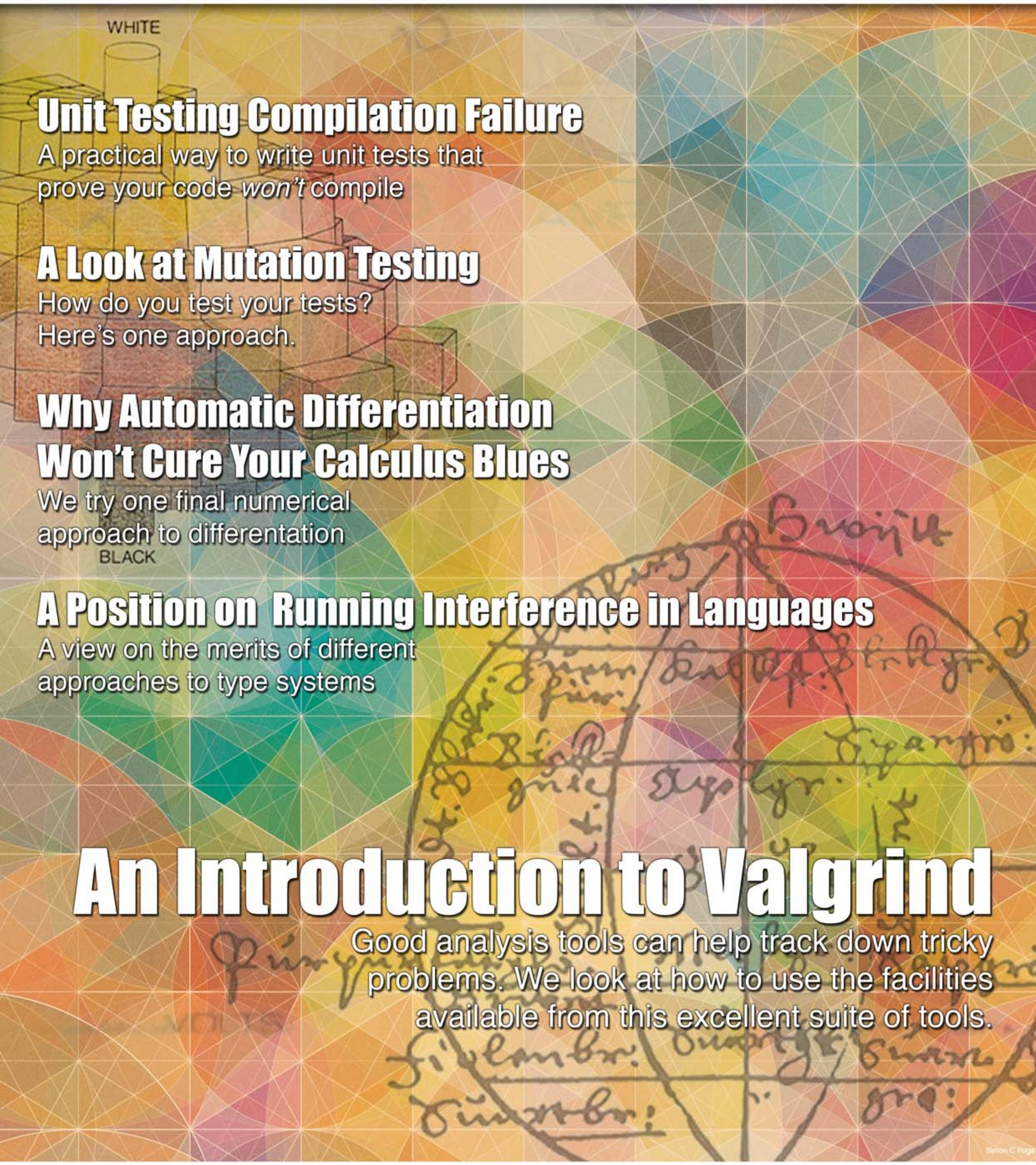
overload 108

APRIL 2012 £3



Unit Testing Compilation Failure

A practical way to write unit tests that prove your code *won't* compile

A Look at Mutation Testing

How do you test your tests? Here's one approach.

Why Automatic Differentiation Won't Cure Your Calculus Blues

We try one final numerical approach to differentation

A Position on Running Interference in Languages

A view on the merits of different approaches to type systems

An Introduction to Valgrind

Good analysis tools can help track down tricky problems. We look at how to use the facilities available from this excellent suite of tools.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 108

April 2012

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 109 should be submitted by 1st May 2012 and for Overload 110 by 1st July 2012.

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

ACCU

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

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

The Computing Revolution Will Be Televised (Again)

30 years ago we had to plug our first computers into the TV. Ric Parkin looks at how far we've come since then.

April 23 [April23] will be a significant date for many people of my age. Many may know that it's St George's day, a few might even know that it's also the birthday of Max Planck, Prokofiev, and Shakespeare (and also the day he died). But to a generation who are now in IT, a more personal reason is that it is also the day, 30 years ago, that the ZX Spectrum was launched [ZXSpectrum].

This was a time of frantic innovation in the home computer industry – the BBC Micro [Beeb] had come out the previous December, and the Commodore 64 [C64] had been announced and would come out later in August – but for me it'll always be the Speccy. (And I wasn't the only one – see Andy Thomas' personal reminiscences in this issue). It was special for me because I'd saved up and ordered one of the first 48K models. While its rivals were technically superior, I couldn't afford them on my pocket money and 50p a week paper round – even the £175 price of the Spectrum was a significant amount in those days. Of course it didn't arrive for ages – there was a massive waiting list, and production problems meant it took around 6 months before I finally got this small black box with weird rubbery keys. And you had to plug it into the television, which in the era of single-tv households basically meant that no one else could watch anything while you were using it, and you had to hunch down on the floor in front of it because the lead was too short.

But why on earth did I want one? I already had some experience of computers, probably more than most – doing some BASIC programming on VAXs at my father's work, and we had an enthusiastic Chemistry teacher at school who'd brought in a ZX80 and ZX81 for us to see. The school itself had a pretty good 'computer lab' from an early age, with an Apple II and Commodore Pet. In an old stationery cupboard. Even in those days computer rooms were where the people with strange obsessions would hide from the light (mainly to avoid screen glare, but could you convince people of that?)

Most people seemed to want these machines for playing games – this was the peak era of arcade games such as Space Invaders, PacMan, and Defender. Now you could play them (or highly derivative versions) without having a pocket bulging with ten pence pieces. But there was one odd thing about these machines – when you turned them on, they just sat there with a fairly blank screen with cryptic messages and a flashing cursor, and you had to ask them to load your game. The command prompt was king. This was a serious hindrance to casual users, requiring much patient instruction of the baffled elders by the technically savvy children. But it did show the idea that computers were there to be instructed – powerful but dumb machines that required a human to set a task and steer them through it.

Did I just say 'powerful'? That's a relative term, of course. Compared to now they were shockingly puny, but for the time they were amazing feats of engineering, which as we all know is all about compromise and squeezing a quart into a pint pot. Some of the tricks that were used are ingenious, both in the hardware and the software. I've even seen it argued that the reason British software engineering was so good for a while (especially for games) was we couldn't afford the bigger and more powerful PCs and other computers where you just let its raw power take the strain, whereas with some of the compromises and limitations of our homegrown ones you had to *really* try to get anything worthwhile done. So you'd get some truly amazing achievements such as the classic Elite [Acornsoft] and Psion Scrabble, which packed a 11000 word dictionary as well an excellent graphical board and game engine, into 48K.

Yes, I played the games. I bought the magazines (some of which are quietly going mouldy in my garage). I typed in the listings, but also worked out how to modify them. I learnt patience by having to type in pages of hex-dumps over hours to find I'd made a single typo and had to redo everything. And I tried programming myself. It was easy. Who didn't know how to nip into Dixons and quickly bash out something like

```
10 Print "Ric Woz 'ere"
20 Goto 10
```

much to the annoyance of the staff?

I did more serious stuff of course – music playing programmes taking the notes from a table, a competitive version of the 'Simon' reaction game for the school's parents evening, to show off how high tech we were (in Pascal using my first compiler, and a hardware controller to detect when a physical circuit was made by the player hitting metal plates), and as a hardware controller for my Design Technology 'O' Level project.

But I never took the computing 'O' level. I was actually advised not to by the teacher when I asked if I should, who said 'You know it all already'.

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.

Perhaps I should have just asked to sign up for the exam without the lessons.

But this reminds me to a very recent debate. What *should* you teach about IT in schools? There still exist exams in computer science [O Level], but they tend to be for overseas students, and in the UK at GCSE level there's only ICT [GCSE]. Unfortunately these are widely disliked and thought to be dull, off-putting, and not even very good at preparing people to use computers at work – things change so quickly that almost by definition the course is going to be badly out of date, and students who have grown up in the era of widely available computers and smartphones are already going to know an awful lot already.

Thankfully there's an effort to revamp the curriculum and re-introduce proper computing courses [CodeInClass], but we have to accept that it's not going to be interesting to everyone. Let's face it, programming is always going to be fairly niche activity. But what inspired me all those years ago was having this machine that I could command to do stuff quickly and easily, and it would be great to expose people to that to see who's interested in doing the more advanced courses. Thinking back there were languages designed to get quick fun results [Logo], so reintroducing them or more modern languages with a quick off-the-shelf environment to get quick and easy feedback would be a great start. For example, the OU has based its new introductory language, SENSE, on MIT's Scratch [MIT], and the students get a little board to play with too. This is only for an introductory course, but quickly covers all the basics to allow people to go on to further study. For a quick taste of how the language and environment works, there are useful videos demonstrating it in action [TU100].

One of things I love about programming is that it is in fact very creative – you're designing and building something – and if you can get students to experience that buzz then you'll get more and better people going into what is a very important industry.

And it's not just the languages. There's been some excitement around a project to develop extremely low cost computing again – the Raspberry Pi [Pi]. This project was set up by a bunch of the people who'd experienced the original home computer boom (in fact one of the authors of Elite is involved) and wanted to bring some of that excitement and immediacy back. In the age of expensive and over complex PCs, laptops, smartphones and Games Consoles, they'd found that a lot of people were scared to fiddle with them in case they 'broke' them (and get told off), and the development suites were expensive and hugely complex. So their idea was to produce a really cheap simple computer (£25, which means pretty much anyone can afford to just get one on a whim) running open source software, and get the wider community to do something with them. In many ways it's even simpler than the old home computers: while the processor and memory is much better, it's been designed to be a flexible core of a system, so you need to add a keyboard and (once again) plug it into a TV. At least this time around many people will have compatible monitors so you don't have to monopolise the only TV in the house.

And there's already many ideas [Forum], from using python to make enjoyable programming education packages, to running in-car entertainment systems, emulators, and the core of many homebrew hardware controllers. Given the rollout of IPv6, we are getting close to the long predicted Internet Of Things, where masses of small cheap devices connect and collaborate, and tiny boards such as the Pi could be a first taste of this. Ultimately these will need to become entire systems on a chip, including CPU, memory and Wifi connection, and perhaps even generate their own power from ambient energy [EnergyHarvesting]. Then things will get interesting.

But the most fun thing I've seen so far is someone has got a ZX Spectrum emulator running on the Raspberry Pi [Emulator]. Nostalgia has never been more cutting edge. ■

References

[Acornsoft] http://www.bbcmicrogames.com/acornsoft.html

[April23] http://en.wikipedia.org/wiki/April_23

[Beeb] http://bbc.nvg.org/

[C64] A new one! http://www.commodoreusa.net/CUSA_C64.aspx

[CodeInClass] http://www.bbc.co.uk/news/technology-17373972

[Emulator] http://www.retrocomputers.eu/2012/03/23/raspberry-pi-running-the-fuse-zx-spectrum-emulator/

[EnergyHarvesting] http://en.wikipedia.org/wiki/Energy_harvesting

[Forum] http://www.raspberrypi.org/forum/projects-and-collaboration-general/the-projects-list-look-here-for-some-ideas

[GCSE] http://www.cie.org.uk/qualifications/academic/middlesec/igcse/subject?assdef_id=969

[Logo] http://mckoss.com/logo/

[MIT] http://scratch.mit.edu/

[O Level] O Level computing: http://www.cie.org.uk/qualifications/academic/middlesec/olevel/subject?assdef_id=904 and http://www.edexcel.com/quals/olevel/7105/Pages/default.aspx

[Pi] http://www.raspberrypi.org/

[TU100] http://www.youtube.com/watch?v=-gcG5UuYZZk

[ZXSpectrum] http://en.wikipedia.org/wiki/ZX_Spectrum

Image

The image is of a Sinclair 48K ZX Spectrum, and was taken by Bill Bertram: http://commons.wikimedia.org/wiki/User:Pixel8

Why Automatic Differentiation Won't Cure Your Calculus Blues

We've tried and rejected many numerical approaches to differentiation. Richard Harris has one final attempt.

At the beginning of the second half of this series we briefly covered the history of the differential calculus and described the incredibly useful Taylor's theorem, which states that for a function *f*

$$\begin{aligned} f(x+\delta) &= f(x)+\delta\times f'(x)+\tfrac{1}{2}\delta^2\times f''(x)+\dots \\ &\quad +\tfrac{1}{n!}\delta^n\times f^{(n)}(x)+R_n \\ \min\left(\tfrac{1}{(n+1)!}\delta^{n+1}\times f^{(n+1)}(x+\theta\delta)\right) &\leq R_n \\ &\leq \max\left(\tfrac{1}{(n+1)!}\delta^{n+1}\times f^{(n+1)}(x+\theta\delta)\right) \quad \text{for } 0\leq\theta\leq 1 \end{aligned}$$

where *f'*(*x*) denotes the first derivative of *f* at *x*, *f''*(*x*) denotes the second and *f*(*n*)(*x*) the n^{th} and where, by convention, the 0^{th} derivative is the function itself.

We have so far used it to perform a full error analysis of finite difference algorithms and to automate the calculation of their terms with polynomial curve fitting.

We went on to describe the far more accurate polynomial approximation of Ridders' algorithm [Ridders82] which treated the symmetric finite difference as a function of the difference δ.

In the previous article we used expression objects to represent functions as expression trees, from which it was possible to compute expression representing their derivative by applying the relatively simple algebraic rules of differentiation.

Noting that there was little we could do to control cancellation error in the unknown expression for the derivative, assuming we hadn't the appetite to implement a full blown computer algebra system that is, we concluded by combining this with interval arithmetic to automatically keep track of such numerical errors.

Having claimed that short of just such a system our algorithm was as accurate as it was possible to get, you may be a little surprised that there is anything left to discuss.

However, as I also noted, expression objects are fairly memory intensive due to the need to maintain the expression tree. Furthermore, a naïve implementation such as ours is computationally expensive since it puts cheap built in arithmetic operations behind relatively expensive virtual function calls. The latter point isn't particularly compelling since we can in many cases remove the virtual functions by instead using templates.

Finally, we must take care when using expression objects in conjunction with numerical approximations. As was demonstrated, it is quite possible to accurately approximate one function with another whose derivative is significantly different.

Whilst expression objects provide a powerful method for the calculation of derivatives, we should ideally seek an equally accurate algorithm that doesn't require as much memory and uses no virtual functions or heavily nested template types.

The question stands as to whether such an algorithm exists.

Given that I'm still writing, it should be more or less self-evident that it does.

And, in another testament to its tremendous power, it's based entirely upon Taylor's theorem.

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.

Surreal numbers

Like Robinson's non-standard numbers [Robinson74], Conway's surreal numbers [Knuth74] extend the reals in a way that admits a rigorous definition of infinitesimals.

So what's to stop us from actually using infinitesimals in our calculations?

Nothing. That's what!

Much as we extend the reals to the complex numbers by representing numbers with both a real and an imaginary part, we can extend the reals to the surreals by representing numbers with both a real and an infinitesimal part.

$$w = a + b\delta$$

Note that we don't actually need to define δ so long as it stands for the same infinitesimal throughout our calculations.

Listing 1 gives a C++ definition of just such a numeric type.

Listing 2 provides the constructors and property accessors of the **`surreal`** class.

```
class surreal
{
public:
  surreal();
  surreal(double a);
  surreal(double a, double b);

  double real() const;
  double inf() const;

  int       compare(const surreal &w) const;
  surreal & negate();

  surreal & operator+=(const surreal &w);
  surreal & operator-=(const surreal &w);
  surreal & operator*=(const surreal &w);
  surreal & operator/=(const surreal &w);

private:
  double a_;
  double b_;
};
```

Listing 1

a naïve implementation such as ours is computationally expensive since it puts cheap built in arithmetic operations behind relatively expensive virtual function calls

```
surreal::surreal() : a_(0.0), b_(0.0)
{
}

surreal::surreal(double a) : a_(a), b_(0.0)
{
}

surreal::surreal(double a, double b) : a_(a),
                                       b_(b)
{
}

double
surreal::real() const
{
  return a_;
}

double
surreal::inf() const
{
  return b_;
}
```

Listing 2

The **compare** member function first compares the real part of the number and, if they are equal, then compares the infinitesimal part, as shown in listing 3.

Negation, addition and subtraction are trivial operations in which we perform the same action on both parts of the number

$$w_1 = a_1 + b_1\delta$$
$$w_2 = a_2 + b_2\delta$$

$$-w = -a - b\delta$$
$$w_1 + w_2 = (a_1 + a_2) + (b_1 + b_2)\delta$$
$$w_1 - w_2 = (a_1 - a_2) + (b_1 - b_2)\delta$$

The implementation of these operations is given in listing 4.

```
int surreal::compare(const surreal &w) const
{
  if(a_<w.a_) return -1;
  if(a_>w.a_) return  1;
  if(b_<w.b_) return -1;
  if(b_>w.b_) return  1;
  return 0;
}
```

Listing 3

```
surreal &
surreal::negate()
{
  a_ = -a_;
  b_ = -b_;
  return *this;
}

surreal &
surreal::operator+=(const surreal &w)
{
  a_ += w.a_;
  b_ += w.b_;
  return *this;
}

surreal &
surreal::operator-=(const surreal &w)
{
  a_ -= w.a_;
  b_ -= w.b_;
  return *this;
}
```

Listing 4

Multiplication proceeds much as it does for complex numbers, although we shall discard the term involving the square of *δ*.

$$\begin{aligned} w_1 \times w_2 &= (a_1 + b_1\delta) \times (a_2 + b_2\delta) \\ &= a_1 \times a_2 + a_1 \times b_2\delta + a_2 \times b_1\delta \\ &= (a_1 \times a_2) + (a_1 \times b_2 + a_2 \times b_1)\delta \end{aligned}$$

Listing 5 provides the implementation of the multiplication operator.

Division would seem to be a slightly trickier proposition; we're ignoring second and higher order powers of *δ*. If the product of two of our surreal number has a second order term, surely we'll need it when dividing the product by one of the arguments if we are to have any chance of recovering the other. Or of recovering the correct result when dividing it by *any* other surreal number, for that matter.

Well, appearances can be deceptive.

```
surreal &
surreal::operator*=(const surreal &w)
{
  b_ = a_*w.b_ + w.a_*b_;
  a_ *= w.a_;
  return *this;
}
```

Listing 5

something about this **sidestepping** of the need to retain **higher order infinitesimals** seems a little bit fishy

Surreal number division

The trick is one we're going to exploit time and again when implementing arithmetic operations for this type; no matter what real number we multiply δ by, it's *still* infinitesimal, which means that we can still use Taylor's theorem!

What we shall therefore do is work out the Taylor series expansion of reciprocation to first order. We can then multiply the numerator by the reciprocal of the denominator. The required series for the reciprocal is given by

$$\frac{1}{a+b\delta} = \frac{1}{a} + b\delta \times \left.\frac{d}{dx}\frac{1}{x}\right|_a$$

where the vertical bar means 'evaluated at the subscript'.

Hence our rule for reciprocation is

$$\frac{1}{a+b\delta} = \frac{1}{a} - \frac{b}{a^2}\delta$$

Note that this trick is effectively the chain rule for our `surreal` type in that it allows us to calculate the derivative of an expression in which one function is applied to the result of another.

The rule for division is therefore given by

$$\begin{aligned}\frac{w_1}{w_2} &= w_1 \times \frac{1}{w_2}\\ &= \left(a_1 + b_1\delta\right) \times \left(\frac{1}{a_2} - \frac{b_2}{a_2^2}\delta\right)\\ &= \frac{a_1}{a_2} + \left(\frac{b_1}{a_2} - \frac{a_1 \times b_2}{a_2^2}\right)\delta\end{aligned}$$

Now something about this sidestepping of the need to retain higher order infinitesimals seems a little bit fishy. To be truly comfortable that I haven't done something silly we should probably check that multiplying the ratio of w_1 and w_2 by w_2 yields w_1 as expected.

$$\begin{aligned}\frac{w_1}{w_2} \times w_2 &= \left(\frac{a_1}{a_2} + \left(\frac{b_1}{a_2} - \frac{a_1 \times b_2}{a_2^2}\right)\delta\right) \times \left(a_2 + b_2\delta\right)\\ &= \frac{a_1}{a_2} \times a_2 + a_2 \times \left(\frac{b_1}{a_2} - \frac{a_1 \times b_2}{a_2^2}\right)\delta + \frac{a_1}{a_2} \times b_2\delta\\ &= a_1 + \left(b_1 - \frac{a_1 \times b_2}{a_2}\right)\delta + \frac{a_1 \times b_2}{a_2}\delta\\ &= a_1 + b_1\delta\\ &= w_1\end{aligned}$$

```
surreal &
surreal::operator/=(const surreal &w)
{
  b_ = b_/w.a_ - a_*w.b_/(w.a_*w.a_);
  a_ /= w.a_;
  return *this;
}
```

Listing 6

Well, it's clear that my reservations were undeserved; I really should have had more faith in Taylor's remarkable result. We can clearly use this formula with confidence and the implementation in listing 6 does just that.

Whither algorithm?

Now this consistent extension of real arithmetic is all well and good, but how does it help us to develop a machine precision accurate algorithm for computing derivatives?

Well, the answer, perhaps a little unsurprisingly, lies in Taylor's theorem.

The `surreal` result of a function is a first order Taylor series expansion about an infinitesimal deviation from the value of that function. By Taylor's theorem, the coefficient by which δ is multiplied in the result of a function is therefore equal to its first derivative.

We can therefore trivially recover the derivative of a calculation by calling the `inf` member function on its result.

For example consider

$$\left.\frac{d}{dx}\frac{1}{x^2}\right|_2 = -\left.\frac{2}{x^3}\right|_2 = -\frac{2}{8} = -\frac{1}{4}$$

Using our `surreal` type we can simply divide 1 by the square of an infinitesimal deviation from 2

$$\begin{aligned}\frac{1}{(2+\delta)\times(2+\delta)} &= \frac{1}{(4+4\delta)}\\ &= \frac{1}{4} + \left(\frac{0}{4} - \frac{1\times 4}{16}\right)\delta\\ &= \frac{1}{4} - \frac{1}{4}\delta\end{aligned}$$

Clearly the coefficient of the infinitesimal part is equal to the required derivative.

This use of the Taylor series to compute derivatives is known as automatic differentiation, as the title of this article might have hinted at.

Further arithmetic operations

Before we can use our `surreal` type to compute the derivatives of functions in general, we shall need to implement the full suite of C++ arithmetic operations.

as we try to **compute higher and higher derivatives** we will inevitably be forced into using **complex recursive template trickery**

```
surreal
exp(const surreal &w)
{
  const double exp_a = exp(w.real());
  return surreal(exp_a, exp_a*w.inf());
}

surreal
log(const surreal &w)
{
  return surreal(log(w.real()),
                 w.inf()/w.real());
}
```

Listing 7

This would take up rather a lot of time, so we shall content ourselves with a few pertinent examples. Specifically

$$e^w$$

$$\ln w$$

$$w_1^{w_2}$$

The first two functions simply require the first order Taylor's series expansions

$$e^{a+b\delta} = e^a + b\delta \times e^a$$

$$\ln\left(a+b\delta\right) = \ln a + b\delta \times \frac{1}{a}$$

as illustrated in listing 7.

The third is a little less obvious, but we can use the same trick we used for our expression objects and rewrite it as

$$w_1^{w_2} = e^{w_2 \times \ln w_1}$$

as implemented in listing 8.

The remaining arithmetic operations are left as an exercise for the reader.

Higher order derivatives

An immediately obvious problem with our `surreal` type is that, as it stands, we cannot use it to compute second and higher order derivatives.

```
surreal
pow(const surreal &w1, const surreal &w2)
{
  return exp(w2*log(w1));
}
```

Listing 8

We might be tempted to change the `b_ member` from a `double` to a `surreal`, allowing us to recover the second derivative from the infinitesimal part of the infinitesimal part.

However, as we try to compute higher and higher derivatives we will inevitably be forced into using complex recursive template trickery which is something I specifically wanted to avoid.

Fortunately, there's a better way which, as I'm sure you will already suspect, is based on Taylor's theorem.

Note that the coefficient of δ^n in the Taylor series expansion of a function f about a value a is

$$\frac{1}{n!} \times \frac{d^n}{dx^n} f\left(x\right)\Big|_a$$

If we therefore keep track of the first n+1 terms of the Taylor series rather than just the first two we can recover the n^{th} derivative by multiplying the coefficient of δ^n by n!.

The first change we need to make to our `surreal` type is to store an array of coefficients rather than just two, as illustrated in listing 9.

Note that here n represents the order of the Taylor series rather than the number of terms and we must consequently use an array of length n+1.

```
template<size_t n>
class surreal
{
public:
  typedef boost::array<double, n+1> coeffs_type;

  surreal();
  surreal(double a);
  surreal(double a, double b);
  explicit surreal(const coeffs_type &coeffs);

  double coeff(size_t i) const;
  const coeffs_type &coeffs() const;

  int       compare(const surreal &w) const;
  surreal & negate();

  surreal & operator+=(const surreal &w);
  surreal & operator-=(const surreal &w);
  surreal & operator*=(const surreal &w);
  surreal & operator/=(const surreal &w);

private:
  coeffs_type coeffs_;
};
```

Listing 9

```
template<>
class surreal<size_t(-1)>
{
};
```

Listing 10

```
template<size_t n>
surreal<n>::surreal()
{
  coeffs_.assign(0);
}

template<size_t n>
surreal<n>::surreal(const double a)
{
  coeffs_.assign(0);
  coeffs_[0] = a;
}

template<size_t n>
surreal<n>::surreal(const double a,
                    const double b)
{
  coeffs_.assign(0);
  coeffs_[0] = a;
  coeffs_.at(1) = b;
}

template<size_t n>
surreal<n>::surreal(const coeffs_type &coeffs)
   : coeffs_(coeffs)
{
}
```

Listing 11

This means that we might run into trouble if *n* is the maximum value of **`size_t`** so I shall admit one tiny bit of template trickery and specialise the class to protect against this, as shown in listing 10.

We have kept our original constructors and have added one which initialises all of the coefficients. Their implementations are given in listing 11.

Note that we shall now access the coefficients by their ordinal position in the Taylor series and that we shall return a NaN rather than throw when reading past the end of the coefficients as shown in the definition of **`coeff`** in listing 12.

This might seem a little contrary to the C++ way of doing things but the reason for this design choice should become clear a little later on.

```
template<size_t n>
double
surreal<n>::coeff(const size_t i) const
{
  if(i>n) return
     std::numeric_limits<double>::quiet_NaN();
  return coeffs_[i];
}

template<size_t n>
const typename surreal<n>::coeffs_type &
surreal<n>::coeffs() const
{
  return coeffs_;
}
```

Listing 12

```
template<size_t n>
int
surreal<n>::compare(const surreal &w) const
{
  size_t i=0;
  while(i!=n && coeffs_[i]==w.coeffs_[i]) ++i;

  if(coeffs_[i]<w.coeffs_[i]) return -1;
  if(coeffs_[i]>w.coeffs_[i]) return  1;
  return 0;
}
```

Listing 13

Next up is the **`compare`** member function which operates in much the same way as **`std::lexicographical_compare`**, implemented in listing 13.

If you think this looks like I've made the beginners' mistake of reading past the end of the coefficients array, don't forget that it has *n*+1 elements.

The negation, addition and subtraction operators are hardly more complex than the original versions, as shown in listing 14.

Unfortunately we have now effectively introduced a branch which may harm performance. However, since the bounds of the loop counter are known at compile time an optimising compiler has an opportunity to unroll the loops for small *n*.

Generalised surreal multiplication

Multiplication is a slightly more complicated proposition since there may be several pairs of terms from the left and right hand side that yield a given order when multiplied.

The process for multiplying polynomials is known as a discrete convolution in which each coefficient in the result is equal to the sum

$$c_k = \sum_{i=0}^{k} a_i \times b_{k-i}$$

where a_i and b_i are the coefficients of the i^{th} order terms of the arguments and c_i is the same of the result.

```
template<size_t n>
surreal<n> &
surreal<n>::negate()
{
  for(size_t i=0;i<=n;++i)
     coeffs_[i] = -coeffs_[i];
  return *this;
}

template<size_t n>
surreal<n> &
surreal<n>::operator+=(const surreal &w)
{
  for(size_t i=0;i<=n;++i)
     coeffs_[i] += w.coeffs_[i];
  return *this;
}

template<size_t n>
surreal<n> &
surreal<n>::operator-=(const surreal &w)
{
  for(size_t i=0;i<=n;++i)
     coeffs_[i] -= w.coeffs_[i];
  return *this;
}
```

Listing 14

```
template<size_t n>
surreal<n> &
surreal<n>::operator*=(const surreal &w)
{
  for(size_t i=0;i<=n;++i)
  {
    coeffs_[n-i] *= w.coeffs_[0];

    for(size_t j=1;j<=n-i;++j)
    {
      coeffs_[n-i] += coeffs_[n-i-j]
         * w.coeffs_[j];
    }
  }
  return *this;
}
```

Listing 15

This is much like the algorithm we used to multiply bignums, except without the carry, and as was the case then efficient algorithms exist that use the Fast Fourier Transform.

We shan't go to such trouble, but we shall at least arrange to apply the process in-place by iterating backwards over the coefficients of the result, as shown in listing 15.

Note that we can only get away with this because we are discarding all terms above n^{th} order.

Generalised surreal division

As was the case for our first order **`surreal`** type, division is trickier still. Once again we shall multiply the numerator by the reciprocal of the denominator, but we shall need to work out the Taylor series for the reciprocal to n^{th} order.

Fortunately, as will often prove the case for arithmetic operations, this isn't actually too difficult.

$$
\begin{aligned}
\frac{1}{a+\delta} &= \sum_{i\geq 0} \frac{1}{i!} \times \left.\frac{d^i}{dx^i}\frac{1}{x}\right|_a \times \delta^i \\
&= \frac{1}{a} - \frac{1}{a^2}\delta + \frac{2}{2!}\frac{1}{a^3}\delta^2 - \frac{2\times 3}{3!}\frac{1}{a^4}\delta^3 + \ldots \\
&= \frac{1}{a} - \frac{1}{a^2}\delta + \frac{1}{a^3}\delta^2 - \frac{1}{a^4}\delta^3 + \ldots \\
&= \sum_{i\geq 0} \frac{(-1)^i}{a^{i+1}} \times \delta^i
\end{aligned}
$$

Now, we have the further complication that there may be several terms with non-zero powers of δ in w. Fortunately, it doesn't actually matter since their sum is still an infinitesimal and can consequently take the place of the δ in any Taylor series expansion.

The easiest way to do this is to use a **`surreal`** δ equal to the infinitesimal part of w and repeatedly multiply 1 by it to generate its integer powers, as illustrated in listing 16.

Note that the inner loop needn't use coefficients of **`di`** of order less than **`i`** since they must be zero after having multiplied by the 0 valued 0th order coefficient of **`d`**.

This suggests an optimisation in which we replace the multiplicative update step for **`di`** with one which doesn't update coefficients of order less than **`i`** or use coefficients of order less than **`i-1`**, as shown in listing 17.

Note that there are no safeguards against calling this helper function with invalid values for **`size`** and/or **`i`**. Instead we shall rely upon the convention that **`impl`** should be read as 'enter at your own peril'.

That caveat out of the way, we can replace the multiplicative update step with a call to this function as shown in listing 18.

```
template<size_t n>
surreal<n> &
surreal<n>::operator/=(const surreal &w)
{
  const double y = w.coeffs_[0];
  double si = 1.0;
  surreal rw(1.0/y);
  surreal di(1.0);
  surreal d(w);
  d[0] = 0.0;

  for(size_t i=1;i<=n;++i)
  {
    si = -si;
    const double yi = si * pow(y, double(i+1));
    di *=  d;

    for(size_t j=i;j<=n;++j)
       rw.coeffs_[j] += di.coeffs_[j] / yi;
  }
  return *this *= rw;
}
```

Listing 16

Generalised arithmetic operations

As before, reciprocation shines a light on how we might go about implementing other arithmetic operations and, once again, we shall choose as examples

e^w

$\ln w$

We shan't bother with the power this time since, as we have already seen, we can implement it in terms of these two.

The first step is, of course, to work out the Taylor series expansions of these functions. Fortunately this isn't very difficult; the Taylor series of the exponential is given by

$$
\begin{aligned}
e^{a+\delta} &= \sum_{i\geq 0} \frac{1}{i!} \times \left.\frac{d^i}{dx^i} e^x\right|_a \times \delta^i \\
&= e^a + e^a \times \delta + \frac{1}{2!} \times e^a \times \delta^2 + \frac{1}{3!} \times e^a \times \delta^3 + \ldots \\
&= e^a \times \left(1 + \delta + \frac{1}{2!} \times \delta^2 + \frac{1}{3!} \times \delta^3 + \ldots\right) \\
&= e^a \times \sum_{i\geq 0} \frac{1}{i!} \times \delta^i
\end{aligned}
$$

```
namespace impl
{
  template<size_t size>
  void
  update_di(const boost::array<double, size> &d,
            const size_t i,
            boost::array<double, size> &di)
  {
    static const size_t n = size-1;

    for(size_t j=0;j<=n-i;++j)
    {
      di[n-j] = 0.0;
      for(size_t k=1;k<=n+1-(i+j);++k)
         di[n-j] += di[n-(j+k)] * d[k];
    }
  }
}
```

Listing 17

```
template<size_t n>
surreal<n> &
surreal<n>::operator/=(const surreal &w)
{
  const double y = w.coeffs_[0];
  surreal rw(1.0/y);
  double  si = -1.0;
  surreal di(w);

  const double y1 = -y*y;
  for(size_t j=1;j<=n;++j)
     rw.coeffs_[j] += di.coeffs_[j] / y1;

  for(size_t i=2;i<=n;++i)
  {
    si = -si;
    const double yi = si * pow(y, double(i+1));
    impl::update_di(w.coeffs_, i, di.coeffs_);

    for(size_t j=i;j<=n;++j)
       rw.coeffs_[j] += di.coeffs_[j] / yi;
  }
  return *this *= rw;
}
```

Listing 18

and that of the logarithm by

$$\begin{aligned}\ln\left(a+\delta\right) &= \sum_{i\geq 0}\frac{1}{i!}\times\frac{d^i}{dx^i}\ln\left(x\right)\bigg|_a \times\delta^i \\ &= \ln a + \frac{1}{a}\times\delta - \frac{1}{2!}\times\frac{1}{a^2}\times\delta^2 + \frac{1}{3!}\times\frac{2}{a^3}\times\delta^3 - \mathrm{K} \\ &= \ln a - \sum_{i\geq 1}\frac{\left(-1\right)^i}{i\times a^i}\times\delta^i\end{aligned}$$

With these formulae to hand, implementing the exponential and logarithmic functions is but a simple matter of programming, as shown in listings 19 and 20.

Now we are nearly done, but there's one last use of the **surreal** type that I'd like to cover before we conclude.

```
template<size_t n>
surreal<n>
exp(const surreal<n> &w)
{
  double yi = 1.0;
  boost::array<double, n+1> di = w.coeffs();
  boost::array<double, n+1> ew = di;
  ew[0] = 1.0;

  for(size_t i=2;i<=n;++i)
  {
    yi *= double(i);
    impl::update_di(w.coeffs(), i, di);

    for(size_t j=i;j<=n;++j)
       ew[j] += di[j] / yi;
  }

  const double y = exp(w.coeff(0));
  for(size_t i=0;i<=n;++i) ew[i] *= y;
  return surreal<n>(ew);
}
```

Listing 19

```
template<size_t n>
surreal<n>

log(const surreal<n> &w)
{
  const double y = w.coeff(0);
  double si = 1.0;
  boost::array<double, n+1> di = w.coeffs();
  boost::array<double, n+1> lnw = {{log(y)}};

  for(size_t j=1;j<=n;++j) lnw[j] += di[j] / y;

  for(size_t i=2;i<=n;++i)
  {
    si = -si;
    const double yi =
       si * double(i) * pow(y, double(i));
    impl::update_di(w.coeffs(), i, di);

    for(size_t j=i;j<=n;++j)
       lnw[j] += di[j] / yi;
  }
  return surreal<n>(lnw);
}
```

Listing 20

L'Hôpital's rule

Consider the function

$$f\left(x\right) = \frac{\sin x}{x}$$

What is its value when *x* is equal to 0?

On the face of it that would seem to be a meaningless question. Certainly calculating it using floating point will yield a NaN.

However, it is in fact 1.

This rather surprising result can be demonstrated using, you guessed it, Taylor's theorem!

The Taylor series expansions of the numerator and denominator are

$$\begin{aligned}\sin\left(x+\delta\right) &= \sin x + \delta\times\cos x - \delta^2\times\sin x - \delta^3\times\cos x + \mathrm{K} \\ x+\delta &= x+\delta\end{aligned}$$

Setting *x* equal to zero gives

$$\begin{aligned}f\left(0+\delta\right) &= \frac{\sin 0 + \delta\times\cos 0 - \delta^2\times\sin 0 - \delta^3\times\cos 0 + \mathrm{K}}{0+\delta} \\ &= \frac{\delta - \delta^3 + \mathrm{K}}{\delta} \\ &= 1 - \delta^2 + \mathrm{K}\end{aligned}$$

Note that if the first order terms of the numerator and denominator were also zero we could continue on to the second order terms in the same fashion.

Since our surreal type keeps track of the coefficients of the Taylor series we can easily extend division to exploit L'Hôpital's rule, allowing us to correctly evaluate such ratios.

We shall do this by first finding the lowest order for which the coefficient in the Taylor series of either the numerator or denominator is non-zero. We shall then copy the coefficients of the denominator from that point on into the lowest order coefficients of our *δ*, leaving the higher order terms equal to zero.

Finally we shall shift the coefficients in the numerator to the left and fill the higher order terms with NaNs before multiplying by the reciprocal, as shown in listing 21.

```
template<size_t n>
surreal<n> &
surreal<n>::operator/=(const surreal &w)
{
  static const double nan =
     std::numeric_limits<double>::quiet_NaN();

  size_t off=0;
  while(coeffs_[off]==0.0 && w.coeffs_[off]==
     0.0 && off!=n) ++off;

  const double y = w.coeffs_[off];
  surreal rw(1.0/y);
  double  si = -1.0;
  surreal d, di;

  for(size_t i=0;i<=n-off;++i) d[i]
     = di[i] = w[i+off];

  const double y1 = -y*y;
  for(size_t j=1;j<=n-off;++j) rw.coeffs_[j]
     += di.coeffs_[j] / y1;

  for(size_t i=2;i<=n-off;++i)
  {
    si = -si;
    const double yi = si * pow(y, double(i+1));
    impl::update_di(d.coeffs_, i, di.coeffs_);

    for(size_t j=i;j<=n;++j) rw.coeffs_[j]
       += di.coeffs_[j] / yi;
  }

  if(off!=0)
  {
    for(size_t i=0;i<=off;++i)
       coeffs_[i] = coeffs_[i+off];
    for(size_t i=off+1;i<=n;++i) coeffs_[i] =
nan;
  }
  return *this *= rw;
}
```

Listing 21

This final padding with NaNs is the reason why we could leave zeros in our δ; it doesn't matter what value they have since any terms they effect will eventually be multiplied by NaN.

We might improve the efficiency of division by making `update_di` and the final multiplication aware of the truncation of the terms, but since this is likely to be a relatively rare occurrence it hardly seems worth it.

Now, at last, I can justify the design choice of returning NaNs when reading past the end of the coefficients rather than throwing exceptions. The application of L'Hôpital's rule means that we have less information for computing coefficients, so some high order coefficients in the array may be undefined and, in floating point arithmetic, *undefined* is spelt NaN.

Making a distinction between coefficients undefined because we didn't choose a high enough order `surreal` and coefficients undefined because of L'Hôpital's rule doesn't make a great deal of sense to me; both are a consequence of a lack of information[1].

One change that will almost certainly be necessary is to replace the equality comparisons when computing `off` with one that checks whether the terms are very small.

Unfortunately the meaning of very small is rather determined by the problem at hand; it needs to be relative to the order of magnitude of the variables in the calculation.

We should therefore need to add some facility for the user to provide this information, preferably as a template argument so that we cannot perform arithmetic with numbers that disagree on the matter. Since we cannot use floating point values as template arguments we should probably have to content ourselves with an integer representing a power of 10.

The Holy Grail?

So we finally have an algorithm for computer differentiation that is as accurate as possible without a computer algebra system and parsimonious with memory to boot.

I must admit that I have something of a soft spot for this approach having independently discovered it a few years ago. Even though I subsequently discovered that I wasn't the first, I still find it to be more satisfying than symbolic differentiation and it is my weapon of choice for difficult, by which I mean extremely lengthy, differentiation calculations.

Cancellation error

Unfortunately automatic differentiation suffers from cancellation error in the derivative just as much as symbolic differentiation does. Indeed, it is doubly cursed since, unlike symbolic differentiation, we couldn't automatically manipulate the form of the calculation to try to reduce its effect even if we wanted to, although we could at least again use interval arithmetic to monitor its effect.

Numerical approximation

As with symbolic differentiation we must still take care when combining automatic differentiation with numerical approximation; the derivative of an approximation of a function may not be a good approximation of the derivative of that function.

The solution is more or less the same as that we used for expression objects; we must simply compute the derivatives explicitly and use them to calculate the coefficients of the function's Taylor series.

As with expression objects we can do this either using a general purpose algorithm such as the polynomial approximation, or with a specific approximation of the derivative of the function, or, if we're very fortunate, with the actual formula for the derivative.

Clearly this is no better or no worse than taking the same approach with expression objects, but somehow it seems a more natural union to me. With automatic differentiation we are not trying to maintain an exact representation of the mathematical expression so resorting to approximation doesn't seem quite so jarring.

Although, as I have already admitted, I am a little biased.

In conclusion, I declare that automatic differentiation is a rock star duck; hardly perfect, but damn cool! ■

References and further reading

[Knuth74] Knuth, D., *Surreal Numbers; How Two Ex-Students Turned on to Pure Mathematics and Found Total Happiness*, Addison-Wesley, 1974.

[Ridders82] Ridders, C.J.F., *Advances in Engineering Software*, Volume 4, Number 2., Elsevier, 1982.

[Robinson74] Robinson, A., *Non-Standard Analysis*, Elsevier, 1974.

1. Note that the `compare` member function will consequently work at the known, rather than the declared, order since the final less than and greater than comparisons will be false.

Back to School

The Sinclair ZX Spectrum will be 30 years old in April 2012. Andy Thomas recalls how this plucky little home computer shaped his childhood.

'So... can you actually talk to it and just tell it what to do?', I asked a kid at school.

The year was 1983, and my friend had just got a home computer, a Tandy TRS80. I had never seen a computer and was absolutely fascinated.

I visualized sitting in front of this thing and speaking out instructions such as, 'Please calculate 6 times 7.'

'Oh no,' he said. 'You use menus.'

'Menus?' Now I really was confused, and I visualized sitting in a restaurant and reading out a menu to a box on the table. It didn't make any sense!

At school, I was pretty much in bottom classes in everything – except, that was, for art. 'Well, he's good at drawing,' I recall a teacher once telling my parents. I was 12 years old, and I didn't know it then, but a plucky little British home computer was about to change my life.

That computer was no other the Sinclair ZX Spectrum, which I got for Christmas that year. In fact, I had already found where my parents were hiding it and had pilfered the instruction manuals, which I read cover-to-cover, several times, before Christmas day.

The Spectrum came in 16K and 48K flavours, and I was lucky enough to get the 48K one. I can remember standing in the computer shop with my Dad while the salesmen emphasized what a massive amount of memory it had!

'No matter what you do, you will never be able to fill up 48K of RAM,' he said.

Yeah right! Good job I didn't end up with the 16K one is all I can say now.

The Spectrum, or Speccy to its fans, had an unusual rubber keyboard which, as bad as it was, represented an improvement over the membrane keypad of its predecessor – the ZX81. Like most home computers in the UK at that time, the Speccy had its own built-in BASIC programming language. Programming it using the weird keyboard took some getting used to though, as each BASIC keyword had its own key, and there were weird input modes which allowed you to access the shifted keywords.

Programs were loaded and saved onto audio tapes using a cassette player. It could take up to five minutes to load a program, and the loading process often failed four and a half minutes in. But this really made the whole experience worthwhile – if a game was worth loading, then it had to be worth playing. It added value to the whole experience. In any case, while I played games occasionally, it wasn't playing games that really did it for me – it was writing them. Or at least trying to, I should say.

I primarily saw the Spectrum as a kind of creativity tool – a blank canvas on which I could digitally paint interesting stuff, whether it be games, scientific programs or whatever. It was my introduction to computers, and it taught me at a young age that computers should be fun and that programming can be a creative and rewarding enterprise. I have never let go of that and, in later life, I guess it helped me not to be succumb to the notion that software development can also be a bureaucratic and soul destroying experience.

Anyway, back to my childhood and the Speccy...

I soon picked up Spectrum BASIC from the user manual, and before long, I was writing games. I wrote quite a few horizontal scrolling games, to which I gave names such as *Shuttle Defender* and *Sea Harrier Attack*.

Unfortunately, no one told me that the age of games written in Spectrum BASIC had begun and ended in 1982 with the *Horizons* cassette that came free with every Speccy. So I soldiered on, oblivious, and sent off demo cassettes containing games written in BASIC to the likes of Atari and Artic Computing.

Although I once received a courtesy reply, no publishing deals were forthcoming. Eventually I decided there was only one thing for it – to set up my own software house. I called my business venture Saturn Software, and spent a whole week's wages from my paper round on a pack of five audio cassettes. Would five be enough? I wasn't sure, but it was all that I could afford. If the orders came flooding in, then I decided I could always buy some more with the money I would be making.

I put a hand-written advertisement for Saturn Software in the local paper shop and, while I waited for the telephone orders to roll in, I loaded the cassettes with my *Shuttle Defender* game. I painstakingly drew the artwork for each cassette inlay by hand, and for that professional touch, I coloured them in with felt-tip pen.

Tragically, the one and only telephone phone call I got in response to my ad was answered by my Mum...

While I never learned exactly what was said, apparently the caller was rather menacing and threatened to have me prosecuted for software piracy. I was duly made to remove my advertisement from the local shop, and that put paid to my first-ever business venture.

It didn't put me off though, and I carried on writing games and other programs. I had a creative urge, an internal spark, that kept me awake at night dreaming up new projects.

Finally, however, I came to realize that Spectrum BASIC just wasn't going to cut it. If I was ever going to get a game published, then I had to learn machine code. Unfortunately, coming from a small town in Northern England, I didn't know how to access the kind of information and learning resources for this. The Internet would have been really useful to me then, but this was before the Web, or even the Information Super Highway for that matter.

At least back in those days computing was safe, however. There was no danger of me coming into contact with chatroom paedophiles or getting myself extradited to the US because I had clicked on something I shouldn't have. There's a part of me which bemoans the day my personal computer became connected to the rest of the world, or more specifically, when the

Andy Thomas has interests in software development, physics, mathematics, space technology and machine intelligence. Professionally, he works in the telecomms arena where he drinks a lot of coffee and occasionally produces something that works. He can be contacted at andyt@bigangrydog.com.

I could understand and interact with a computer quite easily, so I used it as an outlet for my creativity

rest of the world became connected to my computer. I guess I'm a little nostalgic for a time when your computer was your own.

I did, however, get my hands on some computer books from the local library – both of them. I recall that one was about *Expert Systems*, and while I was fascinated, I could not quite grasp what they were actually for. To be honest, I still don't. Did anybody?

There was one person I could ask. My best friend was an electronics genius, and while I spent my evenings writing Speccy BASIC, he spent his with a soldering iron and an assortment of circuit boards with 555 timers on them. He also knew machine code, and I begged him to teach me. I recall him giving me a verbal crash course in Z80 assembler while we stood on his parent's doorstep. I remember it all had something to do with shift registers and interrupts, but it all went over my head and I didn't understand it.

In fact, I never did have much luck with publishing a game for Spectrum. All my games were written in BASIC and were, well, to be honest, a bit rubbish really. But my efforts were not in vain, and the Spectrum did indeed have a profound affect on my childhood and, I suspect also, on my later life.

When I was given a home computer as a Christmas present at the age of 12, I was pretty much written off at school. Over the next two years, however, my school work dramatically improved and I was lifted out of the bottom classes, and in the end, I did very well. The truth was that, as a child, I didn't really understand how to interact with people and found the whole experience confusing and uncomfortable. So I would just disengage sometimes. But I could understand and interact with a computer quite easily, so I used it as an outlet for my creativity, which otherwise, may have remained locked up inside. In the process, I learned a great deal and became interested in many things.

Undoubtedly the ZX Spectrum helped to shape my childhood for the better, and I wonder sometimes just how my life would have turned out without Sir Clive Sinclair's marvellous little machine. Not as well I think.

Afterthought

Although I never managed to publish a game for the Spectrum, I did do slightly better with the Tatung Einstein and a little outfit called Bell Software who were sympathetic enough to publish a couple of my games. I suspect, however, this was largely due to the lack of competition in the Einstein games market more than anything else. (I recall that I made around £15 in total – just don't tell the taxman!)

In any case, I'm kinda hoping that someone from Bell Software will Google this one day, and that by some miracle, they will still have a disc of my old games because I'd sure would love to see them again, even if they were a bit rubbish. ■

cqf.com

CERTIFICATE IN QUANTITATIVE FINANCE CQF

Expand Your Mind and Career

Designed by quant expert Dr Paul Wilmott, the CQF is a practical six month-part time course that covers every gamut of quantitative finance, including derivatives, development, quantitative trading and risk management.

Find out more at **cqf.com**.

ENGINEERED FOR THE FINANCIAL MARKETS

Valgrind Part 1 – Introduction

Good analysis tools can really help track down problems. Paul Floyd investigates the facilities from a suite of tools.

Valgrind is a dynamic analysis tool for compiled executables. It performs an analysis of an application at run time. Valgrind can perform several different types of analysis (though not at the same time): memcheck (memory checking), cachegrind/callgrind (time profiling), massif (memory profiling), helgrind/drd (thread hazard detection). In addition there are some experimental tools: sgcheck (global/stack overrun detection, prior to Valgrind 3.7.0 this was called ptrcheck), dhat (heap use analysis) and bbv (extrapolating longer executions from smaller samples). There is also nulgrind, which does nothing other than execute your application.

Valgrind can be found at http://www.valgrind.org.

How it works

Unlike some other tools, Valgrind does not require the application under test (AUT) to be instrumented. You don't have to perform a special compile or link, and it will even work with optimized builds, though obviously that makes it more difficult to correlate to source lines.

At the core of Valgrind is a virtual machine, VEX. When Valgrind runs your AUT, it doesn't run it on the underlying OS, it runs it in the virtual machine. This allows it to intercept system calls and memory accesses required for the analysis it is performing. VEX performs on the fly translation of the AUT machine code to an Intermediate Representation (IR), which is RISC like and common to all architectures that Valgrind supports. This IR is more like a compiler IR than machine code.

Running your AUT in a virtual machine does of course have some disadvantages. The most obvious one is that there is a significant time penalty. The same AUT running directly on the OS will typically run about 10 times faster than under Valgrind. In addition to taking more time, the combination of Valgrind and the AUT will use more memory. In some cases, in particular on 32bit OSes, you might find that the AUT alone will fit in memory but that Valgrind+AUT doesn't. In this case, if you can, the answer may be to recompile and test in 64bits. There are some discrepancies between Valgrind's virtual machine and your real machine. In practice, I haven't noticed anything significant other than small differences in floating point calculations. This is most noticeable with 32bit executables using the x87 FPU. Valgrind emulates floating point with 64bit doubles, whilst the x87 uses 80 bits for intermediate calculations.

The worst case problem with the virtual machine is an unhandled op-code. In this case you will get a message like

```
==[pid]== valgrind: Unrecognised instruction at
address [address].
```

and then Valgrind will terminate. Fortunately, unless you are using some exotic/bleeding edge OS or hardware, this is unlikely to happen. Somewhat more likely, in particilar on 32bit OSes, is that you will run out of memory. Unfortunately, Valgrind may not produce any output in that case.

You can't run the AUT directly in a debugger while it is running under Valgrind. It is possible to have Valgrind attach a debugger to the AUT when an error occurs, which gives you a snapshot of the executable rather like loading a core file. There is a second, better alternative that is now possible with Valgrind 3.7.0. This is to attach gdb to the embedded gdbserver within Valgrind. If you do this then you will be able to control the AUT much like a normal AUT under gdb, and also perform some control and querying of Valgrind. It is also possible to compile Valgrind tracing in the AUT which can output information on a specific region of memory. I'll cover these topics in more detail in a later article.

Availability

Valgrind started out on Linux, but it is now available on FreeBSD and Mac OS X. It is also available for some embedded systems. I've read of patches that support Windows, but it isn't officially supported. If you are using Linux, then the easiest thing to do is to install it using your package manager. If you want to have the very latest version, then you can download the current package from the Valgrind website and build it yourself. Generally just running **`./configure`**, **`make`** and **`make install`** are sufficient. If you want to be able to use gdb, make sure that the version that you want to use when running Valgrind is in your PATH when you build Valgrind. If you want the very latest version, then you can get it from the Subversion server. Building is a little more involved, but the Valgrind website gives clear instructions.

Running Valgrind

So you've installed Valgrind. Next, how do you run it? Valgrind has a large number of options. There are 4 ways of passing options: a resource file (`.valgrindrc`) in your home directory, in an environment variable (**`VALGRIND_OPTS`**), in a resource file in the working directory and on the command line, in that order. If an option is repeated, the last appearance overrides any earlier ones. This means that the resource file in your home directory has the lowest priority and the command line has the highest priority.

The options can be broken down into two groups: core options and tool options. The core options are for Valgrind itself, like **`--help`**. The most important of these is **`--tool`**, used to select the tool to use, which must be passed on the command line. However, if **`--tool`** is omitted, the default will be memcheck.

You can specify options for more than one tool at a time by using the prefix **`<toolname>:`** before the option. For instance, to have memcheck report memory that is still reachable (but not leaked) when the AUT terminates, you can use **`--memcheck:show-reachable=yes`**. If you then use massif as the tool, this option will be ignored.

In my experience, it is best to put options that are independent of the test in the `.valgrindrc` and to pass options that may change on the command line.

Paul Floyd has been writing software, mostly in C++ and C, for over 20 years. He lives near Grenoble, on the edge of the French Alps, and works for Mentor Graphics developing a mixed signal circuit simulator. He can be contacted at pjfloyd@wanadoo.fr.

Processes and output

Some of the tools default to sending their output to the console (e.g., memcheck). Others default to a file (e.g., massif). You can specify that the output go to a file with the `--log-file=<filename>` option. In either case, the output lines are prefixed with `==<pid>==` or `--<pid>--` showing the pid (process id) of the AUT. The `<filename>` can contain %p which will be expanded to the PID of the AUT. This is particularly useful if you will be running the same AUT more than once. For instance, say you have two AUTs, 'parent' and 'child', both of which you want to run under Valgrind. 'parent' runs 'child' more than once when it executes. If you let all of the output go to the console, then it might get all mixed up. Instead you could launch

```
valgrind --tool=memcheck --log-file=parent.%p.log
parent
```

and arrange it so that 'parent' launches 'child' in a similar fashion

```
system("valgrind --tool=memcheck
--log-file=child.%p.log child");
```

On completion, you would have something like `parent.1234.log` and `child.1245.log`, `child.1255.log`.

There is also the `--trace-children=yes` option. Whilst it is simpler than the above technique, it will cause all child processes to be traced. This is less useful if your application launches a lot of child processes, and you only want to test a small subset of them.

I'll finish with a quick demonstration using Nulgrind. Here are my two test executables.

```
//parent.c
#include <stdlib.h>
int main(void)
{
  system("./child");
}
```

and

```
// child.c
#include <stdio.h>
int main(void)
{
  printf("Hello, child!\n");
}
```

Let's compile them

```
gcc -o child -g -Wextra -Wall child.c -std=c99
-pedantic
gcc -o parent -g -Wextra -Wall parent.c -std=c99
-pedantic
```

If I run 'parent' alone, I see the expected output. Now let's try that with a minimum of options (Listing 1) and with the option to follow children (Listing 2).

So you can see the two PIDs, 9356 and 9357. 9356 spawns then execs sh, which in turn execs child.

Now if I run

```
valgrind --tool=none --trace-children=yes
--log-file=nulgrind.%p.log ./parent
```

then I get two log files, nulgrind.10092.log and nulgrind.10091.log which basically contain the same information as the parent and child sections above.

Lastly, if I change the 'system' call to

```
system("valgrind --tool=none
--log-file=child.%p.log ./child");
```

and I then run parent without the `–follow-children` option, like this

```
valgrind --tool=none --log-file=parent.%p.log
./parent
```

then I get two log files, `parent.12191.log` and `child.12209.log`, again with the content more or less as above.

That wraps it up for part 1. In part 2 I'll cover the basics of memcheck. ■

```
valgrind --tool=none ./parent
==9131== Nulgrind, the minimal Valgrind tool
==9131== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote.
==9131== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==9131== Command: ./parent
==9131==
Hello, child!
==9131==
```

Listing 1

```
valgrind --tool=none ./parent –follow-children
==9356== Nulgrind, the minimal Valgrind tool
==9356== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote.
==9356== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==9356== Command: ./parent
==9356==
==9357== Nulgrind, the minimal Valgrind tool
==9357== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote.
==9357== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==9357== Command: /bin/sh -c ./child
==9357==
==9357== Nulgrind, the minimal Valgrind tool
==9357== Copyright (C) 2002-2010, and GNU GPL'd, by Nicholas Nethercote.
==9357== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==9357== Command: ./child
==9357==
Hello, child!
==9357==
==9356==
```

Listing 2

Mutation Testing

We all know that testing improves our code, guarding against errors. Filip van Laenen asks how we know that the tests are comprehensive?

Quis custodiet ipsos custodes?
Who is guarding the guards?
Juvenal (Satire VI, lines 347-8)

Unit tests guard the source code, verifying that the behaviour of the system is correct. But how do we know that the unit tests cover all the lines of the source code, all the branches, let alone all the paths through the source code? And how do we know that the unit tests test the important aspects of the source code? Indeed, it could very well be that the unit tests cover all lines, branches and even paths of the source code, but never test anything that really matters. Mutation testing can help provide an answer to all these questions.

What is mutation testing?

Mutation testing starts with source code and unit tests. At some place in the source code, it makes a small change, which we'll call the 'mutation'. Typical mutations include replacing an addition with a subtraction, negating or short-cutting conditions, changing the values of constants and literals, commenting out a line, and many more. The result of the mutation is a new system, which we'll call the 'mutant'. What mutation testing then does is that it runs the original unit tests on the mutant, and checks whether any of them fail. Depending on the mutation and the quality of the unit tests, different things can happen. Sometimes only one or two unit tests fail. Other times, almost all the unit tests fail. And in some cases, none of the unit tests fail.

Let's go through the different scenarios that can occur. The most straightforward case is the one in which only one, or perhaps two or three unit tests fail. This means that the mutation happened in a place that's tested by the now failing unit tests, and that the change in behaviour due to the mutation is detected by the unit tests. In general, this should indicate that the unit tests test an important and relevant aspect of the system's behaviour, and that the part of the source code where the mutation occurred matters for the system's behaviour. We can therefore conclude that everything is in order with this mutant, and generate the next mutant to be tested by the unit tests.

Alternatively, many, or in some cases almost all the unit tests could start to fail as a result of the mutation. This often happens when the mutation occurs in one of the fundamental and heavily reused parts of the system, like e.g. the core engine or a project-specific framework. Again, many failing unit tests is a good sign, because it means that the part of the source code where the mutation occurred matters for the behaviour of the system.

It's also possible that all the unit tests pass when they're run against the mutant. This would obviously be the case when the mutation occurs in a part of the system that isn't covered at all by the unit tests. But it could also be that the line where the mutation occurred is covered by unit tests, but that the behaviour of that particular part of the code isn't tested by any of the unit tests. Imagine, for example, that the mutation happened in a log message. Although correct logging is important, it's often only informative and seldom critical to a project. It's therefore not unusual that a system's unit tests don't test the content of the log messages, and any mutations to the content in log messages will therefore remain undetected.

Filip van Laenen works as a chief technologist at the Norwegian software company Computas, which he joined in 1997. He has a special interest in software engineering, security, Java and Ruby, and tries to inspire his colleagues and help them improve their software quality skills. He can be contacted at f.a.vanlaenen@ieee.org.

```
def sum(n) {
  s ← 0;
  for (i ← 1; i ≤ n; i++) {
       s ← s + n;
  }
  return s;
}

def sum_mutant(n) {
  s ← 0;
  for (i ← 1; i < n; i++) {
       s ← s + n;
  }
  return s;
}
```

Listing 1

There may, however, be something else going on. There are cases where a change to the source code doesn't change the behaviour of the system. Obviously, we're not talking about changes like renaming variables or extracting small functions from larger functions. This sort of refactoring can hardly be called a mutation, and certainly doesn't belong to the same category as replacing a multiplication with a division or switching the order of parameters in a function call. But what about replacing a less-than sign (**<**) with a less-than-or-equal-to sign (≤)?

Let's consider two simple cases. Listing 1 illustrates the first case, where a less-than-or-equal-to sign in a function's loop is replaced with a less-than sign. Clearly, this changes the behaviour of the system, and a simple unit test verifying that `sum(1) = 1` would already detect the mutation. Listing 2 however illustrates another case, where the same type of mutation in a condition doesn't change the behaviour of the system at all. (We assume that we're dealing with immutable objects like e.g. simple integers, and not objects where there may be a difference between equality and identity.) Notice that for certain input values, the execution path through the system may be different between the original system and the mutant, but the end result is still same.

In general, a case like the one described in Listing 2 is called an equivalent mutation, since it results in a mutant that's equivalent to the original system. For certain types of mutations, it's not always easy to establish whether a specific mutation to a system is an equivalent mutation or not. Therefore, most mutation testing tools simply don't include these types of mutation, but some do allow it if the user wants it.

If you have good mutation test coverage of your source code, chances are you're close to 100% test coverage anyway

```
def max(a, b) {
  return (a ≤ b) ? b : a;
}

def max_mutant(a, b) {
  return (a < b) ? b : a;
}
```

Listing 2

Does mutation testing work?

It may sound like a strange concept to make changes to the source code of a system, and then running the original unit tests on the resulting mutant. It seems so obvious that some of the unit tests should start to fail when we do that, yet we already mentioned a few reasons why it doesn't always happen. So does it improve the quality of our software, and if so, how?

As Walsh noted in this Ph.D. Thesis in 1985, 'mutation testing is more powerful than statement or branch coverage'. [Walsh85] Many years later, Frankl, Weiss and Hu stated that 'mutation testing is superior to data flow coverage criteria' [Frankl97]. My personal experience with mutation testing is in line with these two statements. In fact, if I'm able to use mutation testing in a project, I usually don't care that much any more about what my test coverage tool may be reporting. If you have good mutation test coverage of your source code, chances are you're close to 100% test coverage anyway. Actually, mutation testing can often give you an impression of a 'more than 100% test coverage', as a small example that we'll dig into shortly will show.

But why does it work? Andrews, Briand and Labiche give us a first clue about why this is so: 'Generated mutants are similar to real faults' [Andrews05]. Geist et. al. provides us a with a longer explanation, saying that 'in practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault' [Geist92]. Could this be because in some cases, one of the mutants would actually be the correct system, but we simply haven't written the correct unit tests yet to differentiate between the incorrect original system, and the correct mutant? Finally, Wah adds to this that 'complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults' [Wah95]. Replace 'test data set' with 'unit test', and it becomes clear that if you apply many simple mutations to your original source code, and your unit tests can detect all the mutants, the unit tests will be able to detect complex mutations too. Or in other words, if you've done the little things right, you have a good chance that you got the big things right too.

```
Assertion: max([0]) = 0

def max(a) {
  return 0;
}
```

Listing 3

```
Assertion: max([0]) = 0
Assertion: max([1]) = 1

def max(a) {
  return a.first;
}
```

Listing 4

A practical example

Let's walk through a simple, practical example, to show how mutation testing works in practice. Actually, the example is not just an academic exercise I made up for this article, but distilled from an eye-opening experience I had a couple of years ago while working on a similar piece of code.

Assume we would want to implement a function that returns the maximum of a vector of integers. (In reality you should of course use the functions that are already available in your language or the standard libraries.) For simplicity, we ignore the case where the input vector could be `null` or an empty vector, and concentrate on vectors that have actual content. Listing 3 shows the first unit test, represented as an assertion, and the corresponding source code, a trivial implementation.

Listing 4 adds a second unit test, and a simple implementation that would make both unit tests pass. Things get more interesting when we add a third unit test, as shown in Listing 5. In order to make all three unit tests pass, we add what could be the final implementation for our function. But the question is: are we really done yet? And did we really follow the TDD principle that we should always aim for the simplest possible implementation that could work?

Let's have a closer look at the code in Listing 5. First of all, we have three unit tests. Second, test coverage is 100%, both in number of lines and branches, even including the implicit `else` clause of the `if` statement. One could wonder whether there's really more to be wished for. But when we run mutation testing against the source code, we find out that there's still something wrong. In fact, it complains that the mutant shown in Listing 6

```
Assertion: max([0]) = 0
Assertion: max([1]) = 1
Assertion: max([1, 2]) = 2

def max(a) {
  m ← a.first;
  foreach (e ∈ a)
    if (e > m)
      m ← e;
  return m;
}
```

Listing 5

follow the **TDD principle** that we always should aim for the **simplest possible implementation** that could work

```
Assertion: max([0]) = 0
Assertion: max([1]) = 1
Assertion: max([1, 2]) = 2

def max(a) {
  m ← a.first;
  foreach (e ∈ a)
    if (true)
      m ← e;
  return m;
}
```

Listing 6

survived all unit tests. This mutant is the result of short-cutting the condition of the `if` statement, or in other words, removing the condition and keeping only the positive branch of the `if` statement. Practically, this is done by replacing the content of the `if` statement's condition with a simple `true`. But can it really be true that the mutant of Listing 6 survived all unit tests? After all, the unit tests did have 100% test coverage, both for lines and branches!

If we run the unit tests by hand on Listing 5, we find quickly that all the unit tests enter the loop of the function. Since we allocated the value of the first element in the vector to the local variable `m`, we pass into the implicit `else` clause during the first round. Only during the second iteration in the third unit test do we enter the positive branch of the `if` statement, since 2 is greater than 1. However, if we do the same exercise for the source code in Listing 6, we find that the three unit tests still pass. The only difference is that we allocate the value of the first element of the vector twice to `m`, once at the beginning of the loop, and once inside the `if` statement's positive branch. It turns out that with these three unit tests, there's no real need to ever enter the `else` clause of the `if` statement.

Actually, if you think it a bit through, the source code of Listing 6 is equivalent to the source code of Listing 7. In fact, if we really wanted to follow the TDD principle that we always should aim for the simplest possible implementation that could work, we should have gone from Listing 4 to Listing 7, not Listing 5. We were probably still too eager to start programming, even though we tried to do our best to follow the principles of TDD, and jumped into implementing the full function one unit test too early.

So where do we go from Listing 7? We add a new unit test, just like TDD says we should do. Listing 8 shows the fourth unit test – we just pick a vector where the last element isn't the largest one – and we can finally finish the implementation of this function.

```
Assertion: max([0]) = 0
Assertion: max([1]) = 1
Assertion: max([1, 2]) = 2

def max(a) {
  return a.last;
}
```

Listing 7

```
Assertion: max([0]) = 0
Assertion: max([1]) = 1
Assertion: max([1, 2]) = 2
Assertion: max([2, 1]) = 2
def max(a) {
  m ← a.first;
  foreach (e ∈ a)
    if (e > m)
      m ← e;
  return m;
}
```

Listing 8

So where did things really go wrong? There is, in fact, a subtlety in Listing 5 (and Listing 8) that tricked our test coverage tool into believing that we really had 100% branch coverage. Take a look at Listing 9, which is basically the same as Listing 5, but with a small yet important difference in the first line of the loop. Instead of using the value of the first element of the vector as the initial value for the local variable `m`, it uses `-∞`. This may look like a detail, but it does make a huge difference. The most important difference is that with the three unit tests from Listing 5 only, the implicit `else` clause of the `if` statement can't be reached. Indeed, the condition of the `if` statement will evaluate to `true` in every iteration of every unit test. As a result, if we had chosen to implement the function as shown in Listing 9, and then run and checked the test coverage, we would never have said that we had 100% branch coverage, and we would have known that there was still missing at least one test case.

This simple example taught us that even when our test coverage tool reports that we have 100% test coverage both in terms of lines and branches, mutation testing may still find problems. This is the reason why I wrote earlier that mutation testing sometimes gives the impression that it leads to a 'more than 100% test coverage', as if 100% test coverage isn't enough. Fundamentally, the mutant of Listing 6 warned us that there was something wrong with the unit tests – in this case that there was at least one unit test still missing.

```
Assertion: max([0]) = 0
Assertion: max([1]) = 1
Assertion: max([1, 2]) = 2

def max(a) {
  m ← -∞;
  foreach (e ∈ a)
    if (e > m)
      m ← e;
  return m;
}
```

Listing 9

mutation testing can quickly become very time-consuming if we don't have a good strategy to select the right unit tests

As a side-effect, this example also showed that it's all too easy to break TDD's principle that you should write only the simplest possible source code to make the unit tests pass. When I do presentations about mutation testing, the audience usually understands that something must be wrong in Listing 5, but only because otherwise it would just make a silly example. Of course, the presentation 'tricks' the audience into the wrong track, just like this article did with you (or at least tried to), but I'm convinced that most developers would never bother to write more than the first three unit tests if they had to implement a function like this. In fact, it's not easy to explain to people that they really need a fourth unit test for this type of function when they're not familiar with the concept of mutation testing. I consider it one of the main benefits of mutation testing that it sometimes 'disturbs' me when it throws a mutant like the one from Listing 6 at me. By now I've learned that when a mutant refuses to disappear, mutation testing is trying to tell me something, and I'm probably going to learn something new.

Relation to other testing techniques

Before we dive into the techniques used by mutation testing and the tools that exist, how does mutation testing relate to other testing techniques? Obviously, mutation testing depends on **unit testing**, because it needs the unit tests to run against the mutants. But in order to create good unit testing, I think **Test-Driven Development** (TDD) is absolutely necessary too.

We already discussed the relation with **test coverage**, i.e. that mutation testing is superior to test coverage and even can give an impression of a 'more than 100% test coverage'. But that doesn't mean mutation testing eliminates the need to run test coverage as part of your build cycle. As we'll see later, mutation testing can consume a lot of resources, and it's therefore usually a good idea to run test coverage before mutation testing as a sort of smoke test. If test coverage is low, mutation testing will almost certainly find lots of mutants that will pass the unit tests.

As far as I know, there's no relation or overlap between mutation testing and **static code analysis**. One could imagine though that some of the mutants produced by mutation testing will be caught by static code analysis, e.g. when the mutation removed a statement closing a resource. It would therefore make sense to run some sort of static code analysis on the mutants, but I'm not aware of any mutation testing tools that do that actively. It should be noted though that many compilers include some static code analysis, and therefore run it implicitly as part of the compilation.

A lot of people confuse mutation testing with **fuzz testing**, but the two testing techniques are quite different. Fuzz testing produces random sets of input data, feeds them to the system, and then detects whether the system crashes as a consequence of the input data. Traditionally, it has been used a lot as part of the security testing of programmes, but it has other applications too. The main differences between the two techniques are that fuzz testing operates on the input data, whereas mutation testing mutates the source code, and that fuzz testing often is a random process while mutation testing is a deterministic process.

That last aspect – the deterministic nature of mutation testing – often surprises people too. Most people connect the term 'mutation' almost always with cosmic rays, radioactivity, Chernobyl, chickens with three legs, and therefore random processes. But what really happens during mutation testing is that a tool goes through all the code in the system, tries to apply a fixed set of mutations on every piece of code, and then runs the unit tests on the resulting mutants. There's no randomness in where the mutations are applied, and there's no randomness in what sort of mutations are applied.

Is mutation testing new?

The answer to the question is a simple no. The technique was already described in the seventies [Lipton71, Lipton78] but didn't gain much success at that time. The problem was that many obstacles had to be overcome for it to become practical to run in real-life projects. First of all, unit testing didn't break through until the turn of the century, and even then we had to wait a few years before TDD started to become mainstream. But as we'll see, mutation testing can quickly become very time-consuming if we don't have a good strategy to select the right unit tests, and therefore it remained mostly an academic tool until a few years ago. As a consequence of that, nobody ever tried to integrate mutation testing into an IDE, and even today that's a part that's still missing. This means that even though mutation testing was described more than forty years ago, it hasn't been until very recently that it has become possible to actually use it in real projects.

Mutation testing techniques

If you want to build a mutation testing tool, there are three aspects that you have to consider. First of all, you need a way to inject mutations into the original source code in order to create mutants. You also have to consider which types of mutations you want to use. Not all types are useful, and especially the ones that can create equivalent mutations can be problematic. Finally, you need to find a strategy to select the unit tests to be run against the mutants. As we'll see, without a good unit test selection strategy mutation testing quickly becomes infeasible for real systems. The key properties for any mutation testing tool are its efficiency and performance: how good is it at creating mutants, and how fast is it at processing all these mutants?

Starting with the **injection of the mutations**, there are basically two alternatives available: source code and binary code mutation. Source code mutation makes the changes in the source code, and then compiles the mutated code. The big advantage of source code mutation is that it can be done using text manipulation, but the big disadvantage is that compilation takes time, even if it could be done as an incremental compilation on top of the original compiled code. Source code mutation has also the advantage that it's easy to output the generated mutant, e.g. when it survives all the unit tests. On the other hand, with source code mutation the mutation testing tool has to be able to handle the compilation round, including compilation errors that may occur.

Binary code mutation is faster, because it operates directly on the already available original compiled source code, but it requires that you know how to manipulate the binary code correctly. And even then, it's possible a

find an order in which to run the unit tests such that those that fail are run as early as possible

mutation leads to invalid binary code, which has to be handled by the mutation testing tool. Outputting the source code for the mutant isn't a straightforward job, but requires a decompilation round. This decompilation can be as simple as knowing how to map a binary code mutation to a source code mutation though.

But how about those compilation errors, or the invalid binary code? A problem like this can occur when a mutation in the source code leads to lines that can't be reached, and the compiler detects such situations. Also, binary code can become invalid, e.g. due to a mutation such that a literal falls outside its legal range. Both cases are good cases though, because just like a failing unit test, they indicate that the mutated source code mattered for the system's behaviour. In fact, since compilation errors or the invalidity of binary code will be detected long before the first unit test is run against the mutant, they're probably the best thing you can hope for when you're generating a mutant.

It should also be mentioned that there is a middle road between source code and binary code mutation which is particularly relevant for interpreted languages. For these languages, there is no real binary code, but it's often possible to access the parsed code as an abstract syntax tree (AST). This can be very useful in a language like Ruby, which allows in its syntax many ways to express what's fundamentally the same thing in the parsed code. This makes it easier to write the mutation testing tool, while at the same time speeding up the generation of mutants.

The next thing we have to consider when we want to build a mutation testing tool, is what **types of mutations** we want to inject. Basically, mutations fall into three categories. Some mutations never change a system's behaviour, like e.g. changing the value of an internal constant or flag that's only used for flow control. We don't really care whether we used 0, 1 or any other integer value, as long as it's unique within its context. Your unit tests shouldn't care about this either.

Other mutations can change the system's behaviour, but do not always do so. Switching between the less-than (**<**) and less-than-or-equal-to sign (≤) is an example of that, as we already showed in the beginning of this article. The problem with these types of mutations is that it's hard to find out when they do change the system's behaviour, and when they don't. In general, one has to do it manually, and therefore this type of mutations isn't very useful when we want to automate mutation testing.

Finally there are mutations that always change the system's behaviour – or should we say: *should* always change the system's behaviour. We already mentioned switching between the less-than (**<**) and greater-than-or-equal sign (≥). Here's a short list with mutation types that are guaranteed to change a system's behaviour, that is, if the code can be reached and actually matters:

- Negation of comparisons, like switching between = and ≠, < and ≥, and > and ≤.
- Negation of boolean conditions (¬)
- Short-cutting boolean conditions by replacing them with `True` or `False`
- Switching between addition and subtraction, and multiplication and division
- Switching between the increment and decrement operator

Simply removing a line should work too, but not every mutation testing tool tries to do that. Mind though that you have to watch out that you're not removing a variable declaration and therefore simply generating a compilation error. On the other hand, removing a whole line of code is often such a big change to the source code that you're almost guaranteed to find a unit test that will fail. It's therefore usually better to save the effort and rather concentrate on mutations that have a better chance of producing mutants passing all unit tests.

Finally, we have to find a good **strategy to select the unit tests** that we're going to run against the mutant. For every generated mutant, the strategy should find an order in which to run the unit tests such that those that fail are run as early as possible. This is crucial in order for the mutation testing tool to be useful, because a brute-force approach, i.e. just running all unit tests against all mutants in a random order will quickly become very time and resource consuming.

Let's illustrate the complexity of the unit test selection problem with a simple example. Assume our system consists of 50 classes, with 20 unit tests per class, and each unit test taking 1 ms on average. It's easy to verify that each full round with unit tests will take 1 s to complete. Now imagine that for every class, 10 mutants can be generated. If we use the brute-force strategy, i.e. we run all unit tests against all mutants, we'll need up to 6 m 20 s to complete the mutation testing round.

A smarter strategy to run the mutation testing round would be to run only those unit tests that are relevant for the class that was mutated in order to generate the mutant. In that case, we run only 20 unit tests of 1 ms for the 50 classes with 10 mutants, and that will take only 10 s in total.

Now let's scale the system up from 50 to 500 classes to make things a bit more realistic. When we do the same exercise as above, we find out that the brute-force strategy will need up to 14 hours to complete one round of mutation testing, whereas the second strategy will use less than 2 minutes. Clearly, just running all unit tests on all mutants isn't a good strategy. Even for a relatively small system with relatively fast unit tests, running a round of mutation testing becomes quickly infeasible.

We can estimate the complexity of mutation testing in the following manner. Assume the system has c classes, and that there are f function points per class. Clearly, f must be equal to or larger than 1. Assume further that we have t unit test per function point, and can generate μ mutants per function point. Again, both t and μ will be equal to or larger than 1. In fact, in contrast to f, which can be arbitrarily large, t and μ will in general be rather small numbers. There are normally not that many unit tests to be written for a single function point, and in the same way there won't be that many mutants that can be generated per function point.

Using the symbols from above, it is easy to see that there are $c \cdot f \cdot t$ unit tests in the system, and $c \cdot f \cdot \mu$ mutants. This means that for the brute-force strategy, there are potentially $c \cdot f \cdot t \times c \cdot f \cdot \mu = c^2 \cdot f^2 \cdot t \cdot \mu$ unit tests to be run in a single mutation testing round. Notice that this result is quadratic both in

This gives exact information about which unit tests are relevant for mutations in a given source code line, and this in a completely automated manner

terms of the number of classes and the number of function points per class, i.e. the total number of function points. This explains the explosion in the time required to run mutation testing in the example above: multiplying the system's functionality by a factor of 10 multiplies the resources needed to run mutation testing by a factor of 100.

Now consider the smarter strategy from above. It runs only those unit tests that are related to the class that has been mutated, and therefore the number of unit tests is reduced to $f{\cdot}t \times c{\cdot}f{\cdot}\mu = c{\cdot}f^2{\cdot}t{\cdot}\mu$. This number is still quadratic in the number of function points per class, but only linear in the number of classes. The ideal strategy would of course be to run only those unit tests that are related to the function point that has been mutated, and in that case the number is even further reduced to $t \times c{\cdot}f{\cdot}\mu = c{\cdot}f{\cdot}t{\cdot}\mu$. The lesson to be learned from these two results is that it's important to narrow down the number of unit tests that have to be run per mutation as much as possible if we want to keep mutation testing a feasible task even for realistically large systems.

As an aside-note, these results also suggest that it's a good idea to keep f, the number of function points per class, as small as possible in the per-class strategy. This is basically the same thing as saying that your classes should be small, which is a recommendation that shouldn't be too unfamiliar to the reader. It's nice to see that what's considered a good practice anyway can help improve the performance of mutation testing too.

Luckily, there are a number of strategies that can be used to find those unit tests for a mutant that have a reasonable chance of failing, and some of these strategies work quite well. First of all, one can use hints, e.g. in the form of annotations provided by the developer, as to which unit tests relate to which classes, parts of classes, or even modules. This can also be automated, e.g. by package and class name matching. If there's a convention that all unit tests reside in the same package as the source code that they're supposed to test, this can limit the number of unit tests per mutant drastically in an automated manner. If there's a convention that the unit tests for a class named `Foo` reside in a class named `FooUnitTest`, or perhaps in classes with names following the pattern `Foo*Test`, this will limit the number of unit tests per mutant even further down.

Another way to reduce the number of unit tests per mutant is to use a code coverage tool, and record which lines of source code are covered by which unit test. This gives exact information about which unit tests are relevant for mutations in a given source code line, and this in a completely automated manner. On the other side, this requires interaction with a code coverage tool, or if that's not possible, the implementation of a code coverage tool that is part of the mutation testing tool. Usually, the effort required to do this is a good investment, because it will improve the performance of the mutation testing tool dramatically.

One can also use automated learning to record which unit test failed for which mutant. Every time a mutant is generated, the tool looks up which unit test failed in the previous round, and tries to run that unit test again as the first one. Of course, this requires some bookkeeping, but if it works, the number of unit tests to be run in a single mutation round can be reduced to the absolute minimum $c{\cdot}f{\cdot}\mu$, i.e. simply the number of mutants.

```
def factorial(n) {
  i ← 0;
  f ← 1;
  while (i < n) {
    i++;
    f ← f × i;
  }
  return f;
}
def factorial_mutant(n) {
  i ← 0;
  f ← 1;
  while (true) {
    i++;
    f ← f × i;
  }
  return f;
}
```

Listing 10

So far we've dealt with how we can inject mutations, what kind of mutations we can inject and how we select the unit tests that we want to run against the mutant. In order to do this, we've had to deal with compilation errors, invalid binary code, equivalent mutations and the quadratically growing complexity of the problem. Could there be even more problems that we have to handle in order to build a good mutation testing tool?

Actually, there's still one problem left. Have a look at an implementation of the factorial function in Listing 10, together with one of its more interesting mutants. The mutant is generated by short-cutting the condition of the `while`-loop.

At first glance, this mutant has an infinite loop. However, when you think of it, there are a number of things that can happen, and not only does it depend on the programming language, but sometimes also on the compiler or the interpreter. If we're really lucky, the compiler spots the infinite loop, throws a compilation error, and we don't even have to run a unit test at all. Otherwise, we might get an exception when the multiplication hits the upper limit of the range of integers. Or it could be that the programming language allows arbitrarily long integers, and that the system continues to run until the multiplication generates such a large number that it doesn't fit into memory any more. Alternatively, the multiplication overflows, and the function continues to run forever. This means that there is indeed a potential for a real infinite loop in this mutant, but even if it's not completely infinite in the programming language, the compiler or the interpreter of your choice, it may take a long time before the final exception is thrown.

This means that we have to be able to detect infinite (or pseudo-infinite) loops. Luckily, that's not so difficult: we just have to measure the time a unit test normally takes, and decide how much longer it can take when we run it against a mutant before we interrupt it, and assume that the mutant contained a (pseudo-)infinite loop. A time factor of three may be reasonable, but we should also take care that we measure actual CPU time, and not just time elapsed. We don't want to interrupt a unit test that took a long time just because the operating system decided to allocate the CPU to another process with a higher priority.

Once we've detected a (pseudo-)infinite loop, we should be happy though. Sure, it takes some extra time compared to a unit test that passes, but again we discovered that the mutation at hand changed something in the system that mattered, and we can move on to the next mutant.

Mutation testing tools

Unfortunately, there aren't so many good tools available to run mutation testing, but the situation has been getting better and better these last years. If you're a Ruby programmer, you should check out Heckle [Ruby]. It integrates with Test::Unit and rSpec, and is usually run from the command line to run a set of unit tests on a method or a class. Mutations include booleans, numbers, string, symbols, ranges, regular expressions and branches. It has good to-the-point reporting and writes out the first mutant it can find that passes all the unit tests in a clear manner, with good performance. There is, however, virtually no documentation available. I've been able to write a manually configurable Rake task for Heckle, but only by letting Ruby call out to the command-line and catching and parsing whatever is output by Heckle. It's possible there's a better way to do this, but getting it right can be a difficult job without any real documentation.

Boo_hiss [BooHiss] is another mutation testing tool for Ruby, but since it hasn't had a single check-in for the last three years now, the project doesn't look like it's very much alive.

I've recently started to experiment with PIT [Pitest], a Java tool that integrates with both JUnit and TestNG. It can be run from the command-line, but it also has a plug-in for Maven. It's highly configurable, has some good defaults, and a website with clear documentation. PIT operates on the byte code and runs pretty fast thanks to a pre-round with a test coverage tool in order to map out which unit tests are relevant for mutations in which parts of the source code. Mutations include conditionals boundary, the negation of conditions, mathematical operators, increment operators, constants, return values, void and non-void method calls, and constructor calls. If you're a Java programmer and want to have a try at mutation testing, this is the tool I recommend for you. And since the project started not so long ago, there's still a lot of activity going on and many chances to get involved in the development of the tool.

Other mutation testing tools for Java include Jester [Jester], Jumble [Jumble], Javalanche [Javalanche] and µJava [Mujava]. None of the projects seem to be alive any more, and I wouldn't say they're an alternative for PIT. Jester integrates only with JUnit, and operates on the source code, which means that it's rather slow. Jumble is faster than Jester because it manipulates the Java byte code, but since it has no plug-in for Maven yet, it has to be run from the command-line. Javalanche and µJava have always seemed cumbersome to me, and seem mainly to be two academic exercises.

Jester has two ports to other languages: Nester [Nester] for C# and Pester for Python. A colleague of mine tested out Nester, and found quickly out that it works only for an older version of .Net. It's therefore probably not very relevant for any ongoing project, but could be a nice to play with and a good start if you want to create a new mutation testing tool for C#. Mutagenesis [Mutagenesis] is a mutation testing tool for PHP, but since I'm not familiar with PHP, I don't know whether it's any good.

I'm not a C++ programmer either, so I can only point the readers to CREAM [Cream] and PlexTest [Plextest] and encourage them to try the two tools out. If anybody does, or finds another tool for C++, I'm pretty sure the editor of *Overload* would be happy to receive an article with an experience report.

Improvements

As the reader may have guessed, I'm already glad there are mutation testing tools available that are usable at all. Indeed, these last years the evolution has been huge. Not so long ago, even the best mutation testing tools could not be considered as much more than proofs of concept, whereas today it is fully possible to use mutation testing tools in real-life projects. But there is still lots of room for improvement. Integration with common development environments is high on the list, together with better reporting on the mutants that don't make any of the unit tests fail. Unit test selection strategies seem to be becoming better and better. Parallellisation is certainly an issue that will be addressed in upcoming versions and future tools, since mutation testing is such a CPU intensive process.

Personal experiences and recommendations

So how can you get started with mutation testing? First of all, select a good tool. If you're a Java programmer, try PIT, and if you're a Ruby programmer, try Heckle. It's important that the tool is easily configurable and flexible, and that it can output mutants that survive the unit tests in a clear and meaningful manner. Start on a small code base, and ideally, start using mutation testing from day 1 in the project. If you start using it on an on-going project, you'll have the same problem as when you run static code analysis for the first time in the middle of the project: the tool will find an overwhelmingly long list of problems.

Once you've started using mutation testing, believe the tool when it reports that a mutant passed all the unit tests. And if you don't believe it, try to prove that it's wrong before you dismiss the mutant. It has happened to me a couple of times that the tool presented me with a mutant that I simply couldn't believe passed all the unit tests, but these have also been the occasions where I learned something new about programming and code quality.

As with other code quality techniques, when the tool finds a problem, fix the problem. And embrace the 'more than 100% test coverage'. Mutation testing leads to path coverage, more than simple line or branch coverage. When you use it, you'll notice that you'll write less, but more condensed code, with more and better unit tests. The reason for this is simple: if you write an extra line of code, there will be a couple of extra mutants, and some of them may pass the unit tests. Mutation testing will also influence your coding style, such that even when you work on a project where you can't use mutation testing, you'll write better code. I've used mutation testing for a couple of years now, mostly Heckle in Ruby, but recently also PIT in Java, and I feel it has helped me become a much better programmer than otherwise would have been the case. ■

References

[Andrews05] Andrews, Briand, Labiche, ICSE 2005
[BooHiss] See https://github.com/halorgium/boo_hiss
[Cream] See http://galera.ii.pw.edu.pl/~adr/CREAM/index.php
[Frankl97] Frankl, Weiss, Hu, *Journal of Systems and Software*, 1997
[Geist92] Geist et. al., 'Estimation and Enhancement of Real-time Software Reliability through Mutation Analysis', 1992
[Javalanche] See http://www.st.cs.uni-saarland.de/~schuler/javalanche/
[Jester] See http://jester.sourceforge.net/ and http://sourceforge.net/projects/grester/
[Jumble] See http://jumble.sourceforge.net/index.html
[Lipton71] R. Lipton, *Fault Diagnosis of Computer Programs*, 1971
[Lipton78] R. Lipton et. al., *Hints on Test Data Selection: Help for the Practicing Programmer*, 1978
[Mujava] See http://cs.gmu.edu/~offutt/mujava/
[Mutagenesis] See https://github.com/padraic/mutagenesis
[Nester] See http://nester.sourceforge.net/
[Pitest] See http://pitest.org/
[Plextest] See http://www.itregister.com.au/products/plextest_detail.htm
[Ruby] See http://rubyforge.org/projects/seattlerb/ and http://docs.seattlerb.org/heckle/
[Wah95] K. Wah, 'Fault Coupling in Finite Bijective Functions', 1995
[Walsh85] Walsh, Ph.D. Thesis, State University of New York at Binghampton, 1985

Unit Testing Compilation Failure

We usually test that our code does what we expect. Pete Barber tries to prove that his code fails to compile.

There is an issue with the use of Window's COM types being used with STL containers in versions of Visual Studio prior to 2010. The crux of the matter is that various COM types define the **`address-of`** operator and instead of returning the real address of the object they return a logical address. If the STL container re-arranges its contents this can lead to corruption as it can unwittingly manipulate an object at the incorrect address.

The prescribed solution to this problem is to not place the COM types directly into the STL container but instead wrap them with the ATL **`CAdapt`**[1] class. The merit or not of this approach is not the subject of this article. Instead, assuming this path is being followed, there are two important aspects to address:

- Firstly, is there a way to prevent incorrect usage of the containers?
- Secondly, assuming this is possible, is there a way to make sure it covers the various permutations of STL containers and COM types that are being used?

It turns out that the first issue is easily solved using template specialization. As shall be shown, this works very well to the extent that it prevents compilation. This is fine except that to write a set of unit tests for this requires them to handle the success case which is compilation failure!

The particular issue required finding uses of the following classes of which all but the first are class templates:

- **`CComBSTR`**
- **`CComPtr<>`**
- **`CComQIPtr<>`**
- **`_com_ptr_t<>`**

When used with the following STL containers:

- **`std::vector`**
- **`std::deque`**
- **`std::list`**
- **`std::map`**
- **`std::multimap`**
- **`std::set`**
- **`std::multiset`**

It wasn't necessary to explicitly handle **`std::stack`**, **`std::queue`** and **`std::priority_queue`** as these are implemented in terms of the previous classes.

In order to simplify the explanation the scope of the problem will be reduced to the use of the following custom type:

```
class Foo {};
```

used in **`std::vector`** , i.e.[2]

```
std::vector<Foo> vecOfFoo;
```

1. http://msdn.microsoft.com/en-us/library/bs6acf5x(v=VS.90).aspx
2. This class does not exhibit the **`address-of`** problem of the COM classes. This doesn't matter as the techniques to be shown do not detect the presence of the **`address-of`** operator but instead detect named classes.

```
template<typename A> class std::vector<Foo, A>
{
private:
  vector();
};
```

Listing 1

Part 1 – Detecting the problem

Preventing usage of **`std::vector<Foo>`** is fairly simple. All that's required is to provide a specialization of **`std::vector`** for **`Foo`** that contains a private constructor so it can't be constructed (for example, see Listing 1).

This now causes a compilation error, e.g.

```
error C2248: 'std::vector<_Ty>::vector' : cannot
access private member declared in class
'std::vector<_Ty>'
```

The specialization needs to be present before any usage. This is achieved by creating a new file containing the specialization called `prevent.h` that also includes `vector`. It also contains the declaration class **`Foo;`** to avoid circular dependencies. `Foo.h` is then modified to include `prevent.h`.

The specialization is in fact a partial specialization as the type parameter for the allocator remains. If it were fully specialized then the specialization would be

```
std::vector<Foo, std::allocator<Foo> >
```

This would not be sufficient to catch an instantiation that uses a custom allocator or, bizarrely, **`std::allocator`** instantiated with a type different to **`Foo`**.

In the real world situation `prevent.h` would be extended to include all of the specified containers and contain specializations for all the combinations of container and payload. This has the downside that every inclusion of `prevent.h` pulls in the headers for all the STL containers and requires parsing all the specializations.

This isn't a complete solution. The **`Foo`** class is equivalent to **`CComBSTR`** above as they're both concrete classes. Preventing the use of a class template in a container which is also a template is a little more complex. The scope of the example problem can be extended to a class template, e.g.

```
template<typename T> class Bar {};
```

To prevent this being used as a type parameter to **`std::vector`** the partial specialization in Listing 2 can be used. This will detect the use of any type used with **`Bar`** (for example, as shown in Listing 3).

Pete Barber has been programming in C++ since before templates and exceptions, C# since V1 & BASIC on a ZX Spectrum before then. He currently writes software for a living but mainly for fun. Occasionally hye writes about it too at petebarber.blogspot.com and CodeProject.com. He can be contacted at pete.barber@gmail.com.

various COM types **define the address-of operator** and instead of returning the real address of the object they **return a logical address** instead

```
template<typename T, typename A>
class std::vector<Bar<T>, A>
{
private:
  vector();
};
```

Listing 2

Part 2 – Testing the detection mechanism

Some basic manual testing with the previous samples (well, the real types in question) gives a general feeling that the mechanism is working, i.e. any use causes a compilation failure. However, when all the different combinations are considered there are too many to test manually. This is very true if a test program is written that contains examples of each, as compilation will fail at the first usage: a manual cycle of compile, fail, comment-out failed code and move onto the next failure is required.

This calls for some form of automated testing. At this level the most familiar and applicable is Unit Testing. Unit Testing is generally considered to be automatic testing, which causes a problem as automating the manual tests is not just a case of creating a set of fixtures. This is because a successful test results in a compilation failure!

Using various template based mechanisms and techniques, it is possible to check that the detection mechanism does indeed capture all of the combinations without explicitly compiling – well, failing to compile – each combination.

As it seems impossible at the normal compilation level to detect code that will not compile, the next best thing is to attempt to detect the presence of code that will prevent compilation. In this case it means testing for the presence of the specializations.

SFINAE

The first mechanism required is Substitution Failure Is Not An Error (SFINAE) [Wikipedia]; a concept introduced by Davide Vandevoorde [Vandevoorde02]. To facilitate this, a line of code must be added to the partial specialization. Any other specialization (full or partial) would also require this line adding (see Listing 4).

```
class Dummy {};

void OtherUsesOfBar()
{
  std::vector<Bar<int*> > vecOfBarIntPtr;
  std::vector<Bar<Dummy> > vecOfBarDummy;
  std::vector<Bar<const Dummy*> >
    vecOfBarDummyConstPtr;
}
```

Listing 3

```
template<typename A>
class std::vector<Foo, A>
{
private:
  vector();
public:
  typedef std::vector<Foo, A> self_t;
};
```

Listing 4

A class is created with an overloaded method which returns **`true`** if a specialized version of **`std::vector`** is detected and **`false`** otherwise (see Listing 5).

Its use is as follows:

```
typedef std::vector<Bar<int>> BarInt_t;

const bool detectVectorOfBarInt =
  DetectPreventionWithSFINAE::Detect<BarInt_t>(
  BarInt_t());
```

If the type passed to **`DetectPreventionWithSFINAE`** does not contain the **`typedef self_t`** then the first version of **`Detect`** is invalid code. Rather than this being an error, the compiler just removes this overload from the available overload set and the method is not instantiated. This leaves the other overload, which as it has ellipses for its argument will match anything, but only if nothing more specific is available. In this case it is the only method, meaning it will be used and when invoked it will return **`false`**. The removal of the invalid code rather than a compilation error is SFINAE.

In the case where the type passed does contain the **`typedef self_t`**, i.e. it is one of the specializations, then both overloads will be valid but as the first one is the better match it will be chosen. When this method is invoked it will return **`true`**.

```
class DetectPreventionWithSFINAE
{
public:
  template<typename U>
  static bool Detect(const typename U::self_t)
  {
    return true;
  }

  template<typename U>
  static bool Detect(...)
  {
    return false;
  }
};
```

Listing 5

Some of these techniques, despite being fairly old, are still relatively unknown and their use somewhat obscure

NOTE: The use of `self_t` is not fool proof. If its definition is neglected from a specialization then an instance of that specialization will not match that case. Conversely if an altogether different specialization of `std::vector` is created which contains the same named `typedef` to itself this will also match.

sizeof

Using SFINAE the presence of the specialization can be detected, but unfortunately this requires creating an instance of it, which of course can't be done. In fact the example above is misleading as the code will not compile. Additionally, assuming the code could compile, passing an object to a method with ellipses as its argument has undefined results so this code couldn't be reliably used.

This seems to be almost back at square one: the presence of the prevention code can be detected but it requires an instance of the specialization that can't be compiled in order to do so! However, examining the use of the instance it becomes clear that the instance isn't actually used. It's only needed in order to interrogate its type. This is the realm of Template Meta Programming (TMP). A fair amount of TMP can be performed without creating instances of types but in this case – as SFINAE is being used, which is tied to the overloading of function templates – an instance that can be passed as a parameter to a function template is required.

Fortunately the innovative use of `sizeof` as introduced by Andrei Alexandrescu [Alexandrescu01] in his seminal work *Modern C++ Design* can be used.[3,4] This is shown in an updated version of the `DetectPrevention` class (see Listing 6).

```
template<typename T> class DetectPrevention
{
  typedef char Small;
  class Big { char dummy[2]; };

public:
  static bool Detect()
  {
    return sizeof(DetectImpl<T>(MakeT())) ==
       sizeof(Small);
  }

private:
  template<typename U> static
     Small DetectImpl(const typename U::self_t);
  template<typename U> static
     Big DetectImpl (...);

  static T MakeT();
};
```

Listing 6

The key point is that if a type is used within the `sizeof` operator then the compiler does not have to actually create an instance of it in order to determine its size. As shown by Alexandrescu, the real magic is that `sizeof` is not just limited to taking a type. It can be given an entire expression which, as long as it results in a type (so a type name, a function invocation, i.e. calling not just referencing a method that returns a value, so non-void or a logical expression resulting in a Boolean), works.

Another essential bit of magic is the `MakeT` method. Even though an instance is required for the overloaded `DetectImpl` methods, as these are only ever referenced within the `sizeof` operator the instance doesn't actually need creating. If `T()` rather than the call to `MakeT` were specified in the definition of `Detect`, e.g.

```
static bool Detect()
{
  return sizeof(DetectImpl<T>(T())) ==
     sizeof(Small);
}
```

then this won't compile as the compiler checks that a constructor is available, and this is not the case as the specialization deliberately makes it private. However, as `sizeof` only needs to calculate the size as opposed to constructing an instance of `T`, the `MakeT` method provides the *potential* to create one, which is sufficient. This is demonstrated by the fact there is no definition.

The 'no definition' theme is extended to the `DetectImpl` methods, as these too are never executed but purely evaluated at compile time. This is important given the earlier statement that passing an object, in this case `T`, to a method taking ellipses could lead to undefined results.

In the previous non-working example, the `Detect` methods return a Boolean to indicate whether the specialization was detected or not. As the `DetectImpl` methods have no implementation and are not invoked, it is not possible for them to return a value. Instead, another technique from Alexandrescu is used. This again returns to the use of `sizeof`, which in this case evaluates the size of the return type of whichever overload of `DetectImpl` was chosen. This is then compared to the size of the type returned by the chosen method. If these are equal then it means the overload of `DetectImpl` that requires the specialization was chosen; if not, the other method was chosen. This use of the size comparison then converts the application of SFINAE into a Boolean result, which can be used at compile time.

Any differently sized types could have been used for return values; however, as some can vary on a platform basis explicitly defining them avoids any future problems, hence the use of `Big` and `Small`.[5]

At this point the presence or lack of the specialization can be detected as shown in Listing 7.

3. The explanation of how `sizeof` works in this context is derived from this book, quite possibly almost verbatim. The application is different.
4. The combination of SFINAE and `sizeof` as used here was first discovered by Paul Mensonides.
5. Typedefs of `yes` and `no` are also common.

```
#include <iostream>
#include <vector>
#include "prevent.h"
#include "bar.h"
#include "DetectPrevention.h"

int main()
{

  bool isPreventer =
     DetectPrevention<std::vector<Bar<int>>>
     ::Detect();

  std::cout << "isPreventer:" << isPreventer
     << std::endl;
}
```

Listing 7

Part 3 – What about the other STL Containers and types?

The problem faced was the prevention and detection of a limited number of types against a limited number of STL Containers. For the purposes of the example the types to be placed in the containers are:

- **`Foo`**
- **`Bar<>`**

The partial specialization will catch every use of **`Bar<T>`** in production code. As the set of **`T`** is effectively infinite, testing the detection of **`Bar<T>`** used within the containers is limited to a representative set of types. In the real world scenario, as the problematic template types were all wrappers for COM objects, just using them with **`IUnknown`** sufficed. For this example **`int`** will continue to be used. If a warmer feeling is required that different instantiations are being caught then the examples can be extended to use additional types.

At the moment, the use of **`Foo`** and **`Bar<>`** can be prevented and detected in **`std::vector`**. This can be easily extended to the remaining applicable STL Containers: **`deque`**, **`list`**, **`map`**, **`multimap`**, **`set`** and **`multiset`**, by creating specializations for **`Foo`** and **`Bar<>`** in a similar manner to that of **`std::vector`**.

The source is too long to show here for all the containers but is included in the example code. In Listing 8 are the partial specializations for **`std::vector`** and **`std::map`** to handle **`Foo`** and **`Bar<>`**.

To avoid repeating lots of test code another template mechanism is used. This is template-templates. The code sample in Listing 9 shows a reduced set of tests for detecting the prevention of **`Foo`** and **`Bar<int>`** when used to instantiate a **`vector`**, **`deque`**, **`list`**, **`map`** and **`multimap`**.

The tests for the **`vector`**, **`deque`** and **`list`** are identical so rather than repeating the code for each container type the container type is a template parameter. This requires the use of template-template parameters as rather than passing an instantiated template, i.e. a concrete type, a container to test is passed instead. This is then instantiated with different types within the test method, e.g. **`Foo`** and **`Bar<int>`**.

Conclusion

Attempting to create a Unit Test (well some form of automated test) to determine if code wouldn't compile without actually stopping it from compiling was a daunting task. The application of various template techniques showed that in combination this problem could be addressed. Some of these techniques, despite being fairly old, are still relatively unknown and their use somewhat obscure. This article shows just the simple application of them can be used to solve practical problems. ■

```
// Prevent use of std::vector with Foo
template<typename A>
class std::vector<Foo, A>
{
private:
  vector();

public:
  typedef std::vector<Foo, A> self_t;
};


// Prevent use of std::vector with Bar<T>
template<typename T, typename A>
class std::vector<Bar<T>, A>
{
private:
  vector();

public:
  typedef std::vector<Bar<T>, A> self_t;
};


// Prevent use of std::map with Foo
template<typename T1, typename T2, typename T3>
class std::map<Foo, T1, T2, T3>
{
private:
  map();

public:
  typedef std::map<Foo, T1, T2, T3> self_t;
};

template<typename T, typename T2, typename T3>
class std::map<T, Foo, T2, T3>
{
private:
  map();

public:
  typedef std::map<T, Foo, T2, T3> self_t;
};

template<typename T2, typename T3>
class std::map<Foo, Foo, T2, T3>
{
private:
  map();

public:
  typedef std::map<Foo, Foo, T2, T3> self_t;
};


// Prevent use of std::map with Bar<T>
template<typename T, typename T1,
         typename T2, typename T3>
class std::map<Bar<T>, T1, T2, T3>
{
private:
  map();

public:
  typedef std::map<Bar<T>, T1, T2, T3> self_t;
};
```

Listing 8

```
template<typename T, typename T1,
         typename T2, typename T3>
class std::map<T, Bar<T1>, T2, T3>
{
private:
  map();

public:
  typedef std::map<T, Bar<T1>, T2, T3> self_t;
};

template<typename T, typename T1,
         typename T2, typename T3>
class std::map<Bar<T>, Bar<T1>, T2, T3>
{
private:
  map();

public:
  typedef std::map<Bar<T>, Bar<T1>, T2, T3>
self_t;
};
```

Listing 8 (cont'd)

```
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include "prevent.h"
#include "foo.h"
#include "bar.h"
#include "DetectPrevention.h"

using namespace std;

template<template<typename, typename> class CONT>
bool TestSimpleContainer()
{
  const bool detectFoo =
     DetectPrevention<CONT<Foo,
        allocator<Foo> > >::Detect();
  const bool detectBarT  =
     DetectPrevention<CONT<Bar<int>,
        allocator<Bar<int> > > >::Detect();
  return detectFoo && detectBarT;
}

template<template<typename, typename,
         typename, typename> class MAP>
bool TestMap()
{
  typedef int Placeholder_t;

  typedef MAP<Foo, Placeholder_t, less<Foo>,
     allocator<pair<Foo,
     Placeholder_t> > > FooAsKey_t;
  typedef MAP<Placeholder_t, Foo,
     less<Placeholder_t>,
     allocator<pair<Placeholder_t,
     Foo> > > FooAsValue_t;
  typedef MAP<Foo, Foo, less<Foo>,
     allocator<pair<Foo,
     Foo> > > FooAsKeyAndValue_t;
```

Listing 9

```
  const bool detectFooAsKey =
     DetectPrevention<FooAsKey_t>::Detect();

  const bool detectFooAsValue =
     DetectPrevention<FooAsValue_t>::Detect();

  const bool detectFooAsKeyAndValue =
     DetectPrevention<FooAsKeyAndValue_t>
     ::Detect();

  // Bar<int>
  typedef MAP<Bar<int>, Placeholder_t,
     less<Bar<int> >, allocator<pair<Bar<int>,
     Placeholder_t> > > BarAsKey_t;

  typedef MAP<Placeholder_t, Bar<int>,
     less<Placeholder_t>,
     allocator<pair<Placeholder_t,
     Bar<int> > > > BarAsValue_t;

  typedef MAP<Bar<int>, Bar<int>, less<Bar<int>>,
     allocator<pair<Bar<int>,
     Bar<int> > > > BarAsKeyAndValue_t;

  const bool detectBarAsKey =
     DetectPrevention<BarAsKey_t>::Detect();

  const bool detectBarAsValue =
     DetectPrevention<BarAsValue_t>::Detect();

  const bool detectBarAsKeyAndValue =
     DetectPrevention<BarAsKeyAndValue_t>
     ::Detect();

  return detectFooAsKey          &&
         detectFooAsValue        &&
         detectFooAsKeyAndValue  &&
         detectBarAsKey          &&
         detectBarAsValue        &&
         detectBarAsKeyAndValue;
}

int main()
{
  cout << "vector:"
     << TestSimpleContainer<vector>() << endl;
  cout << "deque:"
     << TestSimpleContainer<deque>() << endl;
  cout << "list:"
     << TestSimpleContainer<list>() << endl;
  cout << "map:"
     << TestMap<map>() << endl;
  cout << "multimap:"
     << TestMap<multimap>() << endl;
}
```

Listing 9 (cont'd)

References

[Alexandrescu01] Alexandrescu, Andrei. *Modern C++ Design: Generic Programming and Design Patterns* Applied. 2001.

[Vandevoorde02] Vandevoorde, David & Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. 2002.

[Wikipedia] 'Substitution failure is not an error.' Wikipedia. http://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error

Refactoring Towards Seams in C++

Breaking dependencies in existing code is hard. Michael Rüegg explains how seams can help and provides new automated refactorings for C++ to achieve them.

Unwanted dependencies are a critical problem in software development. We often have to break existing dependencies before we can change some piece of code. Breaking existing dependencies is also an important preliminary to introduce unit tests for legacy code — according to Feathers definition code without unit tests [Feathers04].

Feathers' *seams* help in reasoning about the opportunities that exist when we have to break dependencies. The goal is to have a place where we can alter the behaviour of a program without modifying it in that place. This is important because editing the source code is often not an option (e. g., when a function the code depends on is provided by a system library).

C++ offers a wide variety of language mechanisms to create seams. Beside the classic way of using subtype polymorphism which relies on inheritance, C++ also provides static polymorphism through template parameters. With the help of the preprocessor or the linker we have additional ways of creating seams.

Once we have broken dependencies in our legacy code base by introducing seams, our code is not relying on fixed dependencies anymore, but instead asks for collaborators through dependency injection.

Not only has our design improved much, but we are now also able to write unit tests for our code.

The seam types discussed in this article are often hard and time-consuming to achieve by hand. This is why automated refactorings and IDE support would be beneficial. The problem is that current C++ IDE's do not offer this in the extent we describe it here. Therefore, we will explain how to create these seam types by applying refactorings which we have implemented for our IDE of choice, Eclipse C/C++ Development Tooling (CDT).

The classic way: object seam

Object seams are probably the most common seam type. To start with an example, consider Listing 1 where the class **GameFourWins** has a hard coded dependency to **Die**. According to Feathers' definition, the call to **play** is not a seam because it is missing an enabling point. We cannot alter the behaviour of the member function **play** without changing its function body because the used member variable **die** is based on the concrete class **Die**. Furthermore, we cannot subclass **GameFourWins** and override **play** because **play** is monomorphic (not virtual).

This fixed dependency also makes it hard to test **GameFourWins** in isolation because **Die** uses C's standard library pseudo-random number generator function **rand**. Although **rand** is a deterministic function since calls to it will return the same sequence of numbers for any given seed, it is hard and cumbersome to setup a specific seed for our purposes.

Michael Rüegg is a scientific assistant at the Institute for Software of University of Applied Sciences Rapperswil. He is currently doing his Master Thesis under the supervision of Prof. Peter Sommerlad. The outcome of this thesis is Mockator.

What is a seam?

Feathers characterises a *seam* as a place in our code base where we can alter behaviour without being forced to edit it in that place. This has the advantage that we can inject the dependencies from outside, which leads to both an improved design and better testability. Every seam has one important property: an *enabling point*. This is the place where we can choose between one or another. There are different kinds of seam types. We focus on object, compile, preprocessor and link seams in this

The classic way to alter the behaviour of **GameFourWins** is to inject the dependency from outside. The injected class inherits from a base class, thus enabling subtype polymorphism. To achieve that, we first apply the refactoring *Extract Interface* [Fowler99]. Then we provide a constructor to pass the dependency from outside (we could also have passed an instance of **Die** to play directly). The resulting code is shown in Listing 2.

This way we can now inject a different kind of **Die** depending on the context we need. This is a seam because we now have an enabling point: the instance of **Die** that is passed to the constructor of **GameFourWins**.

Leverage your compiler: compile seam

Although object seams are the classic way of injecting dependencies, we think there is often a better solution to achieve the same goals. C++ has a tool for this job providing static polymorphism: template parameters. With template parameters, we can inject dependencies at compile-time. We therefore call this seam *compile seam*.

```
// Die.h
struct Die {
int roll () const ;
};
// Die.cpp
int Die :: roll () const {
  return rand () % 6 + 1;
}
// GameFourWins .h
struct GameFourWins {
  void play (std :: ostream & os);
private :
  Die die;
};
// GameFourWins .cpp
void GameFourWins :: play (std :: ostream & os
   = std :: cout ) {
  if ( die. roll () == 4) {
    os << "You won !" << std :: endl ;
  } else {
    os << "You lost !" << std :: endl ;
  }
}
```

Listing 1

The goal is to have a place where we can **alter the behaviour of a program** without modifying it in that place

```
struct IDie {
  virtual ~ IDie () {}
  virtual int roll () const =0;
};
struct Die : IDie {
  int roll () const {
    return rand () % 6 + 1;
  }
};
struct GameFourWins {
  GameFourWins ( IDie & die) : die(die ) {}
  void play (std :: ostream & os = std :: cout ) {
    // as before
  }
private :
  IDie & die;
};
```

Listing 2

The essential step for this seam type is the application of a new refactoring we named *Extract Template Parameter* [Thrier10]. The result of this refactoring can be seen in Listing 3. The enabling point of this seam is the place where the template class **`GameFourWinsT`** is instantiated.

One might argue that we ignore the intrusion of testability into our production code: we have to template a class in order to inject a dependency, where this might only be an issue during testing. The approach taken by our refactoring is to create a **`typedef`** which instantiates the template with the concrete type that has been used before applying the refactoring. This has the advantage that we do not break existing code.

The use of static polymorphism with template parameters has several advantages over object seams with subtype polymorphism. It does not incur the run-time overhead of calling virtual member functions that can be unacceptable for certain systems. This overhead results due to pointer indirection, the necessary initialisation of the vtable (the table of pointers to its member functions) and because virtual functions usually cannot be inlined by compilers. Beside performance considerations, there is also an increased space-cost due to the additional pointer per object that has to be stored for the vtable [Driesen96].

```
template <typename Dice =Die >
struct GameFourWinsT {
  void play (std :: ostream &os = std :: cout ){
    if ( die. roll () == 4) {
      os << "You won !" << std :: endl ;
    } else {
      os << "You lost !" << std :: endl ;
    }
  }
private :
  Dice die;
};
typedef GameFourWinsT <Die > GameFourWins ;
```

Listing 3

What is compile-time duck typing?

Duck typing is the name of a concept that can be shortly described as follows:

> If something *quacks like a duck* and *walks like a duck*, then we treat it as a **`duck`**, without verifying if it is of type *duck*.

Although duck typing is mainly used in the context of dynamically typed programming languages, C++ offers duck typing at compile-time with templates. Instead of explicitly specifying an interface our type has to inherit from, our *duck* (the template argument) just has to provide the features that are used in the template definition.

Beside performance and space considerations, inheritance brings all the well-known software engineering problems like tight coupling, enhanced complexity and fragility with it [Sutter04, Meyers05]. Most of these disadvantages can be avoided with the use of templates. Probably the most important advantage of using templates is that a template argument only needs to define the members that are actually used by the instantiation of the template (providing compile-time duck typing). This can ease the burden of an otherwise wide interface that one might need to implement in case of an object seam.

Of course, there are also drawbacks of using templates in C++. They can lead to increased compiletimes, code bloat when used naively and (sometimes) reduced clarity. The latter is because of the missing support for concepts even with C++11, therefore solely relying on naming and documentation of template parameters. A further disadvantage is that we have to expose our implementation (template definition) to the clients of our code which is sometimes problematic.

Apart from these objections, we are convinced that this seam type should be preferred wherever possible over object seams.

In case of emergency only: preprocessing seam

C and C++ offer another possibility to alter the behaviour of code without touching it in that place using the preprocessor. Although we are able to change the behaviour of existing code as shown with object and compile seams before, we think preprocessor seams are especially useful for debugging purposes like tracing function calls. An example of this is shown in Listing 4 where we exhibit how calls to C's **`malloc`** function can be traced for statistical purposes.

The enabling point for this seam are the options of our compiler to choose between the real and our tracing implementation. We use the option **`-include`** of the GNU compiler here to include the header file `malloc.h` into every translation unit. With **`#undef`** we are still able to call the original implementation of **`malloc`**.

We strongly suggest to not using the preprocessor excessively in C++. The preprocessor is just a limited text-replacement tool lacking type safety that

Although from the language standpoint the **order in which the linker processes the files given is undefined**, it has to be specified by the tool chain

causes hard to track bugs. Nevertheless, it comes in handy for this task. Note that a disadvantage of this seam type is that we cannot trace member functions. Furthermore, if a member function has the same name as the macro the substitution takes place inadvertently.

Tweak your build scripts: link seam

Beside the separate preprocessing step that occurs before compilation, we also have a post-compilation step called linking in C and C++ that is used to combine the results the compiler has emitted. The linker gives us another kind of seam called *link seam* [Feathers04].

Myers and Bazinet discussed how to intercept functions with the linker for the programming language C [Myers04]. Our contribution is to show how link seams can be accomplished in C++ where name mangling comes into play. We present three possibilities of using the linker to intercept or shadow function calls.

Although all of them are specific to the used tool chain and platform, they have one property in common: their enabling point lies outside of the code, i.e., in our build scripts. When doing link seams, we create separate libraries for code we want as a replacement. This allows us to adapt our build scripts to either link to those for testing rather than to the production ones [Feathers04].

Shadow functions through linking order

In this type of link seam we make use of the linking order. Although from the language standpoint the order in which the linker processes the files given is undefined, it has to be specified by the tool chain [Gough04]: 'The traditional behaviour of linkers is to search for external functions from left to right in the libraries specified on the command line. This means that a library containing the definition of a function should appear after any source files or object files which use it.'

The linker incorporates any undefined symbols from libraries which have not been defined in the given object files. If we pass the object files first before the libraries with the functions we want to replace, the GNU linker prefers them over those provided by the libraries. Note that this would not work if we placed the library before the object files. In this case, the linker would take the symbol from the library and yield a duplicate definition error when considering the object file.

As an example, consider these commands for the code shown in Listing 5.

```
$ ar -r libGame.a Die.o GameFourWins.o
$ g++ -Ldir /to/ GameLib -o Test test.o
shadow_roll.o -lGame
```

The order given to the linker is exactly as we need it to prefer the symbol in the object file since the library comes at the end of the list. This list is the enabling point of this kind of link seam. If we leave `shadow_roll.o` out, the original version of `roll` is called as defined in the static library `libGame.a`.

We have noticed that the GNU linker for Mac OS X (tested with GCC 4.6.3) needs the shadowed function to be defined as a weak symbol; otherwise the linker always takes the symbol from the library. Weak symbols are one of the many function attributes the GNU tool chain offers. If the linker comes across a strong symbol (the default) with the same name as the weak one, the latter will be overwritten. In general, the linker uses the following rules [Bryant10]:

1. Not allowed are multiple strong symbols.
2. Choose the strong symbol if given a strong and multiple weak symbols.
3. Choose any of the weak symbols if given multiple weak symbols.

With the to be shadowed function defined as a weak symbol, the GNU linker for Mac OS X prefers the strong symbol with our replacement code. The following shows the function declaration with the `weak` attribute.

```
struct Die {
  __attribute__ (( weak )) int roll () const ;
};
```

This type of link seam has one big disadvantage: it is not possible to call the original function anymore. This would be valuable if we just want to wrap the call for logging or analysis purposes or do something additional with the result of the function call.

```
// malloc .h
# ifndef MALLOC_H_
# define MALLOC_H_
void * my_malloc ( size_t size ,
   const char * fileName , int lineNumber );
# define malloc ( size ) my_malloc (( size ),
   __FILE__ , __LINE__ )
# endif
// malloc .cpp
# include " malloc .h"
# undef malloc
void * my_malloc ( size_t size ,
   const char * fileName , int lineNumber ) {
  // remember allocation in statistics
  return malloc ( size );
}
```

Listing 4

```
// GameFourWins .cpp and Die.cpp as in Listing 1
// shadow_roll .cpp
# include " Die.h"
int Die :: roll () const {
  return 4;
}
// test .cpp
void testGameFourWins () {
  // ...
}
```

Listing 5

we try to achieve a **generic solution** and do not want to specify a **specific library** here

```
extern "C" {
  extern int __real__ZNK3Die4rollEv ();
  int __wrap__ZNK3Die4rollEv () {
    // your intercepting functionality here
    // ...
    return __real__ZNK3Die4rollEv ();
  }
}
```

Listing 6

Wrapping functions with GNU's linker

The GNU linker **ld** provides a lesser-known feature which helps us to call the original function. This feature is available as a command line option called **wrap**. The man page of **ld** describes its functionality as follows: 'Use a wrapper function for **symbol**. Any undefined reference to **symbol** will be resolved to **__wrap_symbol**. Any undefined reference to **__real_symbol** will be resolved to **symbol**.'

As an example, we compile GameFourWins.cpp from Listing 1. If we study the symbols of the object file, we see that the call to **Die::roll** – mangled as **_ZNK3Die4rollEv** according to Itanium's Application Binary Interface (ABI) that is used by GCC v4.x – is undefined (**nm** yields **U** for undefined symbols).

```
$ gcc -c GameFourWins.cpp -o GameFourWins.o
$ nm GameFourWins.o | grep roll
U _ZNK3Die4rollEv
```

This satisfies the condition of an undefined reference to a symbol. Thus we can apply a wrapper function here. Note that this would not be true if the definition of the function **Die::roll** would be in the same translation unit as its calling origin. If we now define a function according to the specified naming schema **__wrap_symbol** and use the linker flag **-wrap**, our function gets called instead of the original one. Listing 6 presents the definition of the wrapper function. To prevent the compiler from mangling the mangled name again, we need to define it in a C code block.

Note that we also have to declare the function **__real_symbol** which we delegate to in order to satisfy the compiler. The linker will resolve this symbol to the original implementation of **Die::roll**. The following demonstrates the command line options necessary for this kind of link seam.

```
$ g++ -Xlinker -wrap = _ZNK3Die4rollEv -o Test
test.o GameFourWins.o Die.o
```

Alas, this feature is only available with the GNU tool chain on Linux. GCC for Mac OS X does not offer the linker flag **-wrap**. A further constraint is that it does not work with inline functions but this is the case with all link seams presented in this article. Additionally, when the function to be wrapped is part of a shared library, we cannot use this option. Finally, because of the mangled names, this type of link seam is much harder to achieve by hand compared to shadowing functions.

```
# include <dlfcn .h>
int foo(int i) {
  typedef int (* funPtr )( int);
  static funPtr orig = nullptr ;
  if (! orig ) {
    void *tmp = dlsym ( RTLD_NEXT , " _Z3fooi ");
    orig = reinterpret_cast <funPtr >( tmp );
  }
  // your intercepting functionality here
  return orig (i);
}
```

Listing 7

Run-time function interception

If we have to intercept functions from shared libraries, we can use this kind of link seam. It is based on the fact that it is possible to alter the run-time linking behaviour of the loader **ld.so** in a way that it considers libraries that would otherwise not be loaded. This can be accomplished by the environment variable **LD_PRELOAD** that the loader **ld.so** interprets. Its functionality is described in the man page of **ld.so** as follows: 'A white space-separated list of additional, user-specified, ELF shared libraries to be loaded before all others. This can be used to selectively override functions in other shared libraries.'[1] With this we can instruct the loader to prefer our function instead of the ones provided by libraries normally resolved through the environment variable **LD_LIBRARY_PATH** or the system library directories.

Consider we want to intercept a function **foo** which is defined in a shared library. We have to put the code for our intercepting function into its own shared library (e. g., **libFoo.so**). If we call our program by appending this library to **LD_PRELOAD** as shown below, our definition of **foo** is called instead of the original one.

```
$ LD_PRELOAD = path /to/ libFoo .so; executable
```

Of course this solution is not perfect yet because it would not allow us to call the original function. This task can be achieved with the function **dlsym** the dynamic linking loader provides. **dlsym** takes a handle of a dynamic library we normally get by calling **dlopen**. Because we try to achieve a generic solution and do not want to specify a specific library here, we can use a pseudo-handle that is offered by the loader called **RTLD_NEXT**. With this, the loader will find the next occurrence of a symbol in the search order *after* the library the call resides.

As an example, consider Listing 7 which shows the definition of the intercepting function **foo** and the code necessary to call the original function. Note that we cache the result of the symbol resolution to avoid the process being made with every function call. Because we call a C++ function, we have to use the mangled name **_Z3fooi** for the symbol name. Furthermore, as it is not possible in C++ to implicitly cast the void pointer returned by **dlsym** to a function pointer, we have to use an explicit cast.

1. This indeed sounds like a security hole; but the man page says that **LD_PRELOAD** is ignored if the executable is a setuid or setgid binary.

Some of these techniques despite being fairly old now are still **relatively unknown** and **their use somewhat obscure**

The advantage of this solution compared to the first two link seams is that it does not require re-linking. It is solely based on altering the behaviour of `ld.so`. A disadvantage is that this mechanism is unreliable with member functions, because the member function pointer is not expected to have the same size as a void pointer. There is no reliable, portable and standards compliant way to handle this issue. Even the conversion of a void pointer to a function pointer was not defined in C++03[1].

Note that environment variables have different names in Mac OS X. The counterpart of `LD_PRELOAD` is called `DYLD_INSERT_LIBRARIES`. This needs the environment variable `DYLD_FORCE_FLAT_NAMESPACE` to be set.

Our IDE support for Eclipse CDT

Based on the seam types shown in this article, we have implemented Mockator. Mockator is a plug-in for the Eclipse CDT platform including a C++ based mock object library. The plug-in contains our implementation of the two refactorings *Extract Interface* and *Extract Template Parameter* to refactor towards the seam types object and compile seam.

For the preprocessor seam, our plug-in creates the necessary code as shown in Listing 4 and registers the header file with the macro in the project settings by using the `-include` option of the GNU compiler. Activating and deactivating of the traced function is supported by clicking on a marker that resides beside the function definition.

The plug-in supports all link seam types presented in 'Tweak your build scripts: link seam'. The user selects a function to be shadowed or wrapped and the plug-in creates the necessary infrastructure. This includes the creation of a library project with the code for the wrapped function, the adjustment of the project settings (library paths, definition of preprocessor macros, linker options) based on the chosen tool chain and underlying platform and the creation of a run-time configuration necessary for the run-time function interception. To implement the presented link seams, we had to mangle C++ functions. Because we did not want to call the compiler and analyse the result with a tool like nm which would lead to both performance problems and unnecessary tool dependencies, we decided to implement name mangling according to the Itanium ABI in our plug-in.

For an upcoming publicly available release of Mockator, we plan to support further tool chains in Eclipse CDT beside GCC like Microsoft's C++ compiler. Our plug-in will be bundled with our unit testing framework of choice, CUTE [Sommerlad11]. Beside its sophisticated support for unit testing, CUTE will then also assist the developer in refactoring towards seams and in creating mock objects.

Conclusion and outlook

Although we are convinced that the described practices in this article are valuable especially in testing and debugging, they are not used as much as they should be. We think this is because of the large amount of manual work that needs to be done by the programmer which is both tedious and error-prone. With our Eclipse plug-in, we automate these tasks as far as possible which has the benefit that the programmer can focus on what really counts: Refactor its code to enable unit testing and finding bugs in its legacy code base.

Refactoring towards seams enables us to unit test our code. For our unit tests we sometimes want to use test doubles like fake or mock objects instead of real objects to control dependencies. In the upcoming article we will discuss how we think unit testing with mock objects should be done leveraged by the new language features C++11 give us. ■

1. This has changed now with C++11 where it is implementation-defined.

What is dlsym?

Symbols linked from shared libraries are normally automatically available. The dynamic linker loader `ld.so` offers four library functions `dlopen`, `dlclose`, `dlsym` and `dlerror` to manually load and access symbols from a shared library. We only use `dlsym` in this article. This function can be used to look up a symbol by a given name. The dynamic linker loader yields a void pointer for the symbol as its result. `dlsym` has the following function prototype:

```
void* dlsym (void *handle, char *symbol_name);
```

Note that `dlfcn.h` has to be included and the compiler flag `-ldl` is necessary for linking to make this work.

References and further reading

[Bryant10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2nd edition, 2010.

[Driesen96] Karel Driesen and Urs Hoelzle. 'The Direct Cost of Virtual Function Calls in C++'. *SIGPLAN Not.*, 31:306–323, October 1996.

[Feathers04] Michael C. Feathers. *Working Effectively With Legacy Code*. Prentice Hall PTR, 2004.

[Fowler99] Martin Fowler. *Refactoring*. Addison-Wesley, 1999.

[Gough04] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[Myers04] Daniel S. Myers and Adam L. Bazinet. 'Intercepting Arbitrary Functions on Windows, Unix, and Macintosh OS X Platforms'. *Technical report*, Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, 2004.

[Meyers05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, May 2005.

[Sutter04] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, November 2004.

[Sommerlad11] Peter Sommerlad. 'CUTE – C++ Unit Testing Easier'. http://www.cute-test.com, 2011.

[Thrier10] Yves Thrier. *Clonewar – Refactoring Transformation in CDT: Extract Template Paramete*r. Master's thesis, University of Applied Sciences Rapperswil, 2010.

Compiling a Static Web Site Using the C Preprocessor

Sometimes the obvious way is still too complex. Sergey Ignatchenko relates how 'No Bugs' Bunny found an unexpectedly simple approach to creating a web site.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with opinions of the translator or *Overload* editors; please also keep in mind that translation difficulties from Lapine (like those described in [LoganBerry2004]) might have prevented from providing an exact translation. In addition, both translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Quite recently, I was told by one of my fellow rabbits that they're compiling their web site; moreover, that they're doing this using the C preprocessor. My first reaction? 'You guys must be crazy!' My second thought was 'This is so crazy it might just work', so I've taken a deeper look into it, and the more I looked the more I liked it (of course, it is not a silver bullet, but in some cases it can certainly save significant time and effort).

It was so interesting that I felt obliged (with the kind permission of the authors) to share it with the audience of *Overload*. This approach shall be most interesting for C- and C++-oriented readers, the preprocessor is a tool of the trade for most of them.

The task in hand

Once upon a time, there was a software development company, mostly specializing in C/C++ coding. And at some point they needed to create a web site. Nothing fancy, just your usual small company web site with predominantly static content and updates no more frequent than once a week. And we're in 201x, the website needed to be 'Web2.0-ish' and (more importantly for our purposes) needed to be usable on PCs, Macs and on smartphones. This is where our story starts.

Desktop != mobile

Rather quickly it was realized that if you want to ensure a reasonable user experience both on desktops and mobiles, there is absolutely no way you can use the same HTML for both sites. After some deliberation they decided to use jQuery Tools for the desktop site and jQuery Mobile for the mobile one, but the exact choice is not important for the purposes of this article; what is important is that these days we tend to have to assume that the HTML will be different for the mobile and desktop versions. Unfortunately, despite all the efforts of CSS, merely changing the CSS to switch from desktop to mobile is good enough only in textbook examples; it doesn't mean that CSS shouldn't be used – in fact, it was used very extensively in this case (in particular, both jQuery Tools and jQuery Mobile rely on CSS heavily), but CSS is not enough – there are way too many differences between desktop and mobile sites, from potentially different numbers of HTML pages, to very different navigation. And when you have two HTML codebases with essentially the same content, any update needs to be copied very carefully to two places, which as we all know, is not a good thing. In HTML it becomes even worse, as a missing **`</div>`** can easily break the whole thing and figuring out what went wrong can be difficult even if source control is used. Also, in a static website there are lots of similar (usually navigational) fragments across all the pages, and maintaining them manually (style changes, if page is added, moved etc.) certainly means a lot of mundane, tedious and error-prone work.

The classical approach: PHP+MySQL

Usually these problems are addressed by using a server-side engine, like PHP/ASP/Perl/etc., with content essentially residing in a database and formatted into HTML on each request. This would certainly work in this case too, but it was argued that for a site with just a dozen predominantly static pages, having to study one more programming language, installing and maintaining the database (with backups etc. etc.) and dealing with the additional security issues, is not exactly desirable. As I was told, at that point the attitude was 'yes, this is a solution but it is really bulky for such a simple task; is there a chance to find something simpler and more familiar?'

When there is a will, there is a way

Os e layth Frithyeer hyaones, on layth zayn yayn dahloil

If it's sunny today, we'll go and find dandelions

Eventually somebody had a thought: if we describe what we need in familiar terms, then the task can be stated as follows: we need to compile the web site from source texts into multiple HTML targets, one being for desktops, another for mobile devices. As soon as this was stated, things moved rather quickly and the solution came up pretty soon; due to background of the guys involved, the solution was based on the familiar C preprocessor and on sed.

C preprocessor for compiling web sites? You guys must be crazy!

According to the idea of compiling the web site from source texts into HTML, the source code was structured as follows:

- there are text files. These contain all the site's textual content and are allowed to have only very limited HTML (usually restricted to stuff like **`<b>`**, **`<p>`**, **`<h*>`**, occasional **`<a>`** and so on); they are the same for both desktop and mobile versions.
- there are a few HTML template files (in this specific case they were given `.c` extensions – I was told to avoid some obscure problem with applying GCC to .html files). These are regular HTML files but with C preprocessor directives allowed, the most important being **`#include`**. These template files are specific to the desktop/mobile version, and in particular it is easy to have different layouts which is

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

of course, it is not a silver bullet, but in some cases it can certainly save significant time and effort

important for dealing with jQuery Mobile. These HTML template files `#include *.txt` files to include specific content.

The devil is in the details

'Es lay elil?' e laynt meth.

'Are you an enemy?' he said.

The basic file structure described above is enough to start developing in this model, but as usual figuring out more subtle details may take a while.

First of all, in practice 99.9% of the text content is the same for both sites, but the remaining 0.1% needs to be addressed. To deal with these cases, usual C-style macros like `DESKTOP_ONLY` and `MOBILE_ONLY` were used (with a parameter telling what to insert). There is a caveat though – with the C preprocessor one cannot use arbitrary strings as parameters unless they're quoted, and when you use quoted strings the quotes themselves are inserted into the resulting HTML. To get around this the following solution (IMHO quite a dirty one, but it does work) was used:

- the macro is defined in the following manner:

  ```
  #define DESKTOP_ONLY( x ) @@x@@
  ```
- the macro is used like `DESKTOP_ONLY( "<br>" )`. Then after the preprocessor is run it becomes `@@"<br>"@@`
- after running the preprocessor, the Unix-like `sed` with a rule like

  ```
  s/@@"\([^"]*\)"@@/\1/g
  ```

  is run over the post-preprocessed code. This changes our `@@"<br>"@@` into `<br>`, which is exactly what is needed. Note that `@@` has no special meaning (importantly not in the C preprocessor, nor in `sed` and not in HTML); it is used merely as a temporary escape sequence which should not normally appear, but if your HTML does include it, you can always use another escape sequence.

Obviously this whole exercise only makes sense if `DESKTOP_ONLY` is defined this way only when generating desktop HTML templates, and is defined as

```
#define DESKTOP_ONLY( x )
```

for mobile HTML templates. Also it should be mentioned that while this solution is indeed rather 'dirty', it doesn't clutter the text files, and this is what really important.

Another similar example occurs if you need to concatenate strings (note that the usual C technique of putting 2 string literals next to each other is not recognized in HTML). So, for example, a macro to insert an image may look like

```
#define IMG( x ) <img src="images/"@*@x>
```

with an additional `sed` rule being

```
s/"@*@"//g
```

Many other situations can be handled in a similar way.

Additional benefits

Now, as string stuff has been handled, the system is ready to go, and there are several additional (and familiar for C developers) features which can be utilized.

In many cases there will be repeated HTML parts on many HTML pages (headers, footers, navigation and so on), and with this approach this repeated stuff can be easily moved either into `#include`s or macros. I've seen it, and it is indeed a great improvement over repeating the same stuff over and over if editing HTML manually.

Another such feature is conditional compilation; as practice has shown it is particularly useful when dealing with navigation. One typical pattern I've seen was used to have the same navigation pattern for a set of the pages while using macros to disable links back to itself using the following technique:

- in HTML template file:

  ```
  #define NO_HOME_LINK
  #include "navigation.inc"
  ```

- in `navigation.inc`:

  ```
  ...
  #ifndef NO_HOME_LINK
  <a href="/">Home</a>
  #else
  <b>Home</b>
  #endif
  ...
  ```

While the same thing can be achieved with Javascript, why not do it once instead of doing it on each and every client computer (not to mention that `#ifdef` is a much more familiar for developers with a C or C++ background)?

Using `sed` also provides its own benefits: for example, to reduce the size of the resulting file, the following `sed` rule is useful:

```
/^$/d
```

If you don't want to look for a preprocessor command-line switch which removes generated `#line` statement, the following `sed` rule will help:

```
/^#/d
```

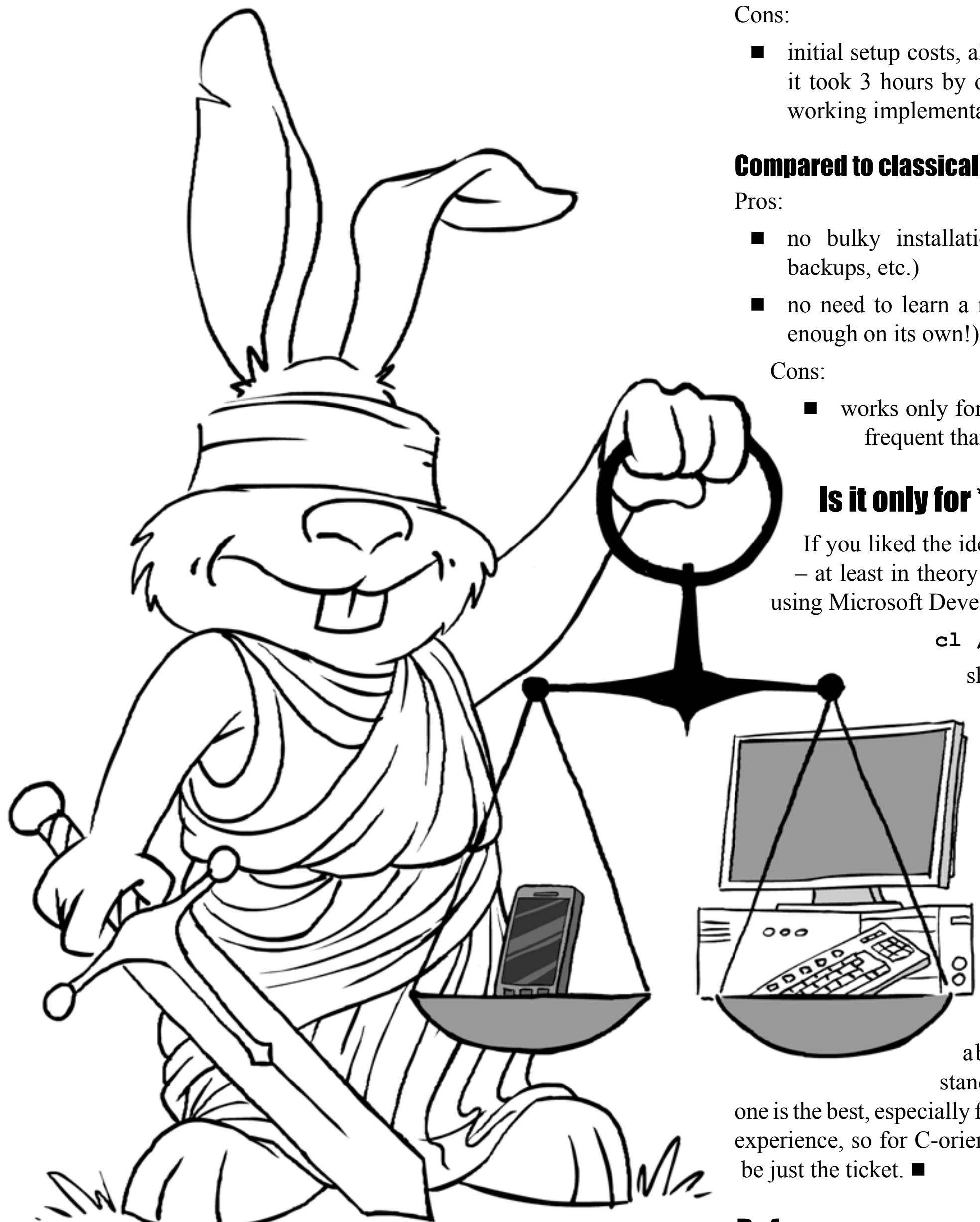
And for an easy fix of problems with handling `&apos;` (which works everywhere, except for Internet Explorer) – the following `sed` rule will help:

```
s/&apos;/\&#039;/g
```

Pros and cons

So, now we've described the system, the question is: what are advantages and disadvantages of this approach?

While the same thing can be achieved with Javascript, why not do it once instead of doing it on each and every client computer



Compared to manual HTML programming

Pros:

- single content source in text files, easily editable (even by management)
- much easier maintenance due to better HTML code structuring (using **#include**s / macros)

Cons:

- initial setup costs, although these seem low: the guys have told me it took 3 hours by one person from coming up with the idea to a working implementation

Compared to classical 'PHP+MySQL' stuff

Pros:

- no bulky installation (PHP or another engine, database with backups, etc.)
- no need to learn a new programming language (Javascript is bad enough on its own!)

Cons:

- works only for rather static sites, where updates are not more frequent than around once per day

Is it only for *nix?

If you liked the idea but are working with Windows, don't worry – at least in theory the same approach should work too. If you're using Microsoft Developer Studio, something like

```
cl /P /EP
```

should perform the preprocessing, and **sed** for Windows can be found, eg [GnuWin]. You'll still need to learn how to deal with a command line though ;).

Is the C preprocessor the best tool for the job?

While this example has been concentrating on the C preprocessor, it is not the only tool which can help. In fact, almost any kind of text processor can be used to compile web sites in a similar way to what has been described above (for example, m4 or some kind of standalone XSLT processor can be used), but which one is the best, especially for small projects, depends a lot on your previous experience, so for C-oriented audience the C preprocessor might indeed be just the ticket. ■

References

[LoganBerry2004] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[GnuWin] http://gnuwin32.sourceforge.net/packages/sed.htm

A Position on Running Interference in Languages

There is much debate about the merits of different approaches to type systems. Teedy Deigh considers how to make the most of them.

The language of type systems is strewn with polarising metaphors. Do you want your type system to be strong? Do you want it to be muscled and rippling or would you rather it was weak, feeble and a bit of a lightweight, pushed around in its sandbox? On the other hand, would you prefer your type system to be dynamic, with an élan and drive that makes it charming and witty, agile and flexible? You could instead have a static type system, all staid and stuck in the mud.

> 'If you used a proper language, with a static type system, you wouldn't need to write all those unit tests.'
>
> 'Your code must either do very little or not compile if you think types eliminate the need for tests. If you used a proper language, with a dynamic type system, you might actually write something interesting and, hey, perhaps even ship it!'
>
> 'Duck typing is for monkeys so they can patch their running systems because they're so busy reshipping to cover their mistakes they only talk about testing without actually doing any. A checked type system gives you at least a reality check rather than a maintenance bill.'
>
> 'Strong typing skills are what you need with statically typed languages. Why write two lines when you can spend twenty lines plea-bargaining with the compiler?'

The discussion over type models in languages can get quite heated at times, making you want to change the subject to something less contentious – religion, politics, football, operating systems, editors, etc.

In the spirit of compromise, static type systems have loosened up a little, with type inference used to deduce types the compiler knows full well. Bureaucratic pedants might quibble, believing that if you want to use an object you should fill out a form in triplicate so you can be properly sure you're getting the right type, but most C++ programmers have found themselves marvelling at the recycled **`auto`** keyword. C# programmers, meanwhile, have been excited by the ability to use **`var`** to make their code look like JavaScript, and yet still have a clue as to what's actually going on, or to use **`dynamic`** to introduce the element of surprise. Haskell programmers have looked on unimpressed.

Going the other way, we see that declared type systems have crept into more libertine languages, bringing tie-and-tease – if not fully bound BDSM – into their programming discipline. PHP has type hints, which bring to the language a system of typing that ingeniously combines the limitations of static typing with the weaknesses of dynamic typing. More recently Dart has presented programmers with a meh-based type model that allows you to declare your types boldly and defiantly, but in a way that they can be ignored politely. Lisp programmers have stammered indifference.

But these type institutions and movements mask a deeper issue for the modern programmer: all of these refinements and additions are intended to make programmers' lives easier and, therefore, their livelihoods less secure and their day-to-day negotiations with the compiler and runtime configuration intrinsically less challenging and more boring. I propose that we need a new school of thought in programming language models: type interference. Type interference is not specifically a kind of type model, but is more a property that certain type systems can exhibit.

A great example of type interference is the head-on collision between C++'s template system and the traditional procedural worldview of most compilers. The resulting debris yields pages of cryptic hieroglyphs. This kind of interference has resulted in many lost hours – even days – of programmer time, while at the same time sparking a secondary industry in tools and expertise to decrypt such apocrypha. It has allowed some programmers to rise above others by learning to read the tea leaves of the compiler's runic ramblings, acquiring deep knowledge of arcane arts rather than actually developing software or learning how to improve their people skills.

The discovery that C++'s template system was Turing complete can be seen to be at the heart of this type interference devolution. Indeed, it is widely acknowledged that a compilation message from a particularly gnarly template that was missing a comma was the first to pass the Turing test.

Disappointingly compiler writers have been making progress on reducing the level of noise, allowing some messages to be contained even on something as small as a 1080p HD screen!

A form of type interference that has taken the Java world by Sturm und Drang is dependency injection, a form of substance abuse that is spilling over into other languages. The term *dependency injection* sounds a lot more exciting than 'lots of XML to create objects and pass arguments around'. Its countercultural overtones make it both attractive and addictive. Rather than creating objects in an obvious and visible way, where they are needed and using the principal programming language of a system, programmers instead dump various object and type relationships into less readable XML files that are separate from the rest of their code. This can make even the simplest instantiation an act of careful orchestration between different frameworks, languages and phases of the moon. In large organisations it can lead to regional conflicts and protracted political wrangling over files.

It is expected that one day DI will be unmasked as an April Fool's joke, a cunning con that has deceived programmers into stating that their systems are loosely coupled, highly cohesive and statically typed, while at the same time deferring all type decisions to runtime and sweeping all the dependency problems under the rug of XML files. It is also worth noting that XML, like the C++ template system, is a tenebrous exercise in obscurity. The waves of angle brackets across the screen look not so much like a formation of migrating geese as an Agincourt of arrows, threatening any who dare attempt understanding.

It has long been known that the majority of professional programmers write only a few lines of working code every day. Type interference helps with that, allowing them to claim a debugged template fragment or XML stanza as the major part of their five-a-day. Type systems need to balance the needs of the many with the needs of the few, favouring developmental obscurity and job security over other measures of productivity and worthiness. ■

Teedy Deight dabbles in programming language design in the way a cat dabbles with a trapped mouse. You never know the details, but the outcome is rarely good for the mouse. The remains will turn up and surprise you where and when you're least expecting it, leaving you wondering how to dispose of it.