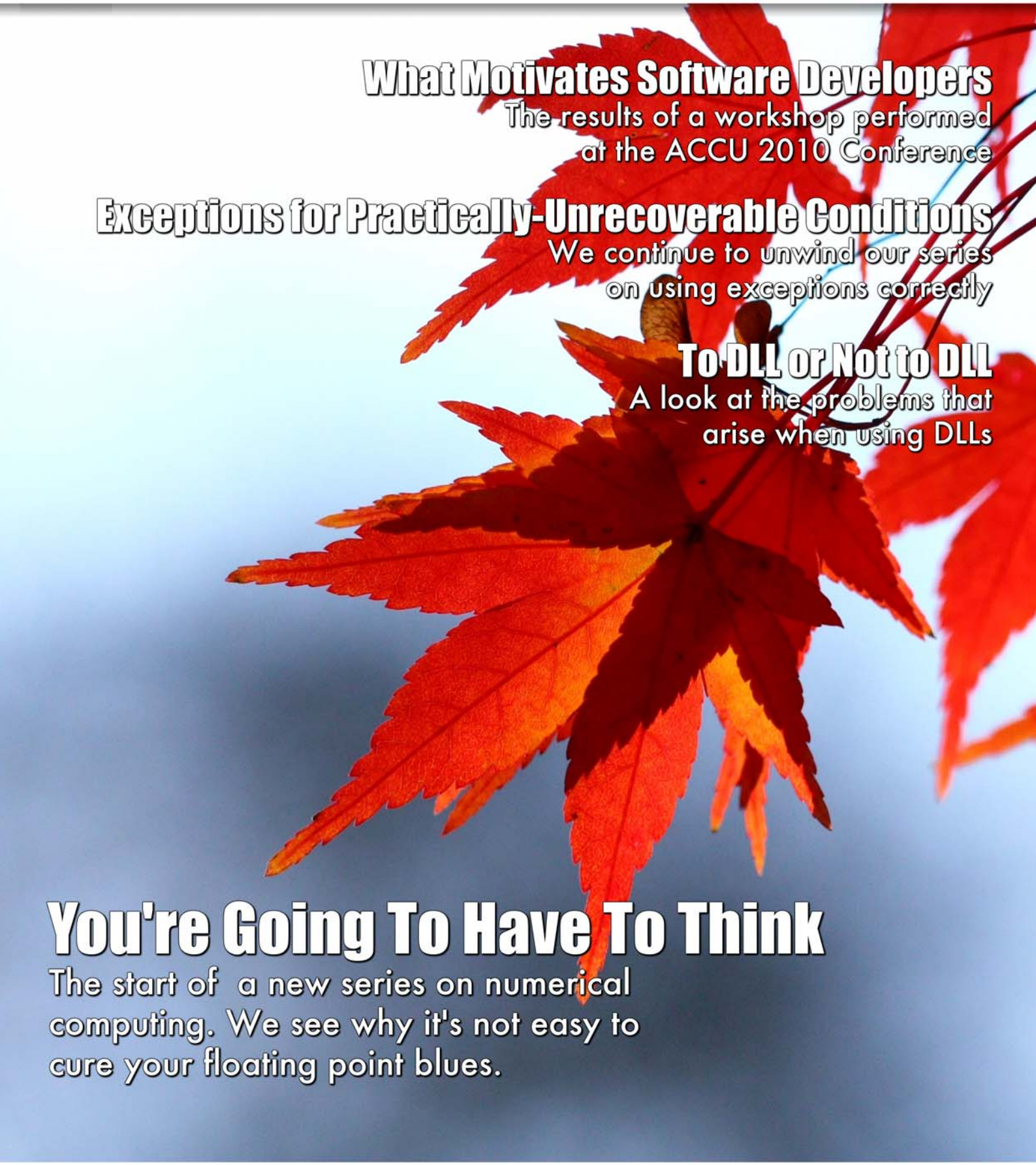
overload 99

OCTOBER 2010 £3



What Motivates Software Developers

The results of a workshop performed at the ACCU 2010 Conference

Exceptions for Practically-Unrecoverable Conditions

We continue to unwind our series on using exceptions correctly

To DLL or Not to DLL

A look at the problems that arise when using DLLs

You're Going To Have To Think

The start of a new series on numerical computing. We see why it's not easy to cure your floating point blues.

A magazine of ACCU

ISSN 1354-3172

OVERLOAD 99

October 2010

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 100 should be submitted by 1st November 2010 and for Overload 101 by 1st January 2011.

ACCU

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

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

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

Lies, Damn Lies, and Statistics

Making a good decision is vital. Ric Parkin looks at what information we use.

Our boss has been asking that really horrible question: 'When will the new release be out?' This is a non-trivial thing to answer at the best of times, so I started to think about what we knew to work out what the influences were. And it's really not encouraging.

I've worked through many release cycles and, while they all have many differences, certain things recur often enough to suggest there are some lessons we can learn and keep an eye out for.

The first is that most estimates are almost certainly wrong. This is not just me being cynical – there are many reasons for errors to be made (and usually in the undesired direction of being unduly optimistic). An obvious one is that estimates tend to be made at the start of a project, as this information is needed to decide whether to approve a project, estimate its likely budget, and plan for coordinated activities such as marketing efforts. Furthermore, software development is in many ways a learning activity – collecting requirements, and finding out how to turn them into a working system – and so by definition you don't know what you'll actually find out, and so cannot reliably estimate how long it will take! Taking previous experiences into account can help a lot, but unless your projects are similar to ones you've done before, the uncertainties remain high. You could ask the 'experts', but there can be conflicts of interest (eg they're the ones proposing a costly development so will tend to be overly optimistic), and senior developers are often promoted into management roles leading them to have less knowledge of how a system actually works in practice than a programmer working on it every day. So is there a better way of estimating taking into account as much information as possible?

I recently got around to reading the whole of *The Wisdom Of Crowds* [Surowiecki], which suggested to me an interesting approach to this sort of complex estimation. Instead of asking an expert for an opinion, ask everybody and aggregate the answers. The reasoning behind this is that no one can know everything, but everyone will know something about the system. By combining the guesses, say by averaging the project time estimates, the idea is that you'll capture what people know, but the errors will tend to cancel out (this is due to the Central Limit Theorem [CLT], with the caveats about under which circumstances it holds). Note that this is by no means an excuse to avoid taking responsibility for the final estimate! But I think it could be a useful exercise to find out what the group as a whole expects, and not just the 'experts'. You have to ask a wide group of people because, in order for this to work, you need a diverse and independent group to sample their opinion – just asking one subset fails to capture other opinions, and people must be asked privately to avoid the danger of people being influenced by the others. In particular, it should include a wide range of people who wouldn't normally be involved in estimation exercises, such as testers, technical writers doing documentation, and the groups who polish off the product and make it ready for release. This is because it's often these people slogging through the bug list and getting reports from users who really understand how much effort goes into those final stages, often more than the architects and developers. This is often because while such developers might design and write the framework and the bulk of the functionality they will often have moved on to design and write the next project, leaving others to finish off the release even though such efforts can take a similar amount of time again.

I'm reminded of an old project where I was involved in maintaining and releasing several versions of a product while the next major rework was being designed and developed. Having had to evolve the previous code to reflect what was actually required, I had a good gut feel of how much complexity was actually present and how much effort was needed to take a functionally complete project to production quality. My estimate of how long it would take was three times longer than the value which had been used to okay the project. I take no pleasure to note that I turned out to be optimistic. This isn't even an isolated example – it's happened severely to projects taking tens of man years at least twice in my personal experience, and to some extent on most projects. While a large company can probably take the hit, for small start-ups it can be devastating to their balance sheets and customer trust.

Another problem is that quite often the cost of the later stages depend on how well things have gone before, what utilities people have put together, and how stable some key code is. This can't be known up front and project plans and estimates should take this into account by being more vague and conservative the further in the future things are. This implies that detailed planning and estimates happen on a rolling basis, and shorter release cycles are encouraged. In other words big over-detailed plans are discouraged, and short agile cycles are the norm which allow plans to respond to circumstances in a much more flexible fashion.

Once the initial estimate has been made and the project starts, it is important to keep track of progress and update the estimated completion date. The usual use for this is to spot potential problems causing overruns as early as possible, but it also gives a sense of progress to the team which can be a vital tool in keeping morale up and momentum going. An interesting question is how detailed should you measure the progress. I often think that people tend to be unrealistically precise, such as estimating individual tasks and measuring progress in terms of hours, often encouraged by the project planning tools. I've had good results by only breaking tasks down to the granularity of a day, or even a week, and only measure in terms of Not Done/In Progress/Done, as that allows for some flexibility and doesn't over-burden people with tiresome paperwork and endless Gantt chart updating. Another good trick was to get three estimates instead of one – as well as the normal 'How long will this task

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.

take', you also ask for a best and worst case estimate. By taking a weighted average (with ratios such as 1:4:1, which roughly follows a normal distribution) you get the expected time that tends to be a bit longer than the most likely estimate. This is because if things go well they'll be a bit quicker, but if they go badly they take a lot longer. For example, I might reckon it'll probably take 10 days, but could be as short as 8, or as long as 18. My expected time is (8 + 4x10 + 18)/6 which is 11 days. That means that if you use your original estimate only, every two weeks you will most likely fall behind by a day. Another thing this exercise gives you is a clue of how well understood the development is – very large estimates, or a wide spread are a sign that there is a lot of unknown risk, and that area should be investigated some more. And returning to the wisdom of crowds, perhaps you should get several people to estimate and combine.

A picture's worth a thousand words

After a while you've slogged through the feature list and the project is complete. Not so fast! It still has to be polished off ready for release. This is the point where I start to look at graphs of bug numbers. Hopefully you'll have already been using a bug tracking system to capture found defects and schedule them to be fixed (yet another source of project slippage – people say a task is complete and so it is closed. But bugs are sure to be found, and fixing them now takes place during time that was expected to be used for some other task, so making that late). During the main development I try not to worry too much about the total number and its fluctuations, as they tend to be dominated by one-off factors – for example, to start with not many people will be actively using a fast changing early system so only major bugs are reported. When functionality has settled, more people test it in detail and start reporting smaller UI glitches, and your bug count will go through the roof. But eventually you'll run out of new areas to report on and things will stabilise, and then it's worth getting your tools to investigate the trends. Recently I've been using the absolute number, and the number open that are assigned to the current sprint. I've thought about using a weighted total according to how difficult the bugs are thought to be (or using the Story Points used in agile planning), but I worry that the overhead of keeping such information correct could make the results unreliable. I'd be interested to hear if anyone does something like this though.

Then you have to interpret the graphs. This is going to depend a lot on your local situation as everyone has different patterns of bug reporting and speed of fixing issues. But I think there are some common things to look out for. Ideally your sprint graphs should look nicely triangular with a fairly steady slope down to zero at the target. If you repeatedly miss then it could be a sign to adjust how much to put into each sprint. The total is a bit trickier. I've found that to start with it will lurch up as people suddenly test a new area with lots of bugs, and down when people fix a lot of simple small tweaks. Apart from this chaotic churn overall numbers tend not to change very much, as test/fix resources are applied or eased off accordingly. Then what you want to look for is The Corner, where the lurching has died down and a solid downward trend appears. What's happened is that despite a continued test effort it's proving hard to find anything new, and yet the bug fix rate has continued. This is good news. Once that trend has settled in, you can look at the slope and estimate when it will reach zero (although remember that the last few bugs will tend to be the difficult ones, so the slope will level off a bit at the bottom until they're fixed or you decide they are not release-stopping bugs). Congratulations! You can now say with some confidence what the release date will be. Unfortunately it will be in the near future so your oracular powers of prediction won't be as appreciated as highly as you'd like.

Professionalism vs profession

And finally, I found an interesting take on an old problem – what does being 'professional' mean, and should there be a formal body to enforce standards? In computing there are certain bodies who have been given a charter to grant such a professional status, but many companies don't insist on it, trusting on people to be professional in their dealings rather than being part of a formal professional body. Well, it turns out that Canadian engineers have been doing both. There is the usual professional body that grants Professional Engineer status, but individuals can also go through The Ritual Of The Calling Of An Engineer [Ritual] (created by Rudyard Kipling no less), where they are presented with an Iron Ring to wear on the little finger [Ring] to remind them of the responsibility and humility of their professional dealings. This ritual is a more recent version of ones such as the Hippocratic Oath [Hippocrates] which are intended to impress the serious nature of the calling, and to establish a basis for ethical standards. I thought it was a wonderful idea, very much in keeping with what we as a industry ought to aspire to.

References

[CLT] http://en.wikipedia.org/wiki/Central_limit_theorem

[Hippocrates] http://en.wikipedia.org/wiki/Hippocratic_Oath

[Ring] http://en.wikipedia.org/wiki/Iron_Ring

[Ritual] http://www.ironring.ca/

[Surowiecki] ISBN 0385721706 http://en.wikipedia.org/wiki/The_Wisdom_of_Crowds

You're Going To Have To Think!

Numerical computing has many pitfalls. Richard Harris starts looking for a silver bullet.

The dragon of numerical error is not often roused from his slumber, but if incautiously approached he will occasionally inflict catastrophic damage upon the unwary programmer's calculations.

So much so that some programmers, having chanced upon him in the forests of IEEE 754 floating point arithmetic, advise their fellows against travelling in that fair land.

In this series of articles we shall explore the world of numerical computing, contrasting floating point arithmetic with some of the techniques that have been proposed as safer replacements for it. We shall learn that the dragon's territory is far reaching indeed and that in general we must tread carefully if we fear his devastating attention.

On the classification of numbers

As programmers we are probably aware that integers and floating point numbers have different properties, even if we haven't spent a great deal of time thinking about their precise nature.

However, I rather suspect that we are somewhat less aware of how they fit into the general mathematical classification of number types.

Therefore, before we start looking at the various techniques for representing numbers with computers I should like to explore what exactly it is we mean by *Number*.

The concept of Number has been refined throughout history as generations of mathematicians have time and again stumbled across inconsistencies in their understanding.

A hierarchy of number types as we currently understand them is provided in figure 1.

Traversing this tree from left to right, we more or less recover the sequence of development in our concept of Number from prehistory to the modern day.

The story of Number begins with the integers, or more accurately the natural numbers; those whole numbers greater than zero. Animal studies have shown that primates, rats and even some birds have a rudimentary ability to count; presumably using neural circuitry similar to that we use to distinguish at a glance between three and four objects, but not between nineteen and twenty. It is not unreasonable, therefore, to suppose that awareness of the natural numbers predates man.

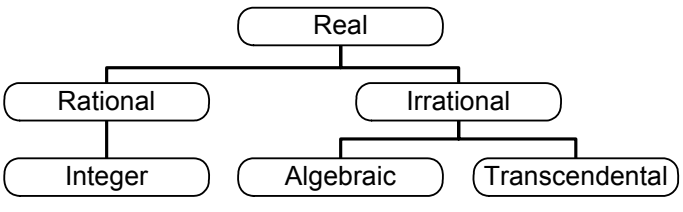


Figure 1

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.

The negative numbers are, comparatively, an example of striking modernity having been discovered in India just a few millennia ago.

The integers, as important as they are, are not particularly useful for measurement; the distance between the ziggurat and the brothel is never quite a whole number of cubits, for example. For this task we instead employed the fractions, or rationals; those numbers equal to the ratio of two integers. Note that the rationals are a superset of the integers; every integer is trivially the ratio of itself and 1.

For many years it was thought that the rationals comprised the totality of Number. Legend has it that a member of the school of Pythagoras discovered that the square root of 2 could not be expressed as a fraction and that his compatriots were so put out by this fact that they drowned him (we shall revisit this in a later article).

The algebraic irrationals are those numbers which are roots of polynomial equations with rational, or equivalently integer, coefficients. By roots we mean those real values, if any, at which the polynomial equates to zero.

The square root of 2 is a root of the polynomial x^2-2, for example. Technically, the algebraic numbers are a superset of the rationals since the latter are solutions to linear equations with integer coefficients.

The final breed of numbers, the transcendentals, is the most elusive. These are the numbers which are *not* solutions of polynomial equations with rational coefficients and include such notable numbers as π and e. They are so difficult to identify that it is still not known whether the sum of π and e is itself transcendental. Despite this, it is known that the transcendentals form the vast majority of numbers; if you were to throw a dart at a line representing the numbers between 0 and 1, you would almost certainly hit a transcendental.

To understand why, we need to discuss the mathematics of infinite sets.

Transfinite cardinals

In the late 19th century Georg Cantor perfected the theory of infinite sets. The transfinite cardinals are not, as their name suggests, characters in a Catholic science fiction blockbuster, but are in fact those infinite numbers that denote the size of infinite sets.

Cantor identified the smallest of the transfinite cardinals, the size of the integers, as $\aleph_0$. This is known as the countable infinity since we can imagine an algorithm that, given infinite time, would step through them sequentially, counting them off one at a time.

He then asked the question of whether the rationals were larger than the integers; whether they were uncountable. His proof that they were not is one of the most elegant in all of mathematics.

When we say a set is countable, we strictly mean that it can be put into a one to one correspondence with the non-negative integers. For example, the integers are countable since we can map from the non-negative integers to them with the rules

the reals, and consequently the transcendentals, are **uncountably infinite**

	1	2	3	4	5	…
1	1/1	2/1	3/1	4/1	5/1	…
2	1/2	2/2	3/2	4/2	5/2	…
3	1/3	2/3	3/3	4/3	5/3	…
4	1/4	2/4	3/4	4/4	5/4	…
5	1/5	2/5	3/5	4/5	5/5	…
…	…	…	…	…	…	…

Figure 2

$$0.x_{00}\, x_{01}\, x_{02}\, x_{03}\, x_{04}\, x_{05}\cdots$$
$$0.x_{10}\, x_{11}\, x_{12}\, x_{13}\, x_{14}\, x_{15}\cdots$$
$$0.x_{20}\, x_{21}\, x_{22}\, x_{23}\, x_{24}\, x_{25}\cdots$$
$$0.x_{30}\, x_{31}\, x_{32}\, x_{33}\, x_{34}\, x_{35}\cdots$$
$$0.x_{40}\, x_{41}\, x_{42}\, x_{43}\, x_{44}\, x_{45}\cdots$$
$$0.x_{50}\, x_{51}\, x_{52}\, x_{53}\, x_{54}\, x_{55}\cdots$$

Figure 3

1. if n is even, n maps to $\frac{1}{2}n$
2. if n is odd, n maps to $-\frac{1}{2}(n+1)$

Enumerating this sequence yields

0, -1, 1, -2, 2, -3, 3, …

which clearly counts through the integers, one at a time.

Cantor laid out the rationals such that the numerator (the number on top of the fraction) was indicated by the column and the denominator (the number on the bottom of the fraction) was indicated by the row, as shown in figure 2.

What Cantor realised was that, whilst each row and column stretched on forever and so couldn't be counted one after the other, the diagonals between the first column of a given row and the first row of the corresponding column were all finite and hence countable. For example, we could iterate over the first row, counting diagonally backwards through the table until we hit the first column yielding the sequence

1/1, 2/1, 1/2, 3/1, 2/2, 1/3, 4/1, 3/2, 2/3, 1/4, …

If we skip any number we have seen before, we have the sequence

1, 2, 1/2, 3, 1/3, 4, 3/2, 2/3, 1/4, …

So, rather surprisingly, despite there being an infinite number of fractions between any two different integers, the sizes of the set of fractions and the set of integers are in fact equal.

Cantor proceeded to demonstrate that the set of polynomial equations with integer coefficients is also countable and, since each has a finite number of roots, so are the algebraic numbers.

He did this by defining a function, we shall call it c, that takes a polynomial with integer coefficients and returns a positive integer. It operates by adding together the absolute values of the coefficients and the largest power to which the variable is raised, the order of the polynomial, minus one.

For example

$$c(2x^2 - 3x + 4) = 2 + 3 + 4 + (2 - 1) = 10$$

Note that we can insist that the term with the highest order is positive, since multiplying a polynomial by minus one doesn't affect its roots.

Cantor realised that every possible value of this function is shared by a *finite* number of such polynomials. For example, there are just 4 such polynomials for which this function yields 2.

$$2x$$
$$x + 1$$
$$x - 1$$
$$x^2$$

So we can count off these polynomials by counting through the positive integers and, for each of them in turn, enumerating the members of the finite set of them for which Cantor's function returns that value.

We are left with the question of whether or not the transcendental numbers are of the same size.

If the transcendental numbers are countable then the real numbers, being the union of both they and the algebraic numbers, must be countable too since we could simply alternate between the sequences of each of them.

Cantor noted that if the reals were countable we could construct a list of them as they are generated by the mapping from the integers. Figure 3 illustrates what this list might look like for the numbers between 0 and 1.

Now starting after the decimal point in the first row and moving diagonally down and to the right we can construct a new number

$$0.\,(x_{00}+2)\%10\ (x_{11}+2)\%10\ (x_{22}+2)\%10\ (x_{33}+2)\%10\ (x_{44}+2)\%10\ \ldots$$

This number is clearly between 0 and 1, but must differ from every number in the list at no less than one digit. Note that we add 2 to each digit rather than 1 to avoid the irritating corner case of recurring nines, such as 0.099999... being equal to 0.1.

We have thus found a number between 0 and 1 that was not in our list and hence the list is incomplete. It is not, therefore, possible to construct such a list and hence the reals, and consequently the transcendentals, are uncountably infinite. Being more sizable than the other numbers, their cardinal number is denoted by $\aleph_1$.

Double precision floating point numbers have precisely the same layout as single precision floating point numbers

IEEE 754-1985

So now we know the mathematical classification of numbers we are ready to start looking at how we might implement numeric types with computers.

The IEEE standard [IEEE] defines floating point numbers to have a format similar to the scientific notation many of us will recognise from our calculators and spreadsheets. In the familiar decimal base 10 this means a number between 1 and 10 multiplied by 10 raised to the power of another number.

For example, the number of days in a year is approximately 365.

Dividing by this 100 gives us a number between 1 and 10, namely 3.65.

Since 100 is 10 raised to the power of 2, the number of days in the year can be written as 3.65×10^2, or commonly 3.65E2.

The number that we multiply by the power of 10 is known as the mantissa and the power of 10 by which we multiply it is known as the exponent.

Since base 10 is rather inconvenient from a computing perspective, IEEE floating point numbers are defined in the binary base 2. Specifically, numbers are defined as $\pm b \times 2a$ with, in the single precision format, the sign taking one bit, the exponent a taking 8 bits and the mantissa b taking 24. Much as in decimal the mantissa must lie between 1 and 10, so in binary it must lie between 1 and 2. The leading digit must therefore be 1, and we can represent b with 23 bits rather than the full 24.

There is, in fact, a special case when we assume the leading digit is 0 rather than 1. This occurs when the exponent takes on its most negative value, yielding the very smallest floating point numbers. Since the leading digits of these numbers, known as subnormal or denormalised numbers, may be 0 there may consequently be fewer bits left to represent the mantissa resulting in fewer significant digits of accuracy, or equivalently in lower precision. In contrast recall that normal numbers have an implied leading digit of 1 and consequently have the full 24 bits with which to represent the mantissa.

In addition to the normal and subnormal numbers, IEEE 754 defines bit patterns to represent the positive and negative infinities and a set of error values for invalid calculations known as the NaNs, for Not a Number. Many of us are probably aware of the quiet and signalling NaNs identified by `std::numeric_limits`, but perhaps not of the fact that there are actually 2^{24}-2 of them in the single precision format, allowing for error codes to be embedded in invalid results.

Figure 4 enumerates the full set of IEEE 754 single precision floating point numbers, $\pm a_1a_2a_3\ldots a_8b_1b_2b_3\ldots b_{23}$

Note that since the mantissa is finite, floating point numbers are actually a finite subset of the rational numbers and it is vitally important not to confuse them with real numbers.

Double precision

Double precision floating point numbers have precisely the same layout as single precision floating point numbers, differing only in that they have an 11 bit exponent and a 53 bit mantissa. Recall that one of the bits in the mantissa is implied, so that these and the sign bit fill 64 bits.

Binary Exponent ($a_1\ldots a_8$)	Decimal Exponent	Binary Value
00000000	0	$\pm 0.b_1b_2b_3\ldots b_{23}\cdot 2^{-126}$
00000001	1	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{-126}$
00000010	2	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{-125}$
00000011	3	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{-124}$
…	…	…
01111111	127	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{0}$
10000000	128	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{1}$
…	…	…
11111100	252	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{125}$
11111101	253	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{126}$
11111110	254	$\pm 1.b_1b_2b_3\ldots b_{23}\cdot 2^{127}$
11111111	255	$\pm\infty$ if b_f=0 NaN otherwise

Figure 4

Henceforth, we shall assume that the double precision format is being used.

Now that we have covered the mundane implementation details of floating point numbers it is time to start looking at the rather more important topic of their precise behaviour.

Not a number

The NaNs infect any calculation they come into contact with since the result of any operation upon a NaN yields a NaN.

Furthermore, any comparison involving a NaN is always false, even an equality comparison between two NaNs.

If you keep this in mind when designing loops and branches, you can ensure that your algorithms will behave predictably in the face of invalid arithmetic operations.

Overflow

Floating point numbers overflow in a satisfyingly predictable way, namely to plus or minus infinity.

Dividing any finite number by an infinity will yield zero and dividing any non-zero number by zero will yield an infinity of the same sign as that number. Adding or subtracting any finite number to or from an infinity will result in that infinity.

we should prefer to test whether two floating point numbers are similar to each other rather than the same

These properties mean that many numerical algorithms can implicitly cope with numerical overflow since arithmetic operations and comparisons are internally consistent and, accompanied by some vigorous hand waving, mathematically sound.

Note that dividing 0 by 0, dividing an infinity by an infinity, multiplying an infinity by 0 and subtracting an infinity from itself all yield NaNs.

Rounding error

One of the most common surprises facing the programmer using floating point arithmetic stems from the fact that there are a fixed number of bits with which to represent the mantissa.

We can illustrate the problem by considering decimal notation. Say we restrict ourselves to 4 figures after the decimal point. Assuming that we have chosen the closest number in this representation, x, to a given number we can only say that its true value lies somewhere within $x \pm 5E10^{-5}$. For example, given π to 4 decimal places, 3.1416, we can only state with certainty that it lies between 3.14155 and 3.14165.

Similarly, for an IEEE double precision floating point number x with an exponent of a, we can only be sure that the true value is between $x \pm 2^{a-53}$. Conveniently, since normalised floating point numbers have a implicit leading digit of 1, these bounds can be written as $x(1 \pm 2^{-53})$ or, conventionally, as $x(1 \pm \frac{1}{2}\varepsilon)$.

Of course, this means that operations on denormalised numbers will introduce proportionally even greater errors but we shall ignore this fact in our analyses and effectively treat them as if they behave in the same fashion as zero.

If an algorithm really must treat denormalised numbers with the same respect as normalised numbers, it will require much more careful analysis.

The mathematical operations of addition, subtraction, multiplication, division, remainder and square root are required by the IEEE standard to be accurate to within rounding error. Specifically, they must return the correctly rounded representation of the result of performing the actual calculation with *real* numbers. This means that, if using round to nearest, they will introduce a proportional error no larger than $(1 \pm \frac{1}{2}\varepsilon 0)$.

Note that because of these accumulated rounding errors, equality comparisons between floating point numbers often behave counter-intuitively; values of unlike expressions that should mathematically be equal may have accumulated slightly different rounding errors.

In general, we should prefer to test whether two floating point numbers are similar to each other rather than the same.

Condition number

It is important to note that the rounding guarantees of the IEEE arithmetic operations do not take into account any rounding error in their arguments.

We can capture the sensitivity of the result of a function f to rounding errors in its argument x with the condition number, given by

Given a real value x and the nearest normal floating point x^* we have

$$\left|\frac{x - x^*}{x}\right| \leq \varepsilon$$

The relative error in f is given by

$$\left|\frac{f(x) - f(x^*)}{f(x)}\right|$$

Dividing by the relative error in x, we have

$$\begin{aligned}
\left|\frac{f(x) - f(x^*)}{f(x)}\right| \Big/ \left|\frac{x - x^*}{x}\right| &= \left|\frac{f(x) - f(x^*)}{f(x)} \Big/ \frac{x - x^*}{x}\right| \\
&= \left|\frac{f(x) - f(x^*)}{f(x)} \times \frac{x}{x - x^*}\right| \\
&= \left|\frac{f(x) - f(x^*)}{x - x^*}\right| \times \left|\frac{x}{f(x)}\right| \\
&= |f'(x)| \times \left|\frac{x}{f(x)}\right| \\
&= \left|\frac{f'(x)}{f(x)} \times x\right|
\end{aligned}$$

Derivation 1

$$\kappa_f(x) = \left|\frac{f'(x)}{f(x)} \times x\right|$$

where f' is the derivative of f and the vertical bars mean the absolute value of the expression between them.

This value is approximately equal to the absolute value of the ratio between the relative error of $f(x)$ and the relative error of x, as shown in derivation 1. Note that it assumes that f can be calculated exactly and so the condition number does not take into account rounding during the calculation or of the result.

As an example, consider the exponential function e^x whose derivative is equal to e^x for all x. Its condition number is therefore $|x|$, meaning that its relative error at x before rounding is approximately equal to $|\frac{1}{2}\varepsilon x|$.

When the condition number is large, a calculation is said to be poorly conditioned and we cannot trust that it is accurate to many digits of precision.

Noting that the number of digits of precision is approximately equal to the logarithm of the reciprocal of the absolute relative error, we can use the condition number to estimate the number of decimal digits of precision of a calculation.

Specifically, we use

$$-\log_{10}\left(\frac{1}{2}\varepsilon\right) - \log_{10}(\kappa_f(x))$$

Whilst rounding error might sneak up upon us in the end, cancellation error is liable to beat us about the head with a length of two by four

Assuming that the floating point epsilon has n decimal leading zeros, for a given real number and its closest normal floating point number we have

$$\varepsilon \leq 10^{-n}$$

$$x = b \times 10^{a}$$

$$x^* \in \left(b \pm \frac{1}{2}\varepsilon\right) \times 10^{a}$$

where b is between 1 and 10.

Now

$$\log_{10}\left(\left|\frac{x}{x - x^*}\right|\right) = \log_{10}(|x|) - \log_{10}(|x - x^*|)$$

$$\geq \log_{10}(|b|) + a - \log_{10}\left(\frac{1}{2}\varepsilon\right) - a$$

$$= -\log_{10}\left(\frac{1}{2}\varepsilon\right) + \log_{10}(|b|)$$

$$\approx n$$

Defining the absolute relative error in the result of a function f as

$$\varepsilon_f(x) = \left|\frac{f(x) - f(x^*)}{f(x)}\right| \approx \kappa_f(x) \times \frac{1}{2}\varepsilon$$

we have

$$\frac{1}{\varepsilon_f(x)} = \frac{1}{\kappa_f(x) \times \frac{1}{2}\varepsilon}$$

and hence

$$\log_{10}\left(\frac{1}{\varepsilon_f(x)}\right) = \log_{10}\left(\frac{1}{\kappa_f(x) \times \varepsilon}\right)$$

$$= -\log_{10}\left(\kappa_f(x) \times \frac{1}{2}\varepsilon\right)$$

$$= -\log_{10}\left(\frac{1}{2}\varepsilon\right) - \log_{10}(\kappa_f(x))$$

Derivation 2

where $\log_{10}$ is the base 10 logarithm, as demonstrated in derivation 2. This is equivalent to subtracting the log of the condition number from the number of digits of precision of the floating point type.

Cancellation error

Given enough operations or poorly conditioned functions, rounding error can significantly affect the result of a calculation, but it is by no means the worst of our troubles.

Far more worrying is cancellation error which can yield catastrophic loss of precision. When we subtract two almost equal numbers we set the most significant digits to zero, leaving ourselves with just the insignificant, and most erroneous, digits.

For example, suppose that we have two values close to π, 3.1415 and 3.1416. These values are both accurate to 5 significant figures, but their difference is equal to 0.0001, or 1.0E-4, and has just 1 significant figure of accuracy.

Whilst rounding error might sneak up upon us in the end, cancellation error is liable to beat us about the head with a length of two by four.

The poster child of cancellation error is the approximation of numerical differentiation with the forward finite difference. The derivative of a function f at a point x is defined as the limit, if one exists, of

$$\frac{f(x + \delta) - f(x)}{\delta}$$

as δ tends to zero.

The forward finite difference replaces the limit with a very small, but non-zero, δ and is a reasonably obvious way to approximate the derivative.

It is equally obvious that we should choose δ to be as small as possible, right?

Wrong!

To demonstrate why not, consider the function e^x whose derivative at 1 is trivially equal to e. Figure 5 plots a graph of minus the base 2 logarithm of the absolute error in the approximate derivative at 1, roughly equal to the number of correct bits, against minus the base 2 logarithm of δ, equal to the number of leading zeros in its binary representation.

Clearly, decreasing δ works up to a point as indicated by an initial linear relationship between the number of leading zeros and the number of accurate bits. However, this relationship seems to break down beyond δ equal to 2^{-25} and the best accuracy occurs when δ is equal to 2^{-26}.

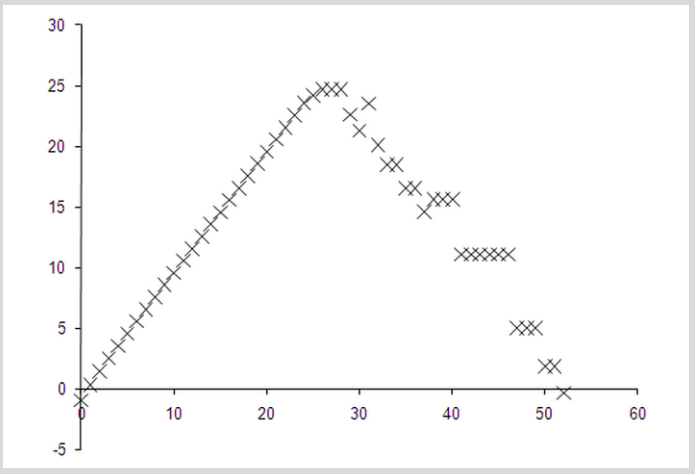


Figure 5

From the Taylor series expansion of f we have

$$f(x+\delta) = f(x) + \delta f'(x) + O(\delta^2)$$

From this we can obtain the result of the approximate derivative

$$\frac{f(x+\delta)-f(x)}{\delta} = \frac{\delta f'(x) + O(\delta^2)}{\delta}$$
$$= f'(x) + O(\delta)$$

Assuming that we can exactly represent both x and $x+\delta$ and that f is accurate to machine precision, the floating point result of this formula will be

$$\frac{\left(f(x+\delta)\times\left(1\pm\frac{1}{2}\varepsilon\right) - f(x)\times\left(1\pm\frac{1}{2}\varepsilon\right)\right)\times\left(1\pm\frac{1}{2}\varepsilon\right)}{\delta}\times\left(1\pm\frac{1}{2}\varepsilon\right)$$

which is equal to

$$\frac{f(x+\delta)\times\left(1\pm\frac{1}{2}\varepsilon\right) - f(x)\times\left(1\pm\frac{1}{2}\varepsilon\right)}{\delta}\times\left(1\pm\frac{1}{2}\varepsilon\right)^2$$

$$= \frac{(f(x)+\delta f'(x)+O(\delta^2))\times\left(1\pm\frac{1}{2}\varepsilon\right) - f(x)\times\left(1\pm\frac{1}{2}\varepsilon\right)}{\delta}\times\left(1\pm\frac{1}{2}\varepsilon\right)^2$$

$$= \frac{\pm\varepsilon f(x) + \delta f'(x) \pm \frac{1}{2}\varepsilon\delta f'(x) + O(\delta^2)}{\delta}\times\left(1\pm\frac{1}{2}\varepsilon\right)^2$$

$$= \frac{\delta f'(x) + O(\varepsilon) + O(\delta\varepsilon) + O(\delta^2)}{\delta}\times\left(1\pm\frac{1}{2}\varepsilon\right)^2$$

$$= \left(f'(x) + O(\delta) + O\left(\frac{\varepsilon}{\delta}\right) + O(\varepsilon)\right)\times\left(1\pm\frac{1}{2}\varepsilon\right)^2$$

$$= f'(x) + O(\delta) + O\left(\frac{\varepsilon}{\delta}\right)$$

Hence, if δ is too large the $O(\delta)$ term will introduce significant errors into the approximation, whereas if it is too small the $O(\varepsilon/\delta)$ will do so instead.

With some vigorous hand waving, we can ignore the constant factors in these terms, and conclude that since a choice of $\delta = \varepsilon^{1/2}$ results in them both having the same order of magnitude it is, in some sense, optimal.

Derivation 3

This is suspiciously close to half of the number of bits in the mantissa of a double precision floating point number. In fact it can be proven, under some simplifying assumptions, that the optimal choice for δ is the square root of ε, which has roughly that many leading zeros.

The reason for this behaviour is that, as δ gets very small, the results of the two calls to f get very close together and their difference introduces a large cancellation error, as shown formally in derivation 3.

Note that since cancellation error results from the dramatic sensitivity of the subtraction of nearly equal numbers to the rounding errors in those numbers, it can be captured by the condition number.

For example, the expression x-1 has a condition number of $|x/(x\text{-}1)|$. As x tends to 1, the condition number tends to infinity, reflecting the growing effect of cancellation error on the result.

Order of execution

The final surprising aspect of floating point numbers is that the exact order in which operations are performed can have a material effect on the result.

For example, suppose that we wish to calculate the cube of the square root of a number, or equivalently the square root of the cube. Starting out with a value x accurate to within rounding, we have

$$x\left(1\pm\frac{1}{2}\varepsilon\right)$$

Next, we take the square root, introducing another rounding error

$$\left(x\left(1\pm\frac{1}{2}\varepsilon\right)\right)^{\frac{1}{2}}\left(1\pm\frac{1}{2}\varepsilon\right) = x^{\frac{1}{3}}\left(1\pm\frac{1}{2}\varepsilon\right)^{\frac{3}{2}}$$

Finally we multiply it by itself twice to recover the cube, introducing two more rounding errors

$$\left(x^{\frac{1}{2}}\left(1\pm\frac{1}{2}\varepsilon\right)^{\frac{3}{2}}\right)^3\left(1\pm\frac{1}{2}\varepsilon\right)^2 = x^{\frac{3}{2}}\left(1\pm\frac{1}{2}\varepsilon\right)^{\frac{13}{2}} \approx x^{\frac{3}{2}}\left(1\pm 3\frac{1}{4}\varepsilon\right)$$

Now let's try it in the reverse order. This time we perform the two multiplications first yielding

$$x^3\left(1\pm\frac{1}{2}\varepsilon\right)^3\left(1\pm\frac{1}{2}\varepsilon\right)^2 = x^3\left(1\pm\frac{1}{2}\varepsilon\right)^5$$

Secondly we take the square root, introducing one additional rounding error

$$\left(x^3\left(1\pm\frac{1}{2}\varepsilon\right)^5\right)^{\frac{1}{2}}\left(1\pm\frac{1}{2}\varepsilon\right) = x^{\frac{3}{2}}\left(1\pm\frac{1}{2}\varepsilon\right)^{\frac{7}{2}} \approx x^{\frac{3}{2}}\left(1\pm 1\frac{3}{4}\varepsilon\right)$$

Surprisingly, this second version of the calculation has accumulated just a little more than half the error that the first had.

Whilst this is a relatively simple example, the fundamental lesson is sound; in order to control errors when using floating point numbers we must plan our calculations with care.

You're going to have to think!

I recently read a comment on a prominent IT internet forum proposing that scientists should not be trusted to implement their own computer models since, presumably unlike the comment's author, they are not trained in computer science and are consequently likely to make mistakes. The given example of such a mistake was the calculation of the average of 20 or so values in the low 20s[1] which was ironic, since a computer scientist should be able to demonstrate that the result of performing this calculation with double precision floating point is correct to about 15 decimal digits of precision!

Such unfair criticism of floating point is not particularly unusual, is often unduly concerned with rounding error and hardly ever mentions the vastly more important topic of cancellation error. One can only assume that many computer science graduates have forgotten their numerical computing lectures and have generalised the very specific and predictable failure modes of floating point arithmetic to the rule of thumb that any use of floating point renders a program broken by design.

These criticisms are generally accompanied by suggestions of alternative arithmetic types that fix the perceived problems with floating point. We shall investigate these in coming articles in this series and we shall learn that if we wish to use computers for arithmetic calculation we shall have to accept the fact that we are going to have to think. ■

References and further reading

[IEEE] *IEEE standard for binary floating point arithmetic*. ANSI/IEEE std 754-1985, 1985.

1.Which, given recent events, should be a fairly big hint as to which scientists were being so criticised.

What motivates software developers: a workshop report

Keeping your team happy should be more than guesswork. Helen Sharp went and asked you.

This article reports the results of a workshop held at ACCU 2009 looking at the question: What motivates software developers? It was the third in a series of workshops [Sharp09, Sharp07] that build on previous research in the field of motivation in software engineering [Hall08] that will form the basis of a larger, more in-depth study into current-day software practice. Software practitioner motivation has been recognised as a key factor in system quality, yet much of what we know about practitioner motivation is based on research conducted in the 1980s and software development has changed considerably since then. Activity in the area is growing again, but the most significant recent studies are focused on the open source community. The workshop was a combination of group discussion and individual reflection; the data collected, and the results obtained should be considered in light of this design.

Motivation in software engineering

Software Development has been an expanding market for over 40 years. It is estimated that the global software market grew by 6.5% in 2008 and is now valued at $303.8 billion [Datamonitor06]. It is also predicted that by 2013 the global software market will be valued at $457 billion [Datamonitor06]. Motivation has been identified as a key factor affecting many important aspects of software development. Such factors include productivity, adherence to budgets, increases in staff retention and reduced absenteeism [Hall08]. The impacts motivation may have on a $300+ billion dollar industry makes the management and identification of key motivational factors crucial for the future improvement of software development and personnel satisfaction.

Several theories of motivation have been proposed and accepted in various contexts, but much work into motivation in software engineering has been based upon just one of these: the Job Characteristics Model [Hackman76]. Cougar and Zawacki tailored this for the software environment [Couger80] and although a significant number of reported studies are based on this modified model, they tend to simply apply the model rather than assess its applicability. In addition, a predominant perspective in motivation research is that of the organisation, focusing on issues such as turnover [Agarwal2000], performance [Darcy05] and absenteeism [Mak01]. Some more recent work focuses on open source software developers (e.g. [Hall08a]), where emphasis on organisational concerns such as turnover and productivity is less important, but this addresses only one area of software development. Only a small number of previous studies identify what is specifically motivating about Software Engineering, and we have found no research focused on understanding the motivation to stay in Software Engineering as a profession [Beecham08]. Previous studies have found that people working in the software industry are motivated by the nature of the job, e.g. change [Burn95, Smits92], technical challenge [Ramachandram06, Tanner03] and problem-solving

Helen Sharp is Professor of Software Engineering in the Department of Computing at the Open University. She has been reseearching the human and social aspects of software practice for many years, and specifically looking at motivation for just over 5 years. She is very active in both the software engineering and interaction design communities and has had a long association with the practitioner-related conferences. She can be contacted at h.c.sharp@open.ac.uk

Motivational factors inherent in software engineering	# of studies reporting this factor
Change	4
Technical challenge	4
Problem Solving	3
Benefit	3
Team Working	2
Science	2
Experiment	2
Development practices	2
Software process/ lifecycle	1

Table 1

Table 1 summarises the motivational aspects of software engineering found in previous studies [Capretz03].

In this paper we present the results of an investigation with experienced software professionals which explored why software practitioners stay in the profession. The next section describes the workshop format, the data collected and the analysis performed, the third section presents the findings, and the fourth section discusses results, limitations and future work.

Data collection and analysis

The data was collected during a workshop at the 2009 ACCU conference. Each workshop attendee was asked to align with one of three role groups (developer, consultant and manager) to align with for the workshop; this design arises from previous research [Beecham08] which indicates that motivation factors vary between different roles. Each individual was given a data collection form containing three sections: the first elicited the individual's background and experience, the second asked questions about why the individual stays in software development and the last asked for three main factors that keep the individual in the software profession. Attendees were asked to complete the first section while waiting for the workshop to begin. The other questions were completed at stages through the workshop as described below. The questions on the form covered:

Section 1

1. What role in software development have you aligned with for today?
2. Are you a practitioner or a researcher or an educator?
3. What is your nationality?
4. How many years experience in software engineering do you have? (if you've had different roles, please list all of them and number of years for each).

The workshop was structured around role group discussion, plenary discussion and individual reflection

5. Please describe briefly the software development projects you've been involved with over the last two years.

Section 2

6. What aspects of your job do you get most satisfaction from?
7. What are the features of a project that make you stay in your job?
8. What factors keep you in software engineering?
9. What makes developing software worthwhile to you?

Section 3

10. Please write down YOUR three most important motivational factors that keep you involved in developing software

The workshop was structured around role group discussion, plenary discussion and individual reflection, as follows. After a brief introduction to the area, role groups were asked to discuss the second section of questions and to record the group's comments and opinions on flip charts. Following this, individuals were asked to write down their own answers on the form. The role groups then exchanged and discussed their responses. Again within role groups, attendees were asked to consider their most important motivational factors – any number of these could be proposed and each one was written on an index card. Then individuals were asked to write down their own three most important factors on the form. Finally, we presented the findings of our systematic literature review [Beecham08] which synthesises previous research work up to 2007.

Responses to questions 6–10 were transcribed into Excel and they were grouped using a simple categorization scheme that emerged from the data itself. The full list of categories is given below:

Big picture	Personal	Problem-solving	Making something
Technical	Variety	Appreciation	Financial
Not software	Habit	Programming	Autonomy
Interesting	Personal	Satisfaction	Enjoyment
Developing	Management	Fear	Challenging
Learning	Creative	People	

Findings

There were 23 usable questionnaire responses ranging over the three roles: developers, technical managers, and consultants. Participants had between 3 and 35 years' experience developing software, with the mean being 15.3 years. The industries within which attendees have worked were quite varied, from embedded systems, petrochemical, insurance, telecoms, government and start-up (to name but a few). Thirteen of the participants declared themselves as British, and the others were a mixture of US, Hungarian, Indian, Norwegian and Dutch.

The following bar charts show the number of unique respondents whose answers fell into each category, i.e. if an attendee had several answers which fell into the same category then they were counted only once, and the number of answers across the sample which fell into each category, i.e. if an attendee had several answers which fell into the same category then they were all counted.

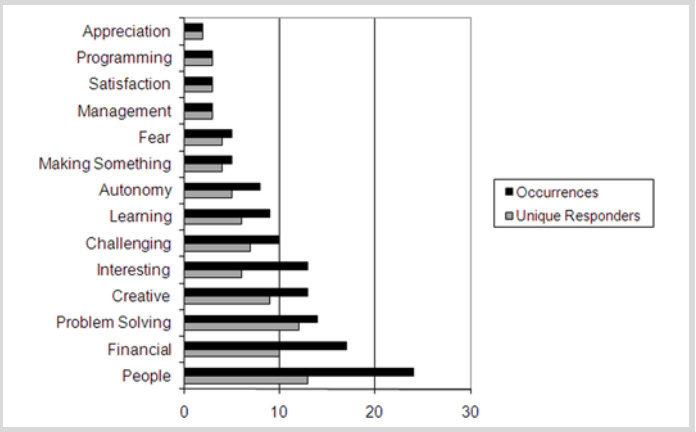


Figure 1

Developers

Fifteen attendees classified themselves as 'developers'. Figure 1 (summarises the responses from these individuals. Example answers from the top category of People include: 'working with bright people'; 'a good team'; 'like-minded colleagues'; and 'talking to others'. Example answers categorized as problem-solving include: 'solving problems other people can't solve'; and 'solving problems – elegant solutions'.

Technical managers

Figure 2 shows the results for technical managers to continue being a software engineering professional. Example answers from the top category of Financial include: 'the money surely helps'; 'it pays well'; and 'money'. Example answers categorized as Challenging include: 'intellectual challenge'; 'the challenges, risky things'; and 'the challenges of customer buy-in'.

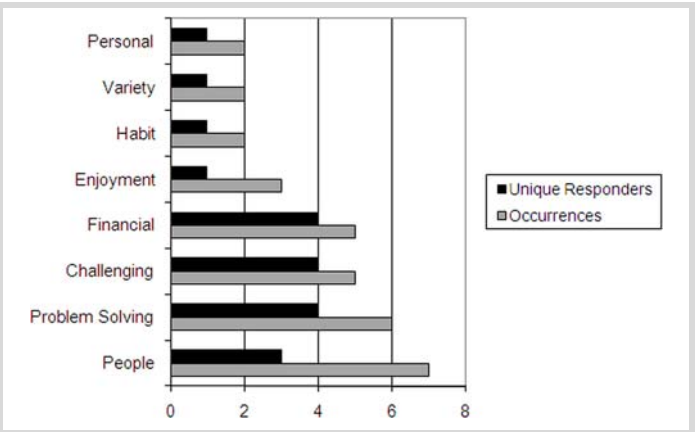


Figure 2

It has long been established that **software engineering focuses on solving problems**

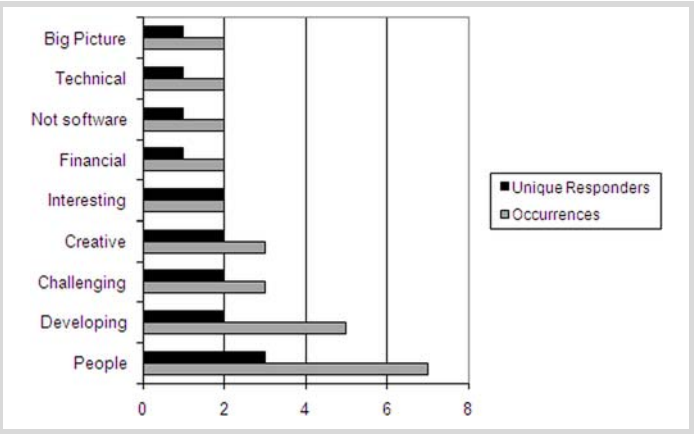


Figure 3

Consultants

Figure 3 shows that there was only consensus between all three consultants on one factor, which was People. These responses are included as a bar chart purely for comparison. The tiny number of respondents would normally mean that such a bar chart would not be produced.

Combined

If we combine all of the categories, then across the sample of 23 practitioners, the results are shown in Figure 4.

Discussion

The data reported here indicates that software engineer practitioners share similar beliefs about what motivates them to continue developing in this sector. The ever-changing nature of software engineering suggests that the results of any study on this sector is likely to vary over time, and will differ from previous and future studies.

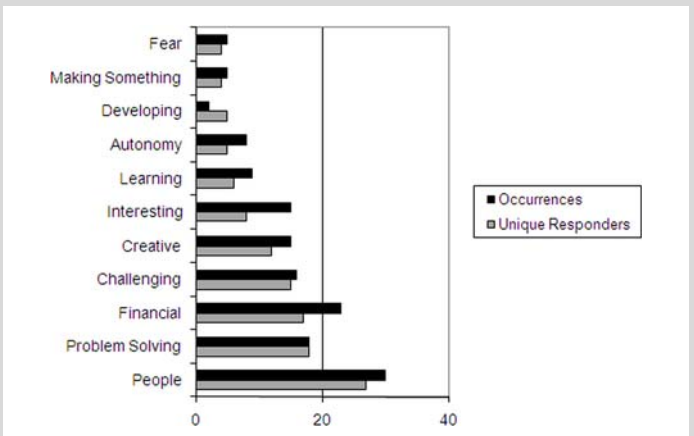


Figure 4

It has long been established that software engineering focuses on solving problems [Tanner03] so the discovery of Problem Solving being a commonly listed factor is expected. According to Hall et al. [Hall08], Challenge and Problem solving have been recognized as motivators for software developers for some time. Hall et al also identified Team work and Benefit (developers create something to benefit others or enhance well-being) as two commonly listed categories. Finding that People is the most commonly listed motivational factor is surprising, as it is not listed in the reviewed literature [Beecham08] although it was identified in the earlier workshops in this series [Sharp07, Sharp09].

Franca & da Silva [Franca09] also found that the factor with the most motivational force was Work with people, followed in 3rd place by Problems resolution. This shows a changing trend when compared to previous work; the inclusion of People as an important and powerful motivational factor is new.

The main implication of this finding for software practitioners is that motivational programmes need to pay more attention to the social environment within which software is developed. The main implication for researchers is to highlight that the social demands of software development are changing.

However this article reports the findings of a small self-selecting group of practitioners (attendees chose to come to this workshop rather than any other session), and is limited in applicability. Future work would aim to involve a larger and more representative sample.

Future work

We aim to conduct further in-depth studies in this area in order to explore what motivates software practitioners in the current software market. We would be very pleased to hear from any practitioner who would be willing to help us pursue this research topic further. Please feel free to contact me at the email address above. ■

Acknowledgements

We would like to thank all our participants for sharing their knowledge and expertise with us. We would also like to thank Mark Dalgarno of Software Acumen for helping to run the ACCU 2009 workshop, and Rien Sach for performing much of the analysis.

References and further reading

[Agarwal2000] Agarwal, R. and T.W. Ferratt, 'Retention and the career motives of IT professionals'. *Proceedings of the ACM SIGCPR Conference*, 2000: p. 158–166.

[Beecham08] Beecham, S., Baddoo, N., Hall, T., Robinson, H. and Sharp, H. (2008) 'Motivation in Software Engineering: A Systematic Literature Review', *Information and Software Technology*, 50, 860–878.

motivational programmes need to pay more attention to the social environment within which software is developed

[Burn95] Burn, J.M., L.C. Ma, and E.M. Ng Tye, Managing IT professionals in a global environment SIGCPR Comput. Pers, 1995. 16(3): p. 11–19.

Bush, C.M. and Schkade, L.L. (1985) 'In search of the perfect programmer', Datamation, 31(6), 128-132.

[Capretz03] Capretz, L (2003) 'Personality types in software engineering', *International Journal of Human-Computer Studies* 58(2), 207–214.

[Couger80] Couger, D.J. and Zawacki, R.A. (1980) *Motivating and Managing Computer Personnel*, John Wiley & Sons.

[Darcy05] Darcy, D.P. and M. Ma (2005) 'Exploring Individual Characteristics and Programming Performance: Implications for Programmer Selection' in *Proceedings of HICSS '05* p. 314a-314a

[Datamonitor06] Datamonitor (2006) 'Software: Global Industry Guide'. Available from: http://www.infoedge.com/product_type.asp?product=DO-4959, accessed 01/02/2010.

[Franca09] Franca, A. and Fabio Q. B. da Silva (2009) 'An empirical study on software engineers motivational factors', in *ESEM 2009*, Lake Buena Vista, FL, USA, pp405–409.

[Hackman76] Hackman, J. R. and Oldham, G. R. (1976). *Motivation through the design of work: test of a theory*. New York, Academic Press.

[Hall08] Hall, T., Sharp, H., Beecham, S., Baddoo, N. and Robinson, H. (2008) 'What Do We Know about Developer Motivation?', *IEEE Software*, 92–94.

[Hall08a] Hall, T., Beecham, S., Verner, J. and Wilson, D. (2008) 'The Impact of Staff Turnover on Software Projects: The Importance of Understanding What Makes Software Practitioners Tick' in *Proceedings of SIGMIS-CPR '08*, 30–39.

Herzberg, F., Mausner, B. and Snyderman, B. B. (1959). *Motivation to Work* (2nd ed.), Wiley, New York.

Lyons, M.L. (1985) 'The DP Psyche', *Datamation*, 31(16) pp103–105, 108, 110.

[Mak01] Mak, B.L. and H. Sockel, 'A confirmatory factor analysis of IS employee motivation and retention'. *Information & Management*, 2001. 38(5): p. 265–276.

[Ramachandram06] Ramachandran, S. and S.V. Rao (2006), 'An effort towards identifying occupational culture among information systems professionals'. *Proceedings of the 2006 ACM SIGMIS CPR conference on computer personnel research: Forty four years of computer personnel research: achievements, challenges & the future*. Claremont, California, USA, p. 198–204

[Sharp07] Sharp, H., Hall, T., Baddoo, N. and Beecham, S. (2007) 'Exploring Motivational Differences between Software Developers and Project Managers' in *Proceedings of European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, September, pp501–504

[Sharp09] Sharp, H. and Hall, T. (2009) 'An initial investigation into software engineers' motivation', in *Proceedings of CHASE 2009*, workshop held at ICSE 2009, Vancouver

Smith, D.C. (1989) 'The personality of the systems analyst: an investigation', *SIGCPR Comput. Pers.* 12(2), 12–14.

[Smits92] Smits, S.J., E.R. McLean, and J.R. Tanner (1992) 'Managing high achieving information systems professionals' in *Proceedings of SIGCPR '92*, Cincinnati, Ohio, April 5–7, A. L. Lederer, Ed, p. 314–327.

[Tanner03] Tanner, F.R. (2003) 'On motivating engineers', in *Engineering Management Conference, IEMC '03. Managing Technologically Driven Organizations: The Human Side of Innovation and Change*, pp 214–218.

To DLL or Not To DLL

Shared libraries provide both benefits and problems. Sergey Ignatchenko introduces a rabbit's-eye view.

Hi all! Let me introduce myself. My name is 'No Bugs' Bunny. I've appeared in two previous issues of *Overload* as a main character in articles about multithreading [Ignatchenko10], and now I've decided to start a writing career of my own. All opinions in my articles are my own, and don't necessarily coincide with opinion of the translator, let alone the editors of the journal. Most of the time, I will be thinking aloud on more or less controversial issues, so please always take my words with a good pinch of salt. I do not aim to tell absolute truths, but rather to raise questions and invite readers to think about their own answers.

Today I will think aloud about a rather contentious DLL issue. Please keep in mind that for the purposes of this article (unless it is specified explicitly) I will use term 'DLL' both for Windows DLLs and for .so libraries in Linux/*nix.

DLL hell

Whenever I need to link a DLL to my application, the very first thing that comes to my mind is 'DLL Hell'. Dependency problems and crashes caused by DLLs are extremely common, and the more installations an application has, the more likely the problems are to appear on some machine.

I will not elaborate on 'DLL Hell' theory, but will provide a few examples from my personal experience. My very first encounter with DLLs was many years ago, when I was a cute little bunny and the very term 'DLL Hell' hadn't even been coined. I had a third-party application which worked perfectly for me for months, and then I installed another application on my system (I think it was electronic dictionary application). Bang! The first application started to crash every time I tried to perform some simple operation. Being a curious little bunny with lots of time to spare, I started to research the problem, and eventually found out that the electronic dictionary I'd just installed had replaced the file `MFC42.DLL` with a 'customized' version; obviously it wasn't 100% compatible and it was the reason for my first application crashes. It was my very first practical lesson about DLLs.

During my career I've seen many applications which had millions of installations, and can tell you that when dealing with DLLs, the famous Murphy's law ('Anything that *can* go wrong, *will* go wrong') is not an exaggeration but an understatement. Not only have I seen when one very specific build of IE5.5 crashed an application which used it merely to show a fancy HTML-based splash-screen (how was QA supposed to test it? By trying ***all*** builds of IE in existence? And the ones that don't exist yet?), and situations where a faulty video card driver (obviously, a DLL too) caused the infamous 'Blue Screen of Death' only when also running a very specific application on a very specific laptop model (the bug has since been fixed by the laptop manufacturer), and bugs in no less a widely used file than `MSVCRT.DLL`. But IMHO the most convincing case was when an application with a few million installations started to use the function **`SetDIBits()`** to load Windows bitmaps (replacing the hand-written BMP file parsing + **`CreateBitmap()`** calls to simplify code); the result was that about 2% of installations just stopped working (and 2% meant 20000 frustrated users per million of installations, and resulted in many hundreds of complaints to the support department). Investigation revealed that while this function is a system one, some video drivers tried to optimize it and this 'optimized' version simply crashed for a certain BMP format (which was a perfectly valid, though not the most common, bitmap variation). This was the last straw for me, and I came to the conclusion: 'if you want your application to run reliably, avoid DLLs for as long as they're possible to avoid'.

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams [Adams].

Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

It might not be your fault, but it is your problem

To make things even worse, if your application crashes the end-user doesn't care if it happened because some ill-behaved 3rd-party application replaced `MFC42.DLL`, or if it happened because of a faulty version of Internet Explorer which is installed on their system: for the end-user it is ***your*** application which crashes, ***your*** application s/he will blame, ***your*** support department s/he will call/write to, and it is ***you*** who will eventually need to deal with it. When the problem with the ill-behaved application installing faulty `MFC42.DLL` occurs, 99.99% of the users will not go into lengthy investigations of the reasons, they will just blame the application that crashes. An application is perceived as a single product, and DLL dependencies are implementation details which the end-user doesn't care about at all. And if the application crashes because I am using a DLL without a good reason, it is indeed my fault; my job is to deliver a product which should work in the best possible way for the end-user, and if that doesn't happen then I didn't do my job properly.

Pro-DLL

Now I hope that I've described the most compelling disadvantages of DLLs (there are more – I haven't mentioned technical issues like more complicated memory management or messy name mangling), I will try to describe reasons why one may want to use DLLs despite these disadvantages. Reasons for implemeting DLLs are traditionally numerous, but IMHO many of them are not valid on closer inspection.

Reasons which are usually used to justify using DLLs are the following:

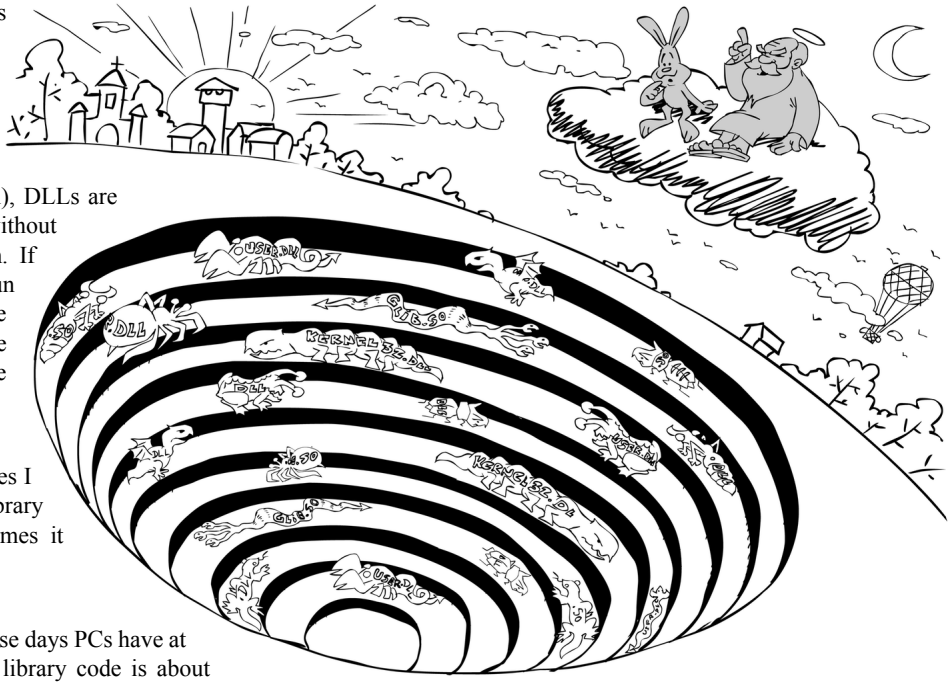
- Using system services that are in a DLL.

  A perfectly valid reason, but it begs the question 'what should we consider to be a system service'? For example, there is no way that file access can be done without using **`kernel32.dll`** on Windows, **`glibc.so`** on Linux (or similar DLLs/.so's), but in cases when more exotic services are used (such as a 'HTML control' or the **`GetDIBits()`** function described above), it becomes less obvious. Usually I'm sticking to the concept 'if you can do it yourself in reasonable amount of time – do it'.

Our primary job as developers is to make things work, and arguments about being 'cool' don't have any standing in my book

- Providing an interface for 3rd-party plug-ins or writing a plug-in.

  Another perfectly valid reason. While you might experience issues with badly behaved plug-ins which can crash your application (and user won't be able to tell difference if it was your application crashing or the plug-in), DLLs are still a primary method of providing plug-ins without the need to recompile the main application. If stability is of real concern, solutions which run plug-ins within separate processes are preferable, but they're much more complicated and are not always worth the trouble.



- The Library I need exists only as a DLL.

  It indeed happens, but personally in such cases I prefer to ask myself: 'do we ***really*** need this library or maybe we can live without it?' Sometimes it helps.

- It saves memory.

  While it was a valid reason back in 1980s, these days PCs have at least 256M of RAM, and the size of static library code is about 1000x less than that. It means that any noticeable effect of static linking instead of DLLs to overall system performance is extremely unlikely, and as a user I would definitely prefer having a statically linked application which doesn't crash instead of an application which uses 200K less RAM but has a 2% chance of crashing. On non-PC platforms analysis can be very different, but for modern PCs I feel memory savings are negligible. In addition, as described in [Anderson00], on Windows effects related to DLL relocation might reduce memory savings.

- Security reasons.

  With the whole software development industry plagued by security problems, having security-related areas separated and independently updated initially sounds like a good idea. Still, on closer consideration this aspect is not that obvious, and needs careful analysis depending on the specific application. First of all, it depends on your application life cycle: if it is routinely updated several times a month, the benefit of DLLs being updated independently is not that great; and in extreme cases of large security holes you can easily recompile and update your entire application. Moreover in some specific cases, when you need to resort to 'security by obscurity' (for example, if you're writing MMORPG and want to prevent 'bots from playing and giving an unfair advantage), using well known DLLs like `OpenSSL.DLL` provide an additional relatively easy vector of attack on your communication protocols. On the other hand, if your application is not going to be updated frequently and has nothing to do with 'security by obscurity', using security DLLs can be indeed a rather good idea.

- Smaller updates

  One common pro-DLL argument is that if you need to apply a minor fix, in the case of DLLs you only need to update a small number of small files rather than the whole large executable. On the other hand, if updating a large executable starts causing problems, it is always possible to use some kind of 'differential update' algorithm, which is able to calculate the differences between two versions of an executable file and then apply such a patch to previous version of the file; if checksums like SHA-1 are checked before and after applying such patch, this method is indeed more reliable than relying on DLL versions (while you can easily produce two different DLLs with the same version number, you will have really difficult time producing two different executables with the same SHA-1). In addition, the effects of larger updates become less relevant with steadily increasing broadband speeds and reduced traffic pricing.

- Static linking is so 1990-ish.

  This argument comes in many forms, including 'static linking is so uncool', 'everybody does it with DLLs these days' etc. etc. As I'm commonly characterized not as a 'cool' Bunny, but as a 'damn hot' Bunny, I really hate 'cool' arguments about technical implementation details, especially when they're causing problems for end-users. Our primary job as developers is to make things work, and arguments about being 'cool' don't have any standing in my book.

what I'm really surprised about is that both Windows and Linux/*nix are borrowing the very worst features from each other!

Mitigation

With the 'DLL Hell' problem being so ubiquitous, numerous ways have been proposed to deal with it:

- Static linking.

  My favourite. If there are no DLLs, they're not able to cause any problems. If concerned about updates, one will need to use 'differential updates' described above, but it is still, IMHO, a very minor effort compared to all the headaches originating from DLL use for large user base.

- Windows file protection.

  This one happens automagically and essentially simply prevents ill-behaved applications from overwriting system DLLs, aiming to address problems like the one I've described above with `MFC42.DLL` being overwritten (as well as security attacks). It does indeed provide some mitigation in certain cases, but is not enough to address all the problems arising from DLL Hell.

- 'private' DLLs.

  If on Windows you put all the DLLs into the same folder where your .EXE resides, your DLLs will become 'private' to your application, and the chances of somebody else messing with them will be minimal. While it indeed helps to deal with some aspects of 'DLL Hell', IMHO this approach doesn't make much sense and should be replaced with static linking, unless (a) a library only exists as a DLL, or (b) this DLL is used very rarely and you load it via `LoadLibrary()` or `dlopen()`. One argument for 'private DLLs' [Anderson00] is that they facilitate software updates, but with the existence of 'differential update' algorithms, it doesn't seem to be a strong argument.

- Allowing different DLL versions to run together.

  In Windows this is known as 'side-by-side assemblies'. [Microsoft] It essentially relies on the ability to specify the version of a DLL needed, which makes your application run reliably provided that user can obtain the required version of the DLL. On the other hand, if you specify the DLL version explicitly, you're putting yourself in a position which is even worse than with 'private DLLs', taking only the disadvantages with no apparent advantages: if you require a specific DLL version it is unlikely to be shared, and you're not able to benefit from security updates etc.; if you specify a major version but will accept minor versions to catch security updates, you no longer have the assurance that your application will run, and are still contributing to the horrible mess with multiple versions. For further analysis of 'side-by-side assemblies', please refer to an excellent recent article in Dr.Dobb's journal [Worthmuller10].

.so/RPM hell

While most of this article was written about DLLs, it would be a big mistake not to mention that *nix, and especially the Linux world, aren't free of similar problems. In particular, on Linux systems, specifying the exact version of an `.so` library is traditionally much more common than on Windows, making the hunt for the right version a particularly annoying exercise. Even if it is handled automagically by a package manager it still causes a horrible mess in installation directories and for deployment/maintenance purposes. In particular, incompatibilities between versions required by different subcomponents of the same executable abound (as just one such example, you can see the discussion about including OpenSSL v1.0 on the Apache mailing list [Apache]).

I don't want to say that Linux or Windows is better in regard of DLLs/.so's, I think that both are a horrible mess, and what I'm really surprised about is that both Windows and Linux/*nix are borrowing the very worst features from each other! *nix was the first to do it, borrowing the whole concept of DLLs as opposed to static linking from Windows – to the best of my knowledge, full support for .so's appeared in *nix as late as SVR4 in 1990 while Windows has had DLLs since Windows 1.0 in 1985. On the other hand, recent Windows 'side-by-side assemblies' seem to borrow from Linux a concept of explicit library version requirement for DLLs/.so's, which has been characterized in [Worthmuller10] as 'We were needing a solution, but we created a monster'.

Bottom line

I know for sure that hardcore fans of neither Windows nor Linux will be fascinated by this article, but that wasn't among my goals (as stated above, my goal was to invite people to think, and whoever can think critically is not a 'hardcore fan' in my book). What I've tried to say is that DLLs (or .so's) are full of inherent dangers, and the decision to use them is not to be taken lightly.

Personally I try to avoid them as long as possible, but the question 'how long is "as long as possible"?' still needs to be solved on case-by-case basis.

Good luck to everybody who needs to tackle DLLs, you'll definitely need it. ■

References

[Adams] http://en.wikipedia.org/wiki/Lapine_language

[Anderson00] 'The End of DLL Hell', Rick Anderson, Microsoft Corporation, 2000

[Apache] Apache HTTP Server Development Main Discussion List, http://mailarchives.apache.org/mod_mbox/httpd-dev/

[Ignatchenko10] 'Single-Threading: Back to the Future?', Sergey Ignatchenko, *Overload* #97/#98 (June/August 2010)

[Microsoft] http://msdn.microsoft.com/en-us/library/ms229072%28VS.80%29.aspx

[Worthmuller10] 'No End to DLL Hell!', Stefan Worthmuller, *Dr. Dobb's Journal*, September 2010, http://www.drdobbs.com/web-development/227300037

Making String ReTRIEval Fast

Sometimes you really do have to optimize your code. Björn Fahller looks at a string lookup data structure.

Ah, summer vacation. Time to toy with ideas that have grown over the year. This summer's project was performance tuning a Trie [TRIE]. Tries are fascinating, while special. They can be used as sets, or by storing values in data nodes, as maps. In a Trie, data is stored in a tree-like structure, where each node holds a part of the key.

Tries are primarily useful if the key is a string, although variants have been successfully used for other purposes, such as storing routing information. While a string is a special case for a key type, it's a frequently used special case, so its utility value should not be underestimated.

The original Trie data structure uses one node for each key character, whereas this variant stores common sub-strings and thus reminds a lot of a HAT-trie [HAT].

So what is a Trie?

Figure 1 shows a Trie with the strings "category" and "catastrophe". Here rectangular nodes denote stored keys and oval nodes are internal 'stepping stones' on the way to key nodes. This means that all leaf nodes are keys, and non-leaf nodes may be keys or internal. Adding more strings can change the structure in three ways.

Adding the string "cat" is trivial – it merely changes the mode of a node from internal to key, as figure 2 shows.

When the word "cathedral" is added, another child is added to "cat", with 'h' as a new distinguishing character, which figure 3 shows.

Inserting the word "catatonic" requires more work. The "catastrophe" node must be split into two nodes, one with an empty internal prefix on 'a' as distinguishing character and, as its child, a node with the suffix "trophe" after 's' as distinguishing character. Then a new node can be created, using 't' as the distinguishing character, and "onic" as suffix. Figure 4 shows the result.

This brief introduction already hints at a number of characteristics of the data structure.

- It can be lean on memory if the keys often share common prefixes (e.g. URLs and fully qualified file names)
- It probably does not work very well for short keys with great variability (e.g. random 32-bit integers)

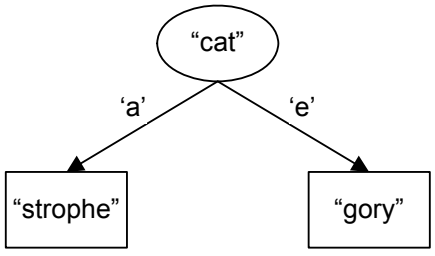


Figure 1

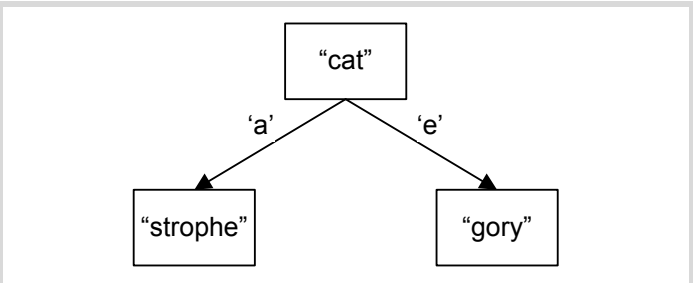


Figure 2

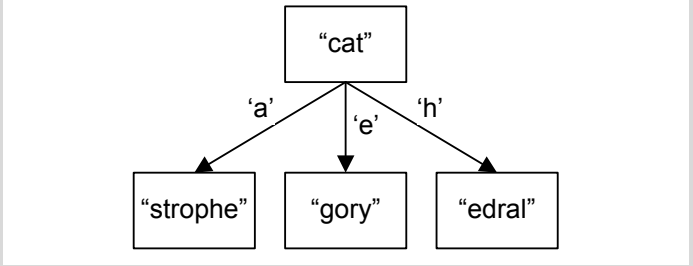


Figure 3

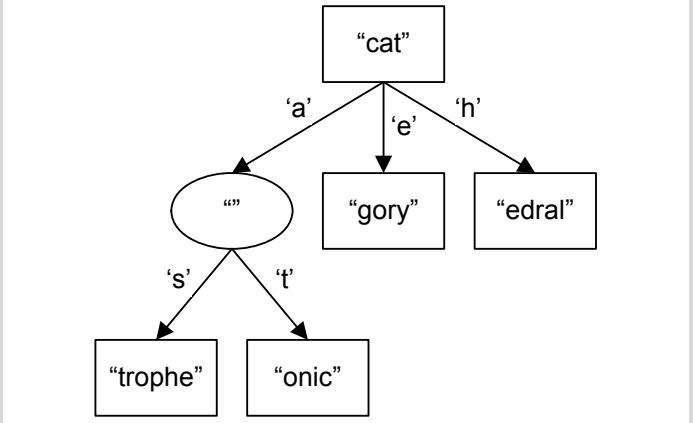


Figure 4

Björn Fahller is a systems analyst for telecommunications software with 13 years' experience as developer of embedded software. Apart from prototyping, programming now mostly competes for time with family, flight instruction and volunteering with flying for damage assessment and search for rescue. Björn can be reached on bjorn@fahller.se

in the real world, the performance of lookups is limited by the height of the structure

- It has the potential for being fast for long keys, since it only needs to read the key once per lookup operation (compared to at least twice for a hash-table).
- The performance of all operations are limited by search performance.
- Longest matching prefix searches are very efficient.

The general idea is that string retrieval time is linear with the key length. This, however, is only true if all accesses have equal cost, but comparing strings requires less computation than jumping between nodes, and the latter is also more likely to suffer from cache misses. So, in the real world, the performance of lookups is limited by the height of the structure, which is typically a logarithm of the number of strings stored.

Tuning in general

Measuring performance on a modern computer can be frustrating. Several levels of caches and a multi-core CPU governed by a preemptive operating system that flushes said caches provide very chaotic timing. It has even been suggested that it is truly random [random]. Even runs with many million searches taking more than 10 minutes will differ by several percent with identical data on a seemingly otherwise idle system.

On Linux for x86/32 and x86/64 there's an invaluable aid in 'valgrind' [valgrind] and its tools 'cachegrind' and 'callgrind'. Most Linux developers know 'valgrind' as a tool for hunting down memory related bugs. The tools 'cachegrind' and 'callgrind' both run your code in a virtual machine which can model your cache. With 'cachegrind' you can see the exact number of instructions used, the number of read and write accesses and number of misses for all cache levels. It often helps to see if your change was an improvement or not, even for reasonably short runs. There are situations when it's difficult to judge, though. Such an example is when the number of instructions increased slightly, but the number of level-1 cache misses decreased slightly. The tool 'callgrind' can be used to visualize details of execution. For example it can show, per code line, or even per CPU instruction, an estimate of cycles consumed, the number of read/write accesses and number of read/write cache misses per level. Unfortunately it is not cycle-accurate, so timing information must be taken with a grain of salt. The result from 'callgrind' is difficult to make sense of when aggressive optimizations are used by the compiler, since it can be difficult to map an instruction to a statement in your code. Fortunately, using less aggressive optimizations rarely change the data access pattern so 'callgrind' can help a lot in pinpointing not only where you have a performance bottleneck, but also why.

Test data and methodology

In order to measure the performance of the implementation representative data must be used. I have used three sources of strings. A number of e-books were downloaded from Project-Gutenberg [gutenberg]. UTF-8 was preferred when available. The majority of books downloaded are written in English, but there's also a fair mix of French, German, Dutch, Italian, Finnish, Swedish and one example of Greek. From those e-books, 2,408,716 unique sentences and 2,063,856 unique words were extracted (very naïvely, just using `operator>>` for `std::string`). Also a snapshot of the file system of a server machine gave 1,907,803 unique fully qualified filenames.

Only search time is measured, since insertion and deletion will be dominated by search when the structure becomes large.

When producing graphs, several runs are made. 5 sets of 1,000,000 unique words, sentences, and file names (obviously with an overlap between the sets) are used. Each set is used for searches when populating from 2 to 1,000,000 unique keys. Every sequence is run 3 times to reduce the randomness introduced by the OS flushing caches when scheduling other processes. The amount of data allocated for the structure is also measured. For the latter it is interesting to know that the average length of words is 9.5 characters, sentences 135 characters, and filenames 105 characters.

The chosen collection for performance measurements is a set. Extending to map, multiset and multimap is trivial. In order to save print real-estate, code samples only include the functions that are necessary to understand lookup and optimizations.

The host used for all performance measurements is an Intel Q6600@2.4GHz Quad Core CPU running 64-bit Linux. All test programs are compiled as 32-bit programs.

The hope is to beat `std::unordered_set<std::string>`, which is a hash-table, on both lookup performance and memory usage.

First attempt

The ideas behind this less than obvious implementation (see Listing 1) are:

- To save memory, and thus increase locality, nodes are made small. Leaves must be special, hence the inheritance.
- Nodes with children allocate each child node on the heap, and store the pointer in the vector children, sorted on the distinguishing character to allow a binary search. The theory is that the number of children will typically be very small.
- A node that has a key stores a `leaf_node` in `children[0]`, hence `n->children[f-v+1]` in `get()`. When implementing a map, the `leaf_node` instance will hold the data.
- To avoid having to lookup the children when searching, the distinguishing character is stored in a separate `char_array`, and is excluded from the child's prefix, as was shown in the introductory example. This avoids holes in the vector, saving memory, but makes lookup logic more complex.
- `char_array` stores short strings locally, and uses a pointer causing indirection only when they are long. This has proven to be a useful optimization in other experiments, so it is used right away.
- `num_chars` refers to the number of chars in `select_set`
- `prefix_len` refers to the length of `prefix`

Figure 5 mercilessly shows that the result was less than impressive. It's faster than a hash-table for long strings, providing there aren't too many

using less aggressive optimizations rarely change the data access pattern

```
struct child_node;
struct node
{
  virtual child_node* get_child() { return 0; }
  virtual ~node() { }
};
struct child_node : node {
  virtual child_node *get_child()
     { return this; }
  union char_array
  {
    char chars[sizeof(char*)];
    char *charv;
  };
  char_array          select_set;
  unsigned            short num_chars;
  unsigned            short prefix_len;
  char_array          prefix;
  std::vector<node*> children;
  const char *get_prefix() const {
    return prefix_len > sizeof(char_array)
           ? prefix.charv
           : prefix.chars;
  }
  const char *get_select_set() const {
    return num_chars > sizeof(char_array)
           ? select_set.charv
           : select_set.chars;
  }
};
```

Listing 1

```
struct leaf_node : public node
{
};
const node* get(const child_node *n,
   const char *p, size_t len)
{
  for (;;)
    {
      if (len < n->prefix_len)
         return 0;
      const char *t = n->get_prefix();
      if (std::strncmp(p, t, n->prefix_len))
         return 0;
      p+= n->prefix_len;
      len -= n->prefix_len;
      if (!len)
         return n->children.size()
            ? n->children[0] : 0;
      const char *v = n->get_select_set();
      const char *f = std::lower_bound(
         v, v + n->num_chars, *p);
      if (f == v + n->num_chars || *f != *p)
         return 0;
      ++p;
      --len;
      node *next = n->children[f - v + 1];
      n = next->get_child();
    }
}
```

Listing 1 (cont'd)

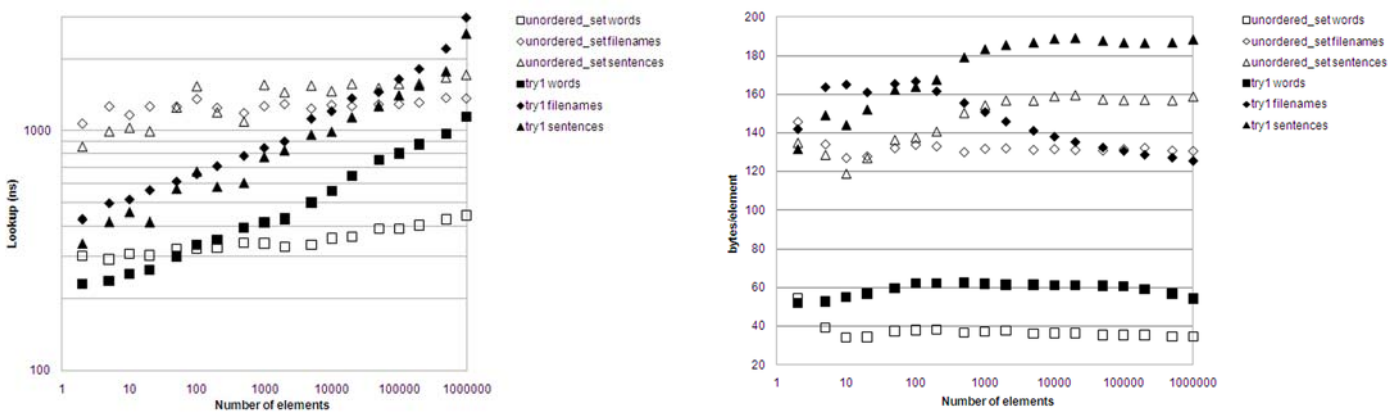


Figure 5

One dynamic dispatch per lookup, instead of one per traversed node, should make a difference

of them. Short string performance is terrible. Not only was insertion very difficult to get right when having to keep the `select_set` and `children` vectors in sync, 'callgrind' also pinpointed the binary search as a major lookup performance killer. It wasn't even lean on memory, only barely beating the hash table for large amounts of filenames.

```
struct data_node;
struct node
{
  virtual const data_node *get_data_node() const
    { return 0; }
  union char_array
  {
    char chars[sizeof(char*)];
    char *charv;
  };
  unsigned short     prefix_len;
  char_array         prefix;
  std::vector<node*> children;
  const char *get_prefix() const;
};

struct data_node : public node
{
  virtual const data_node *get_data_node() const
    { return this; }
};

const data_node *get(const node *n,
   const char *p, size_t len)
{
  for (;;)
    {
      if (!n) return 0;
      if (len < n->prefix_len) return 0;
      const char *t = n->get_prefix();
      if (std::strncmp(p, t, n->prefix_len))
         return 0;
      p += n->prefix_len;
      len -= n->prefix_len;
      if (len == 0) return n->get_data_node();
      unsigned char idx = *p;
      if (idx => n->children.size()) return 0;
      ++p;
      --len;
      n = n->children[idx];
    }
}
```

Listing 2

Second attempt

Binary searching was a time waster that must be gotten rid of. This time the vector will have indices 0–255, casting the characters to `unsigned char` to use directly as index. This wastes vector space, since there will be plenty of holes with 0-pointers, but it should be fast. The inheritance structure is also changed such that the dynamic dispatch will only be required when checking if the terminal node reached is indeed a data node holding a stored key, or if it's an internal node. One dynamic dispatch per lookup, instead of one per traversed node, should make a difference. (See Listing 2.)

Insertion logic was much simplified by this change, since there was no longer any need to shuffle nodes around. Lookup logic is also simple. The distinguishing character is not stored, since it is used as the index into `children` when looking up the next node, hence saving a byte of prefix for each node. Also, getting rid of `select_set` makes the struct 4 bytes shorter on a 32-bit system, probably increasing locality a bit.

There is a considerable lookup performance increase, as figure 6 shows. The improvement is especially noticeable for short strings. For long strings

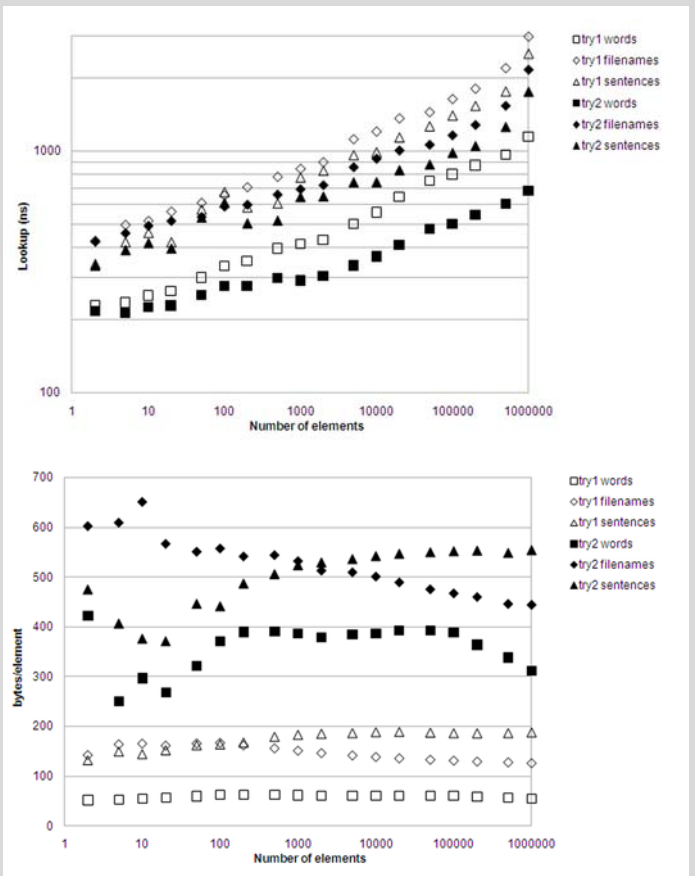


Figure 6

the reduced memory waste should increase the likelihood of cache hits and thus boost performance

the improvements are not as huge. Huge, however, is exactly what the memory requirements are. Time to rethink again.

Third attempt

Something must absolutely be done about the memory consumption. One obvious way is to use an offset for the vector, so that `lookup(i)` is `vector[i - offset]`, provided that `i` is within the legal range. This should reduce the memory consumption a lot without requiring expensive computations. (See Listing 3.)

While the lookup requires extra computation, if ever so little, the reduced memory waste should increase the likelihood of cache hits and thus boost performance.

The results are mixed, as figure 7 shows. With respect to performance, there is actually a slight regression, but memory consumption is way down.

According to 'callgrind', the obvious time waster for lookups is `nodev::operator[]() const`, which spends 42% of its time for the vector indexing, and the rest checking the conditions. Time for thoughts...

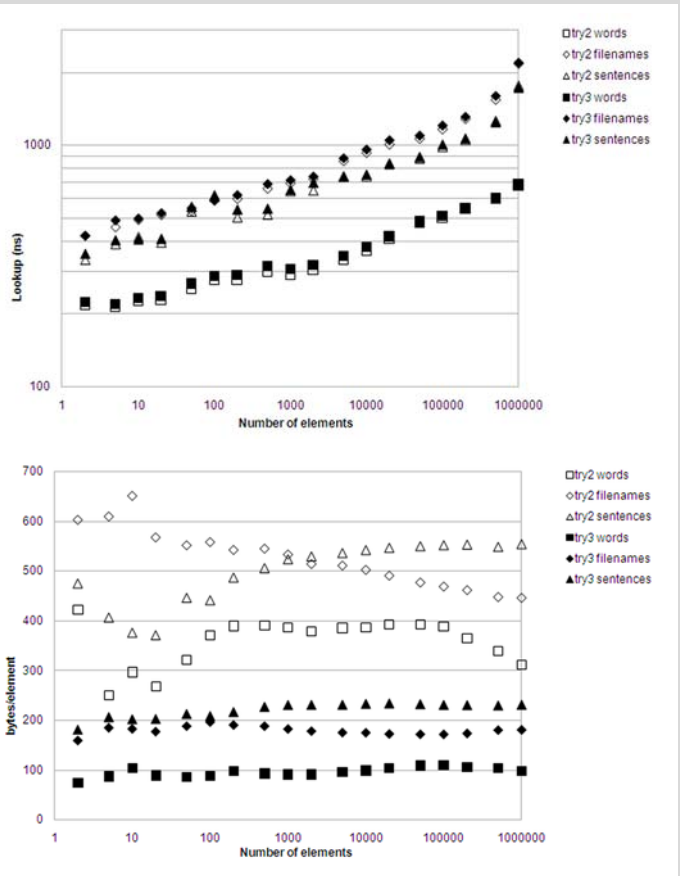


Figure 7

```
struct node;
struct nodev
{
  std::vector<node*> vec;
  unsigned char      offset;
  const node* operator[](unsigned char i) const;
};

struct data_node;
struct node
{
  virtual const data_node *get_data_node() const
    { return 0; }
  union char_array
  {
    char chars[sizeof(char*)];
    char *charv;
  };
  unsigned short prefix_len;
  char_array     prefix;
  nodev          children;
  const char *get_prefix() const;
};

const node * nodev::operator[](
   unsigned char i) const
{
  if (vec.size() == 0) return 0;
  if (i < offset) return 0;
  if (size_t(i - offset) >= vec.size()) return 0;
  return vec[i - offset];
}

const data_node *get(const node *n,
   const char *p, size_t len)
{
  for (;;)
    {
      if (!n) return 0;
      if (len < n->prefix_len) return 0;
      const char *t = n->get_prefix();
      if (std::strncmp(p, t, n->prefix_len))
         return 0;
      p+= n->prefix_len;
      len-= n->prefix_len;
      if (len == 0) return n->get_data_node();
      unsigned char *idx = *p++;
      --len;
      n = n->children[idx];
    }
}
```

Listing 3

The memory consumption is substantially lower, and lookup times have been reduced by around 15% across the board

Fourth attempt

Using a `std::vector<>` for the child nodes wastes space. Both the length and offset for the child vector can be stored as `unsigned char`, which reduces the node size a bit. This leaves an empty hole in the memory layout for the node, which is a waste. Better use it for the inlined prefix by packing the union. The special case that a node has only one child can be taken care of by a direct pointer to it, saving an indirection and thus increasing the chance of a cache hit. The range check for indexing children can be made more efficient by using wrap-around on unsigned integer arithmetics. (See Listing 4.)

Finally some real progress. Figure 8 shows the difference compared to the third attempt. The memory consumption is substantially lower, and lookup times have been reduced by around 15% across the board. The 'valgrind' tools aren't of much help in pinpointing current time consumers, but there is no doubt that there are a lot of cache misses. Hmmm...

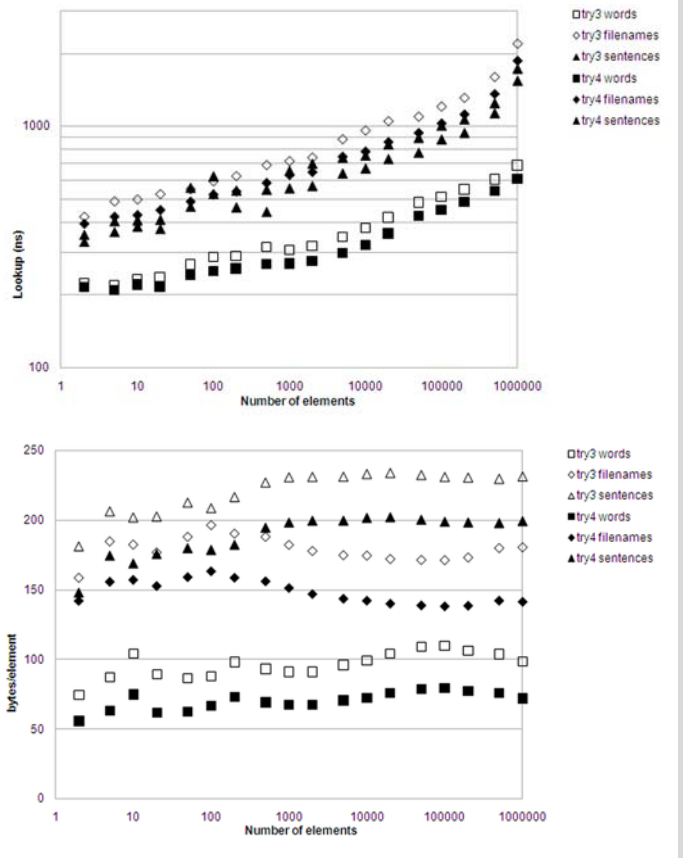


Figure 8

```
struct data_node;
struct node
{
  virtual const data_node *get_data_node() const
     { return 0; }
  const node* at(unsigned char i) const;

  union __attribute__((packed)) char_array
  {
    char chars[sizeof(char*)+2];
    char *charv;
  };
  char_array prefix;
  unsigned   char offset;
  unsigned   char size;
  unsigned   prefix_len;
  union {
    node*    nodep;
    node**   nodepv;
  };
  const char *get_prefix() const;
};

const node *node::at(unsigned char i) const
{
  unsigned char idx = i - offset;
  if (idx < size)  return size == 1
     ? nodep : nodepv[idx];
  return 0;
}

data_node *get(const node *n, const char *p,
   size_t len)
{
  size_t pl;
  while (n != 0 && (pl = n->prefix_len) <= len)
    {
      const char *t = n->get_prefix();
      const char *end = t + pl;
      for (const char *it = t; it != end;
         ++it, ++p)
        {
          if (*p != *it) return 0;
        }
      len-= pl;
      if (len == 0) return n->get_data_node();
      --len;
      n = n->at(*p++);
    }
  return 0;
}
```

Listing 4

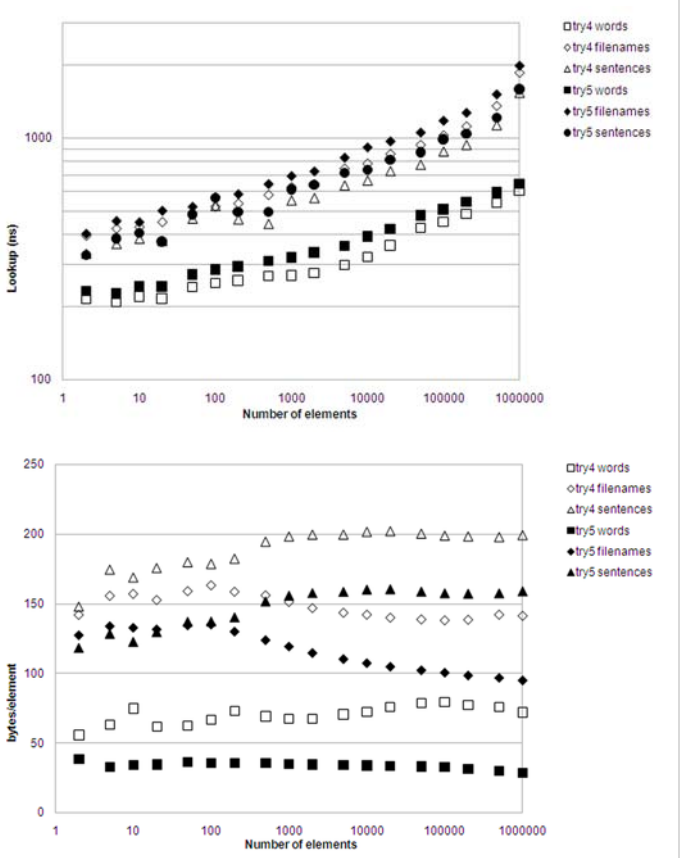


Figure 9

Fifth attempt

Perhaps memory consumption can be reduced by using a simple hash table for the children, instead of direct indexing. It should be possible to reduce the number of unused entries a lot. Since using a hash table may cause several distinguishing characters to reach the same entry, the distinguishing character needs to be stored in the prefix. Fortunately there is no longer any need for the **`offset`** member, so the inlined **`prefix`** member can grow by one **`char`** without growing the node size. To make lookup simple, the hash implementation chosen is just a remainder-operation on the size, and only collision free sizes are allowed. This trivially reaches a maximum size of 256 entries, where it becomes direct indexing. By getting rid of inheritance, and just steal a bit from the **`prefix_len`** member to tell if the node holds data or not, the struct can be further reduced in size, which should increase locality. Limiting the prefix length to 231 characters is unlikely to be a real problem (Listing 5).

Performance regression, bummer. Figure 9 shows that lookup times are slightly longer again.

Look at the memory consumption, though, especially for file names. The average file name is 109 bytes long, yet the average amount of memory consumed per stored file name is only 94.8 bytes for 1,000,000 file names. The size of the entire searchable data structure is, in other words, smaller than the size of its contained data. That's quite impressive.

```
struct node
{
  const node *get_data_node() const {
     return data ? this : 0; }
  node* at(unsigned char i) const;
  union __attribute__((packed)) char_array
  {
    unsigned char chars[sizeof(char*)+3];
    unsigned char *charv;
  };
  char_array    prefix;
  unsigned char size;
  unsigned      prefix_len:31;
  unsigned      data:1;
  union {
    node*       nodep;
    node**      nodepv;
  };
  const unsigned char *get_prefix() const;
};
const node *node::at(unsigned char i) const
{
  if (size <= 1) return nodep;
  unsigned char idx = i % (size + 1U);
  return nodepv[idx];
}
const node *get(const node *n, const char *p,
   size_t len)
{
  unsigned pl;
  while (n != 0 && (pl = n->prefix_len) <= len)
    {
      const unsigned char *t = n->get_prefix();
      const unsigned char *end = t + pl;
      for (const unsigned char *it = t;
         it != end; ++it,++p)
        {
          if ((unsigned char)*p != *it) return 0;
        }
      len-= pl;
      if (len == 0) return n->get_data_node();
      n = n->at(*p);
    }
  return 0;
}
```

Listing 5

	Attempt 4		Attempt 5	
Expression	**L1 misses**	**L2 misses**	**L1 misses**	**L2 misses**
while: n->prefix_len	43,951,604	162	44,112,899	77
*p != *it	1,854,838	11	6,935,480	13
n->get_data_node()	161,291	3	0	0
n->at()	43,870,965	144	39,919,345	49
S	89,838,698	320	90,967,724	139

Table 1

It appears like the reduction in memory consumption went according to plan, without buying anything back in terms of fewer cache misses. This calls for a more detailed study of where the cache misses are.

Surprisingly, Table 1 (a comparison of cache misses in attempts 4 and 5 with 10,000,000 lookups in 100,000 words) shows that the cache miss pattern has changed, but the sum is almost identical. It appears that whenever there is a cache miss in **`n->at()`**, i.e. in obtaining **`nodepv[idx]`**, there is also a cache miss in dereferencing the returned pointer. The reduction in cache misses for **`n->at()`** in attempt 5 is likely to be due to increased locality, but accessing **`*nodepv[idx]`** is seemingly no more likely to be a cache hit than in attempt 4. The considerable rise in cache misses for prefix comparison is a mystery, though.

Most probably the difference in performance is mainly because calculating the remainder of an integer division is a slightly expensive operation. This

is not shown in 'callgrind', though, but remember that its idea of time per instruction is not cycle-accurate.

Sixth attempt

When **`nodepv[idx]`** is a cache miss, a lot of data is read that isn't needed, since a pointer is much shorter than a cache line. The cache lines on the Q6600 CPU are 64 bytes wide. Whenever **`nodepv[idx]`** is a cache miss, 64 bytes are read from the level-2 cache (or worse, from physical memory.) Of those 64 bytes, only 4 are used in 32-bit mode, since a pointer is 4-bytes long. With luck, another lookup will soon refer to a pointer within those 64-bytes, but chances are the line will be evicted for another read before that happens.

Using an array of nodes, instead of an array of pointers to nodes, may waste a lot of memory, but it should save on cache misses. Also, when there is a cache miss, a large part of what is read into the cache line is more likely to be the data in the node struct, which will be used very soon indeed.

It is not obvious that an array of nodes wastes space, though. A counter example is a completely filled array, where the indirection leads to the whole array itself being wasted space, in addition to requiring an unnecessary indirection.

With the above in mind, it is worth staying with the same hash function and see if the expected reduction in cache misses are there and what performance boost it gives. The mystery with increased cache misses for prefix comparison is ignored for the moment. (See Listing 6.)

The lookup hash function remains the same as before. The trick with the **`char_array`** union is just to better use the memory when compiling in 64-bit mode. Having further reduced **`prefix_len`** to 23 bits is getting to the territory where it, at least in theory, may become a problem. However, the problem is not difficult to overcome – just split the prefix into separate nodes every 8,388,607 characters. I doubt the split will be noticeable in performance.

Figure 10 says it all. Good. Very good. The curve is flatter. The performance increase is impressive, especially for large collections, and surprisingly the memory consumption is actually down a little bit.

```
struct node
{
  const node *get_data_node() const {
     return data ? this : 0; }
  const node* at(unsigned char i) const;
  const unsigned char *get_prefix() const;
  union __attribute__((packed)) char_array
  {
    unsigned char chars[2*sizeof(char*)-
       sizeof(uint32_t)];
    unsigned char *charv;
  };
  char_array  prefix;
  uint32_t    prefix_len:23;
  uint32_t    data:1;
  uint32_t    size:8;
  node       *nodep;
};
const node *node::at(unsigned char i) const
{
  unsigned char idx = i % (size + 1U);
  return nodep + idx;
}
const node *get(const node *n, const char *p,
   size_t len)
{
  // same as in fifth attempt
}
```

Listing 6

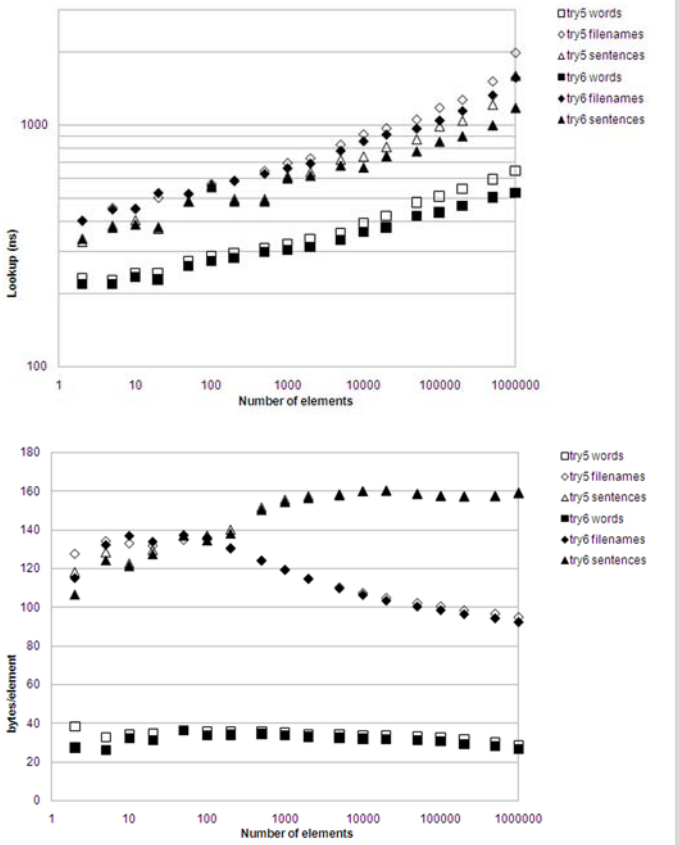


Figure 10

It now comes out favorably both on lookup performance and memory consumption for most situations when compared with **`std::unordered_set<std::string>`**, as figure 11 shows.

Table 2 (a comparison of cache misses in attempts 5 and 6 with 10,000,000 lookups in 100,000 words) shows a comparison of the cache-miss pattern between attempts 5 and 6. As can be seen, the theory of wasted reads seems to have been correct, since the total number of cache misses are down by 40%, since the misses in the **`at()`** function nearly vanished.

Now 'callgrind' pinpoints the calculation of **`i % (size + 1U)`** in **`at()`** as the single most expensive operation in the lookup path. Just out of curiosity, a bit-mask operation must be tried instead. The only code change visible in the lookup path is changing the **`at()`** function to calculate **`idx = i & size`**, where **`size`** is always 2^x-1. Obviously memory consumption will increase severely, but a bit mask operation is much faster than a remainder operation, so lookup times may be considerably reduced.

Woohoo! Illustration 12 gives the proof. Now this is fast, but undoubtedly the memory requirement is rather on the obese side.

	Attempt 5		Attempt 6	
Expression	**L1 misses**	**L2 misses**	**L1 misses**	**L2 misses**
while: n->prefix_len	44,112,899	77	42,991,925	58
n->get_prefix()	0	0	322,580	1
*p != *it	6,935,480	13	7,274,191	11
n->at()	39,919,345	49	4,701,610	1
S	90,967,724	139	55,290,306	71

Table 2

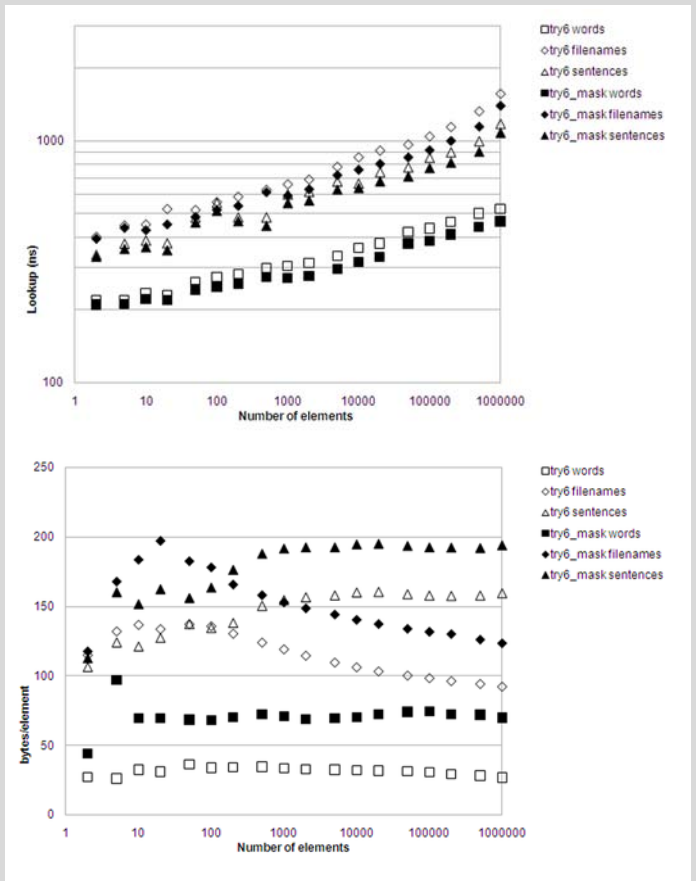


Figure 11

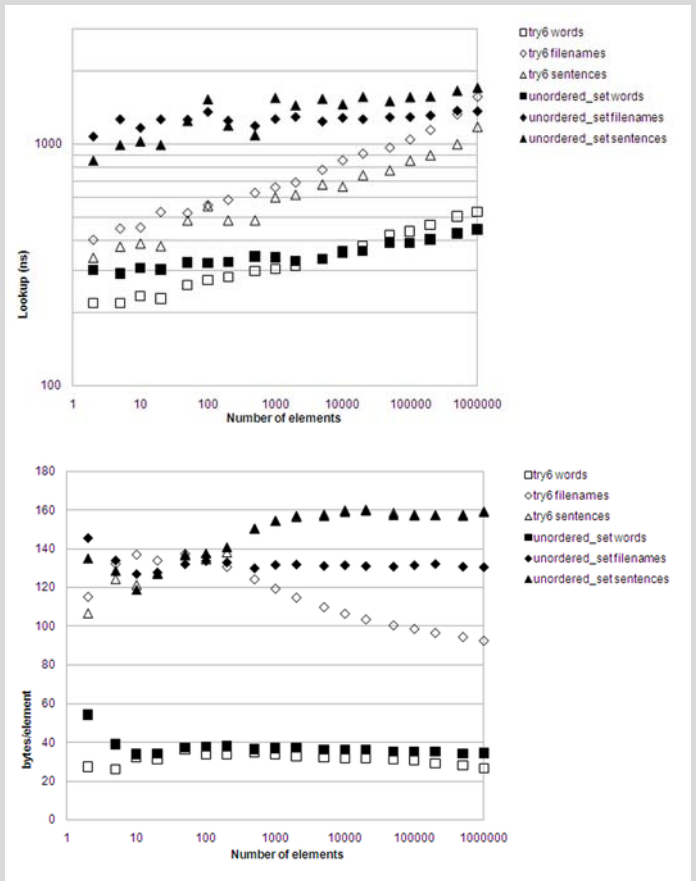


Figure 12

Inconclusion

I would have liked to end this with a final conclusion about how to squeeze the last cycle out of the Trie, but alas summer vacation ended, and with it the available tinkering time.

Some observations, though, with thoughts for further studies:

- Can a different struct layout improve cache hits further?
- The remainder hash function causes an allocation of roughly 3 times as many nodes as needed. The mask hash function results in nearly 6 times as many nodes as needed. Is it possible to find a hash function that has the speed of the mask operation while providing better memory efficiency than the remainder operation?
- Can a hash-table implementation that allows collisions improve lookup performance? The collisions will cost, but if they are rare enough the improved locality of reference may generate a total performance gain.
- Can a global hash table for all nodes, instead of a local hash table for the children of each node, be more memory efficient without sacrificing lookup performance?
- The stored prefix strings are always perfectly aligned for the CPU. Can it be beneficial to compare the prefix and the searched for strings using the largest integer data type possible for the alignment, instead of always using char by char? Good built-in memcmp() implementations do this, but they cannot assume anything about the alignment for any of the strings, whereas an implementation that knows one of the strings is always perfectly aligned could be made a slight bit faster. For very short prefixes it would probably be a waste, but for long prefixes (e.g. filenames and URLs) it might speed up the processing. (Actually calling `memcmp` in attempt 6 with bit mask, causes a 16% slowdown for filenames, and 22% slowdown for words.)
- Can a custom allocator reduce heap fragmentation/overhead and thus improve the chances of cache hits and/or lessen the memory footprint?
- An attempt was made to store all long prefixes in a reference counted structure, to both save memory and increase the chance of cache-hits. It was a resounding failure except for very large collections of file names, where memory usage improved a bit while performance remained unchanged.
- Compiling the same program in 64-bit mode increases its memory consumption, since the larger pointers makes the node struct larger. The lookup performance is generally slightly worse than in 32-bit mode, probably because cache lines are evicted earlier.

One conclusion can be made, however – a Trie has indeed proven to be a very attractive data structure for string keys. It has been shown to beat hash tables in both lookup performance and memory consumption for a number of very real use cases. ■

References

[gutenberg] http://www.gutenberg.org
[HAT] http://crpit.com/confpapers/CRPITV62Askitis.pdf
[random] http://www.osadl.org/fileadmin/dam/presentations/RTLWS11/okech-inherent-randomness.pdf
[TRIE] http://en.wikipedia.org/wiki/Trie
[valgrind] http://www.valgrind.org

Quality Matters #6: Exceptions for Practically-Unrecoverable Conditions

Being robust is harder than you think. Matthew Wilson analyses a classic program.

This is the second in a series of instalments on exceptions. In the last instalment [QM-5] I considered a taxonomic perspective of program states and actions, and suggested a new vocabulary for the four defined states: normative, recoverable, practically-unrecoverable, and faulted. In this instalment I'm going to focus on the simplest proper use of exceptions, for reporting practically-unrecoverable conditions.

It is currently envisaged that there will be two more instalments in this mini-series. The next will deal with the much more challenging situation of using exceptions for recoverable conditions, including the non-trivial issue of deciding whether a given exception should be treated as recoverable or practically-unrecoverable.

The fourth and (hopefully) final part will suggest good practices in exception definition and use, look at how exceptions and threads work together, and consider the effects of the use (or non-use) of exceptions on the software quality characteristics of software libraries and programs.

The major part of this instalment comprises a surprisingly involved look at the classic "Hello, World" program, illustrating how its implicit exception-handling is a bad example for any non-trivial programs. I'll then proffer a practical example from my own work as a production-quality **`main()`** that adequately catches and processes exceptions representing practically-unrecoverable conditions. Finally, I'll look at how implementing C-APIs in C++ brings a blessed discipline to the catching of exceptions, albeit at a significant cost in effort and/or diagnostic flexibility.

Hello, World

In *The C Programming Language* [K&R], the eponymous *hello-world* is given as:

```
// hello-world.0.c
#include <stdio.h>
int main()
{
  printf("hello, world\n");
}
```

It's expressive, transparent, portable, efficient, and it's almost correct. As a basis for all C programs of any sophistication it is reasonable, too. Execution passes to the program via entry to **`main()`** and, unless **`exit()`** (or equivalent) is called by the functions that are called from within **`main()`**, execution completes as **`main()`** returns.

However, as for completeness, specifically for correctness/robustness/reliability, there's a problem: what happens if **`printf()`** fails? (This could be the case if the program's output was redirected to a file that could not receive the 13 or 14 bytes of the message.)

Let's consider the issue by expanding the example. First, let's deal with the implicit return. In case you're not familiar with this form (which I happen to hate with a passion), although **`main()`** must have a return type of **`int`**, it is allowed to have no explicit return statement. The C standard states, in clause 5.1.2.2.3, that 'reaching the } that terminates the main function returns a value of 0'. So, the above code is equivalent to:

```
// hello-world.1.c
#include <stdio.h>
int main()
{
  printf("hello, world\n");
  return 0;
}
```

For the more pedantic, such as your humble author, this should be written with more explicit meaning, as:

```
// hello-world.2.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  printf("hello, world\n");
  return EXIT_SUCCESS;
}
```

The C standard defines (in clause 7.20.4.3;5) the macro **`EXIT_SUCCESS`** (in `stdlib.h`) to be equivalent to the value 0, and that both represent 'successful termination' of the process. (In every case where I've checked, **`EXIT_SUCCESS`** is defined as *being* 0, so you can safely ignore the possibly of two distinct successful termination values.) It also defines the macro **`EXIT_FAILURE`** (also in `stdlib.h`), whose value is also implementation-defined, to represent 'unsuccessful termination'. In every case where I've checked **`EXIT_FAILURE`** is defined as being 1, but that still does not make it appropriate to return 1 in your code. The standard requires an implementation to return an unsuccessful status to the program's calling environment only if **`EXIT_FAILURE`** is returned (or passed to **`exit()`**, which is equivalent).

Now we're getting somewhere. When **`main()`** returns the value **`EXIT_SUCCESS`** (or 0), that's an explicit statement to the calling environment – to the 'world' we're hailing, in fact – that everything succeeded. Unfortunately, the unconditional stipulation of success is unjustified, since there's no guarantee of success here.

We can assume correctness/robustness of the runtime and the implementation of standard library functions. (In fact, we *must* do so, otherwise we have an infinitely insoluble problem of recursive self-guessing; this is another aspect of *irrecoverability* [QM-2, QM-5] that will be dealt with when we get to contract programming. Probably around QM#9 at the going rate …) Contrarily, we *must not* assume *normative* behaviour [QM-5] where that is not guaranteed, which includes cases, like

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

our only contingent action has been to
indicate to the caller, via the return code,
that the program has failed

this one, involving interaction with external entities such as the file-system.

So, strictly, the definitive *hello-world* program is wrong. Ouch! Now, it's entirely appropriate for its authors to claim that the requirements of *hello-world* allow for tacit failure when redirected, or when the kernel's running out of puff, or whatever. However, it is *not* appropriate to say that failing to account for *non-normative* action is justified for the purposes of pedagogy, or that failure is so unlikely in 99.9999% of use-cases that we don't need to bother. Unless that 1-in-a-million user who encounters a blank result can see in the program specification that such an eventuality is possible in certain circumstances (even when that circumstance is not precisely specified), then the program is wrong, and its authors have failed in their task.

Note that it is possible to have correct implementations of **`main()`**, and therefore to stipulate unconditional successful output, as long as we restrict ourselves to only using code that can be asserted as correct, such as:

```
// hello-world.3.c
#include <string.h>
int main()
{
  return (int)strlen("");
}
```

or:

```
// hello-world.4.c
int lnot(int v)
{
  return !v;
}
int main()
{
  return lnot(1);
}
```

But interacting with the file-system involves the possibility of non-normative behaviour, requiring contingent action. Here's an attempt at a simplest robust version involving **`printf()`**:

```
// hello-world.5.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n = printf("hello, world\n");
  return (13 == n) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Ugly, isn't it? A slightly nicer one is possible, using the standard global pseudo-variable **`errno`**:

```
// hello-world.6.c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
  errno = 0;
  printf("hello, world\n");
  return (0 == errno)
     ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

The simplest one I can come up with is:

```
// hello-world.7.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  return (EOF != puts("hello, world"))
         ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Unfortunately, this is still not enough. As I'm sure you're aware, gentle readers, the standard output stream is buffered. Since the C standard (7.20.4.3;4) requires that 'all open streams with unwritten buffered data are flushed', it is entirely possible, indeed likely in all these example cases, that the salutation will not be written prior to leaving **`main()`**. As a consequence, checking the functioning of **`(f)printf()`**/**`(f)puts()`** does not suffice. The smallest clear and robust implementation of *hello-world* in C looks like the following:

```
// hello-world.8.c
#include <stdlib.h>
int main()
{
  if( EOF == puts("hello, world") ||
      0 != fflush(stdout))
  {
    return EXIT_FAILURE;
  }
  else
  {
    return EXIT_SUCCESS;
  }
}
```

Perhaps it's no wonder that programming books don't trouble readers with correct/robust example programs!

Reporting

So far our only contingent action has been to indicate to the caller, via the return code, that the program has failed. We can (and should) also report what has failed, to the degree we are able, via a very simple form of *contingent reporting*, using the standard error stream, via **`perror()`** (see sidebar 'Printing Errors in C'):

```
// hello-world.9.c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main()
```

all interesting events should be subject to diagnostic logging

Printing Errors in C

The C standard library provides two functions for mapping 'error' codes, maintained in the global pseudo-variable **errno**, into human-readable values. The first, **strerror()**, returns a non-**NULL** C-style string mapping any integer value, including all of those defined (both in the standard, and all implementation-defined ones) in `errno.h`, into a human-readable message. For example:

```
strerror(ERANGE); → "Result too large"
strerror(EDOM);   → "Numerical argument out of
                     domain"
strerror(EMFILE); → "Too many open files"
strerror(0);      → "No error detected"
strerror(123456); → "Unknown Error (123456)"
```

It's common to pass the current value of **errno**, to get a string explaining what most recently behaved in a non-normative manner within (the currently executing thread of) your program. There are issues with re-entrancy in the use of **strerror()**; see [STRERROR] for more information.

The second standard library function, **perror()**, is used to print a message that also includes the message associated with the current value of **errno**, separated by ": ", as in:

```
errno = ERANGE;
perror("oops");  → "oops: Result too large
```

```
{
  if( EOF == puts("hello, world") ||
      0 != fflush(stdout))
  {
    perror("failed to say hello");
    return EXIT_FAILURE;
  }
  else
  {
    return EXIT_SUCCESS;
  }
}
```

Reporting can come in two flavours: contingent reports, and diagnostic log statements.

> Definition: A **contingent report** is a block of information output from a program to inform its controlling entity (human user, or spawning process) that it was unable to perform its normative behaviour. Contingent reports are a part of the program logic proper, and are not optional.

Typical contingent reports include writing to the standard error stream, or opening a modal window containing a warning. They are almost always used for issuing important information about *recoverable* or *practically-unrecoverable* conditions.

> Definition: A **diagnostic logging statement** is a block of information output from a program to an optional observing entity (human user/adminstrator, or monitor process) that records what it is doing. Diagnostic logging statements are optional, subject to the principle of *removability* [QM-1], which states: 'It must be possible to disable any log statement within correct software without changing the (well-functioning) behaviour'

Similarly, diagnostic logging statements are often predominantly used for recording contingent events, but this not need be so. In principle, all interesting events should be subject to diagnostic logging, to facilitate detailed historical tracing of the program flow. A good diagnostic logging library should allow for statements of different severities to be selectively enabled with minimal intrusion on performance when disabled.

Even though it's occasionally useful to piggy-back one form of reporting on the mechanism of the other, it's crucial not to confuse or transgress the requirements that the former is part of the program logic and may not be removed and the latter is optional and may be disabled at compile/link/run-time at will.

Hello, world++

What has all this got to do with exceptions, you may wonder? Well, the C++ *hello-world* (this one extracted from *The C++ Programming Language* [TC++PL]) is functionally similar:

```
// hello-world.0.cpp
#include <iostream>
int main()
{
  std::cout << "Hello, new world!\n";
}
```

Unsurprisingly, it has the same defect as the C version: it does not account for failure. Since the IOStreams, like C's Streams library, uses buffered output, the first thing we need to do is to ensure that the standard output stream is flushed, in order that the program is in a position to detect whether the write was successful. That can be done by using the **flush** inserter, as in:

```
// hello-world.1.cpp
#include <iostream>
int main()
{
  std::cout << "Hello, new world!\n"
     << std::flush;
}
```

A more common way of doing this is to express the newline sequence and the flush operation in one, via the **std::endl** inserter:

```
// hello-world.2.cpp
#include <iostream>
```

The programmer doesn't have to lift a finger (to provide any contingent action) and it all just magically works

IOStreams Hello-Worlds

It's no secret that I'm not a fan of IOStreams, and I've written about its many undesirable features before [FF-1]. The one that pertains to our current concern is arguably one of the worst: by default, non-normative behaviour is not reported via exceptions. Instead, you have to use the call-then-test anti-idiom: we must explicitly call the **`basic_ios::fail()`** method, as in:

```
// hello-world.3.cpp
#include <iostream>
int main()
{
  std::cout << "Hello, new world!" << std::endl;
  return std::cout.fail()
         ? EXIT_FAILURE : EXIT_SUCCESS;
}
```

Of course, it's easy to understand how this was the pragmatic choice when moving from a world predominantly without exceptions to a standard-prescribed one with them. But the result is the mess we see before us. (And disrupting programmers during compilation is a lot cheaper than after product deployment …)

Alternatively, you can instruct the stream to throw exceptions in the case where it encounters a non-normative condition:

```
// hello-world.4.cpp
#include <iostream>
int main()
{
  std::cout.exceptions(
  std::ios_base::badbit |
  std::ios_base::eofbit |
  std::ios_base::failbit);
  std::cout << "Hello, new world!" << std::endl;
}
```

While this looks a lot worse, it's actually a lot better, as it applies for the lifetime of the stream, so, as long as you remember to set it early in its lifetime, at least you won't experience any silent failures.

```
int main()
{
  std::cout << "Hello, new world! " << std::endl;
}
```

Now we have the stream flushed. Unfortunately, that's the least of our problems.

The IOStreams is such a horrible undiscoverable library that just getting to make my simple point involves a heap of messing around; see the sidebar 'IOStreams Hello-Worlds' for the hurdle jumping diatribe. Instead, I will use **FastFormat** [FF-1, FF-2, FF-3], which illustrates the point succinctly.

```
// hello-world.5.cpp
#include <fastformat/ff.hpp>
#include <fastformat/sinks/ostream.hpp>
#include <iostream>
int main()
{
  ff::flush(ff::writeln(std::cout,
     "Hello, new world!"));
  return EXIT_SUCCESS;
}
```

This program is robust. The normative behaviour is to output the greeting. The non-normative behaviour, should the output fail to be written, causes an exception to be thrown (by FastFormat's **`std::ostream`** sink), and the program terminates with a non-zero exit code.

Bad reporting

The C++ *hello-world* world sounds great, doesn't it? (At least it does once we get it to the point where exceptions are thrown on failure.) Robustness is achieved by the runtime library performing contingent action in response to the uncaught exception emanating from the **`ff::writeln()`** statement. The programmer doesn't have to lift a finger (to provide any contingent action) and it all just magically works.

This may get us over the line as far as robustness is concerned, but in terms of usability it stinks! The main problem is that the carefully prepared diagnostic information put into the thrown exception is not used. When an uncaught exception makes its way to escape **`main()`**, the language runtime invokes **`std::terminate()`** [TC++PL].

```
// in namespace std
void terminate(void);
```

Note that it takes no parameters – the gratuitous **`void`** is for emphasis. The standard requires it to call **`abort()`**, which causes the process to exit with a non-0 exit code. You can set your own, if you wish, via **`std::set_terminate()`**. But your own version must also return **`void`** and have no arguments.

You might (reasonably) wonder why **`std::terminate()`** doesn't take an argument of type **`std::exception const&`**. Well, that's doubtless because in C++ it is permissible to throw instances of types not derived from **`std::exception`**; it's even possible to throw fundamental type instances: **`void*`**, **`int`**, **`char const*`**, **`double`**, etc. This allows for backwards compatibility with pre-standard exception mechanisms and hierarchies, but it's a pity nonetheless; see the sidebar 'Why Catch-All Clauses are Bad News' for my favourite (of many) reasons why this is a bad idea. Thankfully, newer languages have learned from C++'s experience, and mandate that thrown objects derive from a single, specific, class type.

Thus, we're not going to get the detailed diagnostics we want. Instead you might see a message such as the following, from hello-world.5.cpp compiled with GCC 3.4:

```
This application has requested the Runtime to
terminate it in an unusual way.
Please contact the application's support team for
more information.
```

it's possible to catch access violations, divide-by-zero, and a whole host of critical, and desirably fatal, conditions, and quench them

Why Catch-All Clauses are Bad News

In C++, it's possible to catch all possible exceptions via the catch-all clause, as in:

```
try
{
  . . .
}
catch(...)
{
  fputs("unknown exception\n", stderr);
  throw;
}
```

In principle, this is a great thing. In many cases it's desirable to temporarily intercept a thrown exception in order to issue diagnostic logging/contingent reporting, before rethrowing the exception to be caught by something that knows what to do with it.

Unfortunately, some compilers allow you to do more. On Windows, several compilers piggy-back the C++ exception mechanism on top of the Structured Exception Handling [Richter] mechanism, and, for reasons that must have seemed sensible to someone, somewhere, at some time, allow the user to catch operating system exceptions by C++ catch-all clauses. Microsoft's Visual C++ does this, along with a number of others.

This is a terrible idea, for two reasons. First, and most important, this means it's possible to catch access violations, divide-by-zero, and a whole host of critical, and desirably fatal, conditions, and quench them. Obviously, it's then impossible to trust the program. So, use of a catch-all (that doesn't rethrow, or terminate the process) means that robustness cannot be adjudged.

Second, but also pretty important, this behaviour is not standard, and therefore code that uses the catch-all clause is not portable, either between compilers on a given operating system, or between operating systems.

My advice regarding catch-all clauses is: use them as little as possible, preferably never. If you do use one, it should (almost) always rethrow/terminate, after performing the smallest amount of work possible (i.e. a small impact diagnostic logging/contingent reporting statement). We'll see later in this instalment about the ramifications of this.

Whoa, now Neddy! Have a sugar-lump and calm down. That's a pretty intimidating message. Programmers and non-programmers alike would be concerned to see their program having done that.

And it gets worse still. Some compilers take the minimal approach to fulfilling the standard's requirement for `std::terminate()`, and call `abort()` without issuing any output: CodeWarrior (version 8, on Windows) is one. So the program simply stops, and unless the user is testing the process exit status he/she gets no indication whatsoever that anything has failed.

If you think about it, the minimal approach, while leaving the user non the wiser, is arguably the more correct. Since the exception is, literally, unexpected, the runtime cannot assume the program is in any kind of fit state, not even to issue diagnostic logging output or a contingent report.

Either way, the fact that `abort()` is called, and explanation is scant/missing, means that failing to catch exceptions in `main()` is not intended. Any quality program will, therefore, contain at least one outer try-catch clause. So why don't we see it in C++ textbooks!?

Other languages

The foregoing exposition has fixed firmly on C and C++, for two reasons: they're the languages I know most about; they're the closest to the metal in all this stuff, so represent a very good place to start.

But having raised the spectre of wrongness in pretty much every C/C++ programmer's first introduction to the language, it behoves me not a little to see what's going on in other languages. Simply put, I want to see what happens when all the C#/Java/Python/Ruby *hello-world*s are unable to write to `stdout`. Each will be judged by whether it:

1. Appears to have any awareness that output has failed.
2. Throws an exception to stop the process.
3. Returns a non-zero exit code to the calling environment.

C#

The program used is as follows:

```
class Program
{
  static void Main(string[] args)
  {
    System.Console.Out.WriteLine(
       "Hello, brave new world!");
    System.Console.Out.Flush();
  }
}
```

How does it behave? The good news is that the .NET runtime does register that the write has failed, and throws an exception. Unfortunately, this is hardly handled in what you'd call a graceful way. On a machine where one or more debuggers are resident (including all those I have to hand while writing this article), it causes a 'Select Debugger' (Figure 1) dialog to appear.

If you select **No**, then you get the text shown in Figure 2 on the command-line:

If you're a programmer this is ok. Well, no. Let me rephrase: if you're the programmer of this program, this is useful. If you're a user, it's unnecessarily horrible and scary.

Worse, much worse, is the fact that after all that, the process informs its calling environment that everything has gone swimmingly. Yes, hard as it is to believe, an uncaught exception in a .NET program – specifically, in a .NET 3.5 program, target runtime v2.0.50727 – results in a program exit code of 0, i.e. success! What a load of crap!

Java

You might think, ah well, .NET is really just Windows for people with large PCs and plenty of time to wait for programs to start up, do their thing,

The Java program failed in the simplest way, but either doesn't know, or doesn't tell

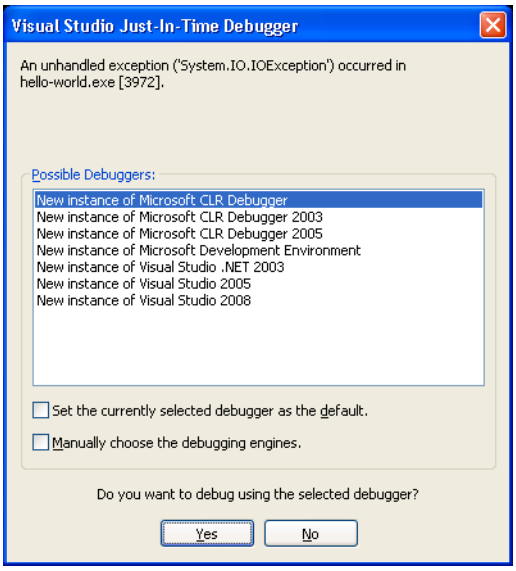


Figure 1

and then shut down again. Never intended for operating systems with sophisticated command-line processing anyway, so it's no loss. Now, Java, that'll show 'em how it's done.

Yeah? Well, prepare for some crow pie. Consider the following program.

```
class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello, JWorld!");
    System.out.flush();
  }
}
```

With Java 1.6.0_05, this fails on all counts. It does not throw an exception when the `flush()` fails to write all 15 or 16 bytes to the standard output stream. Nor does it cause the java.exe process to return an non-zero exit code. It's impossible to know whether the Java runtime even detects the write/flush failure, or just that it doesn't think it worth mentioning. The Java program failed in the simplest way, but either doesn't know, or doesn't tell. Either way, it's a pathetic effort!

Mark both VM languages down as not intended for command-line programming (which, to be fair, we pretty much knew anyway, given the long load times for even the simplest programs).

```
Unhandled Exception: System.IO.IOException: There is not enough space on the disk.
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.__ConsoleStream.Write(Byte[] buffer, Int32 offset, Int32 count)
   at System.IO.StreamWriter.Flush(Boolean flushStream, Boolean flushEncoder)
   at System.IO.StreamWriter.Write(Char[] buffer, Int32 index, Int32 count)
   at System.IO.TextWriter.WriteLine(String value)
   at System.IO.TextWriter.SyncTextWriter.WriteLine(String value)
   at hello_world.Program.Main(String[] args) in
H:\Publishing\Articles\accu\columns\QualityMatters\6-exceptions\code\hello-
world\c#\hello-world\Program.cs:line 10
```

Figure 2

Python

Thankfully, once we come back to the realm of languages that are intended for command-line programs, things get better. The following Python program

```
import sys
print "hello,
   untyped world!"
sys.stdout.flush()
```

gives the output in Figure 3 and the exit code 2.

```
Traceback (most recent call last):
  File "H:\Publishing\Articles\accu\columns\QualityMatters\6-
exceptions\code\hello-world\python\hello-world.py", line 4, in <module>
    sys.stdout.flush()
IOError: [Errno 28] No space left on device
```

Figure 3

Ruby

Similarly, the following Ruby program

```
puts "hello, bejewelled
   world!"
$stdout.flush
```

gives the output in Figure 4 and the exit code 1.

```
H:/Publishing/Articles/accu/columns/QualityMatters/6-exceptions/code/
hello-world/ruby/hello-world.rb:2:in `flush': No space left on device
(Errno::ENOSPC)
      from H:/Publishing/Articles/accu/columns/QualityMatters/6-
exceptions/code/hello-world/ruby/hello-world.rb:2
```

Figure 4

Although neither of the script programs by default give the totally nice and neat output I was seeking – something like '<script-name>: No space left on device' – they both get pretty close. All the compiled languages fail miserably.

Diagnostic logging and contingent reporting is done in least complex manner possible

A production-quality main()

Clearly, the idea of not explicitly handling unrecoverable conditions at `main()`-level in any major language is not going to cut the mustard. Even though it's virtually never covered in text books – which describe only chocolate worlds with marshmallow skies and champagne rain – we must now put aside childish things, and look at doing things properly.

Ok, ok, I'm laying on the opprobrium a bit thick. I understand the need to have succinct examples in textbooks, and I *know* how hard had it is to pack information into a small space and have it still readable, but this whole situation is just not good enough.

In an attempt to redress the balance, and to put my money where my mouth is, I'm volunteering the `main()` I'm using with all the command-line tools I'm creating/enhancing in my main stream of work this year. It's not perfect – it highlights several issues I'll cover later in this instalment, and it also touches on software quality issues yet to be explored in Quality Matters – but it does address the major concern of diagnosis. Take a deep breath, then look at Listing 1.

```
char const TOOL_NAME[];
clasp::alias_t const aliases[] =
{
 . . .
};
int tool_main(clasp::arguments_t const* args);
int main(int argc, char** argv)
{
 struct clasp_log
 {
  static void CLASP_CALLCONV fn(
    void*       /* context */
  , int         severity
  , char const* fmt
  , va_list     args
  )
  {
   pan::pantheios_logvprintf(severity,
      fmt, args);
  }
 };
 try
 {
  clasp::diagnostic_context_t ctxt(NULL,
     &clasp_log::fn, NULL);
  clasp::arguments_t const* args;
  int flags = 0;
  int argsres = clasp::parseArguments(flags,
     argc, argv, aliases, &ctxt, &args);
```

Listing 1

```
  if(0 != argsres)
  {
   pan::log_ALERT("failed to process the command-
line arguments: ", stlsoft::error_desc(argsres));
  }
  else
  {
   stlsoft::scoped_handle<clasp::arguments_t
      const*> scoper(args,
      &clasp::releaseArguments);
   pan::log_DEBUG("entering main(
      ", pan::args(argc, argv), ")");
   return tool_main(args);
  }
 }
 // 1. Always catch bad_alloc first
 catch(std::bad_alloc&)
 {
  pan::logputs(pan::alert, "out of memory");
  ::fputs("out of memory\n", stderr);
 }
 // 2. CLASP
 catch(clasp::unused_argument_exception& x)
 {
  pan::log_INFORMATIONAL("unrecognised command-
line argument: ", x.optionName);
  ff::fmtln(std::cerr, "{0}: invalid argument:
     {1}; use --help for usage", TOOL_NAME,
     x.optionName);
 }
 catch(clasp::clasp_exception &x)
 {
  pan::log_INFORMATIONAL(
     "invalid command-line: ", x);
  ff::fmtln(std::cerr, "{0}: invalid command-
line: {1}; use --help for usage", TOOL_NAME, x);
 }
 // 3. recls
 catch(recls::recls_exception& x)
 {
  pan::log_CRITICAL("exception: ", x);
  ff::fmtln(std::cerr, "{0}: {1}, item={2};
     use --help for usage", TOOL_NAME, x,
     x.get_item());
 }
 // 4. Other standard exceptions
 catch(std::exception& x)
 {
  pan::log_CRITICAL("exception: ", x);
  ff::writeln(std::cerr, TOOL_NAME, ": ", x);
 }
```

Listing 1 (cont'd)

our only contingent action has been to
indicate to the caller, via the return code,
that the program has failed

```
// 5. ...
#ifdef CATCH_UNHANDLED
 catch(...)
 {
  pan::logputs(pan::emergency,
     "unexpected unknown condition");
  ::fputs("unexpected unknown condition\n",
     stderr);
 }
#endif /* CATCH_UNHANDLED */
 return EXIT_FAILURE;
}
```

Listing 1 (cont'd)

Obviously, there's quite a lot going on here, and some of the points encroach on issues that will be covered in later instalments, particularly the minutiae of diagnostic logging content, format and severity. So I'll focus solely on aspects that pertain to exceptions. Before I enumerate the points, I need to cover what is pretty obvious from the code, that it uses the as-yet-unreleased **CLASP** library (**C**ommand **L**ine **A**rgument **S**orting and **P**arsing), which I've mentioned a couple of times here (and in recent CVu articles). Without getting too much into it, the modus operandi of CLASP use is to parse the arguments into a (read-only) arguments structure, which is then passed to a 'real' entry point, which I tend to call `tool_main()`. All the code within the try-clause is pretty self-explanatory; the only thing worth mentioning is my favourite *local-struct-static-method* trick for defining (context-free) local functions, in this case allowing **Pantheios**' logging facilities to be used by CLASP.

Anyway, the important (and relevant) features are all in the catch handlers. In brief:

- Always catch most-derived first, except for special cases.
- All catch-clauses fall out to the single return `EXIT_FAILURE` at the end of `main()`.
- One special-case always present is to catch `std::bad_alloc` before anything else. Diagnostic logging and contingent reporting is done in least complex manner possible, since the program just experienced out-of-memory condition: note the use of C's Streams to avoid the C++ free store [TC++PL].
- *In this case*, since I'm using CLASP for the tools, catch for CLASP exceptions to deal with badly specified command-lines, including, specifically, unrecognised arguments. This relies on using a specific member of the CLASP exception class, one that is not part of the ancestor class(es), hence the need for specific catch clauses, in a specific order.
- *In this case*, since I'm using recls for file-search, catch for recls exceptions to deal with unrecoverable file-system issues, including the specification by the user of invalid directories/patterns.
- Catch `std::exception` last. In principle – on the assumption that every exception type thrown by the program or any of its constituent libraries derives from it `std::exception` – any other exception that is thrown anywhere in the program will be caught here, and a minimum amount of information output.
- The catch-all clause is only included if `CATCH_UNHANDLED` is `#define`d, which it is not by default; see sidebar 'Why Catch-All Clauses are Bad News'. Like the catch for `std::bad_alloc`, very little is attempted, since it must be assumed that the program is *faulted*, and that *nothing* can be relied upon.
- Diagnostic logging and contingent report statements are always provided, and their format and content may differ depending on the likely needs of their respective audiences.
- Diagnostic logging always appears before contingent reporting, since it's possible that the contingent report statements may themselves fail. (Of course, it's also possible that the diagnostic logging statements themselves fail, but with any good diagnostic logging library that is (i) far less likely, and (ii) non-faulted failures are not reported, and therefore do not prevent continuing program execution.)

There's an awful lot more here than in any *hello-world* you're ever likely to see in a C++ textbook. More than one reviewer complained that this example was too much, perhaps even that I am grandstanding. Well, I'm a programmer, so there's bound to be a little of that. But I am making a serious point here: real programs require a substantial amount of contingent logic, invariably involving general and domain-specific cases. Showing you the (only minimally simplified) implementation of a real program keeps it real.

There's actually a better way to do this, involving separation of the general from the domain-specific handling. But it's not as simple as might be imagined, and requires knowledge of issues not yet covered, so I'll leave that for the concluding fourth instalment. For now, Listing 1, while being elaborate, is a solid example of how to have your program handle practically-unrecoverable exceptions.

Implementing C-APIs in C++

Readers of part 2 of *Imperfect C++* [IC++] may recall my assertion that C++ is a fine language for application code and for library implementations, but is often a poor choice for module interfaces, particularly so where programs may be composed of modules compiled by different compilers. This is the C++ ABI issue.

Distilled down to this subject, it's not valid to throw exceptions through C-APIs. A common example of this circumstance is the implementation of COM servers, such as Windows shell extensions. There's no good outcome of letting an exception leak out of any COM interface method: about the best you can hope for is to crash Windows Explorer when it's not doing anything useful.

Therefore, when writing COM in C++, the considerable challenge is to make sure that every possible exception is caught and translated into an appropriate `HRESULT` (the COM result type). Also important is to capture the non-normative action context information, whether for the purposes of

After earlier arguing strongly against quenching exceptions in catch-all clauses, you might wonder why I give users the option

diagnostic logging or contingent reporting. If you imagine a component involving several interfaces and many methods, applying try-catch-…-catch everywhere is a recipe for boredom, mistakes, defects, faults, crashes, career-impacts.

Because COM has a well-defined set of result codes, it is possible to prescribe the appropriate set of responses for a small number of high-level exceptions, covering all common possibilities. The Pantheios library provides a suite of function templates that combine these exception->catch->return-code translations along with diagnostic logging statements, enabling the authoring of logged, exception-safe COM servers without being swamped with boilerplate. I'll illustrate with a short example, from **recls.COM**, the COM mapping for recls; Listing 2. (This version is part of a back-burner rewrite, and not yet available. One day …)

```
// recls.COM.idl
interface IFileSearch3
    : IFileSearch2
{
  . . .
  HRESULT CombinePaths(
    [in, string] BSTR path1
  , [in, string] BSTR path2
  , [out, retval] BSTR *result);
  . . .
// FileSearch.h
class FileSearch
  : IFileSearch3
{
  . . .
  STDMETHOD(CombinePaths)(BSTR path1, BSTR path2,
     BSTR *result);
  . . .
private:
  HRESULT CombinePaths_(BSTR path1, BSTR path2,
     BSTR *result);
// FileSearch.cpp
STDMETHODIMP FileSearch::CombinePaths(BSTR path1,
   BSTR path2, BSTR *result)
{
  return pantheios::extras::com::
     invoke_nothrow_method(this,
     &FileSearch::CombinePaths_, path1, path2,
     result, "CombinePaths");
}
HRESULT FileSearch::CombinePaths_(BSTR path1,
   BSTR path2, BSTR *result)
{
  . . . do "normal" C++, incl. exceptions
}
```

Listing 2

This is all straightforward COM/C++, with the exception of the use of **pantheios::extras::com::invoke_nothrow_method()**, in **FileSearch::CombinePaths()**. This function template (Listing 3) is one of several overloads that cope with different numbers of parameters, providing a similar set of catch-clauses as that shown earlier for the 'production-quality' **main()**. Although it looks like a complex affair, it's actually pretty simple. Call the given member function within a try-catch block (if the compiler's exception-handling support is not disabled), and deal with any exceptions that are thrown. Three conditions are discriminated, via the catch clauses:

- Out of memory. If the exception is **std::bad_alloc**, or the COM component itself returns the **E_OUTOFMEMORY** status code, then a basic log statement is issued and **E_OUTOFMEMORY** is returned to the caller. When compiling in the presence of MFC, it also catches **CMemoryException*** and treats it in the same manner.
- A general exception, caught as **std::exception**, and, in the presence of MFC, **CException***. The exception details (implicitly obtained from the exception instances via string access shims [IC++, XSTLv1, FF-2] by the Pantheios application layer) are included in the diagnostic log statement.
- Everything else, via the catch-all clause. A suitably troubling diagnostic log statement is issued. As discussed previously, catching 'everything' is fraught with danger, so conditional compilation requires intentional buy-in from the programmer to convert into a return code, and even to rethrow; by default, the process is terminated with a call to **ExitProcess()**. Severe, but the only sensible default.

There's also the facility for allowing user-defined catch clauses, via the tersely named **PANTHEIOS_EXTRAS_COM_EXCEPTION_HELPERS_CUSTOM_CLAUSE_0/1** macros.

After earlier arguing strongly against quenching exceptions in catch-all clauses, you might wonder why I give users the option of what behaviour to take. Well, it's just pragmatism, I suppose: it's not possible to know the nature of every use case. For example, it's possible that a COM component's methods cannot emit any operating-system exceptions because they're already using structured exception handlers [Richter], in which case a programmer may wish to capture other (C++) exception types via the catch-all clause.

Other than the restriction that the implementing method must have the exact same signature as the interface method, and that overloads can be a bit of a hassle, this is a pretty big gain for almost no pain.

Furthermore, the destination of the diagnostics here is, in common with any Pantheios client code, independent of the server code; output decisions can be made, for each link-unit, at compile, link, or even run-time. You can log to a file (via the back-end **be.file**) and/or the Windows system debugger (**be.WindowsDebugger**) and/or the Windows Event Log (**be.WindowsEventLog**), and so on. What's especially useful when writing COM servers is to also use **be.COMErrorObject**, which writes the details of the diagnostic log statement to the COM Error Object, a per-

we've considered the ubiquitous 'hello-world' program in a variety of languages, and seen a number of inadequacies

```
template<
  typename R
, typename C
, typename A0
, typename A1
, typename A2
>
inline R invoke_nothrow_method(
  C *pThis
, R (C::*pfn)(A0, A1, A2)
, A0 a0
, A1 a1
, A2 a2
, char const* functionName
)
{
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
  try
  {
  #endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
    HRESULT hr = (pThis->*pfn)(a0, a1, a2);
    if(E_OUTOFMEMORY == hr)
    {
      goto out_of_memory;
    }
    return hr;
  #ifdef STLSOFT_CF_EXCEPTION_SUPPORT
  }
  catch(std::bad_alloc&)
  {
    goto out_of_memory;
  }

PANTHEIOS_EXTRAS_COM_EXCEPTION_HELPERS_CUSTOM_CLA
USE_0
  catch(std::exception& x)
  {
    log(alert, functionName, ": exception: ", x);
    return E_FAIL;
  }
  # ifdef __AFX_H__
  catch(CMemoryException* px)
  {
    px->Delete();
    goto out_of_memory;
  }
  catch(CException* px)
  {
    log(alert, functionName, ": exception: ",
       *px);
    px->Delete();
```

Listing 3

```
    return E_FAIL;
  }
# endif /* __AFX_H__*/

PANTHEIOS_EXTRAS_COM_EXCEPTION_HELPERS_CUSTOM_CLA
USE_1
  catch(...)
  {
    log(critical, functionName,
       ": unexpected exception!");
# if defined(
  PANTHEIOS_EXTRAS_COM_ABSORB_UNKNOWN_EXCEPTIONS)
    return E_UNEXPECTED;
# elif defined(
 PANTHEIOS_EXTRAS_COM_RETHROW_UNKNOWN_EXCEPTIONS)
    throw;
# else
    ::ExitProcess(EXIT_FAILURE);
# endif
  }
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
out_of_memory:
  log(alert, functionName, ": out of memory");
  return E_OUTOFMEMORY;
```

Listing 3 (cont'd)

thread global 'error' context that can be queried by any part of the program. This is a standard mechanism for COM Automation servers to pass so-called 'rich error information' to clients. Clients of any COM servers written using Pantheios can receive such detailed context information about non-normative conditions, with virtually no additional effort from programmer even when it's contained within a thrown exception.

Summary

In this instalment we have begun the exploration of the use of exceptions in software programs and plug-in components. We've examined in detail the effect of uncaught exceptions reporting unrecoverable conditions, and shown that all quality programs must use explicit try-catch code at the application's top-level. Specifically, we've considered the ubiquitous *hello-world* program in a variety of languages, and seen a number of inadequacies in regards to whether failure to write to standard output is detected and, if so, whether it's reported and reflected in program exit code.

In C, it's necessary to explicitly test and report. In other languages that use exceptions, a reliance on the implicit handling of a thrown exception from the chosen output library is incomplete, to different degrees. In C++, execution is terminated and the program exit code reflects the failure, but no (precise) report is provided. Also, if you're using IOStreams you must do an explicit check, or remember to enable exceptions on the stream. C# and Java both record an epic fail. Only Python and Ruby could claim to satisfy the basic requirements of software quality, although even then one would prefer to explicitly handle the exception for the sake of neatness.

it's clear that a program that does not have an explicit top-level try-catch is inadequate

In all cases, it's clear that a program that does not have an explicit top-level try-catch is inadequate. A production-quality `main()` is non-trivial, involving generic and domain-specific aspects. Its order of catch clauses is important. It should provide adequate and appropriately-targeted reporting of the non-normative conditions that have resulted in the caught-exceptions.

We've looked at two different types of reporting – contingent reports and diagnostic log statements – and seen how derived exception classes can (and should) carry additional information to assist with detailed reporting, both for contingent reporting and for diagnostic logging. We'll follow up on this point further in the coming instalments.

We've seen that when implementing C-APIs, exceptions must be caught and translated into return codes, with appropriate contingent reporting and diagnostic logging. This must be done even in the case where an exception represents an unrecoverable condition (e.g. out-of-memory), and the implementer of such an API must trust that its clients faithfully examine and respond to its return codes.

We've considered the use of catch-all blocks in C++, intended to be able to catch any unhandled exception of whatever type, and seen that, 'enhanced' to be able to catch operating-system exceptions on a compiler-specific basis, throwing/catching any type not derived from `std::exception` is a problem.

In the next instalment we'll consider the use of exceptions for recoverable conditions, using some real world examples from my recent work. With one C++ program, we'll see the complexities involved in working with cache memory allocation failures, and the difficulties this brings in deciding what is recoverable and what is practically-unrecoverable. With another, I'll demonstrate that .NET programming for networking software is not the least bit easy as 1-2-3, and have a big swipe at the shipwreck that is .NET's exception hierarchy and the unjustifiable difficulties it imposes on programmers attempting to write robust and cleanly abstracted software.

Parting twist

There's one little perverse aspect to note about the abysmal performance of all the different languages to adequately recognise and/or report a failure of *hello-world*. At least with C, nothing (save from the flushing/closing of files and release of resources back to the operating system) is implicit, so no implicit help is expected. By contrast, the fact that much is implicit with more expressive languages such as C++, C#, and Java has, I believe, lead us to a false expectation, albeit not an unreasonable one in the case of *hello-world*. I conjecture that experienced C programmers may be less caught out by this than experienced programmers in other languages, precisely because their expectations are so much lower. I'd be interested to hear opinions on this, perhaps on the ACCU general mailing list after this is published. ■

Acknowledgements

Thanks to Chris Oldwood, Ric Parkin, and the members of the Overload review team, for helping me out despite another hair's breadth skirting of the deadline.

References and asides

[FF-1] An Introduction to FastFormat, part 1: The State of the Art, Matthew Wilson, Overload 88, February 2009

[FF-2] An Introduction to FastFormat, part 2: Custom Argument and Sink Types, Matthew Wilson 89, April 2009

[FF-3] An Introduction to FastFormat, part 3: Solving Real Problems, Quickly, Matthew Wilson 90, June 2009

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

[K&R] The C Programming Language, Brian Kernighan and Ken Ritchie, Prentice-Hall PTR, 1988

[QM-2] Quality Matters, Part 1: Correctness, Robustness and Reliability, Matthew Wilson, Overload 93, October 2009

[QM-5] Quality Matters, Part 5: Exceptions: The Worst Form of 'Error' Handling, Except For All The Others, Matthew Wilson, Overload 98, October 2010

[Richter] Advanced Windows, Jeffrey Richter, Microsoft Press, 1997.

[STRERROR] Safe and Efficient Error Information, Matthew Wilson, CVu, July 2009

[TC++PL] The C++ Programming Language, Special Edition, Bjarne Stroustrup, Addison-Wesley, 2000

[XSTLv1] Extended STL, volume 1: Collections and Iterators, Matthew Wilson, Addison-Wesley, 2007