overload 109

JUNE 2012 £3

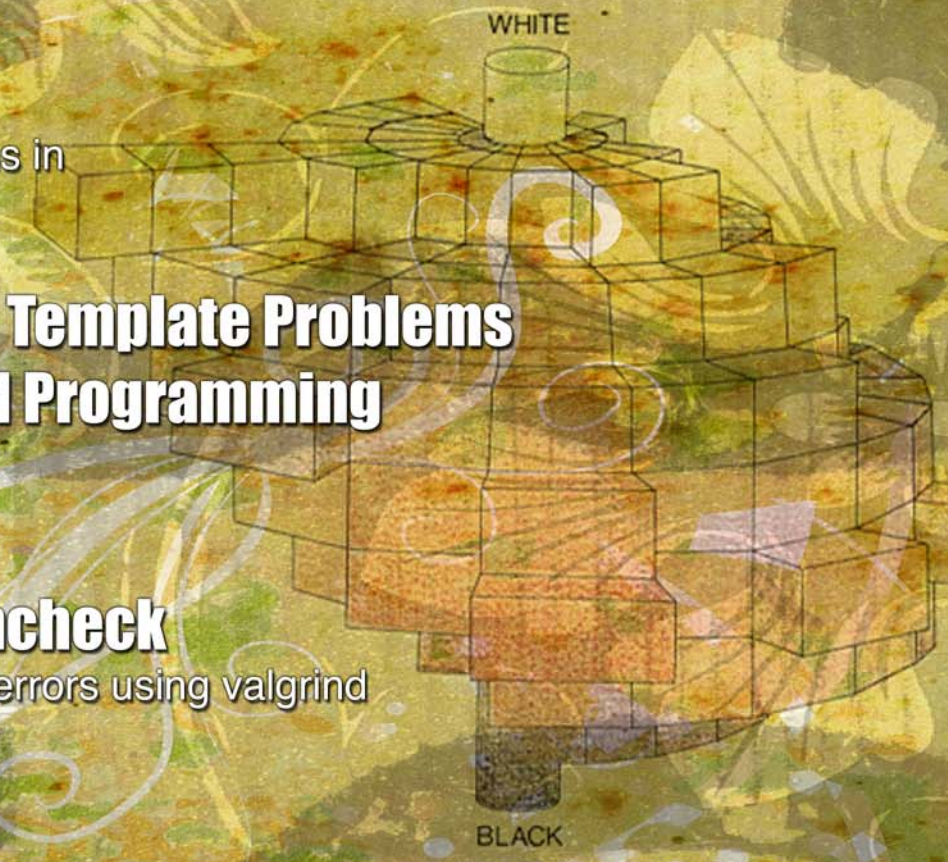
Programming Darwinism
We investigate software's *élan vital*

What's a Good Date?
A profile of different date representations

All About XOR
We the find the hidden depths in this common logic operation

Curiously Recursive Template Problems with Aspect Oriented Programming
How to succesfully combine two useful code techniques

Valgrind: Basic memcheck
We continue to hunt and kill errors using valgrind



Tail Call Optimisation in C++
Stack based languages are powerful, but often can't deal with arbitrary recursion. We find a way around this limitation.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 109

June 2012

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 110 should be submitted by 1st July 2012 and for Overload 111 by 1st September 2012.

ACCU

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

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

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

It's Not What You Know, It's Who You Know

Most human endeavours are not solitary pursuits. Ric Parkin looks at the interconnection of everyone.

Ah the pleasures of a fine late English spring. After a couple of horribly wet and miserable months, May has ended with some gloriously hot and sunny days, which is just as well as I've been sitting in a field drinking a variety of weird and wonderful brews at the 39th Cambridge Beer Festival [CBF]. This is a significant event, as it functions as an informal reunion for the region's IT sector – it seems that large proportion of software developers like their real ale, and so I quite often bump into current and past colleagues and can catch up with what's happening in the industry. Judging by the range of company t-shirts on display, you get a feel for how big and varied is the local tech-cluster known as 'Silicon Fen' [Fen].

The origins of this are usually traced to the founding of the UK's first Science Park back in the early 1970s. By encouraging small high tech startups, often spun out of research from the University, this and many other subsequent business parks in the region have transformed the local (and national) economy, with a critical mass of companies and jobs in sectors such as software and biotech. It's easy to see how such a cluster effect feeds off itself to breed success. By having so many companies nearby, people are more willing to take a risk and start (or join) a small company, because if it goes under (as many do) it's easy to get a job elsewhere. There are also lots of small 'incubators' such as the St Johns Innovation Centre [St Johns] where such companies can get started with lots of advice and help, often from other startups in the next units. Indeed it's this networking effect that really seems to be key – it's comparatively easy to find important staff in the area who've already been through the experience of turning tiny startups into successful businesses, and the region's success encourages people to relocate or stay here after university, providing a rich pool of talent. Indeed it was my realisation of this that made me decide to buy a house and settle here – there are so many companies that I wouldn't have to move in order to change jobs. And I've been proved right – in 18 years I've been in 7 different jobs, usually with small startups, but am still in the same house.

Such has been the success that people try to recreate the effect – locally there is an attempt to build a biotech cluster, centred around a research campus [Genome] (home of the Sanger Institute, one of the pioneers of DNA sequencing and central to the deciphering of the human genome) and Addenbrooke's Hospital to the south of the city. And recently Google and others have set up an incubator hub in east London, dubbed Silicon Roundabout [Roundabout].

But there's more to it than just bringing companies together. There has to be that networking effect, which can range from bumping into someone from the next door company in the shared canteen of the Incubator building, informal meetups such as the Beer Festival, formal networking sites such as LinkedIn and The Cambridge Network [Network], and professional groups such as Software East [East] and of course ACCU itself. Some of these will be of more value to you than others depending on what you want to get out of them. Some are good for contacts, some are good for sharing technical experiences, and some are sadly getting plagued by recruiters wanting to connect with you. But I think they all help by expanding your circle of potential *experience*. By that I mean that if you have something new that you have to do, then you could work things out for yourself but it'd take a long time and you're likely to make lots of mistakes. But if instead you know someone who's done it before, then you can take their advice or get them to help you, and your chances of success will be vastly improved.

Now we've established that pooling experience can increase success, a question is how much of a network do you need? Too much and the effort needed to maintain it will interfere with actually getting on and doing things; too little and you won't have the contacts for when you need them. Well, the good news is that there is a branch of research called Network Theory [Theory] which concerns itself with the properties of networks, such as social interconnectedness. One famous type of network is called a Small-World network, informally known as The Six Degrees Of Separation [Six]. (It has also been turned into a game involving Kevin Bacon, apparently after he commented that he'd worked with everyone in Hollywood, or someone who'd worked with them. [Bacon]). These types of networks have two defining properties – on average each individual has only a few connections; and yet they can connect with anyone in the entire network with only a few 'jumps'. The trick is that you have a few key people who connect to a lot of others, especially other key people. Then all you need to do is know a key person, who knows another key person, who knows your target. (Interestingly, there are hints that we have an 'ideal' social group size [Dunbar]. I wonder if this number is related to how many connections you need for a good small-world network, which would be needed for a successful civilisation to arise). These types of networks appear in many different disciplines, as they are very efficient at creating robust, well connected networks. Examples include neural networks, seismic fault systems, gene interactions, database utility metrics, and the connectedness of the internet (see Figure 1 for a map of the internet from the wikimedia commons [Internet]). Its robustness stems from the fact that if you remove an arbitrary node, the mean path length between any two nodes doesn't tend to change. Problems will only occur if you take out enough key nodes at once).

While a wide social and professional network is useful, in our day-to-day work we tend to work in a much smaller team. We rarely work in isolation as it's not very efficient to do it all yourself – in a similar way to getting advice from a professional connection, we use division of labour and skills

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.

to make up a team. For example, instead of me spending ages learning about databases and struggling to support one to the detriment of my programming, we just hire someone who's a professional DBA. By building such a team we have a group who collectively have all the skills needed.

But now we need to coordinate them. How much of an overhead will that be? Unfortunately models and experience show that this can be disastrously high – by considering a team of *N* people who all talk to each other, we see that there are (*N*-1)! (ie (N-1)x(N-2)x...x2x1) different possible interactions. This gets very big very quickly, which explains why an ideal team size is actually quite low. It has to be large enough to get the benefits of division of labour, but small enough that the overhead of communication and coordination doesn't swamp people. Five to ten people is usually pretty good. But let us reconsider the assumption that everyone needs to talk to everyone else. If we can arrange things such that you generally *don't* need to, then we can have a bigger team with the same overhead.

Small world theory also gives us ideas about how we should be organising them – in small focussed groups with a few key individuals acting as links between the groups. In the past, the groups were often organised by discipline, so for example all the programmers were in one team, the testers in another, sales in another and so on. But this was found to not work very well as the lifecycle of the product cuts across all these teams, whereas there was relatively poor communication channels that mirrored it. Instead, now we tend to make a team with members from all the different disciplines (although things like sales and marketing still tend to be separate from the technical teams – I can see why as a lot of their activities are done after a product is built, but it is a shame that there aren't at least representatives within the technical teams to give input from their point of view, and to understand the product better). You can then see how the modern technical company structure has arisen, with small multi-disciplinary teams building focussed products, coordinated by a few key communicators. Done well, you can grow this model into a large productive company.

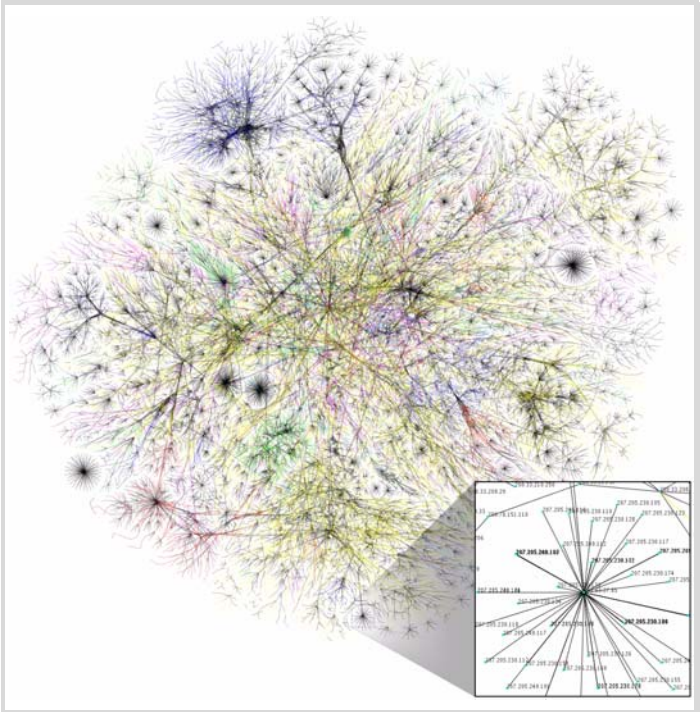


Figure 1

All good things....

I've been editing Overload for just over four years now. It been tremendous fun, but all things must come to an end at some point, and that time has come. From next issue there will be a new editor – I'll leave them to introduce themselves in their own fashion then. I'll be helping out in the review team, and may even find time to write some new articles, so you have been warned! I'd just like to thank all those who've helped make all those issues, from the many authors, the helpful reviewers, Pete's fantastic covers, and especially Alison's excellent (and patient) production.

References

[Bacon] http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

[CBF] http://www.cambridgebeerfestival.com/

[Dunbar] http://en.wikipedia.org/wiki/Dunbar's_number

[East] http://softwareast.ning.com

[Fen] http://en.wikipedia.org/wiki/Silicon_Fen

[Genome] http://www.sanger.ac.uk/

[Internet] http://en.wikipedia.org/wiki/File:Internet_map_1024.jpg

[Network] http://www.cambridgenetwork.co.uk

[Roundabout] http://www.siliconroundabout.org.uk/

[Six] http://en.wikipedia.org/wiki/Six_degrees_of_separation

[SmallWorld] http://en.wikipedia.org/wiki/Small-world_network

[St Johns] http://www.stjohns.co.uk/

[Theory] http://en.wikipedia.org/wiki/Network_theory

Programming Darwinism

Have you ever thought your software had a life of its own? Sergey Ignatchenko wonders whether you might be right.

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.

> *I think anthropomorphism is worst of all. I have now seen programs 'trying to do things', 'wanting to do things', 'believing things to be true', 'knowing things' etc.*
>
> ~ Edsger W. Dijkstra

It is common to refer to programs in terms which are usually reserved for living beings. And yet there are relatively few things we'd refer to as having a 'life cycle'; we have the 'biological life cycle', 'enterprise life cycle', 'product life cycle', 'project life cycle', and 'software development life cycle', and that's about it. There is no such thing as, for example, a 'theorem life cycle' or 'rain life cycle'. And (even understanding that what I'm about to say will contradict Dijkstra [EWD854]) I don't think it is a bad thing. In my opnion, saying that it's a bad thing that people refer to programs in an anthropomorphic way is akin to saying that rain is a bad thing. As I see it, the fact that programs (especially complex programs, and remember that programs have become vastly more complex since 1983 when Dijkstra wrote EWD854, and even more so since his major works back in 70s) do behave similarly to living beings is something which we should accept whether we like it or not.

In fact, academic researchers have recently more or less agreed to acknowledge that complexity of systems and life are closely interrelated ([Chaitin1979], [Gregersen2002], [Kauffman2002]). What scientists (as usual) are disagreeing on, is the question of whether this complexity has arisen spontaneously, or is a result of creative design. Fortunately we don't need to answer this unsolvable question for the purposes of this article; what matters is that the behaviour of complex systems does indeed resemble living beings.

Survival of the fittest?

> *Programming is like sex: one mistake and you're providing support for a lifetime.*
>
> ~ Michael Sinz

Going a bit further with the parallels between programs and living beings, shouldn't we think about applying evolutionary theory?

But how? It might work as follows: program features (this should be understood broadly and include other properties such as reliability, speed, convenience, price, brand, marketing, etc.) are equivalent to biological traits. If users don't buy a program it tends to die, so if a program is desirable it improves the chances of the program surviving. This means that decision of user to buy/not buy program, is equivalent to a process of natural selection, favouring programs with certain desirable traits.

One obvious question, which many of us would like to get an answer to, is 'Is it really the best (most desirable) program that survives?' As much as I would like to see the world where it is true, I should admit that it is not the case. Poor programs abound.

There are at least two major reasons why it happens. First of all, evolutionary theory does not provide any guarantee on the absence of poorly adapted organisms; it just says that sometimes, maybe, if better fit organisms appear they will have a better chance of surviving.

Another, probably even more important reason for poor programs out there, is that even if 'everybody' (which is often used as a synonym to 'community of software developers') agrees that product A is 'better' than product B, it doesn't necessarily mean that product A has the better chance of survival. It is important to remember that it is only end-users (the ones who're paying the money, directly or indirectly, for example, via watching ads in adware) who are directly involved in the process of natural selection. For example, when the software developer community largely agreed that OS/2 was so much better than Windows 3.1, it didn't matter for survivability, as the target market of the operating systems was much wider. What really did matter was the opinion of the end-users, and it was clearly in favour of Windows, which indeed was the one that survived.

Survival of the most agile!

There is a famous [mis]quote attributed to Darwin: 'It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.' While most likely Darwin didn't say it, the principle behind it still stands. Nowadays this property is known as 'evolvability'. For example the article in [TheScience2011] (based on [Woods2011]) says: 'The most successful E. coli strains in a long-term evolution experiment were those that showed the greatest potential for adaptation'; or, as Lone Gunman put it, 'Adaptability trumped fitness' [LoneGunman2011].

Now let's apply this theory to a history of program evolution. In the Paleozoic era of programming, programs were intended to be designed once and then to avoid any changes. These programs are now known as waterfalloptera. Then, in the programming Mesozoic, it became obvious that avoiding any changes is not a viable option (see for example [BBC2012]), and much more lightweight and agile scrumoculates and extremeprogrammerodonts (as well as many others) appeared, with 'responding to a change' being one of their distinguishing features. In [Boehm2004] the authors note that these agile species tend to thrive in environments where frequent changes are the norm. Essentially what we have is that agile species are more adaptable, because they're fast to react to change. This means two things: first, that if evolution theory can indeed be applied to programs, then agile programming species are more likely to survive, and second, that if it is the case we can rephrase the classical

'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

'survival of the fittest' into 'survival of the most agile', at least when applying it to programs.

What about survival of the fattest?

In the programming world, a nasty disease known as 'vendor lock-in' often grows to epidemic proportions. It often comes from a classic positive feedback loop, for example where the more people use MS Office, the more other people need to install it to be able to read the documents, and so even more people install it to read their documents, and so on until everyone is locked into owning MS Office. This is all well known, but with our current analysis there is one big question: how does it correlate with our theory of evolutionary programming? Such species, known as locked-in-asaurus, tend to become irresponsive to change but still may successfully live for ages. So if evolutionary theory holds, how it could possibly happen? There is no definite answer to this question but we will be bold enough to propose the following hypothesis: In biological evolution, there were many cases when less-adaptable species were dominant for a long while until a big 'extinction event' caused drastic changes to the environment. For example, dinosaurs were the dominant terrestrial fauna in the Cretaceous period, and there were no problems on the horizon until the Cretaceous–Paleogene extinction event 65 million years ago (also known as the K-T extinction event, currently attributed to an asteroid impact) demonstrated their inability to cope with the rapidly changed environment and lead to the extinction of dinosaurs and the rise of mammals.

Applying this approach to programs, we may propose the hypothesis that locked-in-asauri are just waiting for an 'extinction event' which would eliminate them, and pave the way for more agile programming species. Unfortunately, it is not possible to test this hypothesis, just as it was not possible to test the hypothesis about the better adaptability of mammals before the K-T extinction event. It is even less possible to predict what could serve as such an extinction event for locked-in-asaurus: it might be the current Eurozone crisis, the rise of smartphones/tablets, or it might be something years from now. In general, evolutionary theory doesn't provide time estimates – it may provide some explanations and predictions, but how fast things go is usually beyond its limits. ■

References

[BBC2012] 'Viewpoint: A rethink in the financing of innovation', http://www.bbc.co.uk/news/business-18062164

[Boehm2004] Boehm, B.; R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2004.

[Chaitin1979] Toward a mathematical definition of 'life', Gregory J. Chaitin, In R. D. Levine and M. Tribus, *The Maximum Entropy Formalism*, MIT Press, 1979, pp. 477–498 , http://www.cs.auckland.ac.nz/~chaitin/mit.pdf

[EWD854] http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD854.PDF

[Gregersen2002] *From Complexity to Life: On The Emergence of Life and Meaning*, Niels Henrik Gregersen (Editor), Oxford University Press, 2002

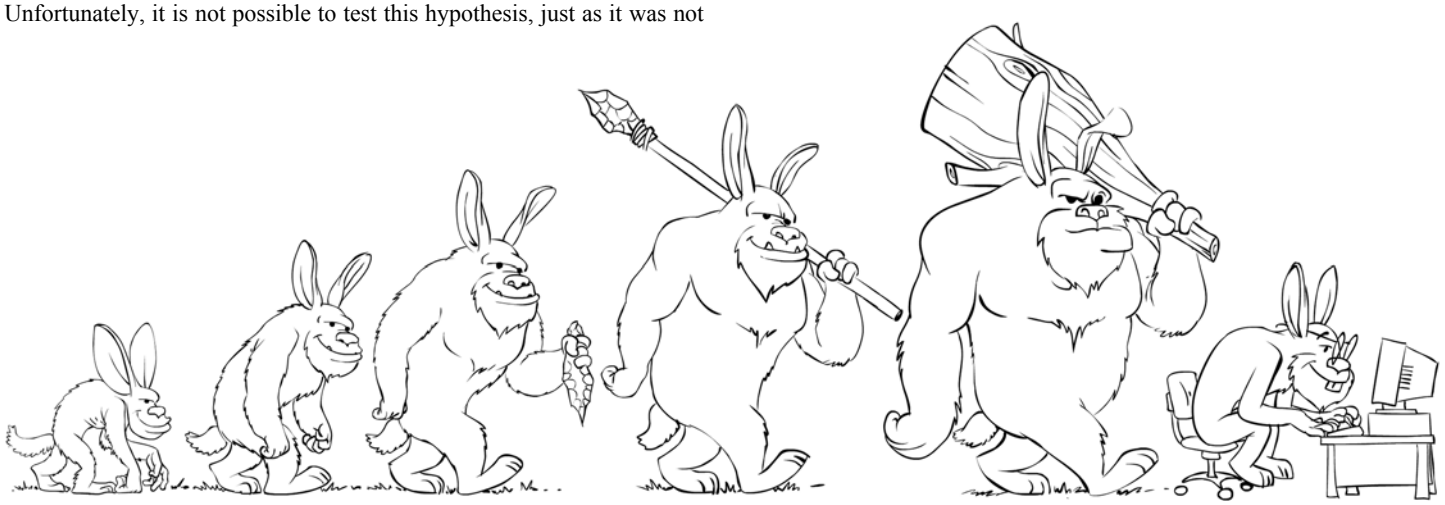
[Kauffman2002] Stuart A. Kauffman, *Investigations*, Oxford University Press, 2002

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

[LoneGunman2011] http://www.lonegunman.co.uk/2011/05/25/in-evolution-adaptability-beats-fitness/

[TheScience2011] 'Evolvability, observed' http://classic.the-scientist.com/news/display/58057/

[Woods2011] R.J. Woods et al., 'Second-order selection for evolvability in a large Escherichia coli population', *Science*, 331: 1433-6, 2011.



What's a Good Date?

Implementing a data type seems simple at first glance. Björn Fahller investigates why you might choose a representation.

One of many things that did not get included in the shiny new C++11 standard is a representation of date. Early this year my attention was brought to a paper [N3344] by Stefano Pacifico, Alisdair Meredith and John Lakos, all of Bloomberg L.P., making performance comparisons of a number of date representations. The paper is interesting and the research well done, but since I found some of their findings rather baffling, I decided to make some measurements myself, to see if I could then understand them.

Types of representation

As a simplification, the proleptic Gregorian Calendar [Proleptic] is used, where we pretend that the Gregorian calendar has always been in use. Given that the introduction of the Gregorian calendar happened at different times in different places, this makes sense for most uses, except perhaps for historians, and also means that the truly bizarre [Feb30] can safely be ignored.

The two fundamentally different representations studied are serial representations and field based representations. The serial representation is simply an integer, where 0 represents some fixed date, and an increment of one represents one day. The field based representation uses separate fields for year, month and day. It is obvious that serial representations have an advantage when calculating the number of days between two dates, and likewise obvious that field based representations have a performance advantage when constructing from, or extracting to, a human readable date format.

Less obvious is the find in [N3344] that sorting dates is faster with serial representations than with field based representations, given that you can store the fields such that a single integer value comparison is possible.

Implementations

None of the implementations check the validity of the input, since [N3344] gives good insights into the cost of such checks.

Listing 1 shows the serial representation. The Julian Day Number [JDN] is used as the date. Today, May 6th 2012, is Julian Day Number 2456054. A 16-bit representation stretches approximately 180 years and thus makes sense to try, but it requires an epoch to add to the number.

Since conversions between year/month/day and julian day number occur frequently, they are implemented in a separate class (not shown in this article) used as a template parameter to the date classes. This also allows the freedom to change calendar without need to change the representations.

```
template <typename T, typename Calendar,
          unsigned long epoch>
struct serial_date
{
  T data;

public:
  serial_date(unsigned y, unsigned m, unsigned d)
    : data(Calendar::to_jdn(y, m, d) - epoch)
  {
  }

  bool operator<(const serial_date& rh) const
  {
    return data < rh.data;
  }
  // other comparison operators trivially similar

  long operator-(const serial_date &rh) const
  {
    return long(lh.data) - long(rh.data);
  }

  void get(unsigned &y, unsigned &m,
           unsigned &d) const
  {
    Calendar::from_jdn(data + epoch, y, m, d);
  }
};
```

Listing 1

For field based representations there are two alternatives. One uses the C/C++ bitfield language feature, in union with an unsigned integer for fast comparisons, the other uses only an unsigned integer type member and accesses the fields using mask and shift operators. With 5 bits for day of month, and 4 bits for month in year, there are only 7 bits left for year on a 16-bit representation – not much, but enough to warrant a try using an epoch. The two field based representations are similar enough to warrant one class template, which does all the calculations based on abstracted fields, and take the concrete representation as a template parameter. It assumes there are named getters and setters for the fields, and an access method for the raw unsigned integral type value for use in comparisons. Listing 2 shows the class template for field based dates.

The bitfield based representation makes use of undefined behaviour, since the value of a union member is only defined if the memory space was written to using that member. However, in practice it works. The layout in Listing 3 makes days occupy the least significant bits and year the most significant bits with g++ on Intel x86.

The bitmask representation in Listing 4 is similar to the bitfield representation, but involves some masking and shifting in the setter/getter functions, and thus avoids any undefined behaviour.

Björn Fahller is a systems analyst for telecommunications software with 14 years' experience as a developer of embedded software. Apart from prototyping, programming now mostly competes for time with family, flight instruction and volunteering with flying for damage assessment and search for rescue. Björn can be reached on bjorn@fahller.se

One of many things that did not get included in the shiny new C++11 standard is a representation of date

```
template <typename T, typename Cal>
class field_date : private T
{
public:
  field_date(unsigned y, unsigned m, unsigned d)
  {
    T::year(y).month(m).day(d);
  }
  bool operator <(const field_date &rh)
  {
    return lh.raw() < rh.raw();
  }
  // other comparison operators trivially similar

  field_date &operator++()
  {
    T::raw_inc();
    if (day() == 0) { day(1); }
    // assume 31 wraps to 0 and inc month.
    if (month() == 13)
      {
        month(1); year(year() + 1);
      }
    else if (day() == 31 &&
        ((month() & 1) == (month() >> 3)))
      {
        day(1);
        month(month() + 1);
      }
    else if (month() == 2 &&
        day() == (29 + Cal::is_leap_year(year())))
      {
        day(1);
        month(3);
      }
    return *this;
  }
  long operator-(const field_date &rh) const
  {
    long ldn = Cal::to_jdn(s_.y_ + epoch,
                           s_.m_, s_.d_);
    long rdn = Cal::to_jdn(rh.s_.y_ + epoch,
                           rh.s_.m_, rh.s_,d_);
    return ldn - rdn;
  }
  void get(unsigned &y, unsigned &m,
           unsigned &d) const
  {
    y = T::year(); m = T::month(); d = T::day();
  }
};
```

Listing 2

```
template <typename T, unsigned epoch_year>
class bitfield_date
{
  struct s {
    T d_:5;
    T m_:4;
    T y_:std::numeric_limits<T>::digits-9;
  };

  union {
    s s_;
    T r_;
  };

public:
  bitfield_date(unsigned y, unsigned m,
                unsigned d)
  {
    s_.y_ = y - epoch_year;
    s_.m_ = m;
    s_.d_ = d;
  }

  bitfield_date& month(unsigned m)
    { s_.m_ = m; return *this; }
  unsigned month() const { return s_.m_; }
  // other access methods trivially similar

  T raw() const { return r_; }
  void raw_inc() { ++r_; }
};
```

Listing 3

Measurements

The three representations are used both as 16-bit versions, with Jan 1st 1970 as the start of the epoch, and with 32-bit versions using 0 as the epoch (i.e. 4800BC for the JDN based serial representation, and the mythical year 0-AD for the field based versions.)

The measurements are:

- **emplace**: time to emplace 743712 random valid dates in year/month/day form into a preallocated vector (see Figure 1).
- **sort**: time to sort the vector (see Figure 2).
- **increment**: time to increment every date in the vector by one day (see Figure 3).
- **diff**: time to calculate the number of days between each consecutive pair of dates in the vector (see Figure 4).
- **extract ymd**: time to get year, month, day from each date in the vector (see Figure 5).

```
template <typename T, unsigned epoch_year>
class bitmask_date
{
  static const T one = 1;
  static const T dayshift = 0;
  static const T daylen = 5UL;
  static const T daymask =
     ((one << daylen) - 1UL) << dayshift;

  static const T monthshift = 5;
  static const T monthlen = 4;
  static const T monthmask =
     ((one << monthlen) - 1U) << monthshift;

  static const T yearshift = 9;
  static const T yearlen =
     std::numeric_limits<T>::digits - 9;
  static const T yearmask =
     ((one << yearlen) - 1U) << yearshift;

  T data;

public:
  bitmask_date& month(unsigned m)
  {
    data = (data & ~monthmask) |
           (m << monthshift);
    return *this;
  }
  unsigned month() const
  {
    return (data & monthmask) >> monthshift;
  }
  // Other setter/getter functions
  // trivially similar

  void raw_inc() { ++data; }
  T raw() const { return data; }
};
```

Listing 4

All measurements are made on an old Samsung NC10 Netbook, running 32-bit Ubuntu 12.04, since it has the smallest cache of all computers I have readily available. The test programs are compiled with g++ 4.6.3 using `-O3 -std=c++0x`. The graphs are the averages of 100 runs of each measurement.

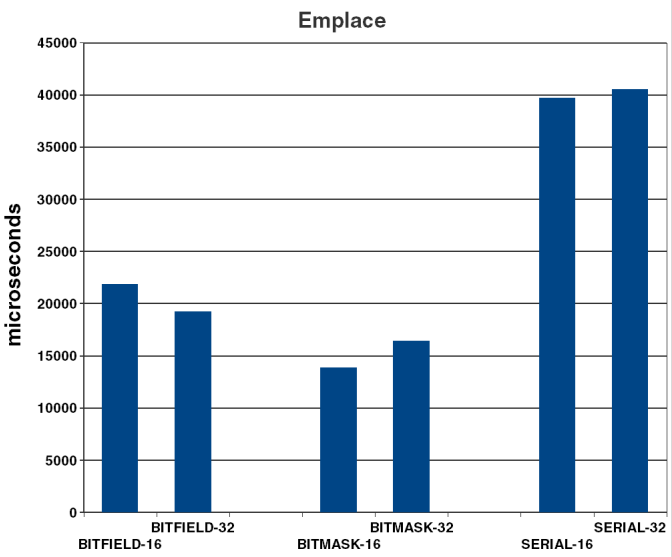


Figure 1

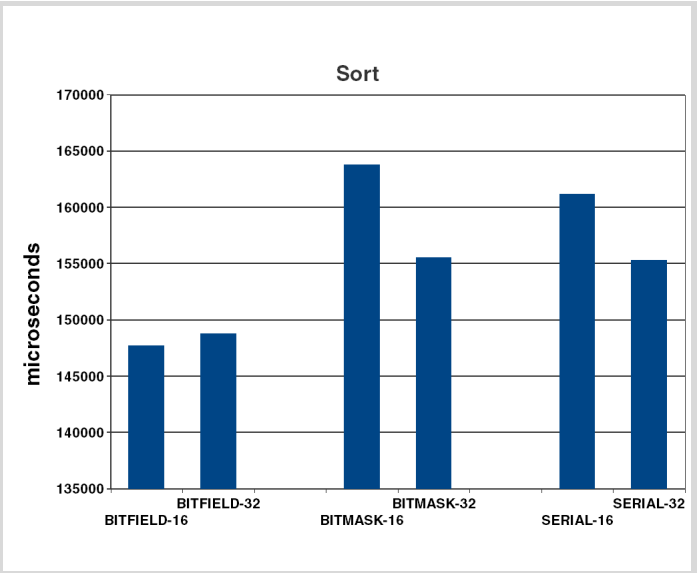


Figure 2

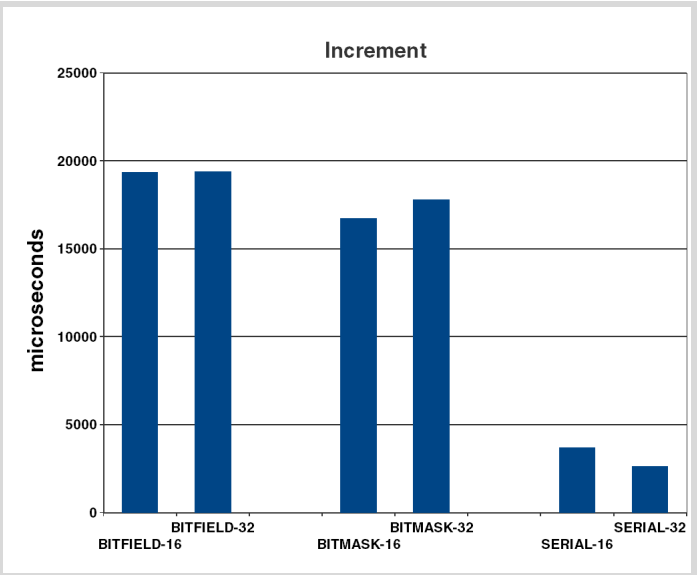


Figure 3

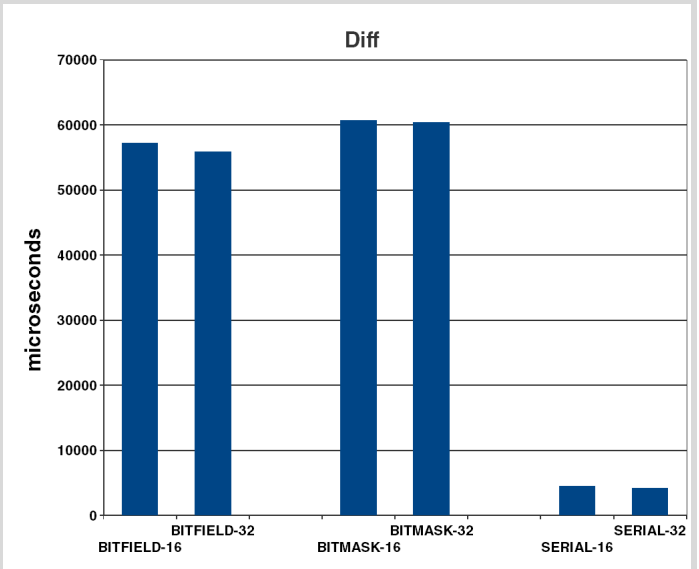


Figure 4

Results

Almost all measurements show a performance advantage for the 32-bit representations, despite a lower number of cache misses (as shown by the valgrind tool 'cachegrind' [cachegrind].) This surprising result is consistent across a 4 different computers tested, all with different versions of x86 CPUs and also with a few different versions of g++.

emplace	That the field based representations are much faster than the serial representation is no surprise. At first I could not understand why the mask/shift version was faster than using the bitfield language feature, but it turns out that the compiler adds unnecessary maskings of the parameter data before insertion into the word since it cannot know that the values are within the legal range. My hand written getter/setter functions do not have that overhead, hence the improved performance.
sort	The difference between the mask/shift field based representation and the serial representation is well within the noise margin for 32-bit implementations, so the performance advantage for sorting of the serial representation shown in [N3344] is not obvious. Why the bitfield version is around 15% faster on the average is a mystery.
increment	Here there is a huge expected advantage with the serial representation, showing approximately five times faster operation. The reason for mask/shift to perform about 10% better than bitfield is unclear, but perhaps it is the above mentioned generated extra masking in the bitfield version that does it.
diff	An even greater advantage of the serial representation, roughly 13 times, where the difference is calculated by a mere integer subtraction, while the field based versions must first compute the JDN of each and subtract those. Why the mask/shift version performs slightly worse than the bitfield versions is not clear.
extract ymd	As expected, this is where the field based representations shine. The performance is in the vicinity of 14 times better for field based representations than serial representations. It is not easy to see in the graph, but the mask/shift version is around 20% faster than the bitfield versions for both the 16-bit and 32-bit versions.

Conclusions

It is not surprising that there probably is no one representation that is best for every need.

For me, the noticeably better performance of hand written bit field management using mask/shift operations was a surprise, but it was understood once the code generated by the compiler was examined. This does not explain why the bitfield version was faster in some of the tests, though. Most peculiar is the substantial performance difference when sorting dates.

It definitely came as a surprise that cutting the size of the data set in half by using 16-bit representations almost always made a difference for the worse. Certainly there is a disadvantage for emplace and extract ymd, and on diff for the field based versions, since the epoch must be subtracted or added, but for all other measurements there is an obvious reduction in number of cache misses, and the increase in number of instructions or number of data accesses are insignificant, and still performance suffers a lot. Indeed, this find was so surprising that a CPU architecture buff of a colleague immediately asked for the source code to analyse. ■

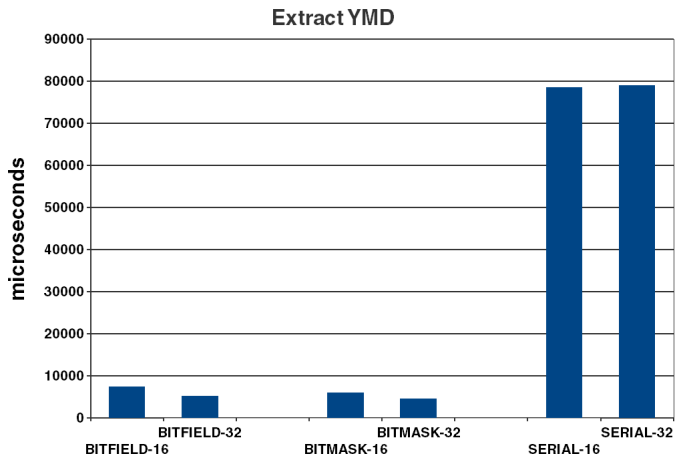


Figure 5

References

[Feb30] See http://en.wikipedia.org/wiki/February_30

[cachegrind] See http://valgrind.org/info/tools.html#cachegrind

[JDN] See http://en.wikipedia.org/wiki/Julian_day

[N3344] See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3344.pdf

[Proleptic] See http://en.wikipedia.org/wiki/Proleptic_Gregorian_calendar

Tail Call Optimisation in C++

Stack based languages can be very powerful, but often can't deal with arbitrary recursion. Andy Balaam finds a way around this limitation.

Some programming languages make recursive programming more practical by providing the tail call optimisation. For a lightning talk at the recent ACCU conference I looked at how we might do something similar in C++. This article attempts a fuller explanation.

Multiplying by two

Here's a toy problem we will use as our example.

Imagine for a second that you want to write a function that multiplies a number by two. OK, we can do that:

```
long times_two_hardware( long value )
{
  return value * 2;
}
```

Now imagine that you don't have the `*` operator.

We're going have to use `+`. We can do that too:

```
long times_two_loop( long value )
{
  long ret = 0;
  for ( long i = 0; i < value; ++i )
  {
    ret += 2;
  }
  return ret;
}
```

(Obviously, this is just a silly example designed to be easy to follow.)

Now imagine that you read somewhere that state was bad, and you could always replace a loop with recursion. Then you might get something like this:

```
long times_two_naive_recursive( long value )
{
  if ( value == 0 )
  {
    return 0;
  }
  else
  {
    return 2 + times_two_naive_recursive
        (value - 1);
  }
}
```

Andy Balaam is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his many open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

This is fine, but what happens when you run it for a large input?

```
$ ulimit -S -s 16
$ ./times_two naive_recursive 100000
Segmentation fault
```

Note that I set my stack size to be very small (16K) to make the point – actually, this will run successfully for very large arguments, but it will eat all your memory and take a long time to finish.

Why does this fail? Because every time you call a function, the state of the current function is saved, and new information is pushed onto the stack about the new function. When you call a function from within a function multiple times, the stack grows and grows, remembering the state all the way down to the place where you started.

So is programming like this useless in practice?

Tail call optimisation

No, because in several programming languages, the compiler or interpreter performs the 'tail call optimisation'.

When you call a function from within some other code you normally need the state of the current code to be preserved. There is a special case where you don't need it though, and this is called a tail call. A tail call is just the situation where you call a function and immediately return its return value as your return value. In this case we don't need any of the state of the current code any more – we are just about to throw it away and return.

The tail call optimisation throws away this unneeded state before calling the new function, instead of after.

In practice, in compiled code, this involves popping all the local variables off the stack, pushing the new function parameters on, and `jmp`ing to the new function, instead of `call`ing it. This means that when we hit the `ret` at the end of the new function, we return to the original caller, instead of the location of the tail call.

Many recursive functions can be re-cast as tail-call versions (sometimes called iterative versions). The one we're looking at is one of those, and Listing 1 is the tail-call version.

It consists of an outer function `times_two_recursive` which just hands off control to the inner function `times_two_recursive_impl`. The inner function uses a counter variable and calls itself recursively, reducing that counter by one each time, until it reaches zero, when it returns the total, which is increased by 2 each time.

The key feature of this implementation is that the recursive function `times_two_recursive_impl` uses a tail call to do the recursion: the value of calling itself is immediately returned, without reference to anything else in the function, even temporary variables.

So, let's see what happens when we compile and run this:

```
$ ulimit -S -s 16
$ ./times_two recursive 100000
Segmentation fault
```

Did I mention that C++ doesn't do the tail call optimisation?

When you call a function from within some other code you normally need the state of the current code to be preserved

```
long times_two_recursive_impl( long total,
                               long counter )
{
  if ( counter == 0 )
  {
    return total;
  }
  else
  {
    return times_two_recursive_impl(
      total + 2, counter - 1 );
  }
}

long times_two_recursive( long value )
{
  return times_two_recursive_impl( 0, value );
}
```

Listing 1

So how would we write code that is tail call optimised in C++? Possibly of more interest to me personally: if we were generating C++ as the output format for some other language, what code might we generate for tail call optimised functions?

Tail call optimised C++

Let's imagine for a second we have some classes, which I'll define later. **`FnPlusArgs`** holds a function pointer and some arguments to be passed to it. **`Answer`** holds on to one of two things: either a **`FnPlusArgs`** to call later, or an actual answer (return value) for our function.

Now we can write our function like Listing 2.

This has the same structure as **`times_two_recursive`**, if a little more verbose. The important point to note though, is that **`times_two_tail_call_impl`** doesn't call itself recursively. Instead, it returns an **`Answer`** object, which is a delegate saying that we have more work to do: calling the provided function with the supplied arguments.

The trampoline

All we need now is some infrastructure to call this function, and deal with its return value, calling functions repeatedly until we have an answer. This function is called a 'trampoline', and you can sort of see why:

```
long trampoline( Answer answer )
{
  while ( !answer.finished_ )
  {
    answer = answer.tail_call_();
  }
  return answer.value_;
}
```

Tail call optimisation and the C++ standard

Tail call optimisation isn't in the C++ standard. Apparently, some compilers, including MS Visual Studio and GCC, do provide tail call optimisation under certain circumstances (when optimisations are enabled, obviously). It is difficult to implement for all cases, especially in C++ since destruction of objects can cause code to be executed where you might not have expected it, and it doesn't appear to be easy to tell when a compiler will or will not do it without examining the generated assembly language. Languages which have this feature by design, like Scheme, can do it more predictably.

While the answer we get back tells us we have more work to do, we call functions, and when we're finished we return the answer.

Now all we need to get this working is the definition of **`Answer`** and **`FnPlusArgs`**, which are shown in Listing 3.

The only notable thing about this is that we use **`operator()`** on **`FnPlusArgs`** to call the function it holds.

```
Answer times_two_tail_call_impl( long acc,
                                 long i )
{
  if ( i == 0 )
  {
    // First argument true means we have finished -
    // the answer is acc
    return Answer( true, null_fn_plus_args, acc );
  }
  else
  {
    // First argument false means more work to do -
    // call the supplied function with these args
    return Answer(
      false,
      FnPlusArgs(
        times_two_tail_call_impl,
        acc + 2,
        i - 1
      ),
      0
    );
  }
}
long times_two_tail_call( long n )
{
  return tail_call( Answer(
      false,
      FnPlusArgs( times_two_tail_call_impl,
                  0, n ),
      0 ) );
}
```

Listing 2

The tail call version can process arbitrarily large input, but how much do you pay for that in terms of performance?

```
struct Answer;
typedef Answer (*impl_fn_type)( long, long );

struct FnPlusArgs
{
  impl_fn_type fn_;
  long arg1_;
  long arg2_;
  FnPlusArgs(
    impl_fn_type fn,
    long arg1,
    long arg2
  );
  Answer operator()();
};

impl_fn_type null_fn = NULL;
FnPlusArgs null_fn_plus_args( null_fn, 0, 0 );

struct Answer
{
  bool finished_;
  FnPlusArgs tail_call_;
  long value_;
  Answer( bool finished,
          FnPlusArgs tail_call, long value );
};

FnPlusArgs::FnPlusArgs(
  impl_fn_type fn,
  long arg1,
  long arg2
)
: fn_( fn )
, arg1_( arg1 )
, arg2_( arg2 )
{
}

Answer FnPlusArgs::operator()()
{
  return fn_( arg1_, arg2_ );
}

Answer::Answer( bool finished, FnPlusArgs
tail_call, long value )
: finished_( finished )
, tail_call_( tail_call )
, value_( value )
{
}
```

Listing 3

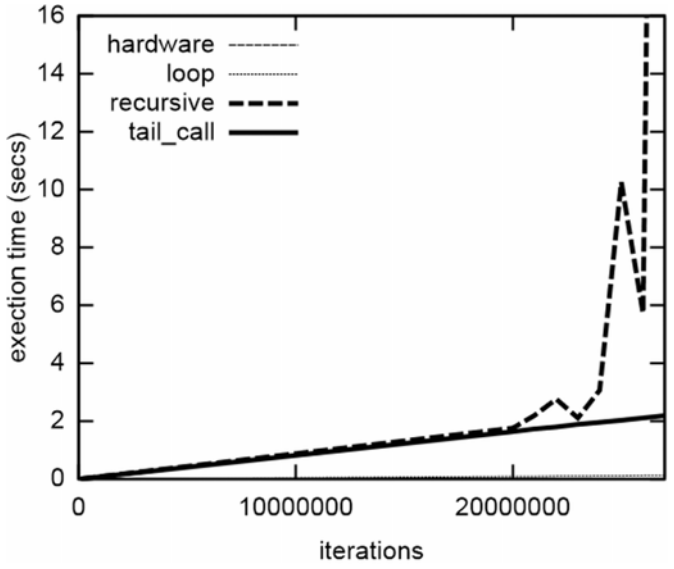


Figure 1

Results

Now, when we run this code, we get what we wanted:

```
$ ulimit -S -s 16
$ ./times_two tail_call 100000
200000
```

(In other words, it doesn't crash.)

So, it turns out that the tail call optimisation is just a **`while`** loop. Sort of.

Performance

You might well be interested in the performance of this code relative to normal recursion. The tail call version can process arbitrarily large input, but how much do you pay for that in terms of performance?

Recall that there are 4 different versions of our function, called **`times_two`**. The first, 'hardware', uses the **`*`** operator to multiply by 2. The second, 'loop' uses a for loop to add up lots of 2s until we get the answer. The third, 'recursive', uses a recursive function to add up 2s. The fourth, 'tail_call' is a reimplementation of 'recursive', with a manual version of the tail call optimisation.

Let's look first at memory usage. The stack memory usage over time as reported by Massif [Massif] of calling the four functions for a relatively small input value of 100000 is shown in Figure 1.

The recursive function uses way more memory than the others (note the logarithmic scale), because it keeps all those stack frames, and the tail_call version takes much longer than the others (possibly because it puts more strain on Massif?), but keeps its memory usage low. Figure 2 shows how that affects its performance, for different sizes of input.

those pesky hardware engineers with their new-fangled * operator managed to defeat all comers with their unreasonable execution times

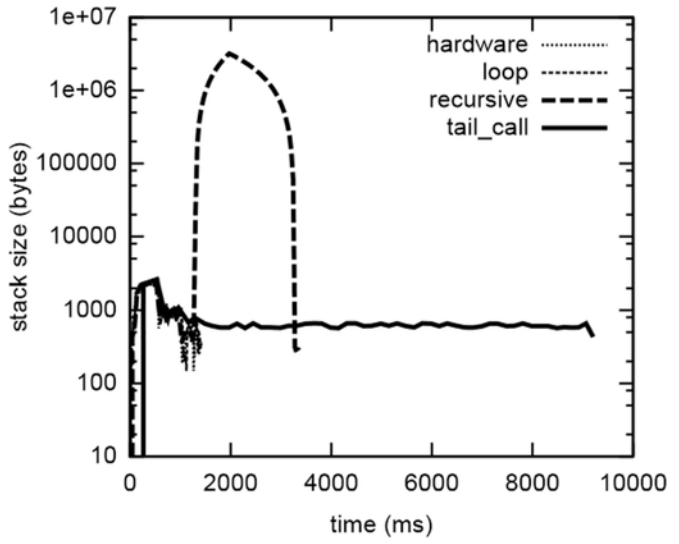


Figure 2

For these much larger input values, the recursive and tail_call functions take similar amounts of time, until the recursive version starts using all the physical memory on my computer. At this point, its execution times become huge, and erratic, whereas the tail_call function plods on, working fine.

So the overhead of the infrastructure of the tail call doesn't have much impact on execution time for large input values, but it's clear from the barely-visible thin dotted line at the bottom that using a for-loop with a mutable loop variable instead of function calls is way, way faster, with my compiler, on my computer, in C++. About 18 times faster, in fact.

And, just in case you were wondering: yes those pesky hardware engineers with their new-fangled ***** operator managed to defeat all comers with their unreasonable execution times of 0 seconds every time (to the nearest 10ms). I suppose that shows you something.

Generalisation

Of course, the code shown above is specific to a recursive function taking two long arguments and returning a long. However, the idea may be generalised. If we make our trampoline a function template, taking the return value as a template parameter, as in Listing 4, which must work with a pointer to an **`IAnswer`** class template like Listing 5, which in turn uses an **`IFnPlusArgs`** class template (Listing 6), we may generalise to functions taking different numbers of arguments, of different types. It is worth noting that only the return type is required as a template parameter. Concrete classes derived from **`IAnswer`** and **`IFnPlusArgs`** may be template classes themselves, but because of the use of these interfaces the types of the arguments need not leak into the trampoline code, meaning that multiple functions with different argument lists may call each other recursively. (Of course, they must all agree on the eventual return value type.) There is an example of how this might be implemented in my blog, along with the full source code for this article [Code].

```
template<typename RetT>
const RetT trampoline_templ
   ( std::auto_ptr< IAnswer<RetT> > answer )
{
  while( !answer->finished() )
  {
    answer = answer->tail_call()();
  }
  return answer->value();
}
```

Listing 4

```
template<typename RetT>
struct IAnswer {
  virtual bool finished() const = 0;
  virtual
     IFnPlusArgs<RetT>& tail_call() const = 0;
  virtual RetT value() const = 0;
};
```

Listing 5

Since this generalisation requires dynamic memory use (because the **`IAnswer`** instances are handled by pointer) this solution may be slower than the stack-only implementation above, but since all the memory is acquired and released in quick succession it is unlikely to trigger prohibitively expensive allocation and deallocation algorithms.

The examples [Code] demonstrate the use of template classes to provide **`IAnswer`** and **`IFnPlusArgs`** objects for each function type signature, and that functions with different signatures may call each other to co-operate to return a value. Generalising the supplied **`Answer2`**, **`Answer3`** etc. class templates to a single class template using C++11 variadic templates or template metaprogramming is left as an exercise for the reader. ■

References

[Code] http://www.artificialworlds.net/blog/2012/04/30/tail-call-optimisation-in-cpp/

[Massif] http://valgrind.org/docs/manual/ms-manual.html

```
template<typename RetT>
struct IFnPlusArgs {
  typedef
     std::auto_ptr< IAnswer<RetT> > AnswerPtr;
  virtual AnswerPtr operator()() const = 0;
};
```

Listing 6

All About XOR

Boolean operators are the bedrock of computer logic. Michael Lewin investigates a common one and shows there's more to it than meets the eye.

You probably already know what XOR is, but let's take a moment to formalise it. XOR is one of the sixteen possible binary operations on Boolean operands. That means that it takes 2 inputs (it's binary) and produces one output (it's an operation), and the inputs and outputs may only take the values of TRUE or FALSE (it's Boolean) – see Figure 1. We can (and will) interchangeably consider these values as being 1 or 0 respectively, and that is why XOR is typically represented by the symbol ⊕: it is equivalent to the addition operation on the integers modulo 2 (i.e. we wrap around so that 1 + 1 = 0)[1] [SurreyUni]. I will use this symbol throughout, except in code examples where I will use the C operator ^ to represent XOR.

XOR Truth Table		
Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1

Certain Boolean operations are analogous to set operations (see Figure 2): AND is analogous to intersection, OR is analogous to union, and XOR is analogous to set difference. This is not just a nice coincidence; mathematically it is known as an isomorphism[2] and it provides us with a very neat way to visualise and reason about such operations.

Important properties of XOR

There are 4 very important properties of XOR that we will be making use of. These are formal mathematical terms but actually the concepts are very simple.

1. *Commutative*: $A \oplus B = B \oplus A$

   This is clear from the definition of XOR: it doesn't matter which way round you order the two inputs.

2. *Associative*: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$

   This means that XOR operations can be chained together and the order doesn't matter. If you aren't convinced of the truth of this statement, try drawing the truth tables.

3. *Identity element*: $A \oplus 0 = A$

   This means that any value XOR'd with zero is left unchanged.

4. *Self-inverse*: $A \oplus A = 0$

   This means that any value XOR'd with itself gives zero.

These properties hold not only when XOR is applied to a single bit, but also when it is applied bitwise to a vector of bits (e.g. a byte). For the rest of this article I will refer to such vectors as bytes, because it is a concept that all programmers are comfortable with, but don't let that make you think that the properties only apply to a vector of size 8.

Interpretations

We can interpret the action of XOR in a number of different ways, and this helps to shed light on its properties. The most obvious way to interpret it is as its name suggests, 'exclusive OR': $A \oplus B$ is true if and only if precisely one of A and B is true. Another way to think of it is as identifying *difference* in a pair of bytes: $A \oplus B$ = 'the bits where they differ'. This interpretation makes it obvious that $A \oplus A = 0$ (byte A does not differ from itself in any bit) and $A \oplus 0 = A$ (byte A differs from 0 precisely in the bit positions that equal 1) and is also useful when thinking about toggling and encryption later on.

The last, and most powerful, interpretation of XOR is in terms of *parity*, i.e. whether something is odd or even. For any n bits, $A1 \oplus A2 \oplus \ldots \oplus A_n = 1$ if and only if the number of 1s is odd. This can be proved quite easily by induction and use of associativity. It is the crucial observation

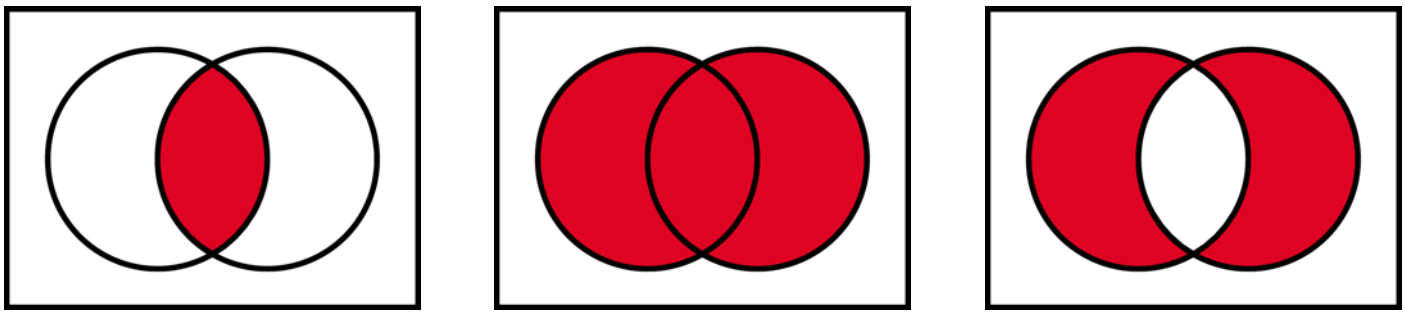


Figure 2

Michael Lewin has had spells in the video games and banking industries, but found his calling with Palantir Government, a platform for making sense of big data in fields such as law enforcement, intelligence and public health. He can be contacted at migwellian@gmail.com

1. In this way, complex logical expressions can be reasoned about and simplified using modulo arithmetic. This is much easier than the commonly taught method of using Karnaugh maps, although OR operations do not map neatly in this way.
2. Formally, the actions of XOR and AND on $\{0,1\}^N$ form a *ring* that is *isomorphic* to the actions of set difference and union on sets. For more details see the appendix.

We can interpret the action of XOR in a number of different ways, and this helps to shed light on its properties

that leads to many of the properties that follow, including error detection, data protection and adding.

Toggling

Armed with these ideas, we are ready to explore some applications of XOR. Consider the following simple code snippet:

```
for (int n=x; true; n ^= (x ^ y))
  printf("%d ", n);
```

This will toggle between two values **x** and **y**, alternately printing one and then the other. How does it work? Essentially the combined value **x ^ y** 'remembers' both states, and one state is the key to getting at the other. To prove that this is the case we will use all of the properties covered earlier:

$B \oplus (A \oplus B)$	(commutative)
$= B \oplus (B \oplus A)$	(associative)
$= (B \oplus B) \oplus A$	(self-inverse)
$= 0 \oplus A$	(identity element)
$= A$	

Toggling in this way is very similar to the concept of a *flip-flop* in electronics: a 'circuit that has two stable states and can be used to store state information' [Wikipedia-1].

Save yourself a register

Toggling is all very well, but it's probably not that useful in practice. Here's a function that is more useful. If you haven't encountered it before, see if you can guess what it does.

```
void s(int& a, int& b)
{
  a = a ^ b;
  b = a ^ b;
  a = a ^ b;
}
```

Did you work it out? It's certainly not obvious, and the below equivalent function is even more esoteric:

```
void s(int& a, int& b)
{
  a ^= b ^= a ^= b;
}
```

It's an old trick that inspires equal measures of admiration and vilification. In fact there is a whole repository of interview questions whose name is inspired by this wily puzzle: http://xorswap.com/. That's right, it's a function to swap two variables in place without having to use a temporary variable. Analysing the first version: the first line creates the XOR'd value. The second line comprises an expression that evaluates to **a** and stores it in **b**, just as the toggling example did. The third line comprises an expression that evaluates to **b** and stores it in **a**. And we're done! Except there's a bug: what happens if we call **s(myVal, myVal)**? This is an example of *aliasing*, where two arguments to a function share the same location in memory, so altering one will affect the other. The outcome is that **myVal == 0** which is certainly not the semantics we expect from a swap function!

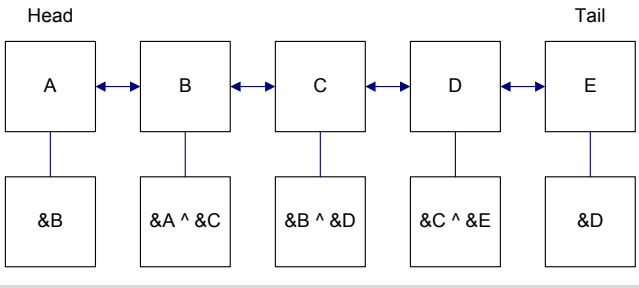


Figure 3

Perhaps there is some retribution for this much maligned idea, however. This is more than just a devious trick when we consider it in the context of assembly language. In fact XOR'ing a register with itself is the fastest way for the compiler to zero the register.

Doubly linked list

A node in a singly linked list contains a value and a pointer to the next node. A node in a doubly linked list contains the same, plus a pointer to the previous node. But in fact it's possible to do away with that extra storage requirement. Instead of storing either pointer directly, suppose we store the XOR'd value of the previous and next pointers [Wikipedia-2] – see Figure 3.

Note that the nodes at either end store the address of their neighbours. This is consistent because conceptually we have XOR'ed that address with 0. Then the code to traverse the list looks like Listing 1, which was adapted from Stackoverflow [Stackoverflow].

This uses the same idea as before, that one state is the key to getting at the other. If we know the address of any consecutive pair of nodes, we can derive the address of their neighbours. In particular, by starting from one end we can traverse the list in its entirety. A nice feature of this function is that this same code can be used to traverse either forwards or backwards. One important caveat is that it cannot be used in conjunction with garbage collection, since by obfuscating the nodes' addresses in this way the nodes would get marked as unreachable and so could be garbage collected prematurely.

Pseudorandom number generator

XOR can also be used to generate pseudorandom numbers in hardware. A pseudorandom number generator (whether in hardware or software e.g. **std::rand()**) is not truly random; rather it generates a deterministic sequence of numbers that appears random in the sense that there is no obvious pattern to it. This can be achieved very fast in hardware using a *linear feedback shift register*. To generate the next number in the sequence, XOR the highest 2 bits together and put the result into the lowest bit, shifting all the other bits up by one. This is a simple algorithm but more

If we know the address of any consecutive pair of nodes, we can derive the address of their neighbours

```
// traverse the list given either the head or
// the tail
void traverse( Node *endPoint )
{
  Node* prev = endPoint;
  Node* cur = endPoint;

  while ( cur )
  // loop until we reach a null pointer
  {
    printf( "value = %d\n", cur->value);
    if ( cur == prev )
    // only true on first iteration
      cur = cur->prevXorNext;
      // move to next node in the list
    else
    {
      Node* temp = cur;
      cur = (Node*)((uintptr_t)prev
         ^ (uintptr_t)cur->prevXorNext);
      // move to next node in the list
      prev = temp;
    }
  }
}
```

Listing 1

complex ones can be constructed using more XOR gates as a function of more than 2 of the lowest bits [Yikes]. By choosing the architecture carefully, one can construct it so that it passes through all possible states before returning to the start of the cycle again (Figure 4).

Encryption

The essence of encryption is to apply some *key* to an input message in order to output a new message. The encryption is only useful if it is very hard to reverse the process. We can achieve this by applying our key over the message using XOR (see Listing 2).

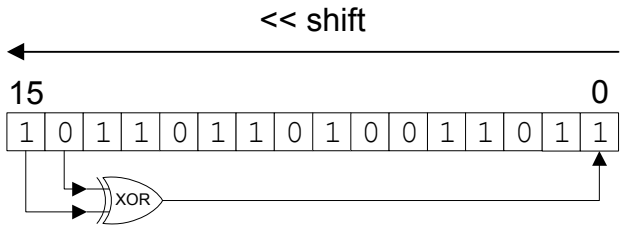


Figure 4

```
string EncryptDecrypt(string inputMsg,string key)
{
  string outputMsg(inputMsg);

  short unsigned int keyLength = key.length();
  short unsigned int strLength =
     inputMsg.length();

  for(int v=0, k=0;v<strLength;++v)
  {
    outputMsg[v] = inputMsg[v]^key[k];
    ++k;
    k = k % keyLength;
  }
  return outputMsg;
}
```

Listing 2

The choice of key here is crucial to the strength of the encryption. If it is short, then the code could easily be cracked using the centuries-old technique of frequency analysis. As an extreme example, if the key is just 1 byte then all we have is a *substitution cipher* that consistently maps each letter of the alphabet to another one. However, if the key is longer than the message, and generated using a '*truly random*' hardware random number generator, then the code is unbreakable [Wikipedia-3]. In practice, this '*truly random*' key could be of fixed length, say 128 bits, and used to define a *linear feedback shift register* that creates a pseudorandom sequence of arbitrary length known as a *keystream*. This is known as a *stream cipher*, and in a real-worl situation this would also be combined with a secure hash function such as *md5* or *SHA-1*.

Another type of cipher is the *block cipher* which operates on the message in blocks of fixed size with an unvarying transformation. An example of XOR in this type of encryption is the International Data Encryption Algorithm (IDEA) [Wikipedia-4].

The best-known encryption method is the *RSA algorithm*. Even when the above algorithm is made unbreakable, it has one crucial disadvantage: it is not a *public key* system like RSA. Using RSA, I can publish the key others need to send me encrypted messages, but keep secret my private key used to decrypt them. On the other hand, in XOR encryption the same key is used to encrypt and decrypt (again we see an example of toggling). Before you can send me encrypted messages I must find a way to secretly tell you the key to use. If an adversary intercepts that attempt, my code is compromised because they will be able to decrypt all the messages you send me.

Error detection

Now we will see the first application of XOR with respect to *parity*. There are many ways to defend against data corruption when sending digital information. One of the simplest is to use XOR to combine *all* the bits

By comparing the received parity bit with the calculated one, we can reliably determine when a single bit has been corrupted

together into a single *parity bit* which gets appended to the end of the message. By comparing the received parity bit with the calculated one, we can reliably determine when a single bit has been corrupted (or indeed any odd number of bits). But if 2 bits have been corrupted (or indeed any even number of bits) this check will not help us.

Checksums and *cyclic redundancy checks* (CRC) extend the concept to longer check values and reducing the likelihood of collisions and are widely used. It's important to note that such checks are *error-detecting* but not *error-correcting*: we can tell that an error has occurred, but we don't know where it occurred and so can't recover the original message. Examples of error-correcting codes that also rely on XOR are *BCH* and *Reed-Solomon* [Wikipedia-5] [IEEEXplore].

RAID data protection

The next application of XOR's parity property is RAID (Redundant Arrays of Inexpensive Disks) [Mainz] [DataClinic]. It was invented in the 1980s as a way to recover from hard drive corruption. If we have n hard drives, we can create an additional one which contains the XOR value of all the others:

$$A^* = A_1 \oplus A_2 \oplus \ldots \oplus A_n$$

This introduces *redundancy*: if a failure occurs on one drive, say A_1, we can restore it from the others since:

$$\begin{aligned}
&A_2 \oplus \ldots \oplus A_n \oplus A^* \\
&= A_2 \oplus \ldots \oplus A_n \oplus (A_1 \oplus A_2 \oplus \ldots \oplus A_n) && \text{(definition of A*)} \\
&= A_1 \oplus (A_2 \oplus A_2) \oplus \ldots \oplus (A_n \oplus A_n) && \text{(commutative and associative: rearrange terms)} \\
&= A_1 \oplus 0 \oplus \ldots \oplus 0 && \text{(self-inverse)} \\
&= A_1 && \text{(identity element)}
\end{aligned}$$

This is the same reasoning used to explain toggling earlier, but applied to n inputs rather than just 2.

In the (highly unlikely) event that 2 drives fail simultaneously, the above would not be applicable so there would be no way to recover the data.

Building blocks of XOR

Let's take a moment to consider the fundamentals of digital computing, and we will see that XOR holds a special place amongst the binary logical operations.

Computers are built from logic gates, which are in turn built from transistors. A transistor is simply a switch that can be turned on or off using an electrical signal (as opposed to a mechanical switch that requires a human being to operate it). So for example, the AND gate can be built from two transistors in series, since *both* switches must be closed to allow current to flow, whereas the OR gate can be built from two transistors in parallel, since closing *either* switch will allow the current to flow.

Most binary logical operations can be constructed from two or fewer transistors; of all 16 possible operations, the only exception is XOR (and its complement, XNOR, which shares its properties). Until recently, the simplest known way to construct XOR required six transistors [Hindawi]: the simplest way to see this is in the diagram below, which comprises three gates, each of which requires two transistors. In 2000, Bui *et al* came up with a design using only four transistors [Bui00] – see Figure 5.

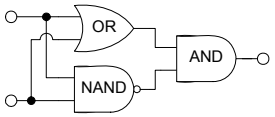


Figure 5

Linear separability

Another way in which XOR stands apart from other such operations is to do with *linear separability*. This is a concept from Artificial Intelligence relating to classification tasks. Suppose we have a set of data that fall into two categories. Our task is to define a single boundary line (or, extending the notion to higher dimensions, a hyperplane) that neatly partitions the data into its two categories. This is very useful because it gives us the predictive power required to correctly classify new unseen examples. For example, we might want to identify whether or not someone will default on their mortgage payments using only two clues: their annual income and the size of their property. Figure 6 is a hypothetical example of how this might look.

A new mortgage application might be evaluated using this model to determine whether the applicant is likely to default.

Not all problems are neatly separable in this way. That means we either need more than one boundary line, or we need to apply some kind of non-linear transformation into a new space in which it is linearly separable: this

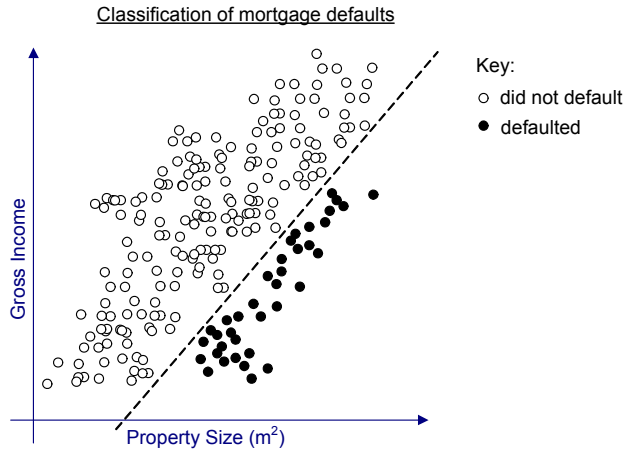


Figure 6

we are able to chain as many of these adders together as we wish in order to add numbers of any size

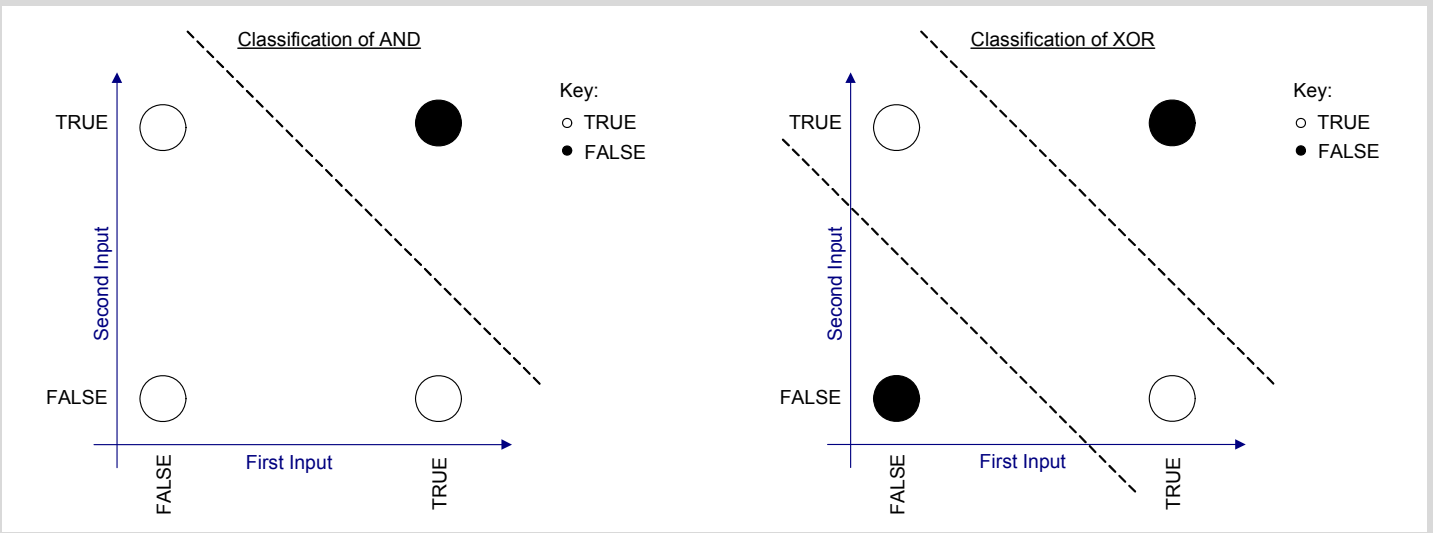


Figure 7

is how machine learning techniques such as neural networks and support vector machines work. The transformation process might be computationally expensive or completely unachievable. For example, the most commonly used and rigorously understood type of neural network is the *multi-layer perceptron*. With a single layer it is only capable of classifying linearly separable problems. By adding a second layer it can transform the problem space into a new space in which the data is linearly separable, but there's no guarantee on how long it may take to converge to a solution.

So where does XOR come into all this? Let's picture our binary Boolean operations as classification tasks, i.e. we want to classify our four possible inputs into the class that outputs TRUE and the class that outputs FALSE. Of all the 16 possible binary Boolean operations, XOR is the only one (with its complement, XNOR) that is not linearly separable with a single boundary line: two lines are required, as the diagram in Figure 7 demonstrates.

Inside your ALU

XOR also plays a key role inside your processor's *arithmetic logic unit* (ALU). We've already seen that it is analogous to addition modulo 2, and in fact that is exactly how your processor calculates addition too. Suppose first of all that you just want to add 2 bits together, so the output is a number between 0 and 2. We'll need two bits to represent such a number. The lower bit can be calculated by XOR'ing the inputs. The upper bit (referred to as the 'carry bit') can be calculated with an AND gate because it only equals 1 when both inputs equal 1. So with just these two logic gates, we have a module that can add a pair of bits, giving a 2-bit output. This structure is called a *half adder* and is depicted in Figure 8.

Now of course we want to do a lot more than just add two bits: just like you learnt in primary school, we need to carry the 'carry bit' along because it will play a part in the calculation of the higher order bits. For that we need to augment what we have into a *full adder*. We've added a third input that enables us to pass in a carry bit from some other adder. We begin with a half adder to add our two input bits. Then we need another half adder to add the result to the input carry bit. Finally we use an OR gate to combine the carry bits output by these two half adders into our overall output carry bit. (If you're not convinced of this last step, try drawing the truth table.) This structure is represented in Figure 9.

Now we are able to chain as many of these adders together as we wish in order to add numbers of any size. The diagram below shows an 8-bit adder array, with the carry bits being passed along from one position to the next. Everything in electronics is modular, so if you want to add 32-bit numbers you could buy four of these components and connect them together (see Figure 10).

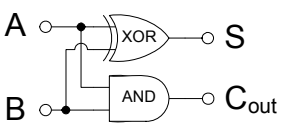


Figure 8

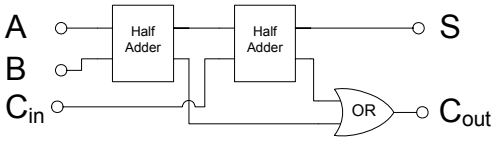


Figure 9

we have defined three isomorphic rings in the spaces of Boolean algebra, modulo arithmetic and set theory

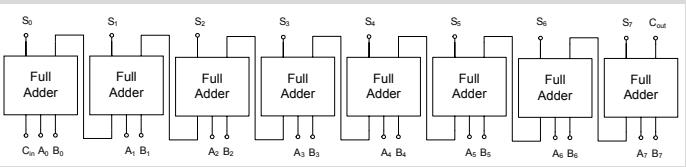


Figure 10

If you are interested in learning more about the conceptual building blocks of a modern computer, Charles Petzold's book *Code* comes highly recommended.

More detail on the Group Theory

For those comfortable with the mathematics, here is a bit more detail of how XOR fits into group theory.

An *algebraic structure* is simply a mathematical object (S, ~) comprising a set S and a binary operation ~ defined on the set.

A group is an algebraic structure such that the following 4 properties hold:

1. ~ is closed over X, i.e. the outcome of performing ~ is always an element of X
2. ~ is associative
3. An identity element e exists that, when combined with any other element of X, leaves it unchanged
4. Every element in X has some inverse that, when combined with it, gives the identity element

We are interested in the operation XOR as applied to the set of Boolean vectors $S = \{T, F\}^N$, i.e. the set of vectors of length N whose entries can only take the values T and F. (I mean *vector* in the mathematical sense, i.e. it has fixed length. Do not confuse this with the C++ data structure `std::vector`, which has variable length.)We have already seen that XOR is associative, that the vector (F, … F) is the identity element and that every element has itself as an inverse. It's easy to see that it is also closed over the set. Hence (S, XOR) is a group. In fact it is an *Abelian* group because we showed above that XOR is also commutative.

Two groups are said to be *isomorphic* if there is a one-to-one mapping between the elements of the sets that preserves the operation. I won't write that out formally (it's easy enough to look up) or prove the isomorphisms below (let's call that an exercise for the reader). Instead I will just define them and state that they are isomorphisms.

The group $(\{T, F\}^N, XOR)$ is isomorphic to the group $(\{0, 1\}^N, +)$ of addition modulo 2 over the set of vectors whose elements are integers mod 2. The isomorphism simply maps T to 1 and F to 0.

The group $(\{T, F\}^N, XOR)$ is also isomorphic to the group $(P(S), \Delta)$ of symmetric difference Δ over the power set of N elements[1]: the isomorphism maps T to 'included in the set' and F to 'excluded from the set' for each of the N entries of the Boolean vector.

Let's take things one step further by considering a new algebraic structure called a *ring*. A ring $(S,+, \times)$ comprises a set S and a pair of binary operations + and × such that S is an Abelian group under + and a semigroup[2] under ×. Also × is distributive over +. The symbols + and × are chosen deliberately because these properties mean that the two operations behave like addition and multiplication.

We've already seen that XOR is an Abelian group over the set of Boolean vectors, so it can perform the role of the + operation in a ring. It turns out that AND fulfils the role of the * operation. Furthermore we can extend the isomorphisms above by mapping AND to multiplication modulo 2 and set intersection respectively. Thus we have defined three isomorphic rings in the spaces of Boolean algebra, modulo arithmetic and set theory. ■

References

[Bui00] H. T. Bui, A. K. Al-Sheraidah, and Y. Wang, 'New 4-transistor XOR and XNOR designs', in *Proceedings of the 2nd IEEE Asia Pacific Conference*

[DataClinic] http://www.dataclinic.co.uk/raid-parity-xor.htm

[Hindawi] http://www.hindawi.com/journals/vlsi/2009/803974/

[IEEEXplore] http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1347837

[Mainz] http://www.staff.uni-mainz.de/neuffer/scsi/what_is_raid.html

[Stackoverflow] http://stackoverflow.com/questions/3531972/c-code-for-xor-linked-list

[SurreyUni] http://www.ee.surrey.ac.uk/Projects/Labview/minimisation/karnaugh.html

[Wikipedia-1] http://en.wikipedia.org/wiki/Flip-flop_%28electronics%29

[Wikipedia-2] http://en.wikipedia.org/wiki/XOR_linked_list

[Wikipedia-3] http://en.wikipedia.org/wiki/XOR_cipher

[Wikipedia-4] http://en.wikipedia.org/wiki/International_Data_Encryption_Algorithm

[Wikipedia-5] http://en.wikipedia.org/wiki/Finite_field_arithmetic

[Yikes] http://www.yikes.com/~ptolemy/lfsr_web/index.htm

1. The *power set* means the set of all possible subsets, i.e. this is the set of all sets containing up to N elements.

2. A *semigroup* is a group without the requirement that every element has an inverse.

Curiously Recursive Template Problems with Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a programming paradigm that makes possible to clearly express programs separated into 'aspects', including appropriate isolation, composition and reuse of the aspect code [Kiczales97]. AOP defines *weaving* as the process of composing the aspects into a single entity.

Independently, there are situations in which a base class needs to know its subclass, e.g. for type-safe downcasts. The CURIOUSLY RECURRING TEMPLATE PATTERN (CRTP) is a C++ idiom in which a class X derives from a class template instantiation using X itself as template argument [Abrahams04]. This way, the base class can know the derived type.

Both AOP and the CRTP are widely adopted C++ programming techniques. In particular, there exists an AOP easy implementation using templates [Spinczyk05]. However, a C++ grammar incompatibility arises when combining AOP and CRTP. While there exists a C++ dialect called AspectC++ [Spinczyk05], we don't evaluate in this work its ability to combine AOP and CRTP since it requires its own compiler extensions and so its not standard C++. Here we look at a simple solution implemented in standard C++ that addresses the issue without any overhead penalty.

Problems combining AOP + CRTP

There are some situations where combining the benefits of AOP and CRTP are desirable; however, as we will show below, some problems arise when applying together the individual standard procedures of each technique.

The code in Listing 1 shows an attempt of adding functionality, through aspects, to a base class named **`Number`**.

We can observe that the return type of ArithmeticAspect's operator **+** and **-** needs to know the 'complete type' (**`FULLTYPE`**) when trying to extend the base class functionality through operator overloading. We will address this issue in the following sections.

Hugo Arregui has been working as professional developer for about seven years. The last two years collaborated at FuDePAN, an NGO/NPO that performs R&D in bioinformatics as a C++ Developer and Regional Coordinator for Buenos Aires province (Argentina). He can be contacted at hugo.arregui@gmail.com

Carlos Castro is a Software Engineer with 5+ years of industry experience, passionate about algorithm design and analysis, who specializes in distributed algorithms and information retrieval. Carlos volunteers as a C++ Developer and Technical Lead at FuDePAN. He can be contacted at castro.carlos@hotmail.es

Daniel Gutson is a software developer with more than 15 years of experience, including 7 years in Motorola, 2 years maintaining the GNU toolchain, and 2 years working on proprietary extensible compilers. He submitted some proposals to the C++ standard committee, attended some of their meetings, and also volunteers at FuDePAN in bioinformatic-related activities. He can be contacted at daniel.gutson@gmail.com

```
//basic class
class Number
{
  protected:
  UnderlyingType n;
};

//aspects
template <class NextAspect>
struct ArithmeticAspect: public NextAspect
{
  FULLTYPE operator+
    (const FULLTYPE& other) const;
    // What type is FULLTYPE?
  FULLTYPE operator-
    (const FULLTYPE& other) const;
  FULLTYPE& operator+=
    (const FULLTYPE& other);
  FULLTYPE& operator-=
    (const FULLTYPE& other);
};

template <class NextAspect>
struct LogicalAspect : public NextAspect
{
  bool operator! () const;
  bool operator&& (const FULLTYPE& other) const;
  bool operator|| (const FULLTYPE& other) const;
};

//decorating Number with aspectual code
typedef LogicalAspect
  <ArithmeticAspect<Number > > MyIntegralType;
```

Listing 1

A minimal solution

The basic principle of this solution does not differ in essence from the traditional solution mentioned before.

Problem

`Number` takes the place of the last aspect in the aspects list. However, **`Number`** itself needs to know (as a template template argument) the aspects list, to which it itself belongs, leading to a 'chicken or egg' grammatical dilemma.

For example, if **`Number`** knew the complete type, it could use it as a return type for its operators as shown in Listing 2.

This shows the weaving of a single aspect with CRTP, which works perfectly:

```
LogicalAspect<ArithmeticAspect<Number<??>>>
```

However, we will look for a more generic way to address this issue avoiding drone code cloning

```
template <template <class> class Aspects>
class Number
{
  typedef Aspects<Number<Aspects>> FullType;
...
};
ArithmeticAspect<Number<ArithmeticAspect>>
```

Listing 2

On the other hand, this exposes the problem when trying to weave one additional aspect, since it requires a template template argument, which the aspects lists can't grammatically fulfill as coded above.

We present two solutions: the first being the simplest using C++11's template alias [Reis], and the second using variadic templates (templates that take a variable number of arguments, recently introduced in C++11 [Gregor]) as the only C++11's feature, which in turn, can also be easily implemented in C++98 as well. Both use a common language idiom introduced next, which aims to be used as a library providing a friendly syntax and reduced reusable code.

The proposed language idiom

A possible solution would be to apply some handcrafted per-case base template aliases, as shown below:

```
//with template alias:
template <class T>
using LogicalArithmeticAspect =
   LogicalAspect<ArithmeticAspect<T>>;

//without template alias:
template <class T>
struct LogicalArithmeticAspect
{
  typedef
     LogicalAspect<ArithmeticAspect<T>> Type;
};
```

and with minor changes in the **`Number`** base class's code, we could write the following declaration:

```
LogicalArithmeticAspect
<
  Number<LogicalArithmeticAspect>
>
```

Although this does the trick it tends to be impractical, and also would increment linearly the number of related lines of code in terms of the amount of combinations to be used, which would cause a copy-paste code bloat.

However, we will look for a more generic way to address this issue, avoiding drone code cloning, and being able to encapsulate the method into a library.

```
template
  <template <template <class> class> class Base>
class Decorate
{
  public:
    template<template <class> class ... Aspects>
    struct with
    {
      //...
    };

    //...

  private:
    struct Apply
    { …
    };
};
```

Listing 3

Therefore, in order to provide a user-oriented and easy to use library, we'll use C++11's new variadic-templates so we can cleanly express our intention: to 'decorate' the base class with a list of aspects. An example of what we intend to achieve is shown below:

```
Decorate<Number>::with<ArithmeticAspect,
                       LogicalAspect>
```

The skeleton of the **`Decorate`** class is shown in Listing 3, the details of which will vary in the solutions below.

In both solutions, the **`with`** nested class and the **`Apply`** internal helper will have different implementations.

Solution 1: Using C++11's template alias

In this solution, the **`Decorate::with`** implementation is as shown in Listing 4, and the internal helper **`Apply`** structure also uses templates aliases (see Listing 5).

Despite the implementation between the two solutions differing, the purpose is the same and the underlying idea is explained next in the second solution.

```
template<template <class> class ... Aspects>
struct with
{
  template <class T>
  using AspectsCombination =
     typename Apply<Aspects...>::template Type<T>;
  typedef
    AspectsCombination
      <Base<AspectsCombination>> Type;
};
```

Listing 4

Binder generates a template template argument – combining Arithmetic with Logical aspects – to be used in the base class

```
template<template <class> class A1,
         template <class> class ... Aspects>
struct Apply<A1, Aspects...>
{
  template <class T>
  using Type = A1
    <typename Apply
       <Aspects...>::template Type<T>>;
};
```

Listing 5

Solution 2: Not using C++11's template alias

In this solution, the `Decorate::with` implementation is as shown in Listing 6.

Combining aspects

Now that we have a list of aspects, how could we combine them? The solution we propose is to create a `Binder` class, as shown in Listing 7.

`Binder` encapsulates an aspect (as a template template argument) within a complete type `Binder<Aspect>`. Additionally, it enables us to do a '`bind`' operation to the next aspect or base class, by accessing to the `Binding` inner class.

The way in which `Binder` finally allows us to construct the whole type is shown below.

```
Binder<ArithmeticAspect,
Binder<LogicalAspect>>::Binding<Number>::Type
```

Let's analyze step-by-step this listing (from the innermost to the outermost definition):

1. `Binder<LogicalAspect>` uses the second definition, it just provides a complete type for the aspect with a possibility to bind to another complete type
2. `Binder<ArithmeticAspect, Binder<LogicalAspect>>` uses the first definition.

   The binding generates a `Binder` to the `ArithmeticAspect`, and binds it to `Binder<LogicalAspect>::Binding<T>` generating a template template argument combining both aspects. (In short, `Binder` generates a template template argument – combining Arithmetic with Logical aspects – to be used in the base class).
3. Finally, the type is injected into the base `Number` class.

Since such an implementation is not immediately obvious, we have provided a simplified version in Listing 8 for illustration.

```
template<template <class> class ... Aspects>
struct with
{
  typedef typename Apply<Aspects...>::Type TypeP;
  typedef typename TypeP::template Binding
  <
    Base<TypeP::template Binding>
  >::Type Type;
};
```

Listing 6

```
struct None
{
};
template <template <class> class A,
   class B = None>
struct Binder
{
  template <class T>
  struct Binding
  {
    typedef
      typename Binder<A>::template Binding
      <
        typename B::template
         Binding<T>::Type
      >::Type Type;
   };
};
template<template <class> class T>
struct Binder<T, None>
{
  template <class P>
  struct Binding
  {
    typedef T<P> Type;
  };
};
```

Listing 7

```
template <template <class> class A,
   class B = None>
struct Binder
{
  template <class T>
  struct Binding
  {
    typedef
       Binder<A>::Binding
       <
         B::Binding<T>::Type
       >::Type Type;
   };
};
```

Listing 8

```
template <template <class> class ... Aspects>
struct Apply;
  template <template <class> class T>
  struct Apply<T>
  {
    typedef Binder<T> Type;
  };

  template<template <class> class A1,
    template <class> class ... Aspects>
  struct Apply<A1, Aspects...>
  {
    typedef Binder<A1, typename Apply
    <
      Aspects...
    >::Type> Type;
  };

  template<template <class> class ... Aspects>
  struct with
  {
    typedef
      typename Apply<Aspects...>::Type TypeP;
    typedef
      typename TypeP::template Binding
      <
        Base<TypeP::template Binding>
      >::Type
    Type;
  };
```

Listing 9

Now we've got a **Binder** that implements the weaving of aspects and finally inject them into the base class.

Applying the list of Aspects to the Number class

The only remaining detail is to apply our **Bind** class to the list of aspects. To do this, we define a helper structure called **Apply**, that recursively applies the **Binder** class to each aspect, as shown in Listing 9.

We use the **Apply** helper struct to generate a resulting aspect as the weaving of all the given aspects, and inject it to the base class.

Then **::Type** contains the type we needed, and that's it!

Using the library

The library that implements this idiom provides two tools: the means to obtain the **FullType**, and the means to build it.

Listing 10 shows a way of obtaining the **FullType** with an **Aspect**.

Let's see a final example, using two of the aspects mentioned before:

```
typedef Decorate<Number>::with<ArithmeticAspect,
          LogicalAspect>::Type
          ArithmeticLogicalNumber;
```

Please note that both solutions presented before expose the same interface so this snippet is equally applicable to them.

```
//basic class
template
   <template <class> class Aspects>
   class Number
{
  typedef Aspects<Number<Aspects>> FullType;
  //...
};

// aspect example
template <class NextAspect>
struct ArithmeticAspect: public NextAspect
{
  typedef typename NextAspect::FullType FullType;
  FullType
     operator+ (const FullType& other) const;
  // ...
};
```

Listing 10

C++98 alternative and complete code

The same idea can be implemented using typelists in previous C++ standards, such as C++98.

The complete code of the library and examples both for C++11 and C++98 can be accessed in http://cpp-aop.googlecode.com

Final comments

We think that the solution to the problem exposed in this article could become straightforward by enhancing the language with a reserved keyword to get the full type. We suggest to consider this problem for the next revision of the language standard. ■

References

[Abrahams04] Abrahams, David; Gurtovoy, Aleksey. 2004. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley. ISBN 0-321-22725-5.

[Gregor] Gregor, Douglas; Järvi, Jaako; Powell, Gary. Variadic Templates (Revision 3). [N2080=06-0150]. Programming Language C++. Evolution Working Group.

[Kiczales97] Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin (1997). 'Aspect-Oriented Programming'. *Proceedings of the European Conference on Object-Oriented Programming*, vol.1241. pp. 220–242.

[Reis] Dos Reis, Gabriel; Stroustrup, Bjarne. Template Aliases (Revision 3). [N2258=07-0118]. Programming Language C++. Evolution Working Group.

[Spinczyk05] Spinczyk, Olaf; Lohmann, Daniel; Urban, Matthias. 'AspectC++: an AOP Extension for C++'. *Software Developer's Journal*, pages 68-76, 05/2005.

Valgrind Part 2 – Basic memcheck

Learning how to use our tools well is a vital skill. Paul Floyd shows us how to check for memory problems.

In the first part of this series I explained what Valgrind is. In this article, I'll start explaining how to use it. Memcheck is the best known of the Valgrind tools. It is a runtime memory checker that validates your use of heap memory and (to a lesser extent) stack memory.

Memcheck detects the following kinds of errors

1. Illegal read/write
2. Use of uninitialized memory
3. Invalid system call parameters
4. Illegal frees
5. Source/destination overlap
6. Memory leaks

Some other memory checking tools have snappier TLA names for the errors that they detect. Some people would like to be able to prioritize the types of errors. I'd say that in general all of these errors can cause either incorrect operation of an application or crashes. Generally the 1st and 4th items on the list are the most likely causes of crashes, but don't take that as advice to neglect the other four types.

General advice

Don't overdo the options. The default options are good for most situations. Some of the options will add significantly to the already high overhead. If you discover a fault and the default output is not enough for you to pin down the error, then consider adding more options. Personally I use memcheck in two ways. Firstly in automatic regression tests each weekend. All of the results get distilled into a single summary. Secondly 'interactively' in a shell, and in this mode I tend to turn up the options.

All of the examples that follow use trivial examples. In real world defects, the locations of the fault, the declaration, the allocation, the initialization and the free may all be far apart.

Illegal read/write errors

Illegal read/write errors correspond to reads or writes to addresses that do not belong to any valid address.

The example in Listing 1 shows reading beyond the end of an array.

Compiling this and running it under memcheck will generate the output shown in Figure 1.

If that had been a long long or a long on a 64 bit platform, then it would have been an `Invalid read of size 8`.

Use of uninitialized memory

You'll get this sort of error if you read memory before assigning any value to it. For instance, if you malloc an array then read an element from it.

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.

Valgrind also propagates the state of initialization through assignments and will only trigger an error if the execution outcome could be affected by the uninitialized state of the memory. This means that harmless errors do not generate any messages (good news) but also it means that the site where memcheck says the error occurs could be far from where the uninitialized memory was allocated.

Listing 2 shows an example of this. I've deliberately made the error propagate through three variables in function `f()` to illustrate that no error is generated until the `if()` condition is reached.

This will result in the output shown in Figure 2.

If the error is in a stack variable rather than in a heap variable, you get a bit less information (see Listing 3)..

This gives just the output in Figure 3a.

Use `--memcheck:track- origins=yes` for more info, but this will increase the Valgrind overhead. Adding this option gives the output in Figure 3b.

OK, so it narrows the search down to `main()`, but it doesn't tell us the name of the variable or the line (the file and line numbers in the output are where teh functions start, not where the problem is).

```
// abrw.cpp
#include <iostream>

void f(int *p2)
{
  int i1 = p2[10]; // write beyond the end of p1
  std::cout << "Hello\n";
}

int main()
{
  int *p1 = new int[10];
  f(p1);
  delete [] p1;
}
```

Listing 1

```
==85258== Invalid read of size 4
==85258==    at 0x400AA4: f(int*) (abrw.cpp:5)
==85258==    by 0x400B5E: main (abrw.cpp:12)
==85258==  Address 0x1c90068 is 0 bytes after a
block of size 40 alloc'd
==85258==    at 0x1006BB7: operator
new[](unsigned long) (in /usr/local/lib/valgrind/
vgpreload_memcheck-amd64-freebsd.so)
==85258==    by 0x400B51: main (abrw.cpp:11)
```

Figure 1

With an unbuffered stream, you see the error immediately rather than when the buffer is flushed

```
// uninit.cpp
#include <iostream>

void f(long *p2)
{
  long l1 = p2[10]; // read beyond end of p1
  long l2 = l1;     // propagates
  long l3 = l2;     // propagate again
  if (l3)           // uninitialized read
  {
    std::cout << "Hello\n";
  }
}

int main()
{
  long *p1 = new long[10];
  f(p1);
  delete [] p1;
}
```

Listing 2

```
==93289== Invalid read of size 8
==93289==    at 0x400AA4: f(long*) (uninit.cpp:5)
==93289==    by 0x400B7E: main (uninit.cpp:17)
==93289==  Address 0x1c90090 is 0 bytes after a
block of size 80 alloc'd
==93289==    at 0x1006BB7: operator
new[](unsigned long) (in /usr/local/lib/valgrind/
vgpreload_memcheck-amd64-freebsd.so)
==93289==    by 0x400B71: main (uninit.cpp:16)
```

Figure 2

Invalid system call parameters

Listing 4 is a `std::fwrite` of memory that is not initialized.

This will generate the output shown in Figure 4a.

Look carefully at the log in Figure 4a and you will see that the error occurs when the file is closed, not when the call to `std::fwrite` is performed. This is because the output is cached. And this can be quite pernicious. If I add a call to `std::setvbuf(f, 0, _IONBF, 0);` after the `std::fopen`, then the log that I get as shown in Figure 4b.

With an unbuffered stream, you see the error immediately rather than when the buffer is flushed.

Illegal frees

An example of this is freeing stack memory (Listing 5). This one is a bit of a no-brainer, the compiler complains about the code and I get a nice core dump if I run the application.

```
// uninit2.cpp
#include <iostream>

void f(long l)
{
  long lb = l;
  long lc = lb;
  long ld = lc;
  if (lc)
  {
    std::cout << "Hello\n";
  }
}

int main()
{
  long la; // uninitialized local scalar
  f(la);
}
```

Listing 3

The corresponding output is in Figure 5.

Let's try a somewhat more likely error, using the wrong `delete` (see Listing 6).

The corresponding output is in Figure 6. Here, memcheck correctly identified that there was an incorrect delete, but it doesn't go as far as saying that the memory was allocated with array new but deleted with scalar delete.

```
==4164== Conditional jump or move depends on
uninitialised value(s)
==4164==    at 0x4009E9: f(long) (uninit2.cpp:8)
==4164==    by 0x400A10: main (uninit2.cpp:17)
by 0x400B71: main (uninit.cpp:16)
```

Figure 3a

```
==4455== Conditional jump or move depends on
uninitialised value(s)
==4455==    at 0x4009E9: f(long) (uninit2.cpp:8)
==4455==    by 0x400A10: main (uninit2.cpp:17)
==4455==  Uninitialised value was created by a
stack allocation
==4455==    at 0x400A00: main (uninit2.cpp:14)
```

Figure 3b

Obviously having two sets of code is not ideal, and this will be a maintenance overhead

```
// syscall.cpp
#include <cstdio>

const std::size_t intArraySize = 3;

int main()
{
   std::FILE *f = std::fopen("output.dat", "w");
   if (f)
   {
      int *intArray = new int[intArraySize];
      std::size_t bytesWritten = 0U;
      intArray[0] = 1;
      // intArray[1] not initialized
      intArray[2] = 3;
      bytesWritten = std::fwrite(intArray,
         sizeof(int), intArraySize, f);
      // omit check
      std::fclose(f);
      delete [] intArray;
   }
}
```

Listing 4

```
==534== Syscall param write(buf) points to
uninitialised byte(s)
==534==    at 0x148C82: write$NOCANCEL (in /usr/
lib/libSystem.B.dylib)
==534==    by 0x148BFC: _swrite (in /usr/lib/
libSystem.B.dylib)
==534==    by 0x10AC16: __sfvwrite (in /usr/lib/
libSystem.B.dylib)
==534==    by 0x15C3C4: fwrite (in /usr/lib/
libSystem.B.dylib)
==534==    by 0x100000E97: main (syscall.cpp:15)
==534==  Address 0x1000040e4 is 4 bytes inside a
block of size 12 alloc'd
==534==    at 0xD6D9: malloc
(vg_replace_malloc.c:266)
==534==    by 0x64F04: operator new(unsigned long)
(in /usr/lib/libstdc++.6.0.9.dylib)
==534==    by 0x64F96: operator new[](unsigned
long) (in /usr/lib/libstdc++.6.0.9.dylib)
==534==    by 0x100000E5E: main (syscall.cpp:11)
```

Figure 4b

```
==468== Syscall param write(buf) points to
uninitialised byte(s)
==468==    at 0x148C82: write$NOCANCEL (in /usr/
lib/libSystem.B.dylib)
==468==    by 0x148BFC: _swrite (in /usr/lib/
libSystem.B.dylib)
==468==    by 0x148B41: __sflush (in /usr/lib/
libSystem.B.dylib)
==468==    by 0x14859A: fclose (in /usr/lib/
libSystem.B.dylib)
==468==    by 0x100000EB6: main (syscall.cpp:16)
==468==  Address 0x100004134 is 4 bytes inside a
block of size 4,096 alloc'd
==468==    at 0xD6D9: malloc
(vg_replace_malloc.c:266)
==468==    by 0x1489ED: __smakebuf (in /usr/lib/
libSystem.B.dylib)
==468==    by 0x148959: __swsetup (in /usr/lib/
libSystem.B.dylib)
==468==    by 0x10ABC8: __sfvwrite (in /usr/lib/
libSystem.B.dylib)
==468==    by 0x15C3C4: fwrite (in /usr/lib/
libSystem.B.dylib)
==468==    by 0x100000EA9: main (syscall.cpp:14)
```

Figure 4a

```
// ifree.cpp
void func()
{
  int stackArray[10];
  delete stackArray; // not even array delete
}

int main()
{
  func();
}
```

Listing 5

```
==72595== Invalid free() / delete / delete[]
==72595==    at 0x1004DDC: operator delete(void*)
(in /usr/local/lib/valgrind/vgpreload_memcheck-
amd64-freebsd.so)
==72595==    by 0x400680: func() (ifree.cpp:4)
==72595==    by 0x400698: main (ifree.cpp:9)
==72595==  Address 0x7ff000240 is on thread 1's
stack
```

Figure 5

As a rule, you're better off fixing your errors than hiding them in a suppression file

```
// ifree2.cpp
void func()
{
  int *heapArray = new int[10];
  delete heapArray; // not even array delete
}

int main()
{
  func();
}
```

Listing 6

```
==72950== Mismatched free() / delete / delete []
==72950==    at 0x1004DDC: operator delete(void*)
(in /usr/local/lib/valgrind/vgpreload_memcheck-
amd64-freebsd.so)
==72950==    by 0x4006DE: func() (ifree2.cpp:4)
==72950==    by 0x4006F8: main (ifree2.cpp:9)
==72950==  Address 0x1c8f040 is 0 bytes inside a
block of size 40 alloc'd
==72950==    at 0x1005BB7: operator
new[](unsigned long) (in /usr/local/lib/valgrind/
vgpreload_memcheck-amd64-freebsd.so)
==72950==    by 0x4006D1: func() (ifree2.cpp:3)
==72950==    by 0x4006F8: main (ifree2.cpp:9)
```

Figure 6

Source/destination overlap

The usual example of this is a **std::strcpy** where the source and destination point within the same char array (Listing 7).

Valgrind's output is shown in Figure 7.

The standard solution to this sort of problem is to use **std::memmove** instead of **std::strcpy** or **std::memcpy**.

Memory leaks

This is the largest of the memcheck error types. Memcheck can detect 3 different types of 'leak'. The definite leak, where the pointer has gone out of scope and the memory is leaked. Next there are possible leaks. This is where there are no longer pointers to the start of the allocated memory, but there are still pointers within the allocated memory. Finally there is still-in-use memory, where both the memory and the pointer to it still exist.

If you use a memory manager (e.g., a pool allocator), then this can complicate leak detection. For instance, if your application has a pool allocator that news blocks of 100MBytes, uses an overloaded operator new that uses this pool, optionally does some overloaded deletes, and then when it terminates deletes all of the pool blocks, memcheck won't be able to detect any leaks, even though your application may be leaking your pool memory in the sense that it wasn't deleted and made available for reuse before the pool was deleted. Furthermore, if you are using an allocator that allocates blocks that are handled as **{length:memory[:guard]}**, so that the pointer obtained by new is adjusted after setting the length, then you're likely to get possible leaks detected rather than definite leaks.

There are two things that you can do in this case. One is to have a special build, where you compile with a macro like **-DDEFAULT_NEW** which disables the memory allocator and uses the standard allocators. Obviously having two sets of code is not ideal, and this will be a maintenance overhead. The alternative is to include the `valgrind.h` header and use the Valgrind **MEMPOOL** macros. More on that in a later article.

A very short example of this in Listing 8. Valgrind's output for this is in Figure 8.

Suppressing errors

Memcheck will use a default suppression file that was generated on the machine where Valgrind was built. This will suppress 'well known' (and hopefully harmless) errors in libc and X11. You can also use user-defined suppression files with the option:

```
-- memcheck:suppressions=<suppression file>
```

This can be used more than once. I would advise that you do this only for harmless errors or errors in third party libraries that you can't fix. As a rule, you're better off fixing your errors than hiding them in a suppression file.

```
// overlap.cpp
#include <cstring>
#include <iostream>

int main()
{
   char *str = new char[100];
   std::sprintf(str, "Hello, world!");
   std::strcpy(str, str+2);
   std::cout << "str " << str << "\n";
   delete [] str;
}
```

Listing 7

```
==74324== Source and destination overlap in
strcpy(0x1c90040, 0x1c90042)
==74324==    at 0x1009A61: strcpy (in /usr/local/
lib/valgrind/vgpreload_memcheck-amd64-freebsd.so)
==74324==    by 0x400BE9: main (overlap.cpp:8)
```

Figure 7

I would recommend that you change this and try to make it something unique

```
// leak.cpp
int main()
{
   int *leak = new int(42);
}
```

Listing 8

```
==76314== 4 bytes in 1 blocks are definitely lost
in loss record 1 of 1
==76314==    at 0x1005F79: operator new(unsigned
long) (in /usr/local/lib/valgrind/
vgpreload_memcheck-amd64-freebsd.so)
==76314==    by 0x400681: main (leak.cpp:3)
```

Figure 8

You can use `--memcheck:gen- suppressions=all` to generate suppression stacks in output log file, which look like this

```
{
   <insert_a_suppression_name_here>
   Memcheck:Leak
   fun:_Znwm
   fun:main
}
```

The opening and closing braces delimit the error callstack. The first line is intended for use as a comment. I would recommend that you change this and try to make it something unique. If you use `valgrind -v`, then in the summary, Valgrind will list all of the suppressions that it used with their comments. This can be used to see which of your suppressions are being used, which allows you to clean out your suppressions files from time to time.

The second line gives the type of error.

The third to last lines are the callstack. Each line has one of the following forms

- `fun`: function name for unstripped functions.
- `obj`: name of library for stripped functions.
- ...: wildcard for any depth. This can be useful for recursive functions that would otherwise need N different suppressions for N possible depths of recursion.

You can use `*` wildcard to make suppressions more generic. For instance, if you want to use the same suppression files on both 32bit and 64bit Linux, then instead of having two separate suppressions for each platform, one with `/opt/mypkg/lib` and the other with `/opt/mypkg/lib64`, you could have just one suppression with `/opt/mypkg/lib*`.

You may want to reduce the amount of callstack that appears in the suppression. This can reduce the number of suppressions that you need (which is OK if they are all the same issue). Don't overdo it though, you don't want to suppress genuine errors.

Errors that memcheck does not detect

Lastly but not least, there are a few types of memory errors that memcheck does not detect.

Reading or writing beyond arrays that are global or on the stack, for instance

```
int x[10]; // local, global or static
x[10] = 1;
```

Try using exp-sgcheck for this sort of error.

Now that we've covered the basics of memcheck, in the next article we'll look at more advanced techniques. ■