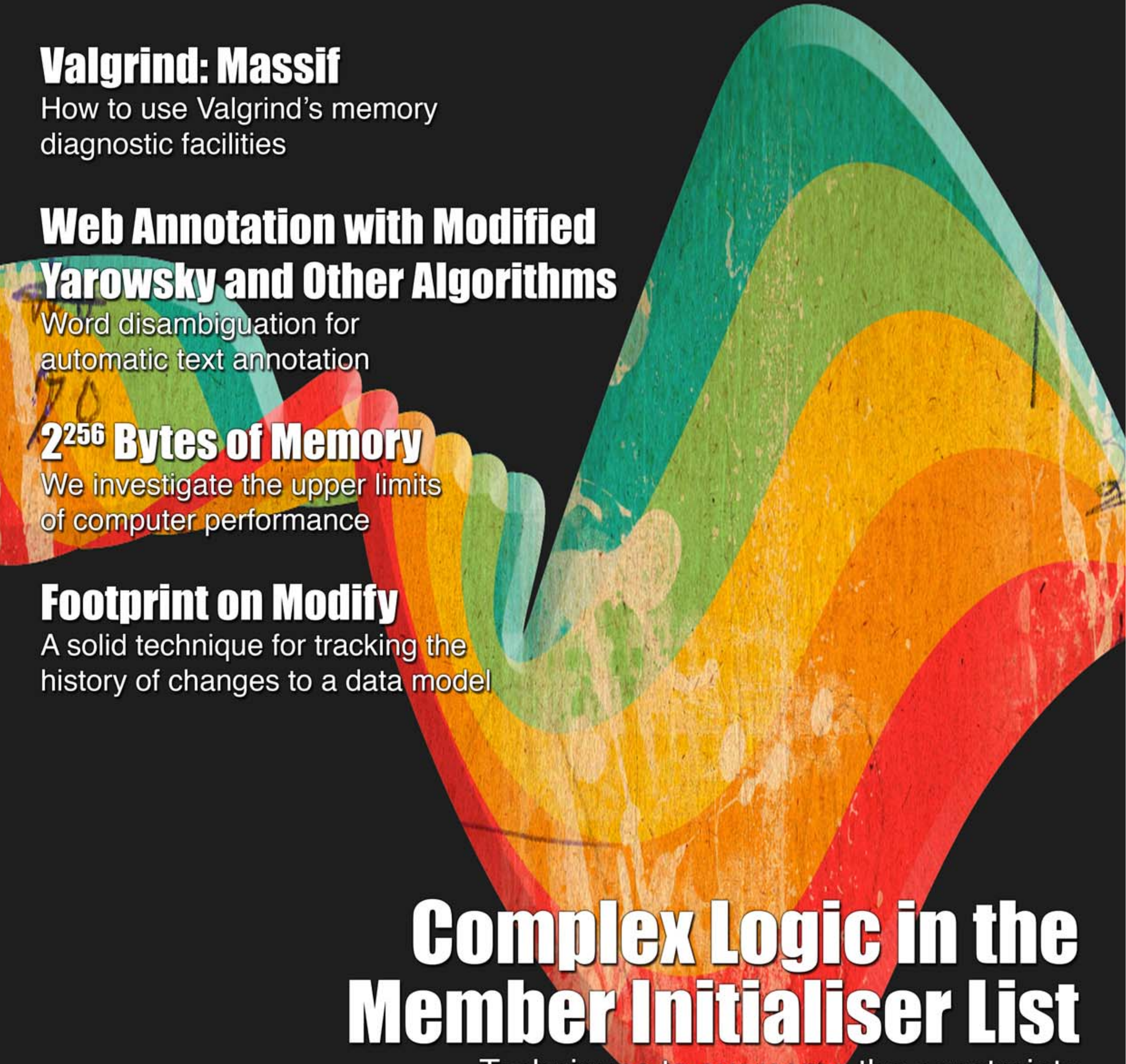
# overload 112

DECEMBER 2012 £3



## Valgrind: Massif
How to use Valgrind's memory diagnostic facilities

## Web Annotation with Modified Yarowsky and Other Algorithms
Word disambiguation for automatic text annotation

## $2^{256}$ Bytes of Memory
We investigate the upper limits of computer performance

## Footprint on Modify
A solid technique for tracking the history of changes to a data model

## Complex Logic in the Member Initialiser List
Techniques to overcome the constraints of C++ member initialiser list syntax

A magazine of ACCU

ISSN: 1354-3172

**OVERLOAD 112**

**December 2012**

ISSN 1354-3172

**Editor**

Frances Buontempo
overload@accu.org

**Advisors**

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

**Advertising enquiries**

ads@accu.org

**Cover art and design**

Pete Goodliffe
pete@goodliffe.net

**Copy deadlines**

All articles intended for publication in Overload 113 should be submitted by 1st January 2013 and for Overload 114 by 1st March 2013.

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

## The ACCU

The 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.

# Originally, Overload Didn't Have an Editorial

Frances Buontempo considers history, predictions about the future and how to shirk off writing an editorial.

Sometimes it's good to look back over history in order to find inspiration, and to see how things have changed over time. *Overload* started with epilogues rather than introductory editorials, and occasional letters to the editor. If we were to revert to this format, that would let me off the hook. If any readers wish to send a letter to the editor, Overload@accu.org, please feel free. Several of the epilogues had brave prediction or questions about the future. How would namespaces work in C++? C++ is here to stay. Is there anybody brave enough to dismiss OO-COBOL? The first editorial appeared in April 1995. This considered the future directions of *Overload* and asked 'When are you lot going to stop messing around with the C++ standard?' [Overload07] Seventeen years later, it seems the answer might be never. Sean Corfield also asked how many of the readers had email, stating 'Please use email, where possible, for submissions – I am allergic to paper' [Overload07]. As I explained last time [Overload111], I am becoming allergic to emails, but articles in an electronic format are certainly easier to deal with than paper ones. How times change. If any readers don't use email, please write in and tell us what you do with all your spare time.

Eventually paper crumbles away, old documents and code, on paper tape, punch cards, floppy disks and various other types of hard copies become unreadable. Either the medium itself decays, or we lose the means to read or understand the information. Taking a long view, I was struck by a BBC news article about proto-Elamite tablets [Proto-Elamite]; very old clay tablets with scribbles on. Though the clay tablets themselves have survived 5000 years, no-one knows what the inscriptions actually mean. It is suspected they might be some form of early accountancy, as many surviving writings from a similar era and area seems to be. I wonder if one day, no-one will be able to read a pdf or a Sage account. We shall see. Rather than writing our records on clay tablets, nowadays many people choose to write blogs on the internet. I suspect the internet will not disappear for a long time, but I wonder if we will lose these glimpses of the everyday at some point. This might lead to another 'Dark Age'. Wikipedia describes the Dark Age as 'a period of intellectual darkness and economic regression that supposedly occurred in Europe' [Wikipedia]. The main reason seems to be few written records have survived from the time. Perhaps people in Europe were writing their own equivalent of blogs, not in the lingua franca of the time, Latin, and these have decayed away. Imagine that one thousand years from now, a historian tries to gather together evidence of how we live today. Will they find copies of *Overload* to use as a source? Or a blog? Of course, I am not suggesting I would rather you send articles in on clay tablets, or carved them into hillsides. I just wonder what now might look like, from the future.

Aside from the problem of using perishable storage media, the proto-Elamite tablets show the problem of communication. The Rosetta stone was a lucky find that allowed translation between Greek and Egyptian hieroglyphs [Rosetta]. For the proto-Elamite tablets, without a triangulation point, we may never know what they say. Rosetta code [RosettaCode] plays on the name to provide a rich resource of code challenges implemented in a variety of programming languages, allowing comparison and potentially is a great learning resource. They claim to have a total of 481 different programming languages, which is phenomenal. I wonder if they've missed any. How many different programming languages are there? I wonder how many different human languages there are. Recently I have been reading my bible, starting at Genesis and have just reached the story of the tower of Babel. It suggests originally 'The whole world had one language and a common speech,' [Genesis 11] but God confuses peoples' language so they no longer understand one another. Certainly, if you are confronted by a program in a language you don't know, if may take a while to figure out how it works. Nonetheless, it is still possible to be bemused by a program written in a language you already know. My colleagues have recently written a tool to reverse engineer our config files, though that is another story. We have seen constant debates and considerations of the importance of naming variables and functions sensibly, in order to communicate our intent clearly. At the heart of this is avoiding the confusion of Babel. In August 2008, Ric Parkin's editorial suggested, developing software is not so much a technical problem as a communication one. [Overload86].

Technology has attempted to make in-roads in to automatic translation between languages to help communication. Various online translators exist, and seem to be improving. I have noticed a few recent news stories about live speech translation, not done by people, but by machines. Specifically, Google Translate has branched out and might now try to translate your spoken words live, presumably allowing you to communicate with colleagues distributed across the world over the phone even if one of you only knows English and the other only Japanese [LiveSpeech]. Had the Dark Ages never happened, and we all still spoke Latin, this wouldn't be necessary. The live speech has grown from Google's machine translation technology, which is a computer-driven pattern recognition algorithm, nudged by feedback from users. We shall see if the live translation takes hold. Technologies come and go. Recently, we have seen the death of Ceefax. Started in 1974, before the internet, it gave instant news, TV listing and weather forecasts on a television set capable of reading and displaying the information feed. The Ceefax pages were created manually – people monitored the incoming information and produced metres of punched tape to upload, after being carried up several flights of stairs to the 'central apparatus room'. We are told, 'It proved an invaluable service for the editor who used to alert his wife that he was

**Frances Buontempo** has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 12 years professionally, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

about to leave Television Centre on his way home by using a back page on Ceefax. [Ceefax]

Watching previous technologies starting to grow and the predictions sparked by these is fascinating. I enjoy reading sci-fi, though I do wonder why these stories still tend to insist on the idea of flying cars. Sometimes such auguries are limited by a lack of imagination, and constrained by the current. As an antidote to ridiculous means of transportation, I have been reading *The Last Man* [Shelly]. Futuristically set at the end of the 21st century, it is free from flying cars. People still use horseback or coach to travel, the English monarchy has only just ceased, and wars are still fought with cannons and swords. The characters and story are played through with more conviction than many sci-fi books though. Heartily recommended for delicious gothic doom and cheer.

It seems that predicting the future is hazardous. "Prediction is difficult, especially about the future." As either Neils Bohr or Yogi Berra once said: no-one seems to be sure who [BohrYogi]. See, predicting the past is hard enough. Would be traders will spend hours backtesting a new strategy, trying to see if they could make money from the historical data they used to form the strategy in the first place. And even getting the present right is difficult. For example, 'nowcasting' the weather is much more difficult than just looking out of the window. "These predictions are very expensive and not available to the public " [Nowcasting1] and, I believe, frequently incorrect. To be fair, nowcasting isn't trying to state what the weather is up to now, but rather what it will be doing in the very short-range, which does require accurate data on what is happening now, to predict rainfall, paths of tornadoes and so on [Nowcasting2]. The met office gathers a huge amount of data and does some serious high performance computing to analyse it, producing thousands of forecasts a day. A variety of ways of trying to elucidate sense from data about now are constantly springing up. Twitter will tell you which subjects are currently trending, but not to be out-done 'Massachusetts Institute of Technology (MIT) associate professor Devavrat Shah has announced the creation of a new algorithm that can predict Twitter trends hours in advance. ' [MIT] That will be hours in advance of twitter noticing, I presume, rather than the tweets actually being tweeted. That really would be something.

Sci-fi stories, along with letters to the editor, epilogues, and occasional stabs at editorials are all attempts to step back, and take stock of the now. They can draw on history, notice current trends, and try to make sense of it all. This is a time consuming activity, and as we have seen is increasingly being opened up to geeks armed with machine-learning algorithms. The next logical step is for the machines to write editorials for us. I have observed some automatic article generators of late. They seem to have started with an automatic Computer Science paper generator, [SCIGen] and sprouted new incarnations, such as a mathematics paper generator [Mathgen]. Some of these papers have been submitted and accepted by peer-reviewed journals [ThatsMaths]. A variant of this code this would get me off the hook. That does not let you, dear reader, off the hook. If you do feel the urge to submit an automatically generated paper, feel free, but rest assured, it will be read by our human review team, and we might just notice. Mind you, if it's interesting, that is fine. I must stop for now, to brush up on my perl skills, in order to hack around the code from SCIGen and Mathgen, to get off having to write an editorial for next time.

## References

[BohrYogi] http://www.peterpatau.com/2006/12/bohr-leads-berra-but-yogi-closing-gap.html

[Ceefax] http://www.bbc.co.uk/news/magazine-20032531

[Genesis 11] http://www.biblegateway.com/passage/?search=Genesis%2011%20&version=NIV

[LiveSpeech] http://www.wired.com/gadgetlab/2011/01/google-translate-adds-live-speech-translation-to-android/

[Mathgen] http://thatsmathematics.com/mathgen/

[MIT] http://www.v3.co.uk/v3-uk/the-frontline-blog/2221958/mit-professor-invents-algorithm-that-can-predict-twitter-trends

[Nowcasting1] http://www.nooly.com/technology/728-2/

[Nowcasting2] http://www.metoffice.gov.uk/learning/science/hours-ahead/nowcasting

[Overload07] http://accu.org/var/uploads/journals/Overload07.pdf

[Overload86] http://accu.org/var/uploads/journals/overload86.pdf

[Overload111] http://accu.org/var/uploads/journals/Overload111.pdf

[Proto-Elamite] http://www.bbc.co.uk/news/business-19964786

[Rosetta] http://en.wikipedia.org/wiki/Rosetta_Stone

[RosettaCode] http://rosettacode.org/wiki/Rosetta_Code

[SCIGen] http://pdos.csail.mit.edu/scigen/

[Shelly] *The Last Man*, Mary Shelley, 1826.

[ThatsMaths] http://thatsmathematics.com/blog/archives/102

[Wikipedia] http://en.wikipedia.org/wiki/Dark_Ages

# Web Annotation with Modified-Yarowsky and Other Algorithms

Annotating text automatically requires word disambiguation. Silas Brown introduces the Yarowsky algorithm to help.

In 1997 I wrote a CGI script in C++ to perform Web 'mediation'. It took the address of a Web page, fetched it, modified the markup so as to simplify complex layouts (so they work better in large print) and to overcome various other disability-related limitations of early Web browsers, and sent the result to the user's browser. Additionally, all links on the page were changed to point back through the system, so the user could surf away on this modified version of the Web without needing to be able to set the proxy settings on the computers they used.

This 'Web Access Gateway' was not the first or the only effort at Web adaptation, but for a time it was, I think, the most comprehensive. For some years it was run on ACCU's server, in order not only to ensure the accessibility of ACCU's site but also as a service to others; this resulted in ACCU being cited in an ACM publication courtesy of IBM's blind researcher Chieko Asakawa [Asakawa]. It also was and still is run by organizations interested in displaying East Asian characters on devices that lack the fonts [EDRDG], since it has a function to replace characters by small bitmap images which are served by a small, single-threaded `select()`-based, HTTP 1.0 server and some public domain fonts.

The Access Gateway became less useful with the advent of Web 2.0 and Javascript-heavy sites. I did try to handle basic navigation-related scripts, but not serious AJAX. However, by this time desktop browsers were improving, and user stylesheets [ACG] became more appropriate than mediators, although user stylesheets still can't solve everything. There was also a demand for mediators to do 'content adaptation' for mobile phone browsers (especially the lower-end, non-smartphone variety), and indeed at one time I (somehow) obtained a part-time job on the development team of a custom server for mobile operators to run [Openwave]. This one was built around the SpiderMonkey Javascript interpreter so it wouldn't have any trouble with AJAX, although we still had to implement the DOM and that was a hard game of 'keep up with the browsers'. Opera Mini had it easier because they already had some browser code. (They also write their own user clients instead of making do with whatever's on the phone. I wish they'd allow larger fonts though.)

Recently I wanted to help a group of smartphone-using friends to access a Chinese-language reference site. I wished to add automatic 'pinyin' pronunciation aids to the site for them. The site was AJAX-heavy and I had not kept any of the Openwave code, but it occurred to me that writing a mediator with modern technologies can be done in a much simpler way. My Openwave no-compete contract has long since expired and I felt free to break out the modern tools and build a 21st-century mediator. It's quite exciting to be able to reproduce in just one or two afternoons of coding something that previously needed many years of development.

**Silas S. Brown** is a partially-sighted Computer Science post-doc in Cambridge who currently works in part-time assistant tuition and part-time for a startup, as well as developing language-related software in spare time since events in Cambridge have led him to acquire fluent Chinese. He has been an ACCU member since 1994. Silas can be contacted at ssb22@cam.ac.uk

## Modern server tools

With apologies to other programming languages, I coded the server in Python. Python makes it quick to try things out, and has many Web-related modules in its standard library. Moreover, it has the Tornado web framework [Tornado], which allowed me to make the entire server (not just the bitmap-serving part) a single-threaded, super-scalable affair with support for HTTP 1.1 pipelining and other goodies thrown in for free. Then there is the Python Imaging Library [PIL] which allowed me to do the character-rendering part in Freetype with better fonts (not to mention more flexible rendering options). For good measure, I added an option to call external tools to re-code MP3 audio to reduce the download size, and to add a text-only option to PDF links. (Both of these can be useful for low-speed mobile links in rural areas.)

How did I call an external processing tool from a single-threaded Tornado process without holding up the other requests? Well it turns out that Tornado can cope with your use of other threads so long as the completion callback is called from the Tornado thread, which can be arranged by calling `IOLoop.instance().add_callback()`. For more details please see my code [Adjuster].

What about handling all the AJAX and ensuring that all links etc are redirected back through the system? This time round, I didn't have to do nearly so much. As the server is Tornado-based and handles all requests to its port (rather than being CGI-based and handling only URIs that start with a specific path), it is possible to mediate a site's URIs without actually changing any of those URIs except for the domain part. Most Javascript code doesn't care what domain it's running on, and it's extremely rare to find a script that would be broken by straightforward changes to any domain names mentioned in its source. Therefore, as long as the browser itself is sufficiently capable, it is not necessary to run Javascript on the server just to make redirection work. If you have a wildcard domain pointing to your server (i.e. it is possible to put arbitrary text in front of your domain name and it will still resolve to your server), you can mediate many sites in this way. There are a few details to get right, such as cookie handling, but it's nowhere near as complex as using a script interpreter.

## Text annotation

For adding the pronunciation aids to the site it was necessary to make a text annotator. In order to make it as easy as possible for others to use their own annotators instead, I kept this in a completely separate process that takes textual phrases on standard input and emits the annotated versions to standard output; for efficiency it is called with all phrases at once, and the results are put back into the HTML or JSON in their appropriate places by the mediator. Therefore the authors of text annotators do not need to worry about HTML parsing, although they still have the option of including HTML in its output. For example, with appropriate CSS styling, HTML's Ruby markup can be used to place annotations over the base text (see the source code to my page on Xu Zhimo's poem [Xu] for one way to do this).

The simplest approach to annotating text is to apply a set of search-and-replace criteria, perhaps driven by a dictionary, but problems can arise

Yarowsky's algorithm for word sense disambiguation used **contextual cues** around a word to try to **guess which meaning it has**

when there is more than one way to match a section of text to the search strings, especially in languages that do not use spaces and there is more than one way to interpret where the word boundaries are. The lexer generator Flex [Flex], which might be useful for 'knocking up' small annotators that don't need more rules than flex can accommodate, always applies the longest possible match from the current position, which might be adequate in many sentences but is not always.

As a result of my being allowed access to its C source, Wenlin software for learning Chinese [Wenlin] now has a function for guessing the most likely word boundaries and readings of Chinese texts, by comparing the resulting word lengths, word usage frequencies according to Wenlin's hand-checked data from the Beijing Language Institute, and some Chinese-specific 'rules of thumb' I added by trial and error. The resulting annotations are generally good (better than that produced by the tools of Google *et al*), but I do still find that some of the obscure multi-word phrases I add to my user dictionary are not for keeping track of any definitions or notes so much as for ensuring that Wenlin gets the boundaries and readings right in odd cases.

## Annotator generator

If you are fortunate enough to have a large collection of high-quality, manually proof-read, example annotations in a computer-readable format, then it ought to be possible to use this data to 'train' a system to annotate new text, saving yourself the trouble of manually editing large numbers of rules and exceptions.

My first attempt at an examples-driven 'annotator generator' simply considered every possible consecutive-words subset of a phrase (word 1, word 2, words 1 to 2, word 3, words 2 to 3, words 1 to 3, etc; it's a reasonable assumption that annotated examples will have word boundaries), and for each case tested to see if the annotation given to that sequence of words is always the same whenever that sequence of words occurs anywhere else in the examples. If so, it is a candidate for a rule, and rules are further restricted to not overlap with each other (this means we don't have to deal with exceptions); the code takes the shortest non-overlapping rules that cover as much as possible of the examples, and turns them into C code consisting of many nested one-byte-at-a-time **`switch()`** constructs and function calls. (When generating code automatically, I prefer C over C++ if reasonable, because C compiles faster when the code is large.) Python was good for prototyping the generator, because it has many built-in functions to manipulate strings and lists of strings, count occurrences of an annotation in a text, etc, and it also has the **`yield`** keyword that can be used to make a kind of 'lazy list' whose next element is computed only when needed (if a function **`yield`**'s values, this creates an iterator over them which returns control to the function when the next value is asked for) so you can stop when enough rules have been accepted to cover the whole of an example phrase. The generator didn't have to run particularly quickly, as long as it could produce a fast C program within in a day or so.

The problem with this approach is that restricting the generator to rules that have no exceptions or overlaps will typically result in rules that are longer than necessary (i.e. require a longer exact match with an example phrase) and that do not achieve 100% coverage of the examples (i.e. would not be able to reproduce all the example annotations if given the unannotated example text). This may be sufficient if you have a reasonable backup annotator to deal with any text that the examples-driven annotator missed, but it does seem like an under-utilisation of the information in the examples. We can however do better, especially if we break away from the idea of matching continuous strings of text.

## Yarowsky-like algorithm

Yarowsky's algorithm for word sense disambiguation [Yarowsky] used contextual cues around a word (not necessarily immediately adjacent to it) to try to guess which meaning it has (Yarowsky's example used the English word 'plant', associating it with either 'plant life' or 'manufacturing plant', and using other words in the vicinity to guess which one was meant). Figure 1 shows how it gradually builds up rules to disambiguate 'plant' in phrases, adding a rule to spot 'animal' nearby. Although Yarowsky was originally talking about meaning, there's no reason why it can't be applied to pronunciation (which is often related to meaning) or to arbitrary other annotations, and there's no reason why it shouldn't work in a language that does not use word boundaries if we modify it to check for characters instead of words and use them to judge which character-based search/replace rules are appropriate and therefore how to decide word boundaries etc.

Yarowsky started with manually-chosen 'seed collocations'. With a fully-annotated set of examples it is possible to automatically list the candidate seed collocations along with a measure of how many correct and incorrect applications of the rule each would result in. (Yarowsky also suggested analysing the exact collocational relationships of the words, such as whether they are in a predicate-argument relationship, but this enhancement is hard to do for arbitrary languages.)

It is then possible to find additional collocations by considering an untagged (unannotated) text. The seed collocations are used to decide the sense of some of the words in that text, and, assuming these decisions to be correct, the system checks what other words are also found near them which might be used as new indicators. This process can be repeated until many other possible indicators have been found. However, if enough annotated examples have been provided it might be possible to skip this step and just use the seed collocations; this has the advantage of applying rules only when we have a greater degree of certainty that we can do so (an 'if in doubt, leave it out' annotation philosophy).

My **`yarowsky_indicators()`** function [Generator] takes the simplified approach of looking only for seed collocations of one or more complete Unicode characters within a fixed number of bytes of the end of the word match, prioritising the ones that are short and that cover more instances of the word, completely excluding any that would give false positives, and stopping as soon as all examples have been covered. Keeping to a fixed number of bytes around the end of the match makes it easier for the C parser to work from a local buffer. The algorithm to find the Yarowsky indicators is shown in Listing 1.

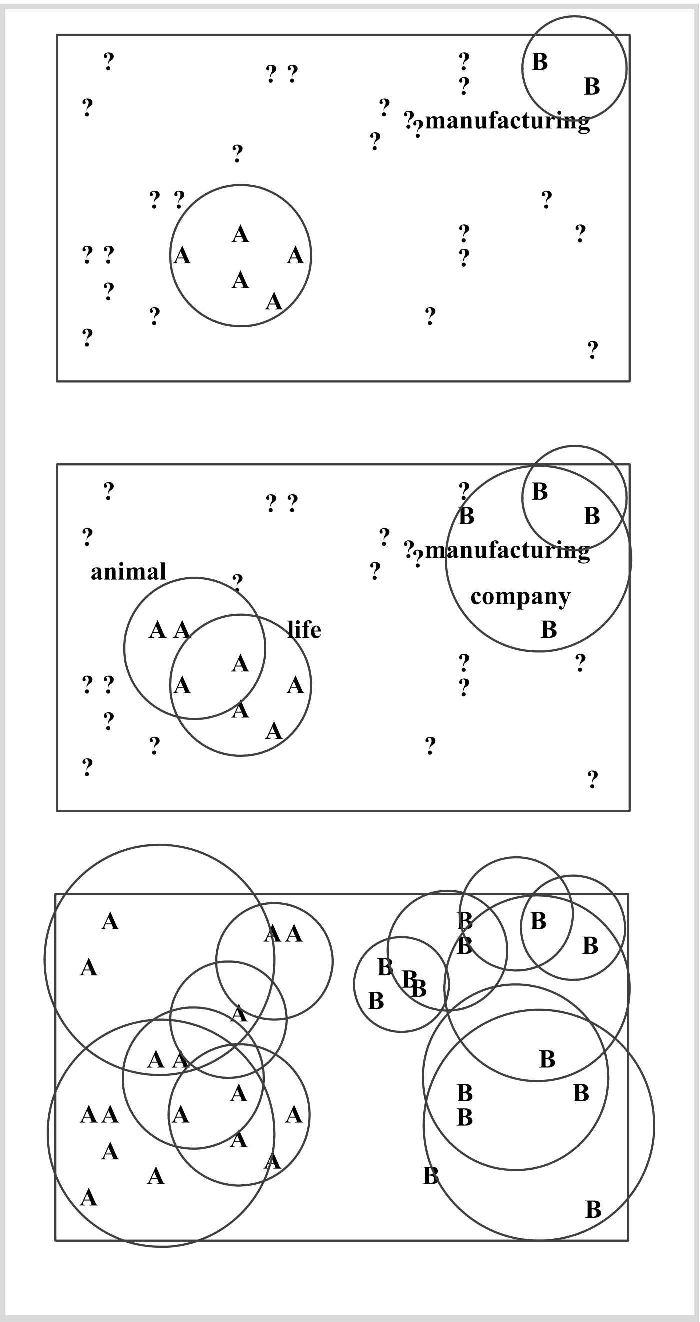


Figure 1 (taken from Yarowsky)

A remaining problem is that it often needs to find too many collocations to make up for the fact that the C parser's handling of rule overlaps is so primitive, greedily matching the longest rule every time. If the parser had something like Wenlin's frequency-driven approach then it might not need to rely on collocations so much, although collocations would still be useful sometimes. The 'collocations' found by `yarowsky_indicators()` are often not real collocations at all, but just strings that happen to be nearby in the example texts; this might cause strange matching behaviour in other texts. I hope to find ways to improve this situation in future. ■

## References

[ACG] Accessibility CSS Generator, http://people.ds.cam.ac.uk/ssb22/css/

[Adjuster] Web Adjuster, http://people.ds.cam.ac.uk/ssb22/adjuster/

[Asakawa] Hironobu Takagi and Chieko Asakawa (IBM Japan). Transcoding proxy for nonvisual web access. ASSETS 2000. http://dl.acm.org/citation.cfm?id=354371 (click on References and check number 12)

[EDRDG] www.csse.monash.edu.au/~jwb/jviewer.html (the actual server is on arakawa.edrdg.org)

[Flex] http://flex.sourceforge.net

[Generator] Annotator Generator, http://people.ds.cam.ac.uk/ssb22/adjuster/annogen.html

[Openwave] http://www.openwave.com/solutions/traffic_mediation/web_adapter/index.html

[PIL] Python Imaging Library, www.pythonware.com/products/pil

[Tornado] www.tornadoweb.org

[Wenlin] www.wenlin.com

[Xu] Xu Zhimo's poem http://people.ds.cam.ac.uk/ssb22/zhimo.html

[Yarowsky] www.cl.cam.ac.uk/teaching/1112/NLP/lectures.pdf pages 55–57

```
# This code will run several times faster if it
# has a dictionary that maps corpus string indices
# onto values of len(remove_annotations(c)) where
# c is the corpus up to that index.
def yarowsky_indicators(word_with_annotations,
            corpus_with_annotations,
            corpus_without_annotations):

  # returns True if the given word's annotation is
  # the majority sense and can be made default, or
  # in minority senses lists the context
  # indicators. Variation of first stage of
  # Yarowsky's algorithm.
  word_without_annotations = \
    remove_annotations(word_with_annotations)

  # First, find positions in
  # corpus_without_annotations which correspond to
  # where word_with_annotations occurs in
  # corpus_with_annotations.
  # Put this into the list okStarts.
  lastS = lenSoFar = 0
  okStarts = []
  for s in \
    re.finditer(re.escape(word_with_annotations),
          corpus_with_annotations):
    s = s.start()
    lenSoFar += len(remove_annotations( \
        corpus_with_annotations[lastS:s]))
    lastS = s
    assert corpus_without_annotations[ \
      lenSoFar:lenSoFar
      + len(word_without_annotations)] \
      == word_without_annotations
    okStarts.append(lenSoFar)

  # Now check for any OTHER matches in
  # corpus_without_annotations, and put them
  # into badStarts.
  okStarts = set(okStarts)
  badStarts = set(x.start() for x in
  re.finditer(re.escape(word_without_annotations),
          corpus_without_annotations)
          if not x.start() in okStarts)

  if not badStarts:
    return True  # this annotation has no false
                 # positives so make it default

  # Some of the badStarts can be ignored on the
  # grounds that they should be picked up by
  # other rules first: any where the match does
```

Listing 1

```
# not start at the start of an annotation
# block (the rule matching the block starting
# earlier should get there first), and any
# where it starts at the start of a block that
# is longer than itself (a longest-first
# ordering should take care of this).  So keep
# only the ones where it starts at the start
# of a word and that word is no longer
# than len(word_without_annotations).
lastS = lenSoFar = 0
reallyBadStarts = []
for s in re.finditer(re.escape(markupStart
           + word_without_annotations[0])
           + '.*?'
           + re.escape(markupMid),
           corpus_with_annotations):
  (s, e) = (s.start(), s.end())
  if e - s > len(markupStart
           + word_without_annotations
           + markupEnd):
    continue  # this word is too long
              # (see comment above)
  lenSoFar += len(remove_annotations( \
      corpus_with_annotations[lastS:s]))
  lastS = s
  if lenSoFar in badStarts:
    reallyBadStarts.append(lenSoFar)
badStarts = reallyBadStarts

if not badStarts:
  return True
  # this annotation has no effective false
  # positives, so make it default

if len(okStarts) > len(badStarts):

  # This may be a majority sense. But be
  # careful. If we're looking at a possible
  # annotation of "AB", it's not guaranteed
  # that text "ABC" will use it - this might
  # need to be split into A + BC (not using the
  # AB annotation). If we make
  # word_with_annotations the default for "AB",
  # then it will be harder to watch out for
  # cases like A + BC later.  In this case it's
  # better NOT to make it default but to
  # provide Yarowsky collocation indicators for
  # it.
  if len(word_without_annotations) == 1:
    # should be safe
    return True

  if all(x.end() - x.start()
       == len(markupStart
       + word_without_annotations)
       for x in
       re.finditer(re.escape(markupStart)
       + (re.escape(markupMid) + '.*?'
       + re.escape(markupStart)). \
         join(re.escape(c)
       for c in
       list(word_without_annotations)),
       corpus_with_annotations)):
    return True
# If we haven't returned yet,
# word_with_annotations cannot be the "default"
# sense, and we need Yarowsky collocations for
# it.
```

Listing 1 (cont'd)

```
omitStr = chr(1).join(bytesAround(s) for s in
    badStarts)
okStrs = [bytesAround(s) for s in okStarts]
covered = [False] * len(okStrs)
ret = []

# unique_substrings is a generator function
# that iterates over unique substrings of
# texts, in increasing length, with equal
# lengths sorted by highest score returned by
# valueFunc, and omitting any where omitFunc is
# true
for indicatorStr in \
  unique_substrings(texts=okStrs,
    omitFunc=lambda txt: txt in omitStr,
    valueFunc=lambda txt: sum(1 for s in
    okStrs if txt in s)):

  covered_changed = False
  for i in xrange(len(okStrs)):
    if not covered[i] and indicatorStr \
      in okStrs[i]:
      covered[i] = covered_changed = \
        True
    if covered_hanged:
      ret.append(indicatorStr)
    if all(covered):
      break

return ret
```

Listing 1 (cont'd)

cqf.com

CERTIFICATE IN QUANTITATIVE FINANCE
CQF

**Expand Your Mind and Career**

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

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

ENGINEERED FOR THE FINANCIAL MARKETS

# Complex Logic in the Member Initialiser List

The syntactic form of the member initialiser list restricts the logic that it contains. Cassio Neri presents some techniques to overcome these constraints.

In C++, during a constructor call, before execution gets into its body all *subobjects* – base classes and non-static data members – of the class are initialised. (In C++11, this rule has an exception which we shall exploit later.) The *member initialiser list* (MIL) lets the programmer customise this initialisation. A subobject is initialised from a parenthesised[1] list of expressions that follows its identifier in the MIL. The MIL of **bar**'s constructor is emphasised in Listing 1.

```
class base {
  ...
public:
  base(double b);
};

class foo {
  ...
public:
  foo(double f1, double f2);
};

class bar : public base {
  const  double x_, y_;
  foo&   r_;
  foo    f_;
  double d_;
  ...
public:
  bar(double d, foo& r1, foo& r2);
};

bar::bar(double d, foo& r1, foo& r2)
: base(d * d), x_(cos(d * d)), y_(sin(d * d)),
  r_(d > 0.0 ? r1 : r2), f_(exp(d), -exp(d))
{
  d_ = d;
}
```

Listing 1

Most often the MIL forwards the arguments to the subobject initialisers. In contrast, **bar** constructor's MIL firstly performs computations with the arguments and then passes the results through. The operations here are still fairly simple to fit in full expressions but had they been more complex (e.g. with branches and loops) the syntactic form of the MIL would be an obstacle.

**Cassio Neri** has a PhD in Applied Mathematics from Université de Paris Dauphine. He worked as a lecturer in Mathematics before becoming a quantitative analyst. Now he works in the FX Quantitative Research at Lloyds Banking Group in London. He can be contacted at cassio.neri@gmail.com.

```
double init_x(double d) {
  const double b = d * d;
  const double x = cos(b);
  return x;
}

bar::bar(double d, foo& r1, foo& r2)
: ... x_(init_x(d)), ...
```

Listing 2

This article presents some techniques that allow more complex logic in the MIL. It's *not* advocating complexity in the MIL, it only shows some ways to achieve this *if you have to*.

Before looking at these methods, we consider the possibility of avoiding the MIL altogether.

## Avoiding the MIL

Notice that **d_** isn't initialised in the MIL. In this case, the compiler implicitly *initialises*[2] **d_** and then we *assign* it to **d** in the constructor's body. Could we do the same for the other subobjects? Not always. Assume that **foo** doesn't have an accessible default constructor. Then, the compiler can't implicitly initialise **f_** and yields an error. We simply don't have a choice and *must* initialise **f_** in the MIL. In addition to subobjects of types without an accessible default constructor, reference members (e.g. **r_**) and **const** members of non class type (e.g. **x_** and **y_**) *must* be explicitly initialised otherwise the compiler complains. Although not enforced by the language, we can add to this list subobjects of *immutable* types – types with no non-**const** methods apart from constructors and a destructor.

It's possible for some subobjects to be default initialised first and then changed in the constructor's body. Nevertheless this two-step set up process might be wasteful. Actually, this argument is the most common stated reason to prefer initialisation in the MIL to assignment in constructor [Meyers05, §4]. For fundamental types, however, there's no penalty because default initialisation does nothing and costs nothing.

## Initialiser functions

The first idea for complex initialisation is very simple and consists of writing an initialiser function that delivers the final result to direct initialise a subobject. Listing 2 shows this technique applied to our example.

We emphasise that, in our toy example, **x_** can be directly initialised in the MIL (as seen in Listing 1). Listing 2 is merely a sample for more complex cases.

1 C++11 also allows the use of braces but their semantics are different and outside the scope of this article. Therefore, we shall consider only parenthesised initialisations and their C++03 semantics.

2 It's unfortunate but according to C++ Standard definitions, sometimes – as in this particular case – initialisation means doing nothing and the value of the object is indeterminate.

**We can bundle some related members into a nested struct and create an initialiser function for the struct rather than for individual members**

Most frequently the initialiser function creates a local object of the same type of the subobject that it initialises and returns it by value. Then the subobject is copy- or move-initialised from this value. Therefore, the subobject's type must be constructible (in particular, it can't be an abstract class) and also copy- or move-constructible.

Calling the copy- or move-constructor might have a cost. Nevertheless, mainstream compilers implement the return value optimisation [RVO] which, under certain circumstances, elides this call. Unfortunately, this doesn't eliminate the need for the subobject's type to be copy- or move-constructible.

In another variation, there are initialisers for various arguments that the subobjects' constructors take. For instance, an initialiser function for base might compute `d * d` and return this value which is then passed to `base`'s constructor. In this way, the argument types, rather than the subobjects, must be constructible and copy- or move-constructible.

It's worth mentioning that when the subobject is a reference member, the initialiser function must return a reference to a non-local object, otherwise the member will dangle. For instance, an initialiser function for `r_` could be as follows.

```
foo& init_r(double d, foo& r1, foo& r2) {
  // r1 and r2 are non-local
  return d > 0.0 ? r1 : r2;
}
```

A positive aspect of having an initialiser function is that it can be used (and it most likely will be) by many constructors. When there's no need to reuse the initialiser, C++11 offers the tempting possibility of writing the initialiser function as a lambda expression as shown below. Notice, however, that readability suffers.

```
x_([&]() -> double {
  const double b = d * d; // d is captured
  const double x = cos(b);
  return x;
} (/* parentheses for calling the lambda */) )
```

Where should the initialiser function be? Assuming that its sole purpose is initialising a class member (so it's not going to be used anywhere else), then placing it in the global or in a named `namespace` is pollution. Making the initialiser a member of the class might come to mind but this isn't ideal because it decreases encapsulation [Meyers00]. Additionally, this requires the initialiser's declaration to be in the class header file forcing on clients an artificial dependency on the initialiser function. The best place for it is inside the class source file (which we're assuming is *not* its header file). Making the initialiser invisible outside the file (by declaring it either static or in an unnamed `namespace`) improves encapsulation and decreases linking time.

Using an initialiser function is the best technique presented in this article as far as encapsulation, clarity and safety are concerned. However, one feature that this solution lacks is the ability to reuse results obtained by one initialiser into another. For instance, the value of `d * d` must be calculated by the initialiser functions of `base`, `x_` and `y_`. In this example, this issue isn't a big deal but it could be if the result was obtained through a very costly operation.

```
class bar : public base {
  struct point {
    double x, y;
  };
  const point p_;
  static point init_p(double d);
  ...
};

bar::point bar::init_p(double d) {
  const double     b = d * d;
  const bar::point p = {cos(b), sin(b)};
  return p;
}

bar::bar(double d, foo& r1, foo& r2)
:  ... p_(init_p(d)), ...
```

Listing 3

Classes can have a member whose only purpose is storing a result to be used by different initialiser functions (e.g. `bar` could have a member `b_` to store `d * d`). This is obviously wasteful and, as in this section, we want partial results to have a short lifetime. The next sections present methods to achieve this goal.

## Bundling members

We can bundle some related members into a nested `struct` and create an initialiser function for the `struct` rather than for individual members. Listing 3 shows relevant changes to bar needed to initialise the two `const` members in one go.

As in the previous section, the type returned by the initialiser function must be copy- or move-constructible and so do the `struct` members.

The initialiser function needs access to the nested `struct`. Ideally, this type will be `private` and the initialiser will be a `static private` member. The initialiser could be a `friend` but, being an implementation detail, hiding it inside the class is advisable. (Unfortunately, it can't be hidden as much as in the previous section.) Alternatively, the initialiser function can be non-member and non-`friend` provided that the `struct` is made `public` but this decreases encapsulation even further.

We can't include base classes in the `struct` and each of them needs a different initialiser function. However, as in our example, the initialiser function of a base class could profit from results obtained by other initialiser functions. The next section shows how to achieve this goal.

## Using an argument for temporary storage

In rare cases we can change the value of an argument to something that is more reusable. Listing 4 is an attempt for our example and consists of changing `d` to `d * d` just before initialising `base`. Unfortunately, this

**A fix for the issue is to use a dummy argument for temporary storage and giving it a default value to avoid bothering clients**

```
bar::bar(double d, foo& r1, foo& r2)
: base(d = d * d),         // d has a new value
  x_(cos(d)), y_(sin(d)), // OK : uses new value
  r_(d > 0.0 ? r1 : r2),  // BUG: uses new value
  f_(exp(d), -exp(d)) {   // BUG: uses new value
  d_ = d;                 // BUG: uses new value
}
```

Listing 4

doesn't work here since initialisations of `r_`, `f_` and `d_` need the original value of `d` but they also get the new one.

A fix for the issue above is to use a dummy argument for temporary storage and giving it a default value to avoid bothering clients. This technique is in practice in Listing 5.

This works because the dummy argument persists for a short period but long enough to be reused by different initialisers. More precisely, its lifetime starts before the first initialisation of a subobject (`base` in our example) and ends after the constructor exits.

A problem (alas, there will be others) with this approach is that the constructor's extended signature might conflict with another one. If it doesn't today, it might tomorrow. As an improvement, we create a new type for the storage. For better encapsulation this type is nested in the `private` section of the class as Listing 6 illustrates.

The simplicity of our example is misleading because the assignment `tmp.b = d * d` can be nicely put in the MIL whereas in more realistic scenarios `tmp` might need a more complex set up. It can be done, for instance, in `base`'s initialiser function by making it take a storage argument by reference as Listing 7 shows.

Notice that `tmp` is passing through the two-step set up process that we have previously advised against. Could we forward `d` to `storage`'s constructor to avoid the default initialisation? For this, `bar`'s constructor requires a declaration similar to

```
bar(double d, foo& r1, foo& r2,
   storage tmp = storage(d));
```

```
class bar : public base {
  ...
public:
  bar(double d, foo& r1, foo& r2, double b = 0.0);
};

bar::bar(double d, foo& r1, foo& r2, double b)
: base(b = d * d),         // b has a new value
  x_(cos(b)), y_(sin(b)), // OK : uses b = d * d
  r_(d > 0.0 ? r1 : r2),  // OK : uses d
  f_(exp(d), -exp(d)) {   // OK : uses d
  d_ = d;                 // OK : uses d
}
```

Listing 5

```
class bar : public base {
  struct storage {
    double b;
  };
  ...
public:
  bar(double d, foo& r1, foo& r2,
      storage tmp = storage());
};

bar::bar(double d, foo& r1, foo& r2, storage tmp)
: base(tmp.b = d * d),
  x_(cos(tmp.b)), y_(sin(tmp.b)), ...
```

Listing 6

Unfortunately, this isn't legal. The evaluation of one argument can't refer to others. Indeed, it's fairly well known that in a function call the order of argument evaluation is undefined. If the code above were allowed, then we could not be sure that the evaluation of `tmp` occurs after that of `d`. Recall that if `storage` consists of fundamental types only, then the default initialisation costs nothing. If it contains a member of non-fundamental type, then the technique presented in the next section applies to prevent default initialisation of a member. The method is general and equally applies to `bar` itself.

A very important warning is in order before leaving this section. Unfortunately, the method presented here is unsafe! The main issue is that the technique is very dependent on the order of initialisation of subobjects. In our example, `base` is the first subobject to be initialised. For this reason, `init_base` had the responsibility of setting up `tmp` before it could be used by `init_x`. The order of initialisation of subobjects is very sensitive to changes in the class. To mitigate this issue you can create a reusable empty class, say, `first_base`, that as its name indicates, must be the first base of a class to which we want to apply the technique presented here. Furthermore, this class' initialiser function will have the responsibility of setting up the temporary storage as shown in Listing 8.

```
double bar::init_base(double d, storage& tmp) {
  tmp.b = d * d;
  return tmp.b;
}

double bar::init_x(const storage& tmp) {
  const double x = cos(tmp.b);
  return x;
}

bar::bar(double d, foo& r1, foo& r2, storage tmp)
: base(init_base(d, tmp)), x_(init_x(tmp)), ...
```

Listing 7

C++11 offers a **loophole that we can exploit** to prevent the compiler calling the default constructor

```
class first_base {
protected:
  explicit first_base(int) { // does nothing
  }
};

class bar : first_base, public base {
  ...
};

int bar::init_first_base(double d, storage& tmp) {
  tmp.b = d * d;
  return 0;
}

double bar::init_base(const storage& tmp) {
  return tmp.b;
}

bar::bar(double d, foo& r1, foo& r2, storage tmp)
: first_base(init_first_base(d, tmp)),
  base(init_base(tmp)), ...
```

**Listing 8**

The use of **first_base** makes the code safer, clear and *almost* solves the problem. Even when **first_base** is the first in the list of base classes, there's still a chance that it's not going to be the first subobject to be initialised. This occurs when the derived class has a direct or indirect virtual base class because virtual bases are initialised first. Experience shows that only a minority of inheritances are virtual and, therefore, this issue is unlikely to happen. However, it's always good to play safe. So, to be 100% sure, it suffices to virtually inherit from **first_base** (always keeping it as the first base in the list). The price that a class has to pay for this extra safety is carrying an extra pointer.

## Delaying initialisation

We arrive at the final technique of this article. The basic idea is delaying the initialisation of a subobject until the constructor's body where more complex code can sit.

Compilers have a duty of trying to ensure that every object of class type is properly initialised before being used. Their way to perform this task is calling the default constructor whenever the programmer doesn't explicitly call one. However, C++11 offers a loophole that we can exploit to prevent the compiler calling the default constructor.

The underlying pattern that supports delayed initialisation is the *tagged union* [TU], also known by various other names (e.g. *discriminated union, variant type*). A tagged union can hold objects of different types but at any time keeps track of the type currently held. Frequently, default initialisation of a tagged union means either no initialisation at all or default initialisation of a particular type (which again might mean no initialisation at all).

```
class bar : public base {
  union { // unnamed union type
    foo f_;
  };
  ...
};

bar::bar(double d, foo& r1, foo& r2)
: ... /* no f_ in the MIL */ {
  const double e = exp(d);
  new (&f_) foo(e, -e);
}

bar::~bar() {
  (&f_)->~foo();
}
```

**Listing 9**

In general, tagged unions are implemented in C/C++ through unions. Unfortunately, the constraints that C++03 imposes on types that can be members of unions are quite strict and implementing tagged unions demands a lot of effort [Alexandrescu02]. C++11 relaxes the constraints on union members and gives more power to programmers. However, this come with a cost: now the programmer is responsible for assuring proper initialisation of union members. The technique that we shall see now relies on C++11. Later we shall see what can be done in C++03.

Class **foo** has no accessible default constructor and we are forced to initialise **f_** in the MIL to prevent a compiler error. We want to postpone the initialisation of **f_** to the constructor's body where we can compute, store and reuse **exp(d)**. This can be achieved by putting **f_** inside an unnamed **union** as shown in Listing 9.

Since the **union** is unnamed all its members (only **f_** in this case) are seen as if they were members of **bar** but the compiler forgoes their initialisations. A member of the **union** can be initialised in the constructor's body through a placement **new**. In Listing 9 this builds an object of type **foo** in the address pointed by **&f_** or, in other words, the **this** pointer inside **foo**'s constructor will be set to **&f_**. Simple, beautiful and efficient – but this isn't the end of the story.

The compiler neither initialises a member of a **union** nor destroys it. Ensuring proper destruction is again the programmer's responsibility. Previously – listings 1–8 – the destruction of **f_** was called when its containing **bar** object was destroyed. To imitate this behaviour, the new **bar**'s destructor calls **~foo()** on the object pointed by **&f_**.

We have just written a destructor, and the rule of three says that we probably need to write a copy-constructor and an assignment operator as well. This is the case here. In addition, there are extra dangers that we must consider. For instance, a new constructor might be added to **bar** and the writer might forget to initialise **f_**. If a bar object is built by this

This leaves the **default constructor empty** and you might wonder **why bother writing this constructor** since the compiler will **automatically implement one** exactly as ours

```
template <typename T>
class delayed_init {
  bool is_init_ = false;
  union {
    T obj_;
  };

public:
  delayed_init() {
  }
  ~delayed_init() {
    if (is_init)
      (&obj_)->~T()
  }

  template <typename... Args>
  void init(Args&&... args) {
    new (&obj_) T(std::forward<Args>(args)...);
    is_init_ = true;
  }
  T* operator->() {
    return is_init_ ? &obj_ : nullptr;
  }
  T& operator*() const {
    if (is_init_)
      return obj_;
    throw std::logic_error("attempt to use "
      "uninitialised object");
  }
  ...
};
```

Listing 10

constructor, then at destruction time (probably earlier) `f_` will be used. The code is then in undefined behaviour situation. To avoid this and other issues, we use a `bool` flag to signal whether `f_` has been initialised or not. When an attempt to use an uninitialised `f_` is made, the code might inform you by, say, throwing an exception. However, `bar`'s destructor can be more forgiving and ignore `f_` if it's uninitialised. (Recall that a destructor shouldn't throw anyway.)

Instead of forcing `bar` to manage `f_`'s usage and lifetime, it's better to encapsulate this task in a generic template class called, say, `delayed_init`. Listing 10 shows a rough draft of an implementation. A more complete version is available in [Neri] but *don't use it* (I repeat, *don't use it*) because Boost.Optional [Optional] is a better alternative. Indeed, it's a mature library that has been heavily tested over the last few years and also works with C++03. `delayed_init` is presented for didactic purposes only. As mentioned above, `union` rules in C++03 are strict and make the implementation of `boost::optional` more complex and difficult to understand. In contrast, `delayed_init` assumes C++11 rules and has a simpler code. See `delayed_init` as a draft of what `boost::optional` could be if written in C++11. Even though, Fernando Cacciola – the author of Boost.Optional – and Andrzej Krzemienski are working on a proposal [Proposal] for `optional` to be added to the C++ Standard Library. This idea has already been praised by a few members of the committee.

Let's see what `delayed_init` looks like. Its member `is_init_` is initialised to false using the new *brace-or-equal initialisation* feature of C++11. Therefore, we don't need to do it in the MIL. This leaves the default constructor empty and you might wonder why bother writing this constructor since the compiler will automatically implement one exactly as ours. Actually, it won't because `delayed_init` has an unnamed `union` member (which is the whole point of this template class).

When the time comes to initialise the inner object, it suffices to call `init()`. This method is a *variadic template* function – another welcome and celebrated C++11 novelty – that takes an arbitrary number of arguments (indicated by the ellipsis `...`) of arbitrary types by *universal reference* [Meyers12] (indicated by `Args&&` where `Args` is deduced). These arguments are simply handed over to `T`'s constructor via `std::forward`. (Take another look at this pattern since it's expected to become more and more frequent.)

Also note the presence of `operator->()`. Essentially, the class `delayed_init<T>` is a wrapper to a type `T`. We wish it could be used as a `T` by implementing `T`'s `public` interface and simply forwarding calls to `obj_`. This is impossible since `T` is unknown. A close alternative is returning a pointer to `obj_` because `T*` replicates `T`'s interface with slightly different syntax and semantics. Actually, pointer semantics fits very naturally here. Indeed, it's common for a class to hold a pointer to an object rather than the object itself. In this way, the class can delay the object's initialisation to a later moment where all data required for the construction is gathered. At this time the object is created on the heap and its address is stored by the pointer. Through `delayed_init`, we are basically replacing the heap with internal storage and, like in a smart pointer, managing the object's lifetime. Finally, the `operator*()` is also implemented. It provides access to `obj_` and throws if `obj_` hasn't been initialised.

## Conclusion

Initialisation in the MIL rather than assignment in the constructor has been advocated for long time. However, in some circumstances, there's genuine need for not so simple initialisations which conflict with the poorness of the MIL's syntax. This article has presented four techniques to overcome this situation. They vary in applicability, clarity and safety. On the way it presented some of the new C++11 features. ■

## Acknowledgements

Cassio Neri thanks Fernando Cacciola and Lorenz Schneider for their suggestions and careful reading of this article. He also thanks the *Overload* team for valuable remarks and feedback.

it's common for **a class to hold a pointer to an object** rather than the object itself

## References

[Alexandrescu02] Andrei Alexandrescu, Generic: Discriminated Unions (I), (II) & (III), *Dr.Dobb's,* June 2002. http://tinyurl.com/8srld2z http://tinyurl.com/9tofeq4 http://tinyurl.com/8ku347d

[Meyers00] Scott Meyers, How Non-Member Functions Improve Encapsulation, *Dr.Dobb's*, February 2000. http://tinyurl.com/8er3ybp

[Meyers05] Scott Meyers, *Effective C++*, Addison-Wesley 2005.

[Meyers12] Scott Meyers, Universal References in C++11, *Overload* 111, October 2012. http://tinyurl.com/9akcqjl

[Neri] Cassio Neri, delayed_init implementation. https://github.com/cassioneri/delayed_init

[Optional] Fernando Cacciola, Boost.Optional. http://tinyurl.com/8ctk6rf

[Proposal] Fernando Cacciola and Andrzej Krzemienski, A proposal to add a utility class to represent optional objects (Revision 2), September 2012. http://tinyurl.com/bvyfjq7

[RVO] Return Value Optimization, Wikipedia. http://tinyurl.com/kpmvdw

[TU] Tagged Union, Wikipedia. http://tinyurl.com/42p5tuz

MSc in

**Software Engineering**

(part-time)

UNIVERSITY OF OXFORD

- **a flexible programme in software engineering leading to an MSc from the University of Oxford**
- **a choice of over 30 different courses, each based around an intensive teaching week in Oxford**
- **MSc requires 10 courses and a dissertation, with up to four years allowed for completion**
- **applications welcome at any time of year, with admissions in October, January, and April**

**www.softeng.ox.ac.uk**

# ~~640K~~ $2^{256}$ Bytes of Memory is More than Anyone Would Ever ~~Need~~ Get

How fast can computers get? Sergey Ignatchenko provides us with some upper limits.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translator or the *Overload* editor. Please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented providing an exact translation. In addition, both the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

There is a famous misquote commonly and erroneously attributed to Bill Gates: "640K of memory is all that anybody with a computer would ever need." Apparently, Gates himself has denied that he has ever said anything of the kind [Wired97]. Reportedly, he went even further, saying "No one involved in computers would ever say that a certain amount of memory is enough for all time." [Wired97] Well, I, 'No Bugs' Bunny, am involved in computers and I am saying that while there can be (and actually, there is) a desire to get as much memory as possible, physics will certainly get in the way and will restrict any such desire.

## Moore's Law vs Law of Diminishing Returns

*What goes up must come down*
proverb

There is a common perception in the computer world that all the current growth in hardware will continue forever. Moreover, even if such current growth is exponential, it is still expected to continue forever. One such example is Moore's Law; originally Moore (as early as 1965, see [Moore65]) was referring to doubling the complexity of integrated circuits every year for next 10 years, i.e. to 1975 (!). In 1975, Moore adjusted his prediction to doubling complexity every two years [Moore75], but again didn't go further than 10 years ahead in his predictions. As it happens, Moore's law has stood for much longer than Moore himself had predicted. It was a great thing for IT and for everybody involved in IT, there is no doubt about it. With all the positives of these improvements in hardware, there is one problem with such a trend though – it has led to the perception that Moore's Law will stand forever. Just one recent example – in October 2012, CNet published an article arguing that this trend will continue for the foreseeable future [CNet12]; in particular, they've quoted the CTO of Analog Devices, who said: "Automobiles and planes are dealing with the physical world. Computing and information processing doesn't have that limitation. There's no fundamental size or weight to bits. You don't necessarily have the same constraints you have in these other industries. There potentially is a way forward."

There is only one objection to this theory, but unfortunately, this objection is that this theory is completely wrong. In general, it is fairly obvious that no exponential growth can keep forever; still, such considerations cannot lead us to an understanding of how long it will continue to stand. In practice, to get any reasonable estimate, we need to resort to physics. In 2005, Moore himself said "In terms of size [of a transistor] you can see that we're approaching the size of atoms which is a fundamental barrier, but it'll be two or three generations before we get that far – but that's as far out as we've ever been able to see." [Moore05] Indeed, 22nm technology already has transistors which are just 42 atoms across [Geek10]; and without going into very different (and as yet unknown) physics one cannot possibly go lower than 3 atoms per transistor.

**'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

## Dangers of relying on exponential growth

*Anyone who believes exponential growth can go on forever in a finite world is either a madman or an economist.*
Kenneth Boulding, economist

In around the 2000s, Moore's Law had been commonly formulated in terms of doubling CPU frequency every 2 years (it should be noted that it is not Moore's formulation, and that he shouldn't be blamed for it). In 2000, Intel has made a prediction that by 2011, there will be 10GHz CPUs out there [Lilly10]; as we can see now, this prediction has failed miserably: currently there are no CPUs over 5GHz, and even the only 5GHz one – POWER6 – is not produced by Intel. Moreover, even IBM which did produce POWER6 at 5GHz, for their next-generation POWER7 CPU has maximum frequency of 4.25 GHz. With modern Intel CPUs, even the 'Extreme Edition' i7-3970XM is mere 3.5GHz, with temporary Turbo Boost up to 4Ghz (see also an extremely enthusiastic article in *PC World*, titled 'New Intel Core I7 Extreme Edition chip cracks 3GHz barrier' [PCWorld12]; the only thing is that it was published in 2012, not in 2002). In fact, Intel CPU frequencies have decreased since 2005 (in 2005, the Pentium 4 HT 672 was able to sustain a frequency of 3.8GHz).

One may say, "Who cares about frequencies with all the cores around" – and while there is some point in such statement (though there are many tasks out there where performance-per-core is critical, and increasing the number of cores won't help), it doesn't affect the fact – back in 2000 nobody had expected that in just 2 years, all CPU frequency growth would hit a wall and that frequency will stall at least for a long while.

It is also interesting to observe that while there is an obvious physical limit to frequencies (300GHz is already commonly regarded as a border of infra-red optical range, with obviously different physics involved), the real limit has came much earlier than optical effects have started to kick in.

## Physical limit on memory

*The difference between stupidity and genius is that genius has its limits.*
Albert Einstein

As we've seen above, exponential growth is a very powerful thing in a physical world. When speaking about RAM, we've got used to doubling address bus width (and address space) once in a while, so after move from 16-bit CPUs to 32-bit ones (which has happened for mass-market CPUs in mid-80s) and a more recent move from 32-bit CPUs to 64-bit ones, many have started to expect that 128-bit CPUs will be around soon, and then 256-bit ones, and so on. Well, it might or might not happen (it is more about waste and/or marketing, see also below), but one thing is rather clear – $2^{128}$ bytes is an amount of memory which one cannot reasonably expect in any home device, with physics being the main limiting factor. Let's see – one cubic cm of silicon contains around $5*10^{22}$ atoms. It means that even if every memory cell is only 1 atom large, it will take $2^{128}/(5*10^{22})*8$ cm$^3$ of silicon to hold all that memory; after calculating it, we'll see that $2^{128}$ bytes of memory will take approximately 54 billion cubic metres (or 54 cubic kilometres) of silicon. If taking other (non-silicon-based) technologies (such as HDDs), the numbers will be a bit different, but still the amount of space necessary to store such memory will be a number of cubic kilometres, and this is under an absolutely generous assumption that one atom is enough to implement a memory cell.

To make things worse, if we're speaking about RAM sizes of $2^{256}$ bytes, we'll see that implementing it even with 1 atom/cell will take about $10^{78}$ atoms. Earth as a planet is estimated to have only $10^{50}$ atoms, so it will take ten billion billion billions of planets like Earth to implement a mere $2^{256}$ bits of memory. The solar system, with $10^{57}$ atoms, still won't be enough: the number we're looking for is close to number of atoms in the observable universe (which is estimated at $10^{79}$–$10^{80}$). In other words – even if every memory cell can be represented by a single atom, we would need 1 to 10% of all the stars and planets which we can see (with most of them being light years afar), to implement $2^{256}$ bytes of memory. Honestly, I have serious doubts that I will live until such a thing happens.

## On physics and waste of space

*Architecture is the art of how to waste space.*
Philip Johnson

It should be noted that the analysis above is based on two major assumptions. First, we are assuming that our understanding of physics is not changed in a drastic manner. Obviously, if somebody finds a way to store terabits within a single atom, things will change (it doesn't look likely in the foreseeable future, especially taking the uncertainty principle into account, but strictly speaking, anything can happen). The second assumption is that when speaking about address space, we are somewhat assuming that address space is not wasted. Of course, it is possible to use as much as a 1024-bit address space to address a mere 64K of RAM, especially if such an address space is allocated in a manner similar to the allocation of IPv4 addresses in early days ("here comes IBM, let's allocate them as small portion of the pool – just class A network, or 1/256 of all IP addresses"). If there is a will to waste address space (which can be driven by multiple factors – from the feeling that space is infinite, like it was the case in early days of IPv4 addresses, to the marketing reason of trying to sell CPUs based on perception that a 128-bit CPU is better than a 64-bit one just because of the number being twice as big) – there will be a way. Still, our claim that '$2^{256}$ bytes of memory is not practically achievable' stands even without this second assumption. In terms of the address bus (keeping in mind that an address bus is not exactly the same as an address space, and still relying on the first assumption above), it can be restated as '256-bit address bus is more than anyone would ever need'. ■

## References

[CNet12] Moore's Law: The rule that really matters in tech. Stephen Shankland, *CNet*, Oct 2012, http://news.cnet.com/8301-11386_3-57526581-76/moores-law-the-rule-that-really-matters-in-tech/

[Lilly10] Where are Intel's 10GHz Processors Hiding? Paul Lilly, 2010 http://www.maximumpc.com/article/news/where_are_intels_10ghz_processors_hiding

[Loganberry04] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[Moore65] 'Cramming more components onto integrated circuits', Moore, G. *Electronics Magazine*, 1965

[Moore75] Progress In Digital Integrated Electronics, Gordon Moore, IEEE Speech, 1975

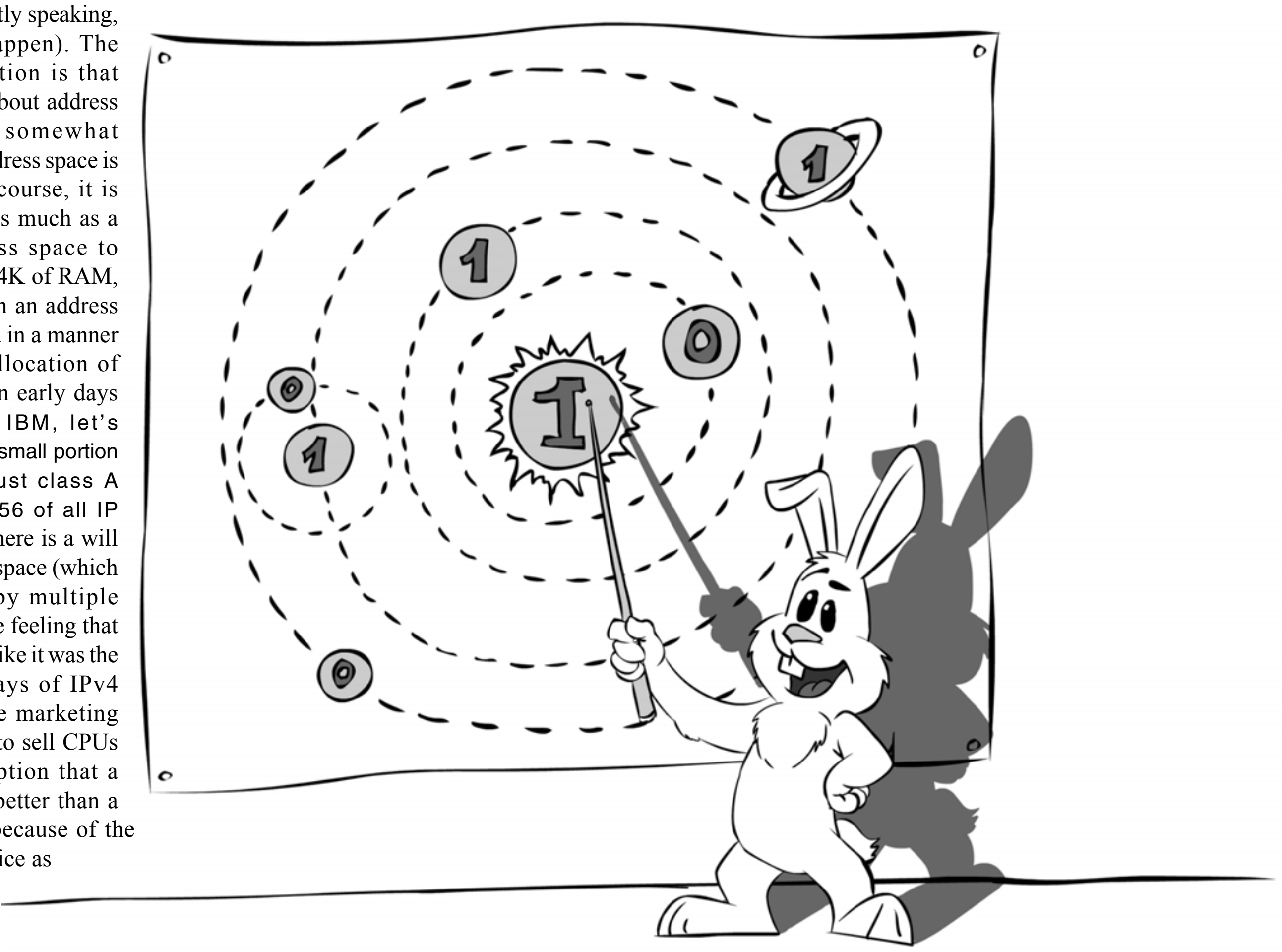
[Moore05] Moore's Law is dead, says Gordon Moore Manek Dubash, *TechWorld* http://news.techworld.com/operating-systems/3477/moores-law-is-dead-says-gordon-moore/

[PCWorld12] New Intel Core I7 Extreme Edition chip cracks 3GHz barrier. *PC World*, Sep 2012, http://www.pcworld.com/article/261873/new_intel_core_i7_extreme_edition_chip_cracks_3ghz_barrier.html

[Wired97] Did Gates Really Say 640K is Enough For Anyone? -- John Katz, *Wired*, 1997

## Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.



# Footprint on Modify

Tracking history can be done in a variety of ways. Andy Balaam describes one technique with many advantages.

Many programs need to solve the problem of keeping track of the history of changes made to a model, and making it possible to navigate backwards and forwards through that history. Perhaps the most obvious example is an interactive program with an undo/redo facility.

Writing code to track history can be done in a variety of ways, each with different sets of constraints, advantages and disadvantages. This article describes one technique which we have been using in our most recent product, and which we have found to have a number of advantages for the particular problem we are solving.

This technique, which we have found ourselves calling 'footprint on modify', involves taking a copy of an object whenever we are about to change it, and inserting it into the historical record in place of the modified object.

In this article we will describe the problem we are solving and some alternative approaches to solving it, before describing our own approach and discussing its advantages and disadvantages in comparison with other options.

We hope, when you come to tackle a similar problem, the issues we cover here will provide you with a richer set of concepts for reasoning about the right solution for your problem area.

## The problem – tracking changes in an object model

Like many programs, our program has an object model – a set of classes which together form a model of the artefact being generated by our users as they use it. Instances of these classes are linked by parent–child relationships (some objects 'contain' others) and references (some objects refer to others).

The problem we must solve is being able to backtrack to the state of the model at a given point in the past. This means we must be able to construct an object model which is identical to the one that existed at that time. We must allow modifying that object model starting from a point in the past, taking a different branch in history. In addition, we are interested in keeping track of this non-linear history, not simply throwing away the previous branch as many undo/redo systems do, but keeping it available for later reference.

This is illustrated in figure 1, which shows a system moving through states 1–4 as changes are made to the model, before backtracking to state 1, and being changed in different ways, resulting in states 2a and 3a. We want to keep the entire history in this case, including states 2, 3 and 4. Users of board-game software which allows exploring different game trees will be familiar with working this way.

There are many different ways of representing object models and the changes they undergo, and we will begin by looking at some of the alternatives we considered before settling on our approach.

**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

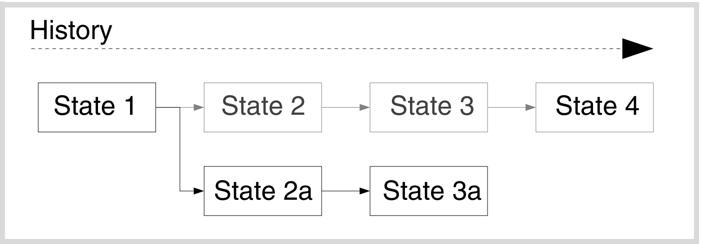


Figure 1

### Alternative solutions

- **Saving complete models**

  The most brute-force method of preserving model history is to store complete models (either on disk or in memory) every time a change is made. This is often simple to implement, but can be expensive both in terms of time taken to copy or save entire object models, and in terms of storage for the saved models.

  This method makes it easy to 'prune' the history, only keeping the most important points when storage becomes limited, and it does not require the invention of a new language to represent model changes – simply a way to save or clone objects in the model.

  It also makes navigation through long distances in the history simple and relatively cheap – we simply restore the complete model which was stored for that point in time.

- **Keeping a change log**

  The classic solution to providing undo/redo behaviour is via a reversible log of actions taken. This amounts to a language that encodes object model modifications, and is often used as an example in textbooks explaining the Command design pattern, since this pattern is well-suited to providing this functionality. Each entry in the log provides a way of changing the model back to the state it was in before a particular change, and a way of moving back again to the after-state. The log entries themselves may be objects with methods capable of modifying the model, or they may be descriptions of how to do it in some language.

  This solution has been shown to work in many contexts. Because it involves storing only the differences between states, it is light-weight in terms of the number of objects held in memory, but can be expensive to move large distances in the history, since the system must pass through all intermediate states in order to reach a particular one.

  In practice, many applications do not require movements of large distances in the history, but in our situation we do need to consider this case because we store a branched tree of history, providing a visualisation to the user through which they can navigate.

  The change log may be seen as somewhat fragile, since if a single point in the log is lost, we are unable accurately to reconstruct states before that time. This is not only a theoretical problem with stability, The key to the solution is **taking copies of objects as they are modified**, and inserting those copies into the **historical record**

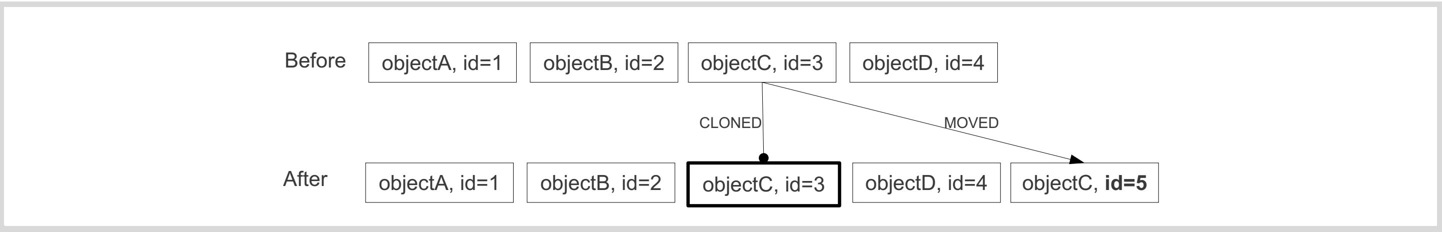


Figure 2

but also makes ‘pruning’ the history to keep only important points more difficult, since every entry in the log is vital. Pruning to reduce the number of log points requires combining multiple points into one, which may be non-trivial.

- **Using copy on write**

  Saving complete models provides a flexible but expensive approach, and one way to gain some of its advantages without so much cost in terms of time and storage is to use the copy-on-write strategy.

  In this method objects are copied when they are modified, leaving the unmodified object stored, allowing us to revert to the old state by looking at the old object.

  This has some of the same advantages as saving complete models. Navigating large distances in the history is cheap since it simply involves restoring the objects that were active at that moment. ‘Pruning’ the history is possible, but more difficult than in the case of saving complete models, because we need to identify which objects are relevant for a given moment in history. We can do this by examining the whole model.

  This solution may be simpler than the change log, because it does not require a language of model changes to be used – instead we only need to know how to copy or store objects, and it will be cheaper than saving complete models because only those objects which are changing need to be copied.

  Depending on the implementation, there is a significant problem with this approach, which is that the user of the object model may be forced to understand what is happening as they manipulate the objects. If objects are copied when they are modified, the user may need to get hold of a reference to the newly-copied object, and stop using the reference to the historical object. This could be inconvenient and error-prone.

  The problem could, of course, be solved by introducing another level of abstraction. It was while we were considering solutions to this problem that we chose to look into the ‘footprint on modify’ approach which is discussed in the rest of this article.

There are, of course, many different alternative solutions, but the three above were the main ones we considered before deciding on our chosen approach.

## The solution – create historical copies as objects change

The approach we chose was inspired by copy on write, but attempts to resolve the problem of allowing the user to manipulate objects without being aware of the building of history, and without adding a further level of abstraction on top of the standard objects.

The key to the solution is taking copies of objects as they are modified, and inserting those copies into the historical record.

### Parent–child relationships

The object model includes parent–child ownership relationships, and the model we are considering has a single root node, which is important to the process of tracking history. In an object model with no single root, we may add one object which is the parent of all the nodes with no parents, and consider that the single root.

The parent–child relationships are stored in the parent object, which holds a list of IDs of its children. It is important that the information is stored in this indirect way, because pointers or references to objects in memory may change in the future (which would effectively change history), whereas the properties of the object referred to by a given ID will always be consistent.

### The object log

The system is built on a structure called the Object Log. This is a collection of all versions of all the objects that have existed, indexed by unique ID. The Object Log is the owner of all objects, including those currently being manipulated by the user.

Every time an object is about to be modified, it is first cloned, and the clone is inserted into the Object Log under its old ID (replacing the object itself). The object’s ID is then changed, and it is re-inserted into the Object Log under its new ID, before being modified. This is illustrated in figure 2.

Because the object’s ID has changed, its parent, which refers to its children by ID, now refers to the old object. Therefore the parent is also cloned and given a new ID, and its list of children is updated to use the new ID for the first object. This process of cloning continues up the tree to the root, meaning that every change in the object model results in a new root node, as illustrated in figure 3.

The number of clones created for every change is limited to the depth of the parent–child tree, which in our model is a maximum of about 5 levels,

**Every class in the object model has the Trackable Object as a base class, and it is this which provides the change-tracking behaviour**

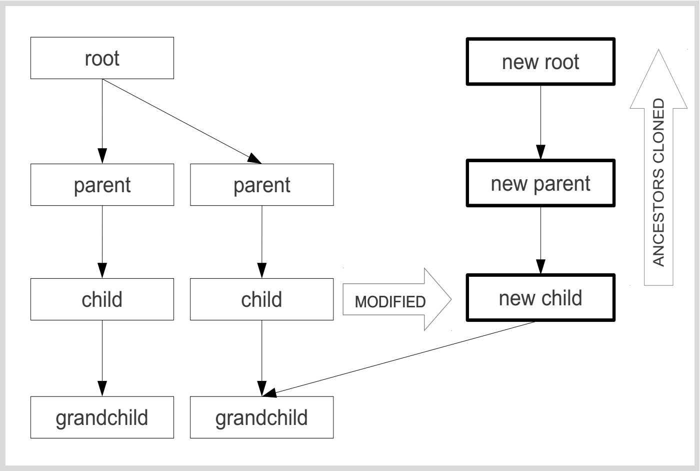


Figure 3

and in many models is of this order. This overhead is acceptable for our system. Crucially, the children and siblings of the modified objects do not need to be cloned. The number of children and siblings of a given node in our model, and many similar models, is unbounded.

### The time point and child–parent relationships

Because each change in the model results in a new root node, it is possible to identify a moment in time simply by storing the ID of the root node at that moment. If we retrieve the object with that ID from the Object Log and examine it to find its children's IDs, retrieve and examine them and continue in this way, we can find all the objects that existed at that moment, and their states.

We provide a Time Point object, which stores the ID of a node which was the root of our model at a given time.

Because a child object may exist in multiple instants in time in different parents (because its parent may have been cloned while being changed, but the child was unchanged), it does not make sense for the information about an object's parent to be stored in that object. Given a Time Point, we may reconstruct the parentage of all objects by walking the tree from the root. In practice we cache that information inside the Time Point object, as shown in figure 4.

### The trackable object

From the point of view of the implementor of a new object in the model, there are two classes which provide the required functionality: the Trackable Object, and the Trackable Collection.

Every class in the object model has the Trackable Object as a base class, and it is this which provides the change-tracking behaviour. Trackable Object keeps a reference to the Object Log which contains this object, and provides a 'footprint' method, which must be called before the object is changed. The footprint method calls a method (also called footprint) on the Object Log, which implements the cloning process described above.

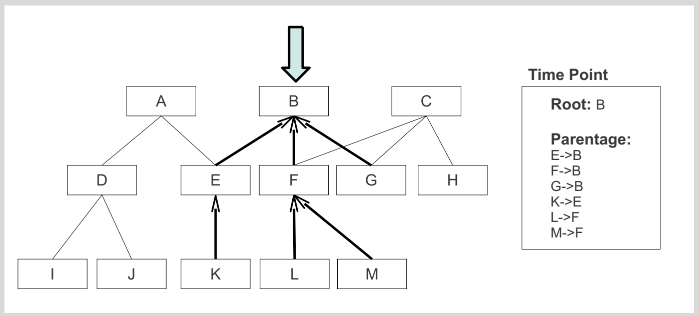


Figure 4

An object's ID is stored in the Trackable Object, and is controlled entirely by the Object Log. In our implementation, objects may be instantiated without an Object Log, which facilitates independent testing, but in this case they have no meaningful ID. Having an ID is tied very closely to having been inserted into an Object Log – indeed it is the job of the Object Log to set and maintain the IDs of the objects, and the objects themselves have 'no interest' in their ID. (For a significant amount of our implementation time, IDs were not stored on objects at all – they were added for efficiency, but could in principle be stored only on the Object Log – they are not really considered part of the public interface of an object.)

Because the Object Log will clone the object during the footprint call, the implementor of an object must provide a way of cloning it. How much effort it is to provide this facility varies widely in different programming languages and environments. In environments with automatic cloning facility, care must be taken to handle the circular reference between a Trackable Object and the Object Log.

### Ownership – the trackable collection

In our object model all objects except the root are owned by some parent object. We represent these relationships by allowing parents to contain one or more Trackable Collection objects. These are lists of children objects (typically with one Trackable Collection for each type of object the parent may contain). The list stores the IDs of children, rather than references to the actual objects. This means that an object containing a Trackable Collection of children will continue to refer to the unmodified child even if one of the objects representing a child is modified and gains a new ID.

This indirection via ID is necessary to ensure that a there is only ever one version of history – a single root ID will always give us the same tree of objects (in terms of IDs), but it does mean that when a child is changed we must make new copies of its parents and grandparents up to the root.

The Trackable Collection class provides convenience methods meaning users of the object model do not have to be aware of the references by ID or the footprinting that happens when an object changes. Methods which modify the Trackable Collection, such as `add`, `replace` and `remove` take care of calling `footprint` on the object which contains the collection. This is illustrated in figure 5.

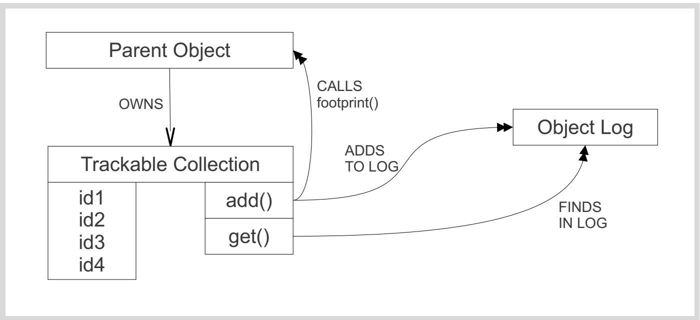


Figure 5

All of the parent–child relationships in our model take the form of resizeable lists of objects of the same type, and so are handled using the Trackable Collection class, but where needed a similar class could be built on the same lines to handle individual children, or fixed-size collections.

### Cross-references

Where objects in our object model need to refer to other objects elsewhere in the hierarchy, we use an unique name which is entirely separate from Trackable Object IDs. This means the reference is independent of changes in the object to which we are referring – we do not want to have to footprint all objects that refer to an object we are changing. The reference is treated the same as the other simple properties of an object like a name or description – the only potential difference is the need to clean up dangling references when an object is deleted, or to correct references if the unique name is changed. These are handled separately from the Trackable Object mechanism, and not covered here.

### Object model classes

Ordinary classes in the object model, which represent aspects of our problem domain, have relatively little to do to fit in with the footprint-on-modify system. They must provide a **`clone()`** method, which copies an object, preserving its properties and keeping references intact (without copying referenced objects or children). They should derive from the Trackable Object base class and hold children objects using Trackable Collections.

Object model classes must enforce that no changes may be made without first calling the **`footprint()`** method on the Trackable Object base class. This is implemented in our model by allowing changes only through setter methods, each of which begins with a call to **`footprint()`**. This is illustrated in figure 6.

With these provisions in place, the object model classes may be written in a familiar way, using any standard language types or custom classes for properties, so long as all properties are copied by the **`clone()`** method.

### External users

Users of the object model classes need not know what is going on underneath. Code that manipulates the model may hold references to objects, read and write properties and add or remove children, either using the **`add()`** and **`remove()`** methods of Trackable Collections, if the collections are exposed by the object model classes, or via specialised methods on the object model classes themselves, which in turn call **`add()`** or **`remove()`** on the Trackable Collections.

As these manipulations go on, the external code will always hold a reference to the latest version of each object, and underneath, each change will cause footprints to be added to the Object Log.

Code that handles undo and redo uses the Object Log directly, asking it for a unique ID to identify a moment in time, and using such an ID later to tell the log to revert back to that moment.

When the Object Log has reverted to a given moment, all existing references to object model objects must be dropped, and new references must be found by starting at the new root node, which can be provided by the Object Log. Given that the object model may have been completely transformed by the revert event, this requirement is not considered onerous.

Once a revert has been performed, the newly-provided object model may be manipulated as normal, and new footprints begin appearing in the Object Log. The old footprints, including those on a separate "branch" of history, are not overwritten, so we may keep a complex branched undo log, and jump back to any point on it at any time. The biggest challenge here is presenting this information in a useful way to the user!

## Discussion

Footprint on modify offers an alternative to change-log-based systems for undo/redo functionality in an object model.

Some advantages of this system include the ability to jump to any moment in history quickly, without the need to traverse intermediate states. This means moving to distant points is fast, there is no need for a language to describe changes in objects, and thus there is no danger that small bugs or inconsistencies in such a language will be propagated through history navigation.

Other advantages include the fact that users of the object model do not have to do anything to ensure history is tracked, unneeded time points in the log may be removed without changing other points, and keeping a history with all branches is just as easy as keeping a traditional linear history.

Disadvantages include the fact that whole objects are copied, rather than just storing changes, which could be a problem if objects are large, and the extra cloning required because the ancestors of objects must be cloned when the objects are changed.

Further potential disadvantages include the need for all objects to be clonable, and to inherit from the Tracking Object base class. Changing an existing object model to use footprint on modify would require significant changes to its implementation.

To build a fully-functional undo/redo mechanism, several areas must be covered which are outside of the scope of this article. The most important area is the structure of the actual undo/redo log, which holds on to root node IDs, and allows navigating between them. In some cases a simple linear stack and marker model will suffice, and in others a tree may need to be presented to the user, along with many other potential features such as named waypoints in history.

Further topics that are of interest, but not covered here, are the mechanisms we could use to prune unneeded time points to reduce storage space, and how to ‘page’ out and in old history to disk or other storage. ■

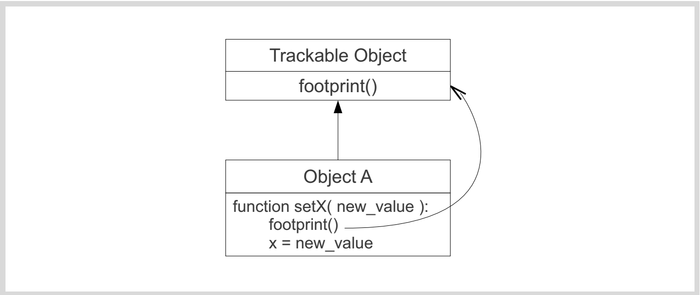


Figure 6

## Acknowledgements

The footprint on modify idea was developed by Edmund Stephen-Smith and Andy Balaam, based on and inspired by a copy-on-write model designed by Ramon Pisters and Ton Steijvers.

## Copyright

# Valgrind Part 5 – Massif

Poor performance can be caused by memory usage. Paul Floyd introduces Valgrind's heap memory profiling tool Massif.

This time there is a bit more good news. You've identified your CPU bottlenecks and have ironed them out. But now you have customers complaining that performance is poor when they are using very big projects. When you hear the word 'big', you should start thinking 'memory'.

## Using system profiling tools

Before you do any lengthy testing, you can get a quick impression of your application's memory use by using 'top' (or a similar process monitor tool).

Here I'm running a simulator on a 48Gbyte server. You should be looking at the **Mem:** and **Swap:** lines in the header block and the **VIRT** (virtual memory) and **RES** (resident memory) in the columns in Figure 1.

You can also get an idea what your system is doing by using a tool like **vmstat** (on UNIX-like systems). If you type "**vmstat 5**" then you'll sample the system every 5 seconds (with the first line being the average since the system booted). You'll see something like Figure 2.

**swpd**, **free**, **buff** and **cache** correspond to similarly named values in the **Mem:** and **Swap:** lines of top. When your performance is really poor, then it's the **si** and **so** values that you should look at. They are the amount of memory that is being swapped in and out from disk per second. Swapping is bad news. Disk drives are many orders of magnitude slower than RAM.

Enough of system profiling, I'm here to write about Valgrind. In particular, Massif, Valgrind's heap memory profiling tool. Massif will give you an overview of the memory used by your application over time. You can use it to help you to identify where and when your application allocates memory on the heap with a view to either reducing the amount of memory that you allocate or freeing it sooner. It can indicate that you have memory leaks, but it will only tell you about memory use, not memory abuse (like unreachable memory due to pointers going out of scope or being overwritten).

I'll start with a very simple example that just allocates and frees some memory (Listing 1).

To select Massif, run valgrind with the **--tool=massif** option. In this case, since the example uses **sleep()** and **usleep()**, I'll add the **--time-unit=ms** option, the default being instructions executed. There are further options to profile the stack, control the call context depth recorded and the number of snapshots taken. There isn't much output to the terminal, just the small piece of header information in Figure 3.

The important information is written to a file named `massif.out.<pid>` (you can control the file name with the **--massif-out-file** option). This isn't really meant for human consumption. For completeness, I'll include an extract, see Figure 4.

The output file is meant to be processed by **ms_print**. This outputs to the terminal. Firstly, there is a short summary, then there is an ASCII art plot of the memory use as a function of time. Figure 5 is what my example code produces.

As expected, the memory use rises monotonically for 10 seconds. The application allocates a bit over 800 million bytes, or 762.94MB. That's

```
top - 17:21:05 up 9 days,  2:16,  2 users,  load average: 1.09, 1.05, 1.01
Tasks: 232 total,   2 running, 230 sleeping,   0 stopped,   0 zombie
Cpu(s): 12.6%us,  0.0%sy,  0.0%ni, 87.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  49414760k total, 21724376k used, 27690384k free,   185660k buffers
Swap: 49151992k total,     2128k used, 49149864k free,  2669128k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12733 paulf     20   0 16.5g  16g  22m R 100.0 34.5 159:06.14 sim64.exe
23657 paulf     20   0 13252 1264  884 R  0.7  0.0   0:02.56 top
```

Figure 1

```
vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0   2128 27686448 185312 2667364    0    0     0     3    0    4 11  3 84  1  0
 1  0   2128 27686876 185312 2667364    0    0     0     0 1136  134 12  0 88  0  0
 1  0   2128 27687488 185312 2667364    0    0     0     0 1088   74 13  0 87  0  0
```

Figure 2

**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.

quite close to the peak shown in the graph. After this, the **`ms_print`** output shows information about snapshots taken during the execution of the application. All of the snapshots show a one line summary of the heap use. By default, every tenth snapshot shows details of where the memory was allocated. Figure 6 is the end of the output from the m1 test application.

We can see that the figure for the useful-heap matches what we expect. The extra-heap figure is the memory allocated for book-keeping. In a real world application, there would be an extensive tree of calls displayed showing where memory was allocated and how much (% of total and number of bytes). In this example, everything is done in **`main()`**, so there's not much context to see. Now that we have this information, what do we want to do with it? Generally, two things: try to free memory earlier and try to find more efficient data structures.

Now, let's go back to the small example and make some changes to improve the memory use. (See Listing 2.)

Now the memory is freed straight after it is used, rather than all at once at the end of the application. The ASCII art graph now looks like Figure 7.

Notice that the peaks where memory is allocated are not evenly spaced. This is an artefact of the sampling, and if your application does a lot of allocation and freeing, you may need to play with the **`--detailed-freq`** or **`--max-snapshots`** options.

As an alternative to the ASCII art, there is a separate tool, 'massif-visualizer' (Figure 8).

I installed massif-visualizer on Kubuntu with the package manager. It wasn't available in the package manager of openSUSE, and I had problems with Cmake dependencies when trying to build it from source. The GUI,

```
// m1.cpp
#include <unistd.h>

const size_t DATA_SIZE = 100U;
const size_t BLOCK_SIZE = 1000000;

int main()
{
  long **data = new long *[DATA_SIZE];

  for (size_t i = 0; i < DATA_SIZE; ++i)
  {
    data[i] = new long[BLOCK_SIZE];
    // do something with data[i]
    usleep(100000);
  }
  sleep(1);

  for (size_t i = 0; i < DATA_SIZE; ++i)
  {
    delete [] data[i];
  }
  delete [] data;
}
```

Listing1

```
 valgrind --tool=massif --time-unit=ms ./m1
==12939== Massif, a heap profiler
==12939== Copyright (C) 2003-2012, and GNU GPL'd, by Nicholas Nethercote
==12939== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==12939== Command: ./m1
==12939==
==12939==
```

Figure 3

```
desc: --time-unit=ms
cmd: ./m1
time_unit: ms
#-----------
snapshot=0
#-----------
time=0
mem_heap_B=0
mem_heap_extra_B=0
mem_stacks_B=0
heap_tree=empty
[content deleted]
#-----------
snapshot=49
#-----------
time=9830
mem_heap_B=784000800
mem_heap_extra_B=345752
mem_stacks_B=0
heap_tree=detailed
n2: 784000800 (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
 n0: 784000000 0x400759: main (m1.cpp:12)
 n0: 800 in 1 place, below massif's threshold (01.00%)
[more content]
```

Figure 4

```
    MB
763.3^                                                                       #
     |                                                             ::@:::::::#
     |                                                           ::: @:      #
     |                                                       ::::::: @:      #
     |                                                    :::::: ::: @:      #
     |                                                 :@::: ::: ::: @:      #
     |                                             :::::@::: ::: ::: @:      #
     |                                          ::::: ::@::: ::: ::: @:      #
     |                                      ::::::::: ::@::: ::: ::: @:      #
     |                                    :@: ::::::: ::@::: ::: ::: @:      #
     |                                 ::::@: ::::::: ::@::: ::: ::: @:      #
     |                             ::::: ::@: ::::::: ::@::: ::: ::: @:      #
     |                          :::::: : ::@: ::::::: ::@::: ::: ::: @:      #
     |                      :::::: ::: : ::@: ::::::: ::@::: ::: ::: @:      #
     |                   :::: :::: ::: : ::@: ::::::: ::@::: ::: ::: @:      #
     |                ::@: :: :::: ::: : ::@: ::::::: ::@::: ::: ::: @:      #
     |             :::: @: :: :::: ::: : ::@: ::::::: ::@::: ::: ::: @:      #
     |          :@@:: : @: :: :::: ::: : ::@: ::::::: ::@::: ::: ::: @:      #
     |      :::::@ :: : @: :: :::: ::: : ::@: ::::::: ::@::: ::: ::: @:      #
     |   ::@:: ::@ :: : @: :: :::: ::: : ::@: ::::::: ::@::: ::: ::: @:      #
   0 +----------------------------------------------------------------------->s
     0                                                                   11.13
```

Figure 5

on top of being a bit prettier than the ASCII drawings, allows you to interact with the snapshots. However, it does not show stack memory if you use the **`--stacks=yes`** option. This leads to an example that does use stack profiling. Listing 3 is the code (from Bertrans Meyer's *A Touch of Class*, p. 477, translated from Eiffel to C++). This is a nice example of stack use that is hard to analyze.

Profiling this with **`valgrind --tool=massif --stacks=yes ./bizarre`** gives Figure 9.

In one last example, let's see what happens if we use some low level allocation like mmap. (Listing 4.)

With this, all I see is the memory allocated for 'data'. (I haven't included it here as it's just an ASCII rectangle showing 808 bytes allocated and a time of 11.14s). Where have the other 763MB gone? By default, massif does not trace memory allocated in low level functions like mmap. If you want to see that memory (which might be the case if you are using a custom allocator that is based on mmap), then you will need to add the **`--pages-as-heap=yes`** option. If I add this option to the above example, then I see all of the memory being allocated (in fact, 777.5MB, since it now includes memory allocated when loading the application).

Note that normally on Linux, glibc will use mmap for malloc requests to allocate above a certain threshold, but Valgrind intercepts the call to

```
 39          7,825      624,275,992      624,000,800       275,192            0
99.96% (624,000,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.96% (624,000,000B) 0x400759: main (m1.cpp:12)
|
->00.00% (800B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 40          7,926      632,279,520      632,000,800       278,720            0
 41          8,126      648,286,576      648,000,800       285,776            0
[content deleted]
 48          9,530      760,335,968      760,000,800       335,168            0
 49          9,830      784,346,552      784,000,800       345,752            0
99.96% (784,000,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.96% (784,000,000B) 0x400759: main (m1.cpp:12)
|
->00.00% (800B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n       time(ms)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 50          9,931      792,350,080      792,000,800       349,280            0
 51         11,134      800,353,608      800,000,800       352,808            0
99.96% (800,000,800B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.96% (800,000,000B) 0x400759: main (m1.cpp:12)
|
->00.00% (800B) in 1+ places, all below ms_print's threshold (01.00%)
```

Figure 6

```
// m2.cpp
#include <unistd.h>

const size_t DATA_SIZE = 100U;
const size_t BLOCK_SIZE = 1000000;

int main()
{
  long **data = new long *[DATA_SIZE];

  for (size_t i = 0; i < DATA_SIZE; ++i)
  {
    data[i] = new long[BLOCK_SIZE];
    // do something with data[i]
    usleep(100000);
    delete [] data[i];
    usleep(100000);
  }

  sleep(1);

  delete [] data;
}
```

Listing 2

malloc, so there is no need to use **--pages-as-heap** unless you are using mmap directly.

That just about wraps up this installment. As you can see, Massif is straightforward to use and it presents a simple view of the memory use of your application.

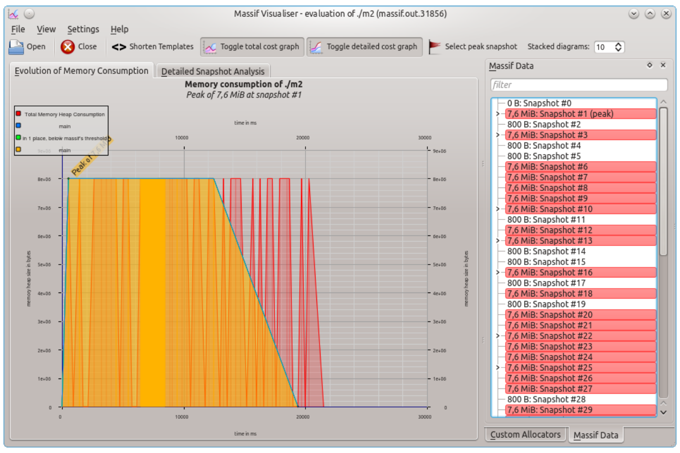
In my next article, I'll cover two Valgrind tools for detecting thread hazards, Helgrind and DRD. ■

```
// bizarre.cpp
#include <iostream>
#include <cassert>

using std::cout;

long int bizarre(long int n)
{
  assert(n >= 1);
  if (n == 1)
  {
    return n;
  }
  else if (0L == n%2L)
  {
    return bizarre(n/2L);
  }
  else
  {
    return bizarre((3L*n + 1L)/2L);
  }
}

int main()
{
  for (long int i = 1L; i < 200000000L; ++i)
  {
    bizarre(i);
  }
}
```

Listing 3

```
    MB
7.634^#::@@::@::::::@::  @@:::::@@    : @:::    ::::@::    ::::::   :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
     |#: @ : @ : : :@::  @ ::: :@     : @::     ::: @:     :: ::    :  :
    0 +---------------------------------------------------------------------->s
      0                                                                  21.15
```

Figure 7


Massif Visualiser - evaluation of ./m2 (massif.out.31856)
File View Settings Help
Open Close Shorten Templates Toggle total cost graph Toggle detailed cost graph Select peak snapshot Stacked diagrams: 10
Evolution of Memory Consumption
Detailed Snapshot Analysis
Memory consumption of ./m2
Peak of 7,6 MiB at snapshot #1
Total Memory Heap Consumption
main
in 1 place, below massif's threshold
main
time in ms
memory heap size in bytes
Massif Data
filter
Custom Allocators
Massif Data

0 B: Snapshot #0
7,6 MiB: Snapshot #1 (peak)
800 B: Snapshot #2
7,6 MiB: Snapshot #3
800 B: Snapshot #4
800 B: Snapshot #5
7,6 MiB: Snapshot #6
7,6 MiB: Snapshot #7
7,6 MiB: Snapshot #8
7,6 MiB: Snapshot #9
7,6 MiB: Snapshot #10
800 B: Snapshot #11
7,6 MiB: Snapshot #12
7,6 MiB: Snapshot #13
800 B: Snapshot #14
800 B: Snapshot #15
7,6 MiB: Snapshot #16
800 B: Snapshot #17
7,6 MiB: Snapshot #18
800 B: Snapshot #19
7,6 MiB: Snapshot #20
7,6 MiB: Snapshot #21
7,6 MiB: Snapshot #22
7,6 MiB: Snapshot #23
7,6 MiB: Snapshot #24
7,6 MiB: Snapshot #25
7,6 MiB: Snapshot #26
7,6 MiB: Snapshot #27
800 B: Snapshot #28
7,6 MiB: Snapshot #29

Figure 8

```
// m3.cpp
#include <unistd.h>
#include <sys/mman.h>

const size_t DATA_SIZE = 100U;
const size_t BLOCK_SIZE = 1000000;

int main()
{
  long **data = new long *[DATA_SIZE];

  for (size_t i = 0; i < DATA_SIZE; ++i)
  {
    data[i] = reinterpret_cast<long *>(mmap(NULL,
       BLOCK_SIZE*sizeof(long),
       PROT_READ | PROT_WRITE,
       MAP_SHARED | MAP_ANONYMOUS, -1, 0));
    // do something with data[i]
    usleep(100000);
  }

  sleep(1);

  for (size_t i = 0; i < DATA_SIZE; ++i)
  {
    munmap(data[i], BLOCK_SIZE*sizeof(long));
  }

  delete [] data;
}
```

Listing 4

```
   KB
18.77^                     #
     |                     #
     |                     #
     |                     #
     |                     #
     |                     #
     |    @                #
     |    @                #
     |    @                #
     |    @                #
     |    @                #
     |    @                #
     |    @                #
     |    @                #
     |    @                #                                        :    :
     |    @                #                                        :  :::  ::
     |    @                #       ::                      :  @@    :  ::: :::
     |    @  ::     ::: ::: #       :  :::    :: :  :: @@ :::  @ ::  :::::::@::
     |    @  : :@@:::  ::  #     :::  :: ::: :  :  :  @ @: :  @ ::  : :::::@:@
     |   :@: : :@ :::  ::  #:::: ::: ::: : :::  : :: :@ @: :::@ ::  : :::::@:@
   0 +----------------------------------------------------------------------->Gi
     0                                                                   548.2
```

Figure 9