a fatal condition, stopping the process, which is a good thing
- are invalid, in that they point into areas in the address range that are reserved for the kernel, are unmapped, write-only, and so forth. In many execution environments this will precipitate a hardware event, and in most cases this will be interpreted as a fatal condition, and the process stopped
- are unintentionally apparently valid, in that they point into some area of the address range in which the memory is read-accessible in a range that includes a (byte that can be interpreted as a) nul-terminator character (**'\0'**). In this case, what is unequivocally a defect (in the caller) will not be detected, and the defective process may continue to execute, with unpredictable outcomes

The possibility of the latter two options makes reasoning about the correctness of **strcmp()** and software entities build in terms of it more complicated than is the case for **invert()**. Specifically, it is possible for **strcmp()** to be passed invalid arguments (as a result of a defect elsewhere within the program), whereas all 'physically' possible arguments to **invert()** are valid.

The next Bet-Your-Life? Test case is **b64_encode()** (see Listing 2).

I'm not going to show the full implementation of this for brevity's sake. (If you're interested you can download the library [B64] and see for yourself.) Like **strcmp()** (and **invert()**), the b64 library has no dependencies on any other software libraries, not even on the C runtime library (except when contracts are being enforced, e.g. in debug builds). This permits a substantial level of confidence in behaviour, because only the b64 software entities themselves are involved in such considerations. Broadly speaking, it means that behaviour, once 'established', can be relied on regardless of other activities in the execution environment. However, it's fair to say that the internal complexity of **b64_encode()** is substantially increased over that of **strcmp()**. Consequently, I think it is impossible in a library such as this to stipulate its correctness based on visual inspection of the code; anyone who would do so would be rightly seen as reckless (at best).

Thus we can see that increasing complexity acts strongly against human-assessed correctness. But there's more to this than correctness. Let's now consider the final member of the Bet-Your-Life? Test cases, **Recls_Search()** from the recls library [RECLS]:

```
RECLS_API Recls_Search(
    char const* searchRoot
,   char const* patterns
,   int         flags
,   hrecls_t*   phSrch
);
```

An incomplete description of the semantics of this function are as follows:

- The function conducts a search of the file-system location specified by **searchRoot**, looking for entries that match the given patterns, according to the given search flags. If a file-system search can be initiated then a search context is created and assigned to ***phSrch** (to be used to elicit search results via other API functions), and a success return code returned; if not, a failure code is returned.
- If **searchRoot** is **NULL** or empty, the current directory is used.
- If patterns is **NULL**, the 'all entries' pattern – **"*"** on UNIX, **"*.*"** on Windows – is assumed.
- Multi-part patterns may be specified, separated by the '**|**' symbol, e.g. **"makefile*|*.c"**

```
size_t b64_encode(
    void const* src
,   size_t      srcSize
,   b64_char_t* dest
,   size_t      destLen
)
{
  . . .
  b64_char_t* p   =   dest;
  b64_char_t* end =   dest + destLen;
  size_t      len =   0;
  for(; NUM_PLAIN_DATA_BYTES <= srcSize;
      srcSize -= NUM_PLAIN_DATA_BYTES)
  {
    b64_char_t
       characters[NUM_ENCODED_DATA_BYTES];
    characters[0] = (b64_char_t)(
       (src[0] & 0xfc) >> 2);
    characters[1] = (b64_char_t)(((src[0] & 0x03)
       << 4) + ((src[1] & 0xf0) >> 4));
    characters[2] = (b64_char_t)(((src[1] & 0x0f)
       << 2) + ((src[2] & 0xc0) >> 6));
    characters[3] = (b64_char_t)(src[2] & 0x3f);
    src += NUM_PLAIN_DATA_BYTES;
    *p++ = b64_chars[(
       unsigned char)characters[0]];
    ++len;
    *p++ = b64_chars[(
       unsigned char)characters[1]];
    ++len;
    *p++ = b64_chars[(
       unsigned char)characters[2]];
    ++len;
    *p++ = b64_chars[(
       unsigned char)characters[3]];
    if( ++len == lineLen &&
        p != end)
    {     {
      *p++ = '\r';
      *p++ = '\n';
      len = 0;
    }
  }

  if(0 != srcSize)
  {
    unsigned char dummy[NUM_PLAIN_DATA_BYTES];
    size_t        i;
    for(i = 0; i < srcSize; ++i)
    {
      dummy[i] = *src++;
    }
    for(; i < NUM_PLAIN_DATA_BYTES; ++i)
    {
      dummy[i] = '\0';
    }
    b64_encode_(&dummy[0], NUM_PLAIN_DATA_BYTES,
       p, NUM_ENCODED_DATA_BYTES * (1 + 2),
       0, rc);
    for(p += 1 + srcSize; srcSize++ <
       NUM_PLAIN_DATA_BYTES; )
    {
      *p++ = '=';
    }
  }
  . . .
}
```

Listing 2

- File entries can be included in the search by specifying the **RECLS_F_FILES** flag. Directories by the **RECLS_F_DIRECTORIES** flag. (Files is assumed if neither specified.)
- etc. etc. …

Clearly the **recls** library (or at least this part of it) has substantial behavioural complexity. That alone makes it, in my opinion, impossible for any reasonable developer to stipulate its correctness. But that's only part of it. Of greater significance is that **recls** is implemented in terms of a great many other software entities, including library components (from STLSoft) and operating system facilities (e.g. the **opendir()** API on UNIX and the **FindFirstFile()** API on Windows). And even that is not the major issue. The predominant concern is that **recls** interacts with the file-system, whose structure and contents can (and do) change independently of the current process. By definition, it is impossible to establish correctness for **recls** or any other software entities who interact with aspects of the execution environment that are subject to change from other, independent software entities.

By now you're probably starting to worry now that I'm asserting that correctness cannot be stipulated. Am I saying that software cannot be correct?

Newtonian software, quantum execution environment

At the risk of embarrassing myself, because it's been 20 years since I did any formal study of the subject, I will now draw parallels between software+hardware and Newtonian+quantum physics.

Consider a point object travelling through an empty universe. In Newtonian physics, the object will continue to travel in the same line, at the same speed, forever more. If there are two point objects, they will influence each other's travel in predictable ways, based on their masses, positions and velocities. But add in a third, fourth, … trillionth object, and the behaviour of the universe becomes complex, and therefore unpredictable (beyond small timescales within which simplifying assumptions may be used to form reasonable approximate results). As is the case in reality, if the objects are non-point, then we have to consider rotation of the bodies, and heat, and a whole lot more besides, including chemistry, biology, even sociology and technology! Thus, even in a Newtonian universe, behaviour is non-linear (and unpredictable) due to the interactions of entities (in part because some of the quantities involved are irrational, and calculations thereby require infinite precision).

In a quantum universe, there are two challenges to our understanding even in the case of a single point object. For one thing, it is, in principle, impossible to state with certainty the position and momentum of the object. Second, it's possible that a virtual particle will spring into existence in any part of the otherwise empty universe at any time. (Here my inadequate training lets me down in understanding whether a virtual particle can have a net effect on our single travelling particle, but I think you get enough of the picture for us to have a working analogy.)

I contend that software is conceived in a Newtonian frame, where we imagine we can rely on perfect (non-defective) execution environments, and that hardware, necessarily, introduces a quantum aspect, due to the imperfect reliability of hardware systems (and the occasional cosmic ray that might flip a bit inside your processor) and the actions of other operating entities (programs, hardware, etc.). Let's look back to the Bet-Your-Life? Test examples from the previous instalment, and consider the behaviours in light of the two perspectives, where imperfect execution environments are subject to 'Quantum' surprises:

- **invert()** is subject only to failures in the execution environment – specifically the processor
- In addition to execution environment failures – in this case processor and memory system (e.g. bus + virtual memory) – **strcmp()** will also fail if it is passed invalid inputs as a result of a defect in another part of the program. The same goes for **b64_encode()**

```
[Test]
public void Test_False()
{
  Assert.IsTrue(BetYourLifeTests.Invert(false));
}
[Test]
public void Test_True()
{
  Assert.IsFalse(BetYourLifeTests.Invert(true));
}
```

Listing 3

- With **`Recls_Search()`**, we have a great many more reliances, both hardware and software. As well as failures in execution environment – processor, memory, disk – **`recls`** must also handle asynchronous external non-failure events of the disk system (such as examining a directory concurrent with files being added/removed from it by a different process/thread). And in pure software terms, it depends on the software entities from the system APIs and other open-source libraries

Specification

I'm not going to engage in discussion about specifications in this instalment, but must at least provide a definition in order that we can properly engage in further reasoning about correctness. Without further ado, a specification is one (or both, if used in concert) of two things:

Specification: A software entity's specification is the sum of all its passing unit-tests.

and

Specification: A software entity's specification is the sum of all its unfired active contract enforcements.

Everything else is fluff and air.

(Note: for today, I'm considering only functional aspects of specifications. Other aspects, such as performance – time and/or resource consumption – are outside the scope of this instalment, and will be discussed at another time. I'm also only going to be talking about measuring specifications in terms of unit-tests.)

Final definitions

Given the forgoing discussion, I'm now in a position to offer my definitions of these three important aspects of software quality.

Correctness: The degree to which a software entity's behaviour matches its specification.

Robustness: The adjudged ability of a software entity to behave according to the expectations of its stakeholders.

Reliability: The degree to which a software system behaves robustly over time.

Correctness

Correctness is exact and measurable. It is the concern of software developers.

When measured (against its specification), the correctness of a software entity 'collapses' from the unknown state to exactly one of two states: correct and defective.

The binary nature of measured correctness is a great thing. For example, consider that we measure the correctness of **`invert()`** as shown in Listing 3 (assuming a C# implementation, with NUnit [NUNIT]).

```
public static class LogicalOperations
{
  public static bool Nor(bool v1, bool v2)
  {
    return BetYourLifeTests.Invert(v1) &&
       BetYourLifeTests.Invert(v2);
  }
}
```

Listing 4

That's a complete functional test for **`BetYourLifeTests.Invert()`**. Informed by this, we could now implement another function, **`Nor()`**, as shown in Listing 4 (sticking with C#).

Knowing that **`Invert()`** is correct, we may choose to assert that **`Nor()`** will faithfully give expected behaviour based on visual inspection. And we could go on to completely measure that correctness with ease, involving just four unit-tests.

Robustness

However, add in just a little complexity and things get sticky very quickly. Consider that we've measured **`strcmp()`**'s correctness against a unit-test suite as shown in Listing 5, this time in C, with xTests [XTESTS].

Clearly, this is not a comprehensive test suite. But the permutations of arguments passed to **`strcmp()`** in the myriad programs built from it will dwarf that found in any unit-test suite. Consequently, we are all using **`strcmp()`** beyond its specification. Specifically, we are using **`strcmp()`** in a state of unknown correctness. How do we get away with it? We apply judgement.

A correct software entity has been proven so by mechanical means. A robust software entity has been judged as likely to behave according to expectations. This judgement is based on our knowledge of the software entity's interface, its likely complexity, its author(s), its published test suite, the skills and experience of the judge, and many other factors.

We can define a principle for robustness as:

```
static void test_equal()
{
  XTESTS_TEST_INTEGER_EQUAL(0, strcmp("", ""));
  XTESTS_TEST_INTEGER_EQUAL(0, strcmp("a", "a"));
  XTESTS_TEST_INTEGER_EQUAL(0, strcmp("ab",
     "ab"));
  XTESTS_TEST_INTEGER_EQUAL(0, strcmp("abc",
     "abc"));
}

static void test_less()
{
  XTESTS_TEST_INTEGER_LESS(0, strcmp("a", "b"));
  XTESTS_TEST_INTEGER_LESS(0, strcmp("ab",
     "bc"));
  XTESTS_TEST_INTEGER_LESS(0, strcmp("abc",
     "bcd"));
}

static void test_greater()
{
  XTESTS_TEST_INTEGER_GREATER(0, strcmp("b",
     "a"));
  XTESTS_TEST_INTEGER_GREATER(0, strcmp("bc",
     "ab"));
  XTESTS_TEST_INTEGER_GREATER(0, strcmp("bcd",
     "abc"));
}
```

Listing 5

The Robustness Principle: A robust software entity is comprised of:

- 0 or more correct software entities
- 0 or more robust software entities
- 0 defective software entities

We must now concern ourselves with how correctness propagates between software entity dependencies. Consider the function **`f()`**, which is implemented in terms of **`strcmp()`**:

```
int f(char const* s)
{
  return strcmp(s, "fgh");
}
```

What can we say about the correctness of **`f()`**? Well, until we test it, by definition it has unknown correctness. But howsoever we make use of it – whether in test or in a software application – we are using **`strcmp()`** outside the bounds of its specification, because **`"fgh"`** is not included in **`strcmp()`**'s test suite. By definition, therefore, we will be using **`strcmp()`** in a manner in which its correctness is unknown.

Consider that we now write a suite of tests for **`f()`**, as in Listing 6.

Since we have a specification for **`f()`**, and **`f()`** meets that specification, we can state that **`f()`** is correct. However, there is something a little strange about having a component that is correct when it is implemented using another component that has unknown correctness.

Taking this notion to extreme, we might wonder whether we can implement a correct software entity in terms of a defective one? Let's imagine that our implementation of **`strcmp()`** always returns a value of **`0`** when passed a string of less than three characters. With this behaviour it would fail four of the tests of its specification and thus be proven defective. But since the test suite for **`f()`** always uses strings of length three, it would still pass all cases. **`f()`** is proven correct, yet is implemented in terms of a defective component. That is more than a little strange, and violates the robustness principle given above.

The answer to this apparent conundrum lies in the notion of robustness. Confidence is placed in a software entity based on a number of factors, knowing that it will be used outside the exact, but necessarily limited, aspects of its specification. An implementation of **`f()`** that uses a correct **`strcmp()`** outside the bounds of its specification is, while common, something that should give pause for thought. An implementation of **`f()`** that uses a defective implementation of **`strcmp()`** violates the robustness principle and, in my opinion, should never be countenanced.

In both cases, the implementation of **`f()`** is brittle. And as each layer of abstraction and dependency is added, this brittleness spreads and compounds, and the combinatorial cracks through which extra-correctness behaviour can permeate increase. Thus, an important part of the skill/art of the software developer list in making judgements about robustness when implementing software entities in terms of others that have unknown correctness (and that must therefore be judged on their robustness).

```
static void test_equal()
{
  XTESTS_TEST_INTEGER_EQUAL(0, f("fgh"));
}
static void test_less()
{
  XTESTS_TEST_INTEGER_LESS(0, f("abc"));
  XTESTS_TEST_INTEGER_LESS(0, f("bcd"));
  XTESTS_TEST_INTEGER_LESS(0, f("cde"));
  XTESTS_TEST_INTEGER_LESS(0, f("def"));
}
static void test_greater()
{
  XTESTS_TEST_INTEGER_GREATER(0, f("ghi"));
  XTESTS_TEST_INTEGER_GREATER(0, f("hij"));
  XTESTS_TEST_INTEGER_GREATER(0, f("ijk"));
  XTESTS_TEST_INTEGER_GREATER(0, f("jkl"));
}
```

Listing 6

Robustness is inexact and subjective. It cannot be measured or proven, and it cannot be automated (beyond a few static analysis tricks). It is equally the concern of software developers, who must provide it, and stakeholders, whose experiences of the software system define it.

Reliability

I am moved to almost completely agree with McConnell's definition of reliability, but I do feel that reliability is a measurable, quantifiable, emergent property of a software system's behaviour. In some senses, it could be thought of as robustness over time, but robustness can't measured, so maybe it's better thought of as apparent robust action over time.

Reliability is more a concern to stakeholders than it is to developers, reflecting the differing perspectives between these groups. To stakeholders, it is almost entirely irrelevant how many constituent software entities were correct versus those adjudged robust. To stakeholders, the proof of the pudding is the eating, and that's its reliability.

Conversely, to software developers, the more correctness that can be adduced the better, because it simplifies the construction of dependent software entities. Reliability, on the other hand, is a distant prospect to a developer, and probably viewed in different ways. For example, I can say that I am motiviated, by pride, to have 0 failures ever; 1 or 10 failures would be equally galling. Conversely, frequency of failures is of proper relevance to a user, who may well tolerate one failure per month if the software can cost him/her significantly less than the version that fails once per year (or never). Many do, and many others just expect software failure, otherwise how do we explain the popularity of certain operating systems, editors, websites, …

Naturally, I'm not suggesting that tuning software failure frequencies is a good thing; I believe that we can all write much more robust software without suffering in the process. That's the raison d'être of this column, and as we proceed I intend to pursue the notion that we should all be aiming for maximum quality all the time.

Summary

At this point I'd intended to go on to examine some of the interesting conflicts between correctness and robustness, and between them and other software characteristics, as well as discussing practical techniques for ensuring robustness when correctness is not achievable. I've even got an argument in favour of Java's hateful checked exceptions. But I've run out of space (and time), and these will have to wait until another instalment.

For the moment, I'll posit a parting rubric that correctness is the worthy aim wherever possible (which is rare), and robustness is the practical must-have in all other circumstances.

I'm not sure what's coming in the next instalment, but I'm determined that it's going to have a lot more code, and a lot less philosophy than this one. It's too exhausting!

See you next time. ■

References and asides

[B64] http://synesis.com.au/software/b64/

[CC] *Code Complete*, 2nd Edition, Steve McConnell, Microsoft Press, 2004

[GRIBBIN] *In Search of Schrödinger's Cat*, John Gribbin, Corgi, 1984

[IC++] *Imperfect C++*, Matthew Wilson, Addison-Wesley, 2004

[NUNIT] http://nunit.org/

[OOSC] *Object Oriented Software Construction*, 2nd Edition, Bertrand Meyer, Prentice-Hall, 1997

[QM-1] 'Quality Matters, Part 1: Introductions, and Nomenclature', Matthew Wilson, *Overload 92*, August 2009

[RECLS] http://www.recls.org/

[SOED] *The New Shorter Oxford English Dictionary*, Thumb Index Edition, ed. Lesley Brown, Clarenden Press, Oxford, 1993.

[XTESTS] http://xtests.org/

The Generation, Management and Handling of Errors (Part 2)

Dealing with errors is a vital activity. Andy Longshaw and Eoin Woods conclude their pattern language.

This is the second part of a paper that presents patterns for handling error conditions in distributed systems. The patterns in the collection are illustrated in Figure 1.

Some of the patterns (SPLIT DOMAIN AND TECHNICAL ERRORS, LOG AT DISTRIBUTION BOUNDARY, UNIQUE ERROR IDENTIFIER) were discussed in the first part of the paper in the previous issue. The remaining patterns are covered in this second part. At the end of the paper, a set of proto-patterns is briefly described. These are considered to be important concepts that may or may not become fully fledged patterns as the paper evolves.

BIG OUTER TRY BLOCK

Problem

Unexpected errors can occur in any system, no matter how well it is tested. Such truly exceptional conditions are rarely anticipated in the design of the system and so are unlikely to be dealt with by the system's error handling strategy. This means that these errors will propagate right to the edge of the system and will appear to 'crash' the application if not handled at that point. This may lead to some or all of the information associated with such unexpected errors being lost, leading to difficulties with the rectification of underlying problem in the system.

Context

A distributed system with a largely 'lay' user community, probably using graphical user interfaces. The interface is likely to be very simple: possibly even a 'kiosk style' interface. Users are mostly on remote sites and will not do much to report errors if they can work around them.

Forces

- If an in-depth error report, particularly for a technical error, is presented to an end user, they are unlikely to be able report its content in enough detail for the underlying problem to be unambiguously identified and so the details of the error are likely to be lost.
- If technical errors are presented to users on a regular basis, they will start to ignore them rather than to go through the process of trying to report them and knowledge of the existence, as well as the details, of these errors will be lost entirely.
- Members of the support staff need to be able to associate user problem reports with logged error information, but detailed error information can be very big and it all looks the same to an end user, making it difficult to report.
- We want to avoid having to write code to handle technical errors at multiple layers of the application but this opens up the risk that such errors will 'leak' through to the user.

Andy Longshaw works for Barclays Bank delivering IT solutions with particular focus on reusability. He has been delivering and explaining technology and system architecture for most of the last decade. He can be contacted at www.blueskyline.com

Eoin Woods is a software architect at Barclays Global Investors, heading the application architecture group.He has been working in software engineering for nearly 20 years and is co-author of the book 'Software Systems Architecture'. He can be contacted at www.eoinwoods.info

Solution

Implement a BIG OUTER TRY BLOCK at the 'edge' of the system to catch and handle errors that cannot be handled by other tiers of the system. The error handling in the block can report errors in a consistent way at a level of detail appropriate to the user base.

Implementation

In the system's ultimate client, wrap the top-level invocation of the system in a BIG OUTER TRY BLOCK that will catch any error – domain or technical – propagating up from the rest of the system. The BIG OUTER TRY BLOCK should differentiate between technical errors (such as databases not being available) and domain errors (such as performing business process steps in the wrong order) as suggested in SPLIT DOMAIN AND TECHNICAL ERRORS.

Technical errors should be logged for possible use by technical support staff and the user should then be informed that something terrible has happened in general terms, making it clear that what has happened is not related to their use of the system.

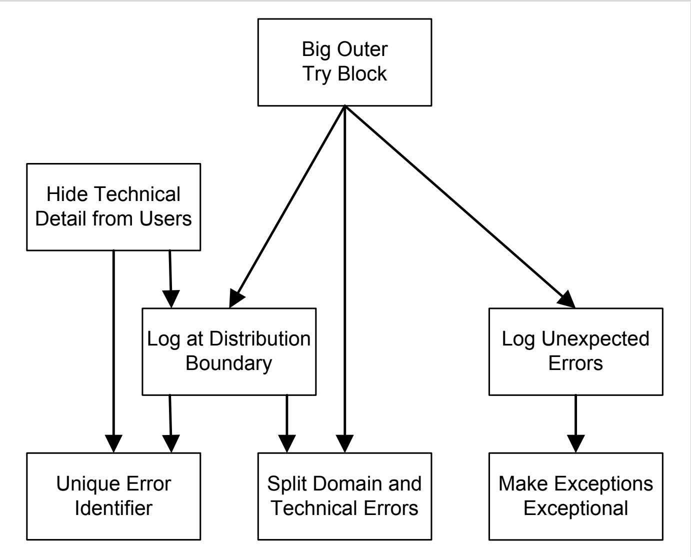
Big Outer Try Block
Hide Technical Detail from Users
Log at Distribution Boundary
Log Unexpected Errors
Unique Error Identifier
Split Domain and Technical Errors
Make Exceptions Exceptional

Figure 1

The technical details of errors that occur are typically of no interest to the end-users of a system.

A domain error that reaches the BIG OUTER TRY BLOCK is probably a failure in the design of the user interface that resulted in an unanticipated business process state being reached and as such should be treated as a system fault. In such cases, again the error should be logged and a user-friendly message displayed, but in this case the message can include details of the problem encountered, as these details are likely to be meaningful to the user since they relate to the business process that they were performing.

Finally, a totally unpredictable error (such as an exception indicating a resource shortage due to having run out of memory) that reaches the BIG OUTER TRY BLOCK is some form of internal or environmental error that could not be handled at a lower level. As with a technical error, a generic error should be displayed to the user and the details of the error logged locally.

An example of the structure of a BIG OUTER TRY BLOCK's implementation is shown in Listing 1.

```
public class ApplicationMain
{
  ...
  public static void main(String[] args)
  {
    try
    {
      ApplicationMain m = new ApplicationMain() ;
      m.initialize() ;
      m.execute() ;
      m.terminate() ;
    }
    catch(AppDomainException de)
    {
     // Domain exceptions shouldn't get to this
     // level as they should be handled in the
     // user interface. If they get here, report
     // the text to the user and log them in a
     // local log file
    }
    catch(AppTechnicalException te)
    {
     // Technical exceptions here are probably
     // user interface problems. Display a
     // generic apology and log to a local log file
    }
    catch(Throwable t)
    {
     // Other exception objects must be internal
     // errors that could not be caught and
     // handled elsewhere. Display a generic
     // apology and log to a local log file
    }
  }
}
```

Listing 1

Positive consequences

- Error information is never lost because of unexpected errors propagating to the edge of the system and 'leaking out'. The error information is always captured in its entirety to allow it to be retrieved for support and diagnostic purposes.
- Users are never surprised by an application simply stopping or crashing, but are always informed that something has gone wrong in a user comprehensible form.
- If the application does fail in an unexpected way, it always handles this condition in a consistent manner.
- Other parts of the system may have simpler error handling as they do not need to include handling for totally unpredictable errors.

Negative consequences

- The outer catch block needs to be carefully implemented so that exception information from the very wide range of possible exception types that it must handle does not get lost when handling that scenario.

Related patterns

- Implementing SPLIT DOMAIN AND TECHNICAL ERRORS makes the implementation of BIG OUTER TRY BLOCK simpler because the different types of error can be easily differentiated and handled differently.
- This pattern can be combined with the HIDE TECHNICAL DETAILS FROM USERS pattern in order to ensure that suitable messages are reported to the user when the BIG OUTER TRY BLOCK is triggered.
- BIG OUTER TRY BLOCK combines with LOG AT DISTRIBUTION BOUNDARY so that the errors that it receives are more relevant and potentially suitable for display to the user.
- This pattern can be combined with UNIQUE ERROR IDENTIFIER in order to ensure that errors logged by the BIG OUTER TRY BLOCK can be clearly identified.
- A BIG OUTER TRY BLOCK is a form of DEFAULT ERROR HANDLING [Renzel97]
- This concept is also mirrored in the Java idiom SAFETY NET in [Haase]

HIDE TECHNICAL ERROR DETAIL FROM USERS

Problem

The technical details of errors that occur are typically of no interest to the end-users of a system. If exposed to such users, this error information may cause unnecessary concern and support overhead.

Context

An application with a largely non-technical user community, probably using the system via some sort of graphical interface.

Forces

- If a detailed error report, particularly for a technical error, is presented to an end user, they are likely to find its content incomprehensible.
- If technical errors are presented to end users or the application simply stops or crashes unexpectedly then this is likely to cause a loss of confidence in the application, possibly leading to a reluctance to use it.
- Inconsistent user error reporting makes the system difficult to support as it confuses the users and prevents them reporting problems accurately and consistently.
- Technical errors generally have a lot of information that is useful for support staff but it is irrelevant to the end user.
- If the system under consideration offers a limited capability user interface (such as that offered by a mobile device), the interface may not be capable of reporting detailed error information in a comprehensible manner.

Solution

Implement a standard mechanism for reporting unexpected technical errors to end-users. The mechanism can report all errors in a consistent way at a level of detail appropriate to the different user constituencies who need to be informed about the error.

Known uses

The authors are aware of a number of instances of this pattern in enterprise systems, although none of them are available for public study. Some examples of using this pattern outside the domain of enterprise systems include the following.

- A number of self service web-sites report a generic error message if an internal error occurs, including a unique error identifier that can be used to report the situation to a helpdesk.
- Some intelligent hardware devices respond to errors that occur by displaying a simple error screen (in some cases including a unique error identifier to allow the error to be uniquely identified by the hardware supplier), that instructs the owner to call a telephone hotline in order to obtain assistance.
- The Microsoft Windows error dialog that is displayed when an application encounters an internal error is an example of the use of this pattern.

Implementation

Within the system's user interface implementation, provide a single, straightforward mechanism for reporting technical errors to end-users. The mechanism is almost certainly going to be a simple API call of the general form:

```
  void notifyTechnicalError(Throwable t) ;
```

The mechanism created should perform two key tasks:

- Log the full technical details of the error that has occurred for possible use by technical support staff.
- Display a friendly, user-centric message to inform the user that something terrible has happened in general terms, making it clear that what has happened is not related to their use of the system. The user message should include some form of unique identifier to allow the user to easily report what has happened, via some form of helpdesk.

  Ideally, the user reporting of the error should be automated in some way (for example using desktop email automation) in order to make the process of reporting as simple as possible and to avoid errors during the process. If the process is automated, this will avoid the problem of users ignoring the errors because reporting them is too much trouble and will ensure accurate reporting of each error.

  From the information in the user's error report, a helpdesk can escalate the problem to an administrator who can access detailed error information elsewhere in the system, using the identifier as a key.

Use this mechanism to handle all technical errors encountered by the system's user interface.

Positive consequences

- Users of the system are never presented with technical error information that could confuse or worry them.
- The system becomes easier to support because support staff can correlate fatal system errors with logged information in order to allow them to understand and investigate the problem.
- Error handling in the GUI implementation is simplified and standardized.

Negative consequences

- Concealing all error information from the end-user means that a knowledgeable end-user is powerless to apply their own knowledge to solve the problem. This could mean that a number of avoidable calls are made to helpdesks, that could otherwise be resolved by the users themselves.
- The implementation of this pattern may require the implementation of a reasonably sophisticated error-handling framework and this may be perceived as a significant overhead within the development process.

Related patterns

- This pattern fits very naturally with the BIG OUTER TRY BLOCK to ensure that technical errors are displayed and logged appropriately.
- Using the LOG AT DISTRIBUTION BOUNDARY pattern to govern where technical errors are logged ensures that the received are suitable for reporting to the end user and include a suitable unique identifier.
- This pattern can alternatively be combined directly with UNIQUE ERROR IDENTIFIER to ensure that errors can be clearly identified and to mitigate the potential confusion arising from one error causing multiple log entries.
- An ERROR DIALOG [Renzel97] forms part of a strategy to hide errors from users.

LOG UNEXPECTED ERRORS

Problem

Much domain code includes handling of exceptional conditions and is designed to recognize and handle each condition according to a business process definition (typically the offending transaction being rejected or a new domain entity being created). If such routine error conditions are logged, this makes real errors requiring operator intervention difficult to spot.

Context

Where systems are created in organizations with complex domain processing, or systems with a large number of routinely expected error conditions to which the processes specify the response.

Forces

- The system should report errors when they occur so that they can be investigated and fixed but it is easy for serious errors to be hidden under large numbers of spurious or trivial problems. It should be obvious when operator intervention is required.
- If all possible error conditions, including those routinely encountered during normal operation, are reported then log management becomes much more difficult due to the speed at which the error logs fill up.

- Recording lots of errors increases the amount of logging code and the number of error messages that need to be managed, which reduces the maintainability of the system.

Solution

Implement separate error handling mechanisms for expected and unexpected errors. Error conditions that are expected to arise in the course of normal domain processing should not be logged but handled in the code or by the user. Hence, any logged error should be viewed as requiring investigation.

Implementation

Throughout the system's implementation, use two distinct error handling approaches for expected and unexpected errors:

- Log unexpected errors according to the other patterns, such as LOG AT DISTRIBUTION BOUNDARY, and put in place a process that ensures the error triggers operator intervention to resolve the situation.
- Do not log expected errors, but handle them as part of the system's normal operation. This may be done in the code itself, maybe by trying different domain logic that may be able to handle the given inputs or scenario or creating alternative domain entities.

Alternatively, the application may interact with the user, inform them of the problem (in appropriate terms – see HIDE TECHNICAL DETAIL FROM USERS) and prompt them to re-start part or all of the current operation.

By following these principles, errors such as 'could not connect to database' are not hidden by hundreds of routine error conditions such as 'no such product code' (perhaps caused by a user misreading a code from a piece of product packaging). As the former error is a significant error requiring investigation, while the second is an expected error condition, the former would be logged and the latter handled algorithmically by the business logic, without logging the condition.

One variation on this approach is to log different types of error message to different places. For example, in terms of the application itself a user failing to authenticate may not be worth recording. However, from the system's point of view (i.e. the operating system) the security policy may require all failed authentications to be logged. This is usually resolved by logging different types of errors to different logs, such as the application event log and security event log provided under Windows. Such partitioning allows different logs to be created to serve the needs of different areas of concern. Another example of this is where knowledge of the patterns in which errors occur would be of interest to developers – large numbers of failed searches at a search engine site may indicate a usability problem. However, such errors are not of interest to the operations team who are responsible for keeping the system running. In this case, the expected errors could be logged to a different location where they will not interfere with the operational errors but can be retrieved later by the development team for further analysis.

A second variation is to log different types of error message in one location but to mark each log message with one or more attributes that allow a set of filters to be created to provide the ability to extract various subsets of the log content on demand to support different uses (such as error monitoring versus usability analysis).

Positive consequences

- Errors that appear in logs always indicate exception conditions and so can be used to initiate support and diagnostic activities.
- Spurious messages indicating that expected conditions have occurred do not prevent easy recognition of the occurrence of exceptional conditions.
- Logs do not quickly fill up with spurious messages created as a result of normal operation.
- Application code is simplified as a result of the reduction in the number of log messages that need to be produced.

Negative consequences

- If the recognition of expected errors is not specific or accurate enough then there is the danger of masking or ignoring exceptional conditions, by incorrectly assuming them to be manifestations of expected error conditions.

Related patterns

- You need to ensure that the correct distinction is made between expected errors and exceptional occurrences as described in MAKE EXCEPTIONS EXCEPTIONAL.
- It may be helpful to classify errors as 'domain' or 'technical' errors, as described in SPLIT DOMAIN AND TECHNICAL ERRORS.
- An approach such as 3 CATEGORY LOGGING [Dyson04] can help to make a log filterable.

MAKE EXCEPTIONS EXCEPTIONAL

Problem

A number of languages include exception handling facilities and these are powerful additions to the error handling toolkit available to programmers. However, if exceptions are used to indicate expected error conditions occurring, then the calling code becomes much more difficult to understand.

Context

Any situation where a language with exception handling built into it is in use.

Forces

- An application should be designed to handle and recover from most domain errors but some unexpected errors will always occur. Examples of the latter include incorrect or missing application data in the database and incorrect or missing values in configuration files.
- The code paths for handling 'recoverable' errors and 'unrecoverable' errors are usually quite distinct so they should be easily differentiated.
- Large numbers of exceptions generated cause problems for the consumer of a class/method – especially in a language that uses checked exceptions.
- We want to avoid convoluted code and algorithm distortion when routine error conditions (such as 'end of list') are encountered.

Solution

Indicate expected domain errors by means of return codes. Only use exceptions to indicate runtime problems such as underlying platform errors or configuration/data errors.

Implementation

When designing the interfaces in your system you should classify errors into two types:

- Conditions that will occur routinely in standard algorithms, which should be indicated by returning a reserved return value. This could be a null pointer, an empty list or a specific return code (**`E_INVALID`**). An example of this would be returning an empty list from a search operation that did not match any items.
- Conditions that will only occur due to unexpected errors, which should be indicated by raising language exceptions. Examples of such conditions include those caused by underlying platform or network failure (cannot connect to database), incorrect configuration (bad database connection string) or bad application data (customer id – not name – could not be found).

Errors of the first type will be handled as part of the standard business logic in the system. On the other hand, errors of the second type will normally be handled by a combination of logging and exiting the current code block via an exception path.

It is worth briefly exploring the differences and the blurring of the boundaries here through an example. Consider a component in a retail system that offers out two methods to look up product information. One of these methods allows you to look up products either by keyword or wildcard text string and returns a list of matching products. The other method requires a numeric product code such as a barcode and returns the single product matching that code. The component is backed by a database containing all the products stocked by the retailer.

The search by keyword/wildcard has no guarantee of finding a matching product. Typically, the keyword/wildcard will be entered by a user and so could be subject to all forms of data problems such as mis-spelling or unrealistic expectations (e.g. entering `"Elton John"` when the retailer just sells food – not CDs). Hence, semantically you could expect no products to be returned – this is an expected business condition, however unhelpful it is to the user of the calling application. Having said that, the user can always get the answer they want by trying again – providing sensible input to the search.

On the other hand, there is more of a semantic implication that the method that requires a product code should find something. Unless users of the system are prone to scanning in barcodes from random products they bring into work, any product scanned in store should be in the database: you should not be able to provide a code that cannot be found. In this case, you could justifiably throw an exception as the only way this condition can occur is if there is a problem with the data in your database. Not only can the user not get the right answer by re-scanning the product (same answer each time…), but in terms of the system this situation needs resolving (i.e. the data in the database needs correcting).

Finally, in either case if the component cannot connect to the database for whatever reason a technical exception should be raised (indeed, the underlying platform will probably raise one for you).

Positive consequences

- Application code is simplified as it does not need to include exception handling constructs within normal algorithms.
- Exceptional conditions in code can all be treated as abnormal situations requiring error handling and as such can be handled via a uniform strategy.

Negative consequences

None

Related patterns

- Expected errors should not be logged, as described in DON'T LOG BUSINESS PROCESS ERRORS, but unexpected errors – whether technical or domain exceptions – should be logged.
- Ward Cunningham's CHECKS pattern language for information integrity [Cunningham] provides a great deal of guidance relating to the design of data validation in user interfaces (i.e. 'Type 1' errors in this pattern's Implementation section).

Proto-Patterns

IGNORE IRRELEVANT ERRORS

- Problem

  Sometimes technical errors or exceptions do not denote a real problem and so reporting them can just be confusing or irritating for support staff.
- Solution

  Assess what action can be taken in response to an error and only log it if there is a relevant course of action. An example is `ThreadAbortException` which is raised under ASP.NET whenever you transfer to another page using `Server.Transfer()`. This is not an error condition – just a side-effect – and so is of no consequence to support staff. Also, you will get lots of these in any busy web-based system.

SINGLE TYPE FOR TECHNICAL ERRORS

- Problem

  There are a myriad different technical errors that may occur during a call to an underlying component.
- Solution

  When you create your exception/error hierarchy for your application, define a single error type to indicate a technical error, e.g. `SystemError`. The definition and use of a single technical error type simplifies interfaces and prevents calling code needing to understand all of the things that can possibly go wrong in the underlying infrastructure. This is especially useful in environments that use checked exceptions (e.g. Java). ■

References

[Cunningham] CHECKS: A Pattern Language of Information Integrity, http://c2.com/ppr/checks.html

[Dyson04] Dyson 2004 *Architecting Enterprise Solutions: Patterns for High-Capability Internet-based Systems*, Paul Dyson and Andy Longshaw, John Wiley and Sons, 2004

[Haase] Java Idioms – Exception Handling, linked from http://hillside.net/patterns/EuroPLoP2002/papers.html.

[Renzel97] 'Error Handling for Business Information Systems', Eoin Woods, linked from http://hillside.net/patterns/onlinepatterncatalog.htm

Acknowledgements

We'd like to thank our EuroPLoP 2005 shepherd, Ofra Homsky for her thorough and valuable feedback during this paper's review process and our original EuroPLOP 2004 shepherd Bob Hanmer for providing very valuable advice on the original paper, the members of the EuroPLoP 2004 and 2004 workshops, and the members of the OT2004 workshop at which this paper was first presented.