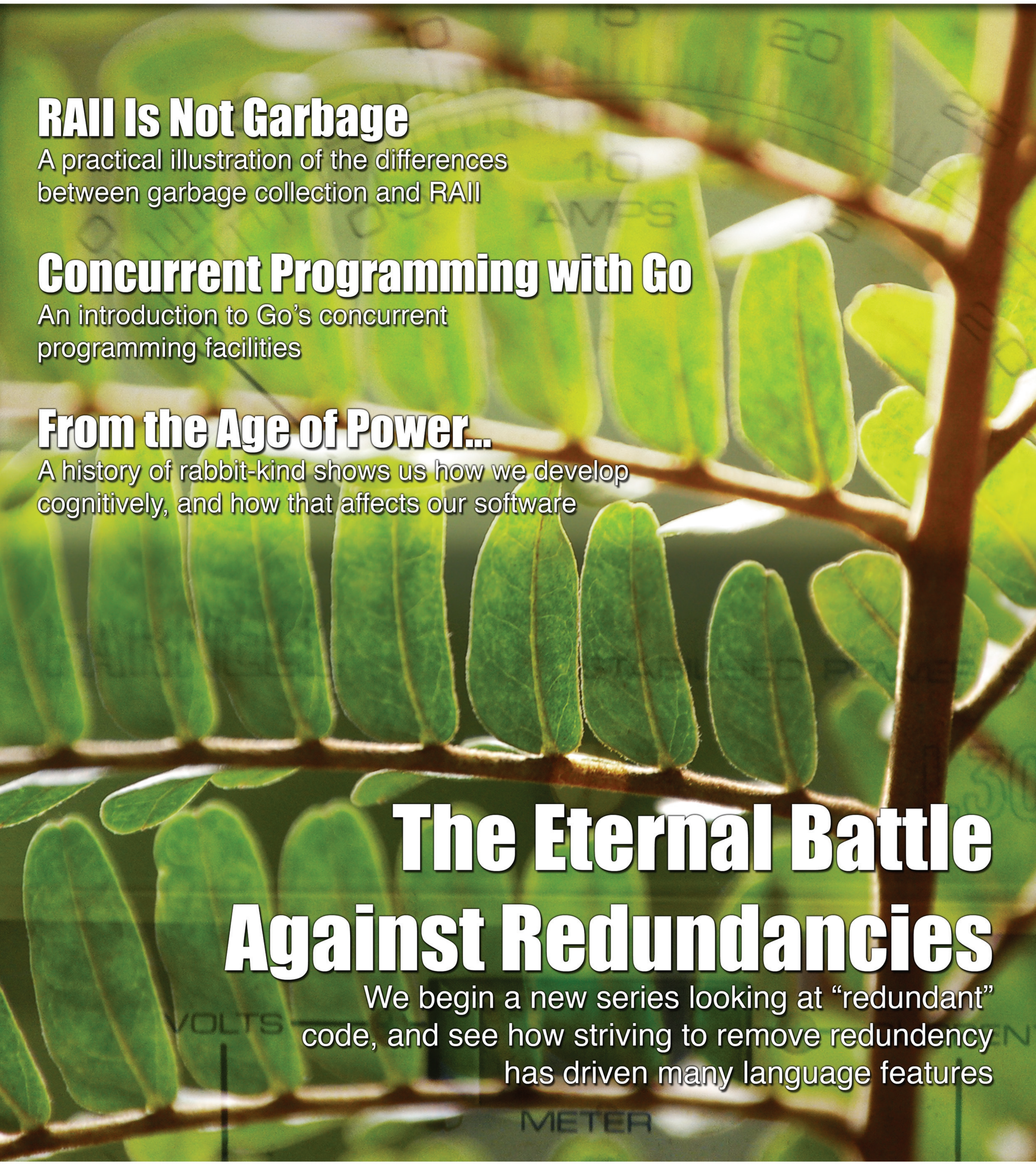
overload 106

DECEMBER 2011 £3



RAII Is Not Garbage

A practical illustration of the differences between garbage collection and RAII

Concurrent Programming with Go

An introduction to Go's concurrent programming facilities

From the Age of Power...

A history of rabbit-kind shows us how we develop cognitively, and how that affects our software

The Eternal Battle Against Redundancies

We begin a new series looking at "redundant" code, and see how striving to remove redundancy has driven many language features

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 106

December 2011

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Richard Blundell
richard.blundell@gmail.com

Matthew Jones
m@badcrumble.net

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 107 should be submitted by 1st January 2012 and for Overload 108 by 1st March 2012.

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

ACCU

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

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

Patently Ridiculous!

Software patents have a chequered history. Ric Parkin looks at some of the problems.

Anyone who has followed the technical news over the last decade or so will have surely have thought that the whole area of software patents is fraught with difficulty, and has led to some cases of sheer absurdity. But is it just me or have things become worse in the last year or so, with news of lots of legal spats over supposed patent infringing?

Disclaimer: as with any legal issues, this whole area is a minefield and should be treated with extreme caution. I've checked things as well as I can but there are bound to be mistakes, misinterpretations, and subjective opinions. If you want proper advice, be prepared to pay for an expert!

What is a patent?

> It consists of a set of exclusive rights granted by a sovereign state to an inventor or their assignee for a limited period of time in exchange for the public disclosure of an invention. [Patent]

Let's examine the important parts

- exclusive rights

  Exclusivity is important – no one else will have these rights. What these rights are will vary between jurisdictions, but the basic idea will be that no one else can use the invention without your permission.
- granted by a sovereign state

  This is really so that there is some legal framework. In particular, this means the legal system will back you up if someone steals your idea or violates some of the rights granted. Note though that this is limited to a particular legal framework – unless there's agreement between different systems, there may well be places that do not recognise these rights.
- an inventor or their assignee

  You can keep these right to yourself, or transfer them, perhaps by licensing or selling outright. Who the 'inventor' is depends – I expect that in the software industry most employment contracts will have a section saying that anything you invent belongs to the company. Some even try to include anything you do outside of your work, although I'm not sure how enforceable that is. There is some variation about who the 'inventor' can actually be – some jurisdictions say that whoever had the idea first is entitled to the patent, which is why it is a good idea to document your engineering ideas as much as possible, so it can be used to back up a claim. However, others go with the first person to apply for the patent, which could in fact lead to an unpatented idea being patented later by someone else, resulting in the original inventor infringing the patent!
- limited time

  These rights are yours, but it is not open ended. What is a suitable timescale is debatable, however.
- in exchange for the public disclosure

  This is the quid pro quo for the granting of the rights – once you've reaped your reward from your idea, it is in the public domain and can now be used by anyone.
- of an invention

  You have to invent something. So it has to be new and never done before, right? Unfortunately this seems to be the most awkward area. Some definitions use phrases like 'novel', 'non-obvious' or 'inventive' to define an invention worthy of such rights. There are also other ideas, such as checking against 'prior art', which roughly means if an idea can be shown to have existed before, and is perhaps widely used already, then you cannot patent it. There's also variation in how rigorously these checks are performed – some places do a proper search for pre-existing patents and prior art, others do much less and assume a legal challenge will be made if there's a dispute. There's also the subtle issue of whether the subject matter is patentable – many restrict it to a physical implementation and exclude abstract ideas, algorithms, and software. But this varies wildly, and is still in flux [SubjectMatter].

So why do patents exist? What purpose do they serve?

From the point of view of the inventor the important thing is that they provide guaranteed ownership rights backed by the relevant legal framework. This gives an incentive to actually try and invent things as you can exclusively use your idea how you like – manufacture and sell it, or license the right to do so, or just give it away – with the protection that others can't steal it. It also encourages you to tell people about it and get it used as widely used as possible – perhaps in return for a fee each time – rather than try and keep it secret so that only you can use it.

From the point of view of the wider society, it encourages people to invent so you get more innovation. But also it encourages a faster and more open spread of good ideas, especially if the inventor is not themselves in a position to use their idea directly. And once the exclusivity expires, the invention is now available for other to exploit, driving down the cost of using it or allowing further uses and refinements.

To illustrate these aspects, consider the Chamberlen family of surgeons. [Chamberlen]. Around 1634 or so, they invented the forceps, and successfully used them to assist in difficult childbirths. Rather than disseminate this discovery so it could be widely used, they kept it a carefully guarded secret (to the extent that the mother would be blindfolded) for approximately 150 years! While this allowed them to have exclusive use of a lucrative invention, many more people's lives could have been saved if it had been more widely used. Lacking the

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

protection of their idea, it made sense to keep it secret. If they'd had a patent, it could have been licensed, manufactured and used widely, and they'd have still profited from their idea.

Contrast this with the current system of drug patents. It is extremely expensive to develop a new drug, mainly due to the huge effort involved in research and testing, especially when you take into account the amortized cost of the ones that fail. By allowing an exclusive patent for a period that money can be recouped. And once the patent has expired, generic copies can be made by others that drives the price down and spreads usage, especially in poorer parts of the world. This isn't perfect by a long shot though – for example this gives incentives to develop drugs for diseases that are relevant to the richer countries that can pay, rather than develop cheap fixes for poorer countries where there would be less profit. But despite such flaws the basic idea of the patent does seem to work to at least some degree.

Software patents

So how do these ideas map work in the world of software? The main issue is that people are still deciding what size is a suitable unit to be patented, or even if it should be. One problem is what is actually patented? The old restriction to the physical implementation doesn't work, and even *defining* a software implementation is difficult. Many software patents go for patenting an algorithm, often for a particular purpose. This can be very controversial though. It seems reasonable to me to be able to patent a complex, truly novel algorithm with useful non-trivial applications. Two examples quickly come to mind: complex encryption algorithms [RSA], and Penrose Tiles [Penrose]. (This last is actually quite controversial if you're a Platonist, but you can argue that the possible applications are indeed patentable.)

Poor examples abound however. There's the notorious patent of using `XOR` to add and remove a screen cursor [XOR], but while researching I found one that is quite frankly amazing. I'll quote the entire abstract:

> A computerized list is provided with auxiliary pointers for traversing the list in different sequences. One or more auxiliary pointers enable a fast, sequential traversal of the list with a minimum of computational time. Such lists may be used in any application where lists may be reordered for various purposes.

Sound familiar? The simplest implementation I can think of that this patent covers is `std::list`.

I hope the claimant submitted this either to show how bad the system is, or perhaps as a 'White Knight' to prevent anyone else getting the patent and trying to charge for it. Can you actually believe this was granted in April 2006! [Wang]

Unfortunately sorting out this mess can be very expensive. Take for example the JPEG format. [JPEG]. Lots of fees were paid, and lots of costly legal cases, and it's not really clear if anyone actually owned suitable patents.

And we seem to be in a period of particularly many loud and high-profile cases as major companies sue and counter-sue, especially over patents relating to smartphones and tablets. There are many examples, such as [Groklaw] and [Galaxy], and given the potential market and the spare money these companies have, I can't see it getting better any time soon.

Release cycles

Some of you might actually wonder what happens to produce an issue of Overload, and what the editor actually does. There are three main parts.

Part 1: getting articles

This is more of a rolling effort, as people submit articles every now and again or ask for feedback on an idea or early draft. As the new submission deadline approaches you get an idea of how much material you have, and perhaps ask around for more articles, or hunt interesting blog entries that can be turned into material. Sometimes a few emails to chase up promises are needed. I add all articles to a GoogleDocs spreadsheet (so I can update it from anywhere), tracking its status and any actions I need to do.

Part 2: reviewing the articles

This tends to happen after the 'new submission' deadline has passed. It consists of about three weeks of to-and-froing reading the articles, passing new versions to reviewers, and returning feedback to the authors. Sometimes an article is pretty much ready to be published, perhaps some rewriting is needed (usually to expand on an important point), and occasionally a heavy edit is needed, perhaps to polish up the language of a non-native speaker. Reviewing articles can be a bit of an effort, but the review team can help out here. But most important here is to be prompt in passing feedback and new versions. Finally, at this stage it's also a good idea to get the author biographies, and check that any graphics are suitable for publication (we print at a quite high DPI and many graphics produced onscreen tend to be a much lower resolution, so we need scalable vector graphics or high resolution bitmaps).

Phase 3: produce the magazine

Once the articles are ready, I zip them all up and send to our production editor. She does the magic of typesetting and laying out everything, and also casts a watchful eye for typos and odd sentences (it's amazing how many things you overlook because you've read an article multiple times, and seeing fresh in a new form can really expose flaws you'd missed). There tends to be a week where I hear little as the bulk of the work happens, and then there's a flurry of PDFs of the articles to proofread. This involves lots of checking for typos, looking for odd code line-wraps, suggestions for pull-quotes, straplines, and contents descriptions, and checking that everything lines up nicely and looks good. Pete, who came up with the graphic design style of the magazine, also helps with this, and in parallel produces the cover so needs to know which articles are in. While this is going on I also have to write the editorial – I should really start it earlier, but always seem to find ways to put it off! Fortunately the layout is relatively easy. A final proofread of the entire magazine, and it's off to the printers to be sent to the addresses provided by the membership secretary. A couple of weeks of quiet, and the next cycle starts again.

I've been doing this for nearly 4 years now (actually just over if you include an issue as guest editor as a try-out), and am thinking it's time to start looking for a new editor. It's good fun, not too much effort if you're organised and prompt, you get to talk to many big names in the industry, and you get some excellent Kudos and lines for your CV. So if you're interested or curious for more information, drop me a line.

References

[Chamberlen] http://en.wikipedia.org/wiki/Forceps_in_childbirth#History
[JPEG] http://en.wikipedia.org/wiki/JPEG#Patent_issues
[Galaxy] http://www.bbc.co.uk/news/business-15956275
[Groklaw] http://www.groklaw.net/articlebasic.php?story=2011111122291296
[Patent] http://en.wikipedia.org/wiki/Patent
[Penrose] http://en.wikipedia.org/wiki/Penrose_tiling and US patent 4133152
[RSA] http://www.rsa.com/rsalabs/node.asp?id=2326
[SubjectMatter] http://en.wikipedia.org/wiki/Patentable_subject_matter
[Wang] US patent Number 7028023
[XOR] Claims 10 and 11 of US patent 4197590

Moving with the Times

The ACCU is primarily a way for programmers to communicate. Alan Griffiths looks at its past, and speculates on its future.

I first became aware of the ACCU a very long time ago: in those dark days I didn't even have a personal email address, the world-wide-web hadn't yet taken off, and the primary means of communicating technical information was in print.

In a recent C Vu article Francis Glassborow has covered much of these early days of ACCU – or the 'C User Group (UK)' as it was called at first. I don't want to repeat too much of that, but I need to set some context for what I have to say. It is worth mentioning that in these early days the association was primarily its journal *C Vu* – there was no website, no Overload, and no conference. And *C Vu* survived only through the heroic efforts of its editor (Francis). I got involved, initially by republishing articles I'd written for internal consumption for my employer (with the employer's consent obviously).

Over the course of the 1990s the organisation grew. I can't now remember the order in which everything happened but fairly early on someone set up a website and mailing lists (on a box left under a desk at a university). Our membership got a boost when we absorbed the Borland C++ User Group (which became the 'C++ Special Interest Group' and provided a second journal: *Overload*). As the membership expanded, and existing members moved into new technology the 'C' roots became diluted and the 'Association of C and C++ Users' was born. Towards the end of that decade the spring conference got started, initially as a few talks and exhibits to make the AGM a bit more of an outing. We even, briefly, had an autumn conference too.

But Francis edited *C Vu*, managed printing of *C Vu* and *Overload*, handled the review books, chaired meetings, and organised the conference. All this thanks to Francis's abundant energy. The only problem with this was that even Francis could only do so much – and he was involved in nearly everything the organisation did. There were a few individuals who helped out: there was a someone handling the webserver and mailing lists, there was a separate editor for *Overload* – there was even, briefly an 'International Standards Development Fund' newsletter. But everything went through Francis – or it didn't get done. Eventually, even Francis reached a limit and the effect of this could be seen in the membership numbers: they stopped rising.

By the end of the 20th century the ACCU was doing as much as Francis could cope with (and his wife was begging people to take some of the load off his shoulders). When Francis announced he'd stand down I thought 'good – someone can reorganize things so that we don't burn one guy out'.

Alan Griffiths has been developing software through many fashions in development processes, technologies and programming languages. During that time, he's delivered working software and development processes to a range of organizations, written for a number of magazines, spoken at several conferences, and made many friends. He can be contacted at alan@octopull.co.uk.

I little thought that it would be me – until the AGM; when the election came and it became apparent that there was no candidate for Chair.

Something needed to be done and I took on the challenge. There was no way I was going to try to do everything, or be involved in everything. I chose to delegate. The next few years were spent building some teams to handle aspects of the organisations activities.

It wasn't always easy – people were used to Francis/the chair doing everything. But somehow things moved forward. John Merrills set up an editorial team for *Overload*; we separated out content editing for both *C Vu* and *Overload* content from the publication work (and then contracted out the publication and distribution). We formed a team to decide the conference content and outsourced the actual running of the conference. I even used one AGM to bully Paul Grenyer into joining the committee to represent the 'mentored developers' projects that he was so keen on ACCU providing.

Each of these teams was represented on the committee and I tried to avoid being directly involved in any of them. The committee were responsible to the membership for getting things done and I made sure that the committee was able to do its job. This model means that a lot of people contributed – I won't try to list them here.

We had a committee meeting every couple of months and most of the committee was able to report progress and go away with a couple of action points. These points were minuted and the minutes distributed in a timely manner so that people were aware of what was expected of them and others. You might think these were long 'iterations', but even the committee have to fit ACCU business around a full time job – the important thing is that they were long enough to get some stuff done and short enough to maintain momentum.

Things got done and the membership numbers started to rise again.

While there were many successes, one thing we attempted didn't go too well – and I feel it is time to mention it. We (the committee) decided the website needed to be revised and the content brought up to date. There was a lot of discussion about how to do this – I was very keen to find a way of producing interesting web content (the best thing we had then was our book review database). Others on the committee felt that the technology supporting the website needed to be replaced first and set about doing that.

Our first attempt at replacing the website was a failure. After this a team driven by Allan Kelly at least got as far as replacing the outdated technology. This took a lot of time and energy – so much so that I quit the chair and Allan the committee before the process of managing the website content got addressed. Sadly, it never has: the book review database has faced repeated problems, and the 'Overload online' content is buried and poorly indexed.

Despite problems with the website, many of the successes live on: teams are still doing the things they did then – there's still a conference team, *Overload* still has an editorial team, nowadays *C Vu* also has an editorial

the constitution only allows voting by those present at a general meeting ... the cost of voting can be a substantial trip. There has to be a better way!

team. We now outsource the web hosting (having replaced the box under a desk), journal publication and conference organisation. The committee is responsible for all of this – and, for the most part, follows the model of having a committee member heading each team.

The spring conference has gone from strength to strength, and is a much more significant part of our activities. We have a second conference again and there are some lively local groups organising regular meetings.

However, over the same period membership numbers have first levelled off and then declined.

The committee has once more decided that the technology behind the website needs to be changed. I'm in no position to argue with that, but we can't afford to repeat our past mistake – we also need to put in place a mechanism for creating and maintaining our web content. Our *Overload* articles and those in *C Vu* (since the new editorial process improved the quality) could form the basis of an attractive web resource.

There is a lot of work needed to make that happen – not just migrating the text and images from one form to another, but also making sure that links and indexes are put in place. That's far more than a part time job for a volunteer! We employ a professional to get the journal material into print, but a professional appearance on the internet is far more important!

Publishing the material we produce on the website does reduce the opportunity for 'selling' the journals. On the other hand the journals are our biggest cost and there's been discussion recently about whether the journals should be published electronically. I would be sad to see the printed version go – as that is still my preferred format for reading material. However, while I read the paper form I don't think that format is the way things are best disseminated these days.

Another thing that has changed is communications technology. The ACCU purports to be a world spanning organisation. Yet almost all of its activities are in the UK and, most particularly, the constitution only allows voting by those present at a general meeting (usually the AGM, but potentially a Special General Meeting). Even for those in the UK the cost of voting can be a substantial trip. There has to be a better way!

To address this there's a need for those not able to attend the AGM in person to be aware of the issues to be voted beforehand, and for them to attend remotely, pre-register votes or appoint proxies. Indeed, there are good reasons for informing members of constitutional motions before the AGM – that may effect a decision to attend and vote.

As a case in point, there was a very significant constitutional motion proposed at the last AGM, and were it not for Allan Kelly starting a discussion on accu-general beforehand the first anyone would have known would be when it was raised at the AGM. Even so, this motion was not even voted on – as people didn't feel informed enough to deal with it then and there.

In the early days of ACCU these constitutional requirements were unexceptional – the only forum for getting together was the AGM, so people made the effort. Physical meetings are less important now: email, blogs, mobile phones, video calls and other media are available and fewer people find the AGM as important as it was.

This doesn't mean I don't think face-to-face meetings are unimportant. In fact, I find it very worrying that the current committee has only met twice since the last AGM. (I also find it worrying that the minutes of the last meeting assign almost all of the 'actions arising' to two individuals.)

We need to understand what the ACCU offers today, and how best to deliver it. A recent survey found the following points (in no particular order):

1. Finding other people who will stimulate, enthuse or enable becoming a better programmer
2. Socialising with other geeks (preferably under the influence of alcohol)
3. Programming tips, techniques, craft and lore
4. Discussion of programming languages (except VB and, possibly, Perl but particularly C++)

In the past ACCU was the way for us to get things published, this is no longer true. There are a couple of recent cases that illustrate this: recently Olve Maudal published his 'Deep C' slides [Deep C], not through ACCU but on slideshare. Similarly, Paul Grenyer is talking of taking his 'Desert Island Books' to another forum. They have good reasons for their choices – but it shows that we've moved a long way from where the ACCU started, as the only way to get the news out.

The ACCU isn't dead yet, but it needs work to keep itself relevant.

That work has to be done by you – an ACCU member – working together with other members. The ACCU can be the forum for that work, but it needs to be updated. Don't expect the people in place to do more – they are already busy doing what they can. Don't even expect them to carry on – they get tired. If everyone does a little a lot can be done. ■

Reference

[Deep C] http://www.slideshare.net/olvemaudal/deep-c

The Eternal Battle Against Redundancies, Part I

The drive to remove redundancies is widely seen as a good thing. Christoph Knabe sees how it has influenced programming languages.

Since the beginning of programming, redundancies in source code have prevented maintenance and reuse. By ‘redundancy’ we mean that the same concept is expressed in several locations in the source code. Over the last 50 years the efforts to avoid redundancies [Wikipedia] have inspired a large number of programming constructs. This relationship is often not obvious to the normal programmer. Examples include relative addressing, symbolic addressing, formula translation, parameterizable subroutines, control structures, middle-testing loops, symbolic constants, preprocessor features, array initialization, user defined data types, information hiding, genericity, exception handling, inheritance, dynamic dispatch, aspect oriented programming, functional programming, and even program generators and relational databases. These constructs are discussed using examples from 14 widely used programming languages. Whosoever understands the common concept is well equipped for the future.

What can the Zuse computer say today?

In 1971 my (high) school inherited a 12-year-old Zuse 22 and I learned programming on it. In the rapidly moving computer domain we would usually consider using such obsolete technology to be a waste of time. But this has provided me with the background for a survey of the programming techniques developed over the last 50 years. The Zuse 22 of the German computer pioneer Konrad Zuse was one of the first mass produced computers in the world (55 machines was a lot in those days!). It had a highly economical construction and was programmed in a machine level language: the Freiburgian Code. The example program in table 1 adds the natural numbers from *n* decrementing to 1, and prints the result (tested by the Z22 simulator of Wolfgang Pavel [Pavel]). In practice only the contents of the Instruction column were punched onto paper tape and read by the computer as a program. The Address column indicates into which word the instruction was stored, and the Comment column corresponds to comments in contemporary languages.

The instructions can have symbolic, combinable operation letters : **B**=Bring, **A**=Add, **S**=Subtract, **T**=Transport, **U**=Umspeichern (store to), **C**=Const-Value, **PP**=if Positive, **E**=Execute from (go to), **D**=Drucken (print), **Z**=Stop. But the addressing was purely numeric with absolute storage addresses. Here the variables *i* and *sum* are stored at the addresses 2048 and 2049 respectively. The algorithm itself is stored from address 2050, where we jump back using the instruction `PPE2050` , if the value of *i* is still positive.

Redundancies appear here in the addresses: 2048 for *i* appears 4 times, 2049 for *sum* 3 times, 2050 for the beginning of the program and the loop twice explicitly and once implicitly (two cells after where the tape content is stored). As a consequence the program is neither relocatable in the working storage nor simply extendable. So there are big difficulties in its maintenance.

Christoph Knabe learned programming at high school on a discarded Zuse 22, studied computer science from 1972, worked as a software developer at www.psi.de, and since 1990 is professor of software engineering at the Beuth University of Applied Sciences Berlin www.bht-berlin.de. Scala is the 14th language in which he has programmed intensively.

Table 1

Address	Instruction	Comment
	T2048T	**T**ransport the following to words 2048 ff.
2048	10'	**i**: Initial value for i is n, here the natural number 10.
2049	0'	**sum**: Initial value is the natural number 0.
2050	B2049	**B**ring the sum into the accu(mulator).
2051	A2048	**A**dd i to the accu.
2052	U2049	Store (**U**mspeichern) accu to sum.
2053	B2048	**B**ring i into the accu.
2054	SC1	**S**ubtract the Constant value 1 from the accu.
2055	U2048	Store (**U**mspeichern) accu to i.
2056	PPE2050	If accu **P**ositive **E**xecute from (go to) 2050
2057	B2049	**B**ring sum into the accu.
2058	D	Print (**D**rucke) accu.
2059	Z0	Stopp
	E2050E	**E**xecute now from 2050

Relative and symbolic addressing

Progress came later for the transistorized Zuse 23 by the development of ‘Relative Addressing’. This enabled a programmer to write a subroutine as if it was located at address 0. A certain prefix instruction told the loading program to store the actual start address in a load-time base register, which was usually register 26. Appending `A26` to an address caused the loading program to add the content of register 26 to the value to form an absolute address before storing the instruction to be executed later. So when using relative addressing the conditional jump instruction to the beginning of the program in table 1 would be `PPE2A26` instead of `PPE2050`. By this means the program has become relocatable. Relative addressing was still very economic: it did not need more resources than the register 26 at load time.

True, relative addressing facilitates relocating a subroutine in working storage, but inside the subroutine it is as inflexible as absolute addressing. If we wanted to extend the example by inserting a prefix action before the calculation loop, we would have to shift the relative jump goal `2A26`, too. Thus ‘symbolic addressing’ was introduced with the Zuse 23 (sold from 1961). See the German programming manual for the Z23 [Zuse23] p. 65ff. The Z23 loading program substituted each bracketed identifier of up to 5 characters by its address. The program from table 1 could be rewritten with the symbolic addresses `(I)`, `(SUM)`, and `(BEGIN)` as in Listing 1.

Now it is possible to insert further instructions at any place without destroying the program. This improvement was such a big one that the assembler for the Siemens 2002 was named after this technique, PROSA (Programming with Symbolic Addresses). Necessary resources for symbolic addressing were an addressing program and a symbol table.

relative addressing facilitates relocating a subroutine in working storage, but inside the subroutine it is as inflexible as absolute addressing

Architecture of the Zuse 22

The design of the Z22 was finished by about 1955, and 55 machines of this type were produced. It formed a whole generation of computer specialists in central Europe. The Z22 was characterized by:

- hardware logic implemented by 600 tubes
- working storage: a magnetic drum of 8192 words @ 38-bit
- registers: a core memory of 14 words @ 38-bit
- peripheral storage: 5 hole punched paper tape
- console I/O: push buttons, glow-lamps, teletype with paper tape reader
- operating frequency: 3 kHz

An instruction consisted of 38 bits: 2 with the value 10, then 5 for conditions, 13 for operations, 5 for a register address, and 13 for a working storage address. Each of the condition and operation bits was programmed by a specific letter and switched a specific gate.

The registers could be accessed by their address, but some were used for special purposes by some operations. Access time for registers was always one CPU cycle, for drum words only if they were accessed in sequence.

Some Registers of the Z22

Number	Special Usage
2	Testable by P or Q if positive or negative
3	Overflow area for accumulator, last bit testable by Y
4	Accumulator, filled by B, added by A, etc., testable by PP etc.
5	Stores return address for subroutines, filled by F

The Zuse 23 of 1961 was logically equivalent, but was implemented using transistors. It had 40-bit words and up to 255 registers.

We are now acquainted with the most common procedure for redundancy elimination: The redundant code part gets a name and we use that name instead of the former, redundant code parts.

```
T2048T
(I) 10'
(SUM) 0'
(BEGIN) B(SUM)
A(I)
U(SUM)
B(I)
SC1
U(I)
PPE(BEGIN)
B(SUM)
D
Z0
E(BEGIN)E
```

Listing 1

Formula translation

In the beginning, technical and scientific calculations dominated computer applications. But as we can see in the words 2050…2053 of the Z22 example, a simple addition needed three instructions (bring, add, store). For the formula (*a*+*b*)*(*a*-*b*) we would need about 7 instructions. So the need quickly arose for simplifying formula calculations. This was enabled by formulae with variable identifiers, literals, operator signs, operator priorities, and parentheses. The programming language FORTRAN got its name by this feature (Formula Translator). Fortran I was defined in 1956. At that time there was no possibility of defining your own operators.

Subroutines

If you needed an instruction sequence several times, on the Zuse 22 you could jump there by using the call instruction **`F`** from different program locations. Besides doing the actual jump, this instruction loaded a ‘jump back’ instruction into register 5. That is why you had to begin each subroutine by copying the contents of register 5 to the end of the subroutine using a U-instruction. This assured a jump back to the calling location when reaching the end of the subroutine.

But often you don’t need identical, but only similar processing. In this case Freiburgian Code had the convention of reserving cells before the subroutine for arguments and results. These had to be filled by the caller before the call instruction and retrieved afterwards, respectively. Then it had to jump to the first instruction of the subroutine, which always had to be **`B5`** followed by a **`U`** with the address of the last instruction cell of the subroutine. So the program from listing 1, converted into a Zuse23 subroutine with entry address **`SUMUP`** for summing up the integer numbers from 1 to *n*, is shown in listing 2.

In order to print the sum of the numbers from 1 to 20, you could call **`SUMUP`** as follows:

```
BC20   U(N)   F(SUMUP)   B(SUM)   D
```

While this had to be obeyed as a convention on the Zuse 23, nowadays it is automated in all higher programming languages by the concept of a subroutine call with an argument list and return value. FORTRAN II and

```
T2048T
(N) 10'
(SUM) 0'
(SUMUP) B5
U(BACK)
B(SUM)
A(N)
U(SUM)
B(N)
SC1
U(N)
PPE(SUMUP)
(BACK)Z0
```

Listing 2

By combining these facilities you could construct arbitrarily complex algorithms

```
C     COMPUTES THE SUM OF THE NUMBERS FROM 1 TO N
      FUNCTION ISUMUP(N)
      ISUM = 0
      DO 99 I = 1, N
   99 ISUM = ISUM + I
      ISUMUP = ISUM
      RETURN
      END
```

Listing 3

Algol introduced the concept of named, parameterized subroutines around 1958. The possibilites for redundancy elimination were enormous and gave rise to the style of procedural programming. A subroutine in FORTRAN II for summing up the numbers from 1 to *n* by a counting loop is shown in listing 3. Identifiers starting with `I` to `N` are considered integers.

Control structures

The building blocks by which programs got their high flexibility and reusability were conditional jumps such as the `PPE` instruction of the Zuse 22. Conditional forward jumps could be used to implement conditional branches. A conditonal backward jump could be used to implement repetition, which would terminate when the condition no longer held. By combining these facilities you could construct arbitrarily complex algorithms. As conditional branches and limited repetitions with or without a control variable were needed frequently, the popular programming languages introduced such constructs. In FORTRAN I (1957) you could sum up the integer numbers from 1 to *n* by the following counting loop:

```
      ISUM = 0
      DO 99 I = 1, N
   99 ISUM = ISUM + I
C  HERE ISUM CONTAINS THE SUM OF THE NUMBERS
C  FROM 1 TO N
```

FORTRAN had half-symbolic addressing. A statement could be identified by a freely electable number, a so-called ‘label’. The statement `DO 99 I = 1, N` incremented the control variable `I` from 1 to `N`, and each time all statements up to and including the statement labeled by 99 are repeatedly executed. For branching FORTRAN I offered the arithmetic `IF`:

`IF` *(expression) negativeLabel,zeroLabel,positiveLabel*

This statement jumps to one of the three enumerated labels depending on the sign of the expression result.

Algol 60 had already introduced nowadays common control structures:

1. A multi-branches cascadable `IF`: `if` *cond* `then` *expr* `else` *expr*
2. A universal loop with control variable, for example:
   - `for i := 1 step 1 until 100 do print(i)` prints the numbers from 1 to 100
   - `for i := 1, i*2 while i<2000 do print(i)` prints the powers of 2 from 1 to 1024.

```
VAR
    x: integer;
    sum: integer := 0;
BEGIN
    readNumber(x);
    WHILE x>0 DO BEGIN
        sum := sum + x;
        readNumber(x);
    END;
    writeln;
    writeln('The sum is ', sum, '.');
END
```

Listing 4

Modern control structures with the purpose of being able to completely avoid jump instructions were introduced by Pascal (1970). Pascal distinguished the pre-testing `WHILE`-`DO`-loop, the post-testing `REPEAT-UNTIL`-loop, and the counting `FOR`-`DO`-loop. Nevertheless Pascal still contained the `GOTO` statement, as non-local termination could not be done otherwise

Unfortunately the `WHILE`-loop, planned for repetitions where the number of iterations is not known in advance, implies redundancies in the following use case, which occurs extremely often in practice: We want to process an unknown number of elements, maybe even none. The code in listing 4 reads positive numbers and prints their sum. The procedure `readNumber` is understood to deliver -1 if it can’t find any further number. The keywords are typed in upper case, although this is not important in Pascal.

We see, that the procedure `readNumber` has to be called redundantly: Once before the `WHILE`-loop, the second time at the end of the loop body, in order to prepare the variable `x` for the next test of the `WHILE`-condition.

That is why C (1973), Modula-2 (1978), and Ada (1980) introduced the possibility of leaving an arbitrary loop by a special jump instruction, in particular an endless loop. Using this we could solve the above task without redundancies (C syntax, listing 5).

This corresponds to the general pattern for processing an unknown-in-advance number of records in a middle-testing loop (listing 6).

Recursion: Recursion is when a function calls itself either directly or indirectly. The ability to do so was introduced by LISP and Algol at around the same time, but the then predominant languages FORTRAN and COBOL did not allow recursion. Some tasks, e.g. traversing trees, are most

```
int sum=0, x;
for(;;){
    readNumber(&x);
    if (x<=0) break;
    sum += x;
}
```

Listing 5

The ability for programmers to define their own control structures was born in LISP

```
initialize
for(;;){
  retrieve
  if (notSuccessful) break;
  process
}
```

Listing 6

elegantly expressed by recursion. Recursion does not directly contribute to the elimination of redundancies. If the recursive call is the last statement of a function, the compiler can replace it by a simple, storage-efficient loop. Compilers of functional programming languages regularly do this optimization.

User-defined control structures

The ability for programmers to define their own control structures was born in LISP (1958), as in this language instructions and data were notated in the same form, the so-called S-expressions. They also had the same internal representation. For example, **`(TIMES N 7)`** on the one hand means the multiplication of **`N`** by 7. On the other hand it is simply a list with the elements **`TIMES`**, **`N`**, and **`7`**. A function in LISP I, which was defined as **`FEXPR`** instead of the usual **`EXPR`**, had a special property. It evaluated its passed arguments not before executing its function body, but left this until the explicit usage of the function **`EVAL`** in the function body. So a function body could arbitrarily control the frequency and order of evaluation of its argments.

Later on when the power of this concept was found out, LISP dialects introduced comfortable notations to define **`FEXPR`**s. For example in MacLISP you could define an **`if-then-else`** construct, which was not contained in early LISP, by the traditional **`COND`** as follows:

```
(defun IF fexpr (args)
  (let ((predicate (car args))
        (then (cadr args))
        (else (caddr args)))
   (cond
    ((eval predicate) (eval then))
    (t (eval else)))))
```

The **`IF`** function gets all arguments unevaluated as a list with the name **`ARGS`**. **`LET`** assigns the individual arguments to the values **`PREDICATE`**, **`THEN`**, and **`ELSE`**. **`COND`** evaluates **`THEN`** or **`ELSE`** depending on the evaluation of **`PREDICATE`**. Example from Steele and Gabriel [LISP].

As a lover of redundancy-free solutions I missed middle-testing loops in the modern object-functional language Scala (2001). But I could define one myself (listing 7).

The function **`loopGetTestProcess`** has 3 parameter lists. In the first it expects an expression **`getItem`** for obtaining a next item, in the second a boolean function **`shouldContinue`** for judging the success of the obtainment, and in the third a function **`processItem`** for processing the obtained item. By using the generic parameter **`[Item]`** it is assured that

```
def loopGetTestProcess[Item]
    (getItem: => Item)
    (shouldContinue: Item=>Boolean)
    (processItem: Item=>Unit)
{
    var item = getItem
    while(shouldContinue(item)){
        processItem(item)
        item = getItem
    }
}
```

Listing 7

the types fit together. The redundant call of **`getItem`** is well encapsulated in the function body and is not visible in the using source code.

As syntactic sugar in Scala you can write an argument list of length 1 with curly brackets thus enabling the following usage:

```
def printLines(in: java.io.BufferedReader){
  loopGetTestProcess(in.readLine())(_!=null){
    println
  }
}
```

In Java this could be done by **`break;`** and would look like in listing 8.

The big achievement in Scala is that the programmer has defined this control structure himself and can define others too.

Constants

In working storage every cell is modifiable. That is why there were no constants in Freiburgian code. By around 1960, the predominant higher programming languages FORTRAN, Algol, and COBOL had only variables and literals, usable in expressions. e.g. the literal **`2`** in the FORTRAN expression **`A**2 + 2*A*B + B**2`** with **`**`** meaning 'to the power of'. A redundancy problem arises, if you must use the same literal several times. In COBOL 66 you could also use literals to dimension a variable, for example a west-German ZIP code, which consisted of four digits, as **`ZIP PICTURE 9(4)`**. But if you had done it this way at several locations in your program code, the reorganisation of the German postal systems in 1993 obliged you to change all these locations to five digits: **`ZIP PICTURE 9(5)`**. In the early higher programming languages there was no possibility to declare variable or array sizes free of redundancy.

```
void printLines(final java.io.BufferedReader
in){
    for(;;){
      final String line = in.readLine();
      if(line==null)break;
      println(line);
    }
}
```

Listing 8

Pascal (1970) solved this problem very cleanly by declarable, symbolic constants, which could be used for dimensioning arrays, as well. E.g.:

```
CONST zipLength: integer := 4;
TYPE ZipCode = PACKED ARRAY[zipLength] OF
 character;
VAR clientZip: ZipCode;
BEGIN
  FOR i := 1 TO zipLength DO write(clientZip[i]);
```

Pascal also introduced the concept of user-defined data types (here the type `ZipCode`). Along with the symbolic dimensioning constants this enabled you to define sizes for a whole software system free of redundancies.

C (1973) solved the dimensioning problem less elegantly, as you had to use preprocessor macros instead of symbolic constants. But on the other hand you could even do it without explicit size constants, if you used the `sizeof` operator. So the same thing expressed in C:

```
typedef char ZipCode[4];
ZipCode clientZip;
int i=0;
for(; i<sizeof(clientZip); i++){
  putchar(clientZip[i]);
}
```

C++ (1983) introduced the ability to declare frozen variables at any location in the program code by the keyword `const` and by that to build calculated 'constants'. Beginning with Java (1995) it became normal to dimension arrays only at runtime, or to work with the growable collections.

Preprocessor features

Often a preprocessor was used in order to avoid redundancies. So it is possible to include declarations, which are needed in the same wording in several compilation units, from one single file. This technique was used in COBOL (`COPY`), FORTRAN (`INCLUDE`), and C (`#include`). In this way you could build a data abstraction module in C. You had to write the function declarations of the module in its header file, and the function definitions along with the managed module variables in its implementation file. As an example see the header file `stack.h` for a stack of characters. The lines starting with `#` contain preprocessor directives.

```
#ifndef stack_h
#define stack_h

#define MAX_STACK_SIZE 10
void stack_push(char item);
char stack_pop(void);

#endif
```

Every client of this stack has to include the header file `stack.h` and then you can call the functions declared therein:

```
#include "stack.h"
...
stack_push('X');
```

About 1985–1995 this was the predominant modularization technique in industry, before it was superseded by object orientation.

Stringize Operator: The C/C++ preprocessor contains the `#` operator, which delivers the name of a given macro argument as a string. With this you can program without redundancies simple dialogs, e.g. for test purposes. So you can use in C++ the following macro `PROMPT_READ` in order to print the name of the passed variable, and to read a value into it afterwards:

```
#define PROMPT_READ(var) \
{ cout << #var << "? ";    cin >> var;}
```

Using this macro you can program a console dialog as in listing 9.

A typical dialog usage could look as follows:

```
name? Ulf
age? 22
Ulf is 22 years old.
```

```
  string name;
  int age;
  PROMPT_READ(name);
  PROMPT_READ(age);
  cout << name << " is " << age <<
     " years old." << endl;
```

Listing 9

Generally preprocessors are considered an inelegant solution. That is why the majority of cases for which you used the preprocessor in C or C++, are solvable in Java by its own language constructs without a preprocessor. However in Java the following is not possible:

- determination of the name of a variable as by the `stringize` operator in C
- delegation of the abortion of a method execution and some accompanying action, e.g. `errno=FAILURE;  return;` to another method
- delegation of a certain pattern of exception handling to another method.

  If we had C-like macros in Java we could write

  ```
  TRANSACTION(myAction1(); myAction2())
  ```

  which would be useful to transform a statement sequence into a database transaction by

  ```
  #define TRANSACTION(stmts) \
  {try{stmts; commit();}catch(Exception e) \
  {rollback(e);}}
  ```

Array initialization

Pascal introduced redundancy-free array dimensioning, but initialization still had a problem. As an example we want to declare and initialize an array in Pascal, which contains codes for three allowed actions. Even in modern GNU Pascal you have to indicate the array size 3:

```
VAR actions: ARRAY[1..3] OF char = ('G','P','D');
```

But the size is redundant in relation to the number of initialization elements. The more actions you need, the more difficult becomes the manual numbering.

C introduced to derive an array's size from the length of its initialization list: `static char actions[] = {'G','P','D'};`. Java inherited this useful feature.

Summary and prospects

The early higher programming languages introduced the possibility of avoiding redundancies by extraction, parameterization, and naming of a repeated code snippet. Such code snippets could be: addresses, values, size indications, declarations, and statement sequences. Some code patterns were invoked by syntactical units of the programming language, e.g. loops. In the better case the programmer could give a freely electable name to a code pattern. If a programming language helped to eliminate redundancies better than a competing language, this was a relevant advantage in the battle for dissemination.

In the 2nd part of this article we will deal with the more modern techniques 'information hiding', genericity, exception handling, object-oriented, aspect-oriented, and functional programming as well as domain specific languages and relational data bases, and how they all contribute to redundancy elimination. ■

References

[LISP] Steele/Gabriel: *The Evolution of LISP*, p. 9: http://www.dreamsongs.com/Files/HOPL2-Uncut.pdf

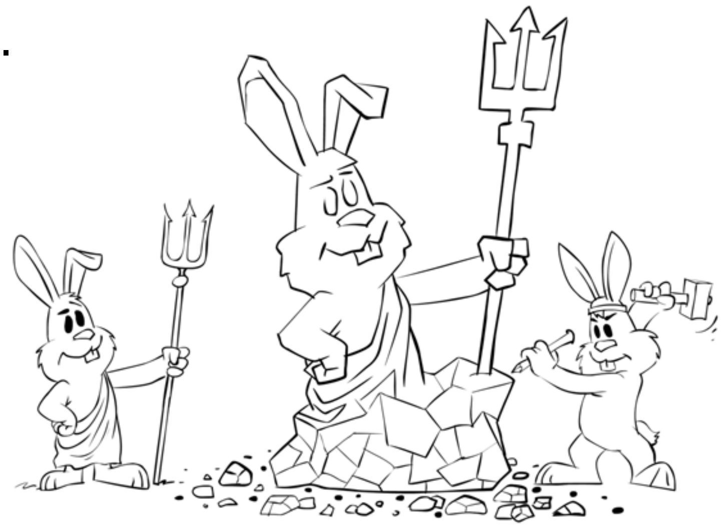
[Pavel] http://www.wpavel.de/zuse/simu/

[Wikipedia] http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[Zuse23] http://www.weblearn.hs-bremen.de/risse/RST/WS04/Zuse23/Z23Programmierungsanleitung.pdf

From the Age of Power to the Age of Magic and beyond...

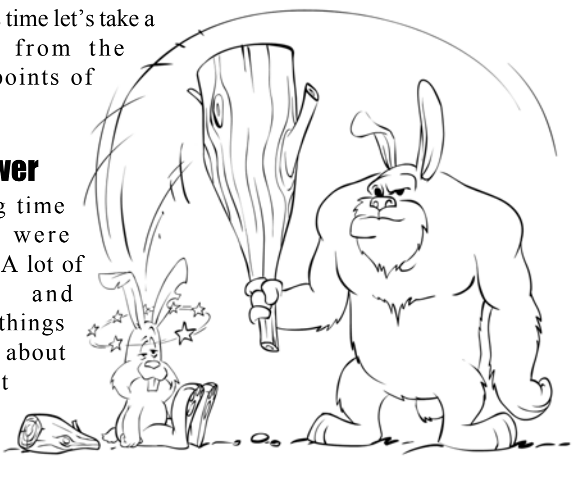
Certain abilities allowed some societies to dominate their peers. Sergey Ignatchenko takes a historical perspective on dominant societies.



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

Hray, u hraithile!
Run, for the thousandth time!

Sometimes, to get a better understanding of where we're standing now, it helps to take a look back, sometimes even a long while back, to the very beginnings of rabbitkind. The history of rabbitkind can be classified in a number of different ways, but this time let's take a look at it from the following points of view.



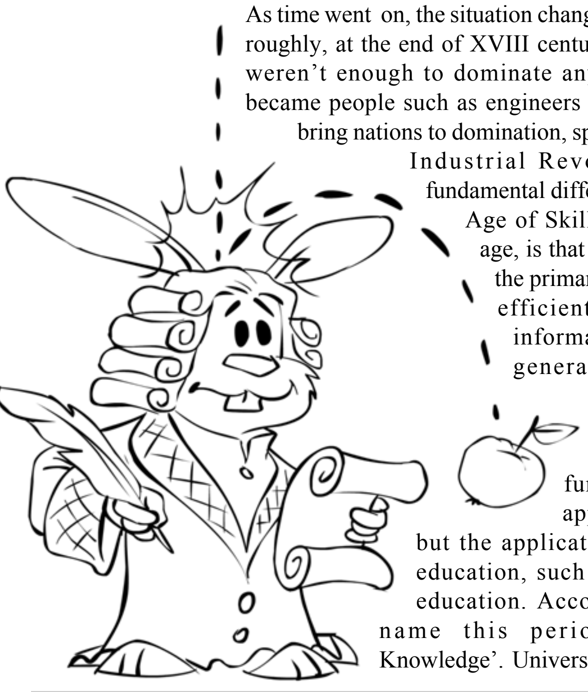
Age of Power

A very long time ago, there were caverabbits. A lot of different and interesting things can be said about them, but what is important for us now is which characteristic was the most important factor in the success of societies back there (we will not address individual success now, concentrating on societies). What we can say rather easily, is that back at the very beginning the most important property which has dominated the world, was power. The most physically powerful and aggressive rabbit tribes easily dominated the others. This raw power wasn't offset by other things like the quality of weapons – with the only weapons being a simple club, it was the rabbit who was able to carry a bigger one that won. Let's name this rough period in the rabbit's history an 'Age of Power'.

Age of Skill

As time has passed, raw power appeared to be somewhat affected by other things, such as military skill and quality of weapons. One prominent example of is ancient Greek and Roman rabbits – while they weren't more physically powerful than their neighbours, they still managed to dominate them mostly because of superior organizational and military skills. Later on, skills continued to affect societies a great deal, with one prominent example (in addition to an obvious Renaissance period) being the Dutch Golden Age of the XVI–XVII centuries, when tiny Netherlands had risen to one of superpowers of that time based mostly on promotion of various skills, in particular naval expertese. Let's name this period the 'Age of Skill'. One major characteristic of this Age is that skills were passed from one generation to another as a part of the process of informal education, mostly based on apprenticeships. It was also the way how skills were improved, and the process was rather slow by today's standards.

Age of Knowledge



As time went on, the situation changed again. Very roughly, at the end of XVIII century, mere skills weren't enough to dominate anymore, and it became people such as engineers who started to bring nations to domination, spearheading the Industrial Revolution. The fundamental difference between Age of Skill and this new age, is that in the new age the primary and the most efficient way to pass information between generations, and to develop skills and knowledge further, was not apprenticeships, but the application of formal education, such as university education. Accordingly, let's name this period 'Age of Knowledge'. Universities have made

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

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

a huge impact on the way skills and knowledge have improved; and one of major improvements achieved by universities was that a lot of people specializing in the same area were brought together, increasing both the exchange of ideas and level of competition (and competition is necessary to achieve top-level results, see, for example, [Parkinson72]). Accordingly, it was people with university education (mostly engineers) who were a symbol of Industrial Revolution and Age of Knowledge.

Age of... ?

...now British people often incorrectly use the term 'Engineer' to describe Plumbers and Mechanics.
~ Wikipedia

By the end of the XX century, things had changed once again. With the advent of the Internet, search engines (and later Wikipedia), knowledge as such has became easily available and therefore has lost its value (the model of supply and demand works surprisingly well even in such matters). Such loss of value is indicated by multiple symptoms, in particular by the loss of respect to engineers (who were once one of the symbols of the Age of Knowledge). In addition, similar to the way the creation of universities has allowed thousands of people to work together, now millions are collaborating at the same time, which again makes a significant impact on the way knowledge is accumulated and created. So, where we are heading now? What is indeed valuable in the modern world?

It looks that (though it is only a rather wild guess on my part) that what is of value now is the ability to combine knowledge from several different sources, to produce a new piece of knowledge. It might be a small step within the grand scheme of things, but I feel it is quite essential to justify calling it a new age. For the purposes of the rest of this article, let's take it as a working hypothesis. But how to name this new age? Age of Combining Knowledge is too bulky, Age of Combination is quite unclear, and Age of Combinatorics is outright misleading. Let's think about it a little bit...

Why not 'Age of Magic'?

Any sufficiently advanced technology is indistinguishable from magic.
~ Arthur C. Clarke

Similar to engineers being a symbol of Age of Knowledge, this new Age, where it is combining knowledge to create new knowledge which really counts, also has a symbol – it is us, software engineers (which, for the purposes for this article, includes both software developers and administrators). The aura around us, software engineers, at the boundary of the XX and XXI centuries is somewhat similar to the aura around engineers a century earlier – we are perceived as difficult to understand people with high salaries (and people are not sure if such salaries are justified, because they cannot understand us), who can do strange things which ordinary people cannot do. Let's try to look at us with the eyes of your neighbour (the one who has nothing to do with IT). If he takes a look (in the movies, or in real life), how will software engineer look to him? We're sitting typing some very strange stuff on our keyboards, mumbling some words which are completely incomprehensible for outsiders (and often the words are incomprehensible even for fellow software developers from another company), and from time to time achieving some things which look completely unrelated to our words or actions. But isn't it exactly the way mages are described in fantasy? They also mumble strange words, make strange gestures with their hands, and sometimes get results which they want – which seem completely unrelated to the nature of the words said or written. Aren't these two things the very same thing if observed from the outside? If you still have any doubts, think how stuff like

```
copy (v.begin(), v.end(),
   ostream_iterator<int>(cout, "\n"));
```

or

```
growisofs -Z /dev/cdrw -r -J /home/nobugs/
archive/20111029
```

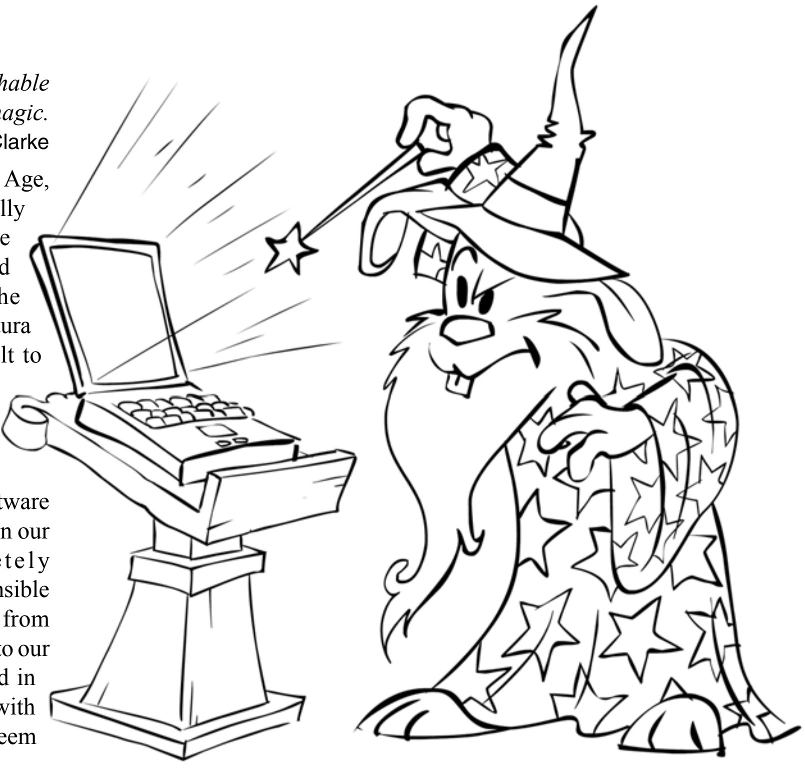
looks to your neighbour? If it is not a spell as described in a fantasy book, what is?

Based on the logic above, I hereby propose to name this new age as the Age of Magic.

Ethics of the Mages

Primum non nocere
~attributed to Thomas Sydenham

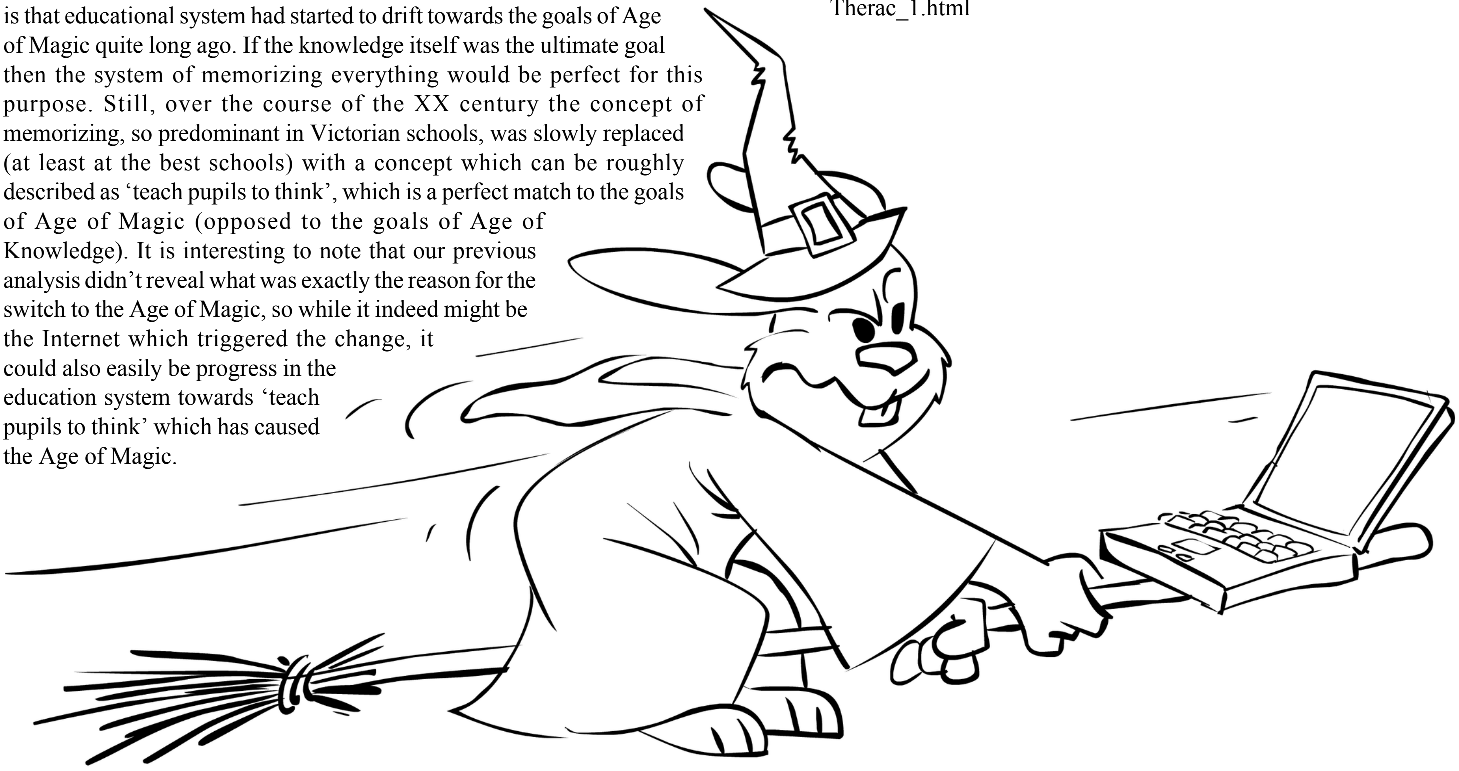
Whenever somebody has a power to affect others significantly, it is vital to have certain ethical standards to follow. Professionals of various flavours have had established ethical standards for centuries. The very first professional ethical standard in the history of rabbitkind is, most likely, the Hippocratic Oath, taken by doctors since around the 5th century BC. Much closer to the Age of Magic are the engineering codes of ethics, which originated in the US around 1910 after series of spectacular bridge disasters. It is interesting to note that the first information about an engineering code of ethics in the UK that I was able to find is from 1991, more than half a century later. The main reason behind these professional codes of ethics is to avoid potential harm which can arise from irresponsible actions of professionals (in the case of doctors it is potential death or harm to the patient, for engineers it is a potential failure like collapsed bridge). But is software any different in this regard? Bugs in software have already caused deaths [Therac-25] [Patriot], and have already caused the loss of huge amounts of money [Ariane 5], which already indicates the potentially dangerous impact of Mages on society, and therefore the need for a code of ethics. It has already been recognized, and associations like IEEE or BSC already have their codes, though quality of them may vary; still, what matters the most for the fellow Mage is not to follow instructions set in formal codes blindly, but to understand their responsibility to society.



Education in the 'Age of Magic'

The mind is not a vessel to be filled, but a fire to be kindled.
~ Plutarch, On Listening to Lectures

As it was noted above when discussing differences between the Age of Skill and Age of Knowledge, the way knowledge is passed between generations is critical for our analysis. The interesting thing about it is that educational system had started to drift towards the goals of Age of Magic quite long ago. If the knowledge itself was the ultimate goal then the system of memorizing everything would be perfect for this purpose. Still, over the course of the XX century the concept of memorizing, so predominant in Victorian schools, was slowly replaced (at least at the best schools) with a concept which can be roughly described as 'teach pupils to think', which is a perfect match to the goals of Age of Magic (opposed to the goals of Age of Knowledge). It is interesting to note that our previous analysis didn't reveal what was exactly the reason for the switch to the Age of Magic, so while it indeed might be the Internet which triggered the change, it could also easily be progress in the education system towards 'teach pupils to think' which has caused the Age of Magic.



In any case, what is rather obvious is that 'teaching pupils to think' (as opposed to 'make pupils memorize tons of stuff') is very important to ensure that society flourishes within the new Age of Magic. Moreover, it can be easily shown that such an approach not only makes societies succeed, but also makes individuals succeed within societies. Unfortunately, the whole system of formal exams is not a good fit with this concept [Hyman2009], which means that this approach, while popular in the best schools, is far from being universal, so we can conclude: 'if you want your children to succeed in life – find a school which will teach them to think'.

The final step? Of course not!

The next question which inevitably arises is: shall we expect that this new Age of Magic is the ultimate step in this direction? Of course not. The history of rabbitkind shows that there is no such thing as an 'ultimate step' in any direction and this area is not an exception. Later on, we can expect that simple combining knowledge won't be enough anymore, and that rabbitkind will move towards the next age, towards an Age of Creativity; at this point it is too early to say what exactly will be the next step on this way, so I just hope that I'll live long enough to see it. ■

References and further reading

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

[Ariane 5] 'The Ariane 5 bug and a few lessons' Les Hatton, Oakwood Computing, U.K. and the Computing Laboratory, University of Kent, UK. www.leshatton.org/Documents/Ariane5_STQE499.pdf

[Hyman2009] Peter Hyman. 'Drop GCSEs. We should be teaching our children to think' http://www.guardian.co.uk/commentisfree/2009/aug/16/peter-hyman-education-teaching-exams

[Loganberry2004] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[Patriot] 'Patriot Missile Software Problem' Andrew Lum http://sydney.edu.au/engineering/it/~alum/patriot_bug.html

[Parkinson72] *The fur-lined mousetrap*, C.N. Parkinson, 1972

[Therac-25] *An Investigation of the Therac-25 Accidents* Nancy Leveson, University of Washington; Clark S. Turner, University of California, Irvine http://courses.cs.vt.edu/cs3604/lib/Therac_25/Therac_1.html

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

RAII is not Garbage

Many think that Garbage Collection frees the programmer from cleanup tasks. Paul Grenyer compares and contrasts it with a classic C++ idiom.

RAII is the greatest contribution C++ has made to software development. ~ Russel Winder

Managed and non-managed programming languages have very different ways of approaching resource management. Ever since I created my first resource leak (Windows handles in an MFC application) I have been fascinated by resource and memory management and the ways of preventing leaks. In this article I am going to compare the ways a non-managed programming language like C++ manages resources compared to managed languages like Java and C#.

Resource Acquisition Is Initialisation

Resource Acquisition is Initialisation, which is often abbreviated to RAII, although badly named is as Dr. Winder says the greatest contribution C++ has made to software development. Unlike garbage collected languages memory in C++ is not cleaned up automatically. If you or something you are using allocates memory on the free-store, you or that other something must delete it when it's finished with. In his article 'Garbage Collection and Object Lifetime', Ric Parkin [Parkin] discusses the different ways memory is handled in C++ and C# in a reasonable amount of detail so I won't go into it too deeply here.

So what is RAII? It is a mechanism that is available as a direct result of C++ having classes with constructors and *automatic*, and *deterministic* destructors. It is probably best demonstrated with a simple example. The **`ScopedFile`** class in listing 1 allocates a C **`FILE`** pointer in its constructor and closes it in its destructor.

If an instance of **`ScopedCFile`** is created on the stack its destructor is guaranteed to be called when it goes out of scope. This is automatic and deterministic destruction. The destructor cleans up by closing the file. As the client of the **`ScopedCFile`** instance you do not have to take any action to ensure clean-up. Of course if you create the instance on the free-store, you become responsible for deleting it. Deleting the instance causes the destructor to be called and ensures clean-up as before. The calling of the destructor is still deterministic, but it is not longer automatic.

Smart pointers such as **`std::unique_ptr`** can be used to manage free-store memory in C++. They are generally stack based objects that employ RAII to make free-store based object deletion automatic. They are not usually restricted to dealing just with memory and can also deal with resources.

Of course the C++ standard library has its own file handling classes that do all of the resource handling for you so you don't actually need to write a **`ScopedCFile`** class in most cases.

Garbage collection and destructors

It seems to me that most modern languages are garbage collected. In fact I would go so far as saying that most languages have some sort of automatic

Paul Grenyer An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com.

```
class ScopedCFile {
private:
  FILE* file;
public:
  ScopedCFile(const std::string& filename) {
    file = fopen(filename.c_str(), "r");
    // ..
  }
  ~ScopedCFile() {
    // ..
    close(file);
  }
};
int main(int argc, char* argv[]) {
  ScopedCFile file(argv[1]);
  return 0;
}
```

Listing 1

memory management and C++ is an example of one of the very few languages that don't (leaving managed C++ to one side). Therefore I am going to pick two languages that I am familiar with, Java and C#, to demonstrate resource management in a garbage collected language.

In garbage collected languages *memory* deletion is done for you automatically. Objects are generally created on the heap, although there are some exceptions. Every so often the garbage collector is invoked and determines which objects are no longer referenced. These objects are then marked for deletion and subsequently deleted. The upshot of this is that after you create an object you don't have to worry about deleting it again as the garbage collector will do it for you. Sounds wonderful, doesn't it? The truth is it's not bad, but it's not perfect either. You have little control over when the garbage collector is called. Even when it is invoked directly the best you can hope for is that the runtime agrees it's a good time to garbage collect. This means that there is no way of determining when or even if an object will ever be destroyed.

What about destructors? C# has destructors and Java has finalizers. Both are methods that are called just before an object is deleted. Therefore it cannot be determined when or even if (in the case of Java) they will be called. C# destructors and Java finalizers are automatic, but *not* deterministic. That's not much good for resource clean up. If for example you're opening a lot of files you may well run out of operating system file handles before the garbage collector runs to free them all up again, if it runs at all.

So how do you make sure resources are cleaned up as soon as they are no longer needed? Both Java and C# support **`try-catch-finally`** (Listing 2).

Clean-up code is placed in the finally block so that even in the presence of exceptions resources are released. C# also has the **`IDisposable`** interface, which together with the **`using`** declaration provides a shorthand for **`try-catch-finally`**.

```
public static void main(String[] args)
   throws IOException
{
  InputStream inputStream =
     new FileInputStream(args[1]);
  try
  {
    // ..
  }
  catch(Exception e)
  {
    // ..
  }
  finally
  {
    inputStream.close();
  }
}
```

Listing 2

```
static void Main(string[] args)
{
  using(var fileStream =
        new FileStream(args[1], FileMode.Open))
  {
    // ..
  }
}
```

In the newly released Java 7 try has been overloaded to provide a similar short hand.

```
public static void main(String[] args)
   throws IOException
{
  try(InputStream inputStream =
      new FileInputStream(args[1]))
  {
    // ..
  }
}
```

RAII vs try-catch-finally

Despite five wonderful years of C++, I am now a big fan of memory managed languages. That's not quite the same as being a fan of garbage collection. What I like is being able to allocate *memory* without having to worry about deallocating it. However the non-deterministic destruction behaviour of garbage collected languages pushes the responsibility for cleaning up *resources* from the encapsulating class to the client of the class.

As I have shown, when using RAII in C++, no clean-up is required by the client:

```
int main(int argc, char* argv[])
{
  ScopedCFile file(argv[1]);
  return 0;
}
```

but in a garbage collected language such as C# clean-up code must be written by the client or left for finalisation, which is never a good idea:

```
static void Main(string[] args)
{
  using(var fileStream = new FileStream(args[1],
        FileMode.Open))
  {
    // ..
  }
}
```

Therefore RAII is the clear winner in terms of resource management, as the client is not relied upon to notice that the class they are using requires clean-up. It also keeps the client code cleaner and more readable as its intention is not littered with code that serves no purpose other than to clean-up.

```
static class FileStreamTemplate {
  public static void Execute( string filename,
     FileMode fileMode,
     Action<FileStream> action) {
    using (var fileStream =
       new FileStream(filename, fileMode)) {
      action(fileStream);
    }
  }
}
static void Main(string[] args) {
  FileStreamTemplate.Execute(args[1],
     FileMode.Open,
     delegate(FileStream fileStream)   {
    // ..
  };
}
```

Listing 3

Execute Around Method

There are ways to move the resource handling from the client into an encapsulating class and Kevlin Henney discusses this in his article 'Another Tale of Two Patterns' [Henney]. The basic idea is that you have a class that manages the resource and passes it to another class, a delegate or something to use it. In the C# example in Listing 3, the **`execute`** method creates a **`FileStream`** object and passes it to a delegate. The implementation of the delegate does what needs to be done with the **`FileStream`** object and when it returns the **`execute`** method cleans up.

For such a simple example as a **`FileStream`** the only advantage is making sure the client cannot forget to clean-up and a timely manner. With more complex resources such as the various classes that collaborate together to access a database the boilerplate becomes a far more useful encapsulation.

Although Execute Around Method in garbage collected languages solves the same problem as RAII In C++ it is still inferior due to the amount of boilerplate required and the complexity and verbosity of the client code.

Finally

It is clear that the encapsulated resource management provided by RAII in C++ is vastly superior to the client responsible approach of the try-catch-finally pattern in garbage collected languages like Java and C#. I would go so far as to say that the language designers of garbage collected languages have dropped the ball where resource management is concerned. In fact C# **`using`** was added as an afterthought. Try-catch-finally is very much treating a symptom, rather than fixing the problem.

This is about the third version of this article. In the previous version I fell into the trap of trying to provide a solution, when really that is an article all to itself. When I started this article I set out only to highlight that what many people see as good enough in garbage collected languages is actually nowhere near enough. Yet all is not lost. In a follow up article I'll describe some of the concerns in creating a solution to encapsulated resource management in garbage collected languages. ■

References

[Henney] Another Tale of Two Patterns: http://www.two-sdg.demon.co.uk/curbralan/papers/AnotherTaleOfTwoPatterns.pdf

[Parkin] Garbage Collection and Object Lifetime: *Overload 68* http://accu.org/index.php/journals/244

Acknowledgements

Thank you to Russel Winder for inspiration and Ric Parkin for review. Further thanks go to Phil Nash, Alan Stokes and Nick Butler.

Why Polynomial Approximation Won't Cure Your Calculus Blues

We're still trying to find a good way to approach numerical computing. Richard Harris tries to get as close as possible.

We began this arc of articles with a potted history of the differential calculus; from its origin in the infinitesimals of the 17th century, through its formalisation with Analysis in the 19th and the eventual bringing of rigour to the infinitesimals of the 20th.

We then covered Taylor's theorem which states that, for a function *f*

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

where *f′*(*x*) stands for the first derivative of f at *x*, *f″*(*x*) for the second and $f^{(n)}(x)$ for the *n*th with the convention that the 0th derivative of a function is the function itself.

We went on to use it in a comprehensive analysis of finite difference approximations to the derivative in which we discovered that their accuracy is a balance between approximation error and cancellation error, that it always depends upon the unknown behaviour of higher derivatives of the function and that improving accuracy by increasing the number of terms in the approximation is a rather tedious exercise.

Of these issues, the last rather stands out; from *tedious* to *automated* is often but a simple matter of programming. Of course we shall first have to figure out an algorithm, but fortunately we shall be able to do so with relatively ease using, you guessed it, Taylor's theorem.

Finding the truncated Taylor series

The first step is to truncate the series to the first *n* terms and make the simple substitution of *y* for *x*+*δ*, giving

$$\begin{aligned} f(y)&\approx f(x)+(y-x)\times f'(x)+\\ &\quad\tfrac{1}{2}(y-x)^2\times f''(x)+\ldots+\tfrac{1}{n!}(y-x)^n\times f^{(n)}(x)\end{aligned}$$

or

$$\begin{aligned} (y-x)&\times f'(x)+\tfrac{1}{2}(y-x)^2\times f''(x)+\ldots\\ &+\tfrac{1}{n!}(y-x)^n\times f^{(n)}(x)=f(y)-f(x)\end{aligned}$$

to *n*th order in *y*-*x*.

Next we evaluate *f* at *n* points in the vicinity of *x*, say y_1 to y_n, yielding

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

$$\begin{aligned} (y_1-x)&\times f'(x)+\tfrac{1}{2}(y_1-x)^2\times f''(x)+\ldots\\ &+\tfrac{1}{n!}(y_1-x)^n\times f^{(n)}(x)=f(y_1)-f(x)\\ (y_2-x)&\times f'(x)+\tfrac{1}{2}(y_2-x)^2\times f''(x)+\ldots\\ &+\tfrac{1}{n!}(y_2-x)^n\times f^{(n)}(x)=f(y_2)-f(x)\\ \ldots&\\ (y_n-x)&\times f'(x)+\tfrac{1}{2}(y_n-x)^2\times f''(x)+\ldots\\ &+\tfrac{1}{n!}(y_n-x)^n\times f^{(n)}(x)=f(y_n)-f(x)\end{aligned}$$

Now the only unknowns in these equations are the derivatives of *f* so they are effectively a set of simultaneous linear equations of those derivatives and can be solved using the standard technique of eliminating variables.

By way of an example, consider the equations

$$\begin{aligned} E_1&:\ 2x+4y+3z=20\\ E_2&:\ 3x+2y+2z=13\\ E_3&:\ 4x+3y+4z=21\end{aligned}$$

To recover the value of *x*, we begin by eliminating *z* from the set of equations which we can do with

$$\begin{aligned} E_4&=E_1-\tfrac{3}{4}\times E_3:\ -x+\tfrac{7}{4}y=\tfrac{17}{4}\\ E_5&=E_2-\tfrac{1}{2}\times E_3:\quad x+\tfrac{1}{2}y=\tfrac{5}{2}\end{aligned}$$

Next, we eliminate *y* with

$$E_6=E_4-\tfrac{7}{2}\times E_5:\ -\tfrac{9}{2}x=\tfrac{17}{4}-\tfrac{35}{4}=-\tfrac{18}{4}=-\tfrac{9}{2}$$

and hence *x* is equal to 1.

At each step in this process we transform a system of *n* equations of *n* unknowns into a system of *n*-1 equations of *n*-1 unknowns and it is therefore supremely well suited to a recursive implementation, as shown in listing 1.

The first thing we need to do before we can use this to compute the derivative of a function is to decide what values we should choose for each y_i.

We can do this by extrapolating our analysis of finite differences; for an nth order approximation, we should choose

$$O(y_i-x)=\varepsilon^{\frac{1}{n+1}}$$

to yield an error of order $\varepsilon^{\frac{n}{n+1}}$.

Following a similar argument to that we used to choose the values of *δ* for our finite difference approximations, we shall choose

$$y_i=x+i\times\left((n+1)\times\varepsilon\right)^{\frac{1}{n+1}}\times\left(|x|+1\right)$$

as we increase the order of our polynomials the **number of accurate digits grow** exactly as expected

```
template<class T>
T
solve_impl(std::vector<std::vector<T> > &lhs,
           std::vector<T> &rhs,
           const size_t n)
{
  for(size_t i=0;i!=n;++i)
  {
    for(size_t j=0;j!=n;++j)
    {
      lhs[i][j] -= lhs[i][n]*lhs[n][j]/lhs[n][n];
    }
    rhs[i] -= lhs[i][n]*rhs[n]/lhs[n][n];
  }
  return n!=0 ? solve_impl(lhs, rhs, n-1)
              : rhs[0]/lhs[0][0];
}
template<class T>
T
solve(std::vector<std::vector<T> > lhs,
      std::vector<T> rhs)
{
  if(rhs.empty())
     throw std::invalid_argument("");
  if(lhs.size()!=rhs.size())
     throw std::invalid_argument("");
  for(size_t i=0;i!=lhs.size();++i)
  {
    if(lhs[i].size()!=rhs.size())
       throw std::invalid_argument("");
  }
  return solve_impl(lhs, rhs, rhs.size()-1);
}
```

Listing 1

```
template<class F>
class polynomial_derivative
{
public:
  typedef F function_type;
  typedef typename F::argument_type argument_type;
  typedef typename F::result_type result_type;

  polynomial_derivative(const function_type &f,
                        unsigned long n);

  result_type operator()(
     const argument_type &x) const;

private:
  function_type f_;
  unsigned long n_;
  argument_type ef_;
};
```

Listing 2

```
template<class F>
polynomial_derivative<F>::polynomial_derivative(
  const function_type &f, const unsigned long n)
: f_(f),
  n_(n),
  ef_(pow(argument_type(n+1)*
          eps<argument_type>(),
          argument_type(1)/argument_type(n+1)))
{
  if(n==0) throw std::invalid_argument("");
}
```

Listing 3

Polynomial derivative approximation

Listing 2 gives the class definition for a polynomial derivative approximation function object based upon these observations.

The constructor is relatively straightforward, as shown in listing 3. Note that the **epsilon** function is represented here by the typesetter-friendly abbreviation ***eps*****<T>**).

Once again we assume that we have specialisations of both **std::numeric_limits** and **pow** for the function's argument type.

Listing 4 provide the definition of the function call operator.

Note that if the argument and result types are different we shall very likely introduce a potential source of numerical error. Consequently we should ideally only use this class for functions with the same argument and result types.

Figure 1 plots the negation of the base 10 logarithm of the absolute error in this approximate derivative of the exponential function at zero, equivalent to the number of accurate decimal places, against the order of the approximation. The dashed line shows the negation of the base 10 logarithm of the expected order of the error in the approximation, $\varepsilon^{\frac{n}{n+1}}$.

This certainly seems to be a step in the right direction; as we increase the order of our polynomials the number of accurate digits grow exactly as expected. We should consequently expect that by increasing the order of the polynomials we should bring the error arbitrarily close to ε, as shown in figure 2.

Something has gone horribly wrong! As we expend ever increasing effort in improving the approximation, the errors are getting *worse*.

The reason for this disastrous result should be apparent if you spend a little time studying our simultaneous equation solver. We are performing large numbers of subtractions and divisions; if this doesn't scream cancellation error at you then you haven't been paying attention!

our higher order polynomial approximations are suffering from massive loss of precision

```
template<class F>
typename polynomial_derivative<F>::result_type
polynomial_derivative<F>::operator()(
   const argument_type &x) const
{
  std::vector<std::vector<result_type> >
     lhs(n, std::vector<result_type>(n));
  std::vector<result_type> rhs(n);
  const argument_type abs_x =
     (x>argument_type(0)) ? x : -x;
  const result_type fx = f_(x);
  for(size_t i=0;i!=n_;++i)
  {
    const argument_type y =
      x +argument_type(i+1)*ef_*(
         abs_x+argument_type(1));
    result_type fac(1);
    for(size_t j=0;j!=n_;++j)
    {
      fac *= result_type(j+1);
      const result_type dn = pow(result_type(y-x),
                                 result_type(j+1));
      lhs[i][j] = dn/fac;
    }
    rhs[i] = f_(y)-f_(x);
  }
  return solve_impl(lhs, rhs, n_-1);
}
```

Listing 4

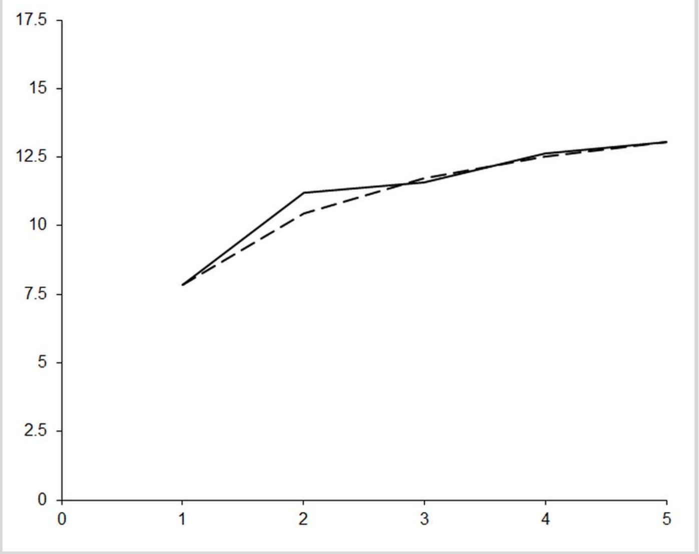


Figure 1

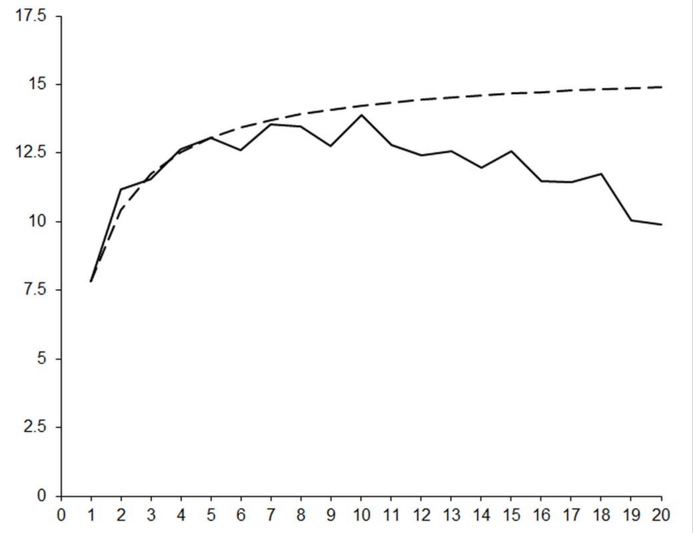


Figure 2

We can demonstrate the problem quite neatly if we use interval arithmetic to keep track of floating point errors. You will recall with interval arithmetic we keep track of an upper and lower bound on a floating point calculation by choosing the most pessimistic bounds on the result of an operation and then rounding the lower bound down and the upper bound up.

Figure 3 plots the approximate number of decimal places of agreement between the upper and lower bound of the interval result of the approximation as a solid line and the expected order of the error as a dashed line.

This clearly shows that our higher order polynomial approximations are suffering from massive loss of precision. We can't plot results above 11th order since the intervals are infinite; these calculations have effectively lost *all* digits of precision.

Note that the intersection of the two curves provides a reasonable choice of 4 for the polynomial order we should use to calculate this specific derivative.

Improving the algorithm

Our algorithm is, in fact, the first step in the Gaussian elimination technique for solving simultaneous linear equations. The second stage is that of back-substitution during which the remaining unknowns are recovered by recursively substituting back to the equations each unknown as it is revealed. We are able to skip this step because we are only interested in one of the unknowns.

However, as it stands, our algorithm is not particularly well implemented.

we have automated the annoying process of working out the formulae

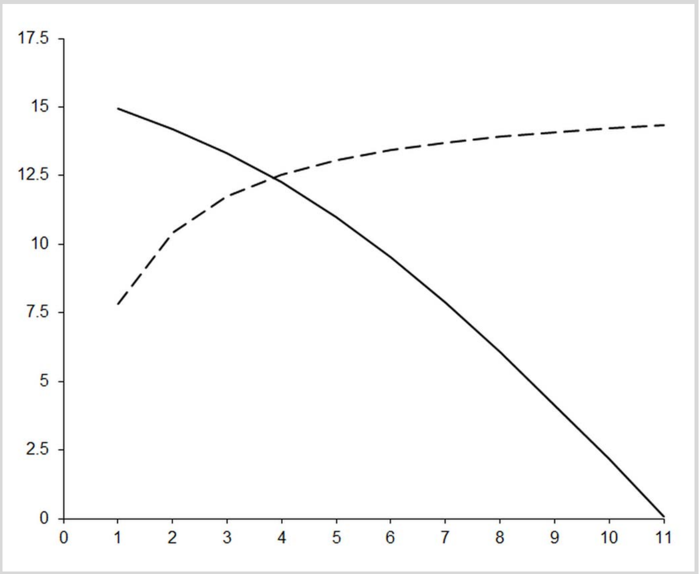


Figure 3

Firstly, if every value in a column in the left hand side is zero, an unlikely but not impossible prospect, we shall end up dividing zero by zero.

Secondly, and more importantly, by eliminating rows in reverse order of the magnitude of δ, we risk dividing by small values and unnecessarily amplifying rounding errors; a far better scheme is to choose the row with the largest absolute value in the column we are eliminating.

Listing 5 provides a more robust implementation of our simultaneous equation solver that corrects these weaknesses.

Unfortunately, whilst this certainly improves the general stability of our algorithm, it does not make much difference to its accuracy.

A more effective approach is to try and minimise the number of arithmetic operations we require for our algorithm. We can do this by exploiting the very same observation that led us to the symmetric finite difference; that the difference between a pair of evaluations of the function an equal distance above and below x depends only on the odd ordered derivatives.

$$\delta \times f'(x) + \tfrac{1}{6}\delta^3 \times f'''(x) + \ldots \\ + \tfrac{1}{(2n-1)!}\delta^{2n-1} \times f^{(2n-1)}(x) = \tfrac{1}{2}\left(f(x+\delta) - f(x-\delta)\right)$$

This will yield an error of the same order as our first implementation with approximately half the number of simultaneous equations and consequently roughly one eighth of the arithmetic operations.

Listing 6 shows the required changes in the **`polynomial_derivative`** member functions.

Figure 4 illustrates the impact upon the number of accurate digits in the results of our improved approximations.

```
template<class T>
T
solve_impl(std::vector<std::vector<T> > &lhs,
           std::vector<T> &rhs,
           const size_t n)
{
  T abs_nn =
     lhs[n][n]>T(0) ? lhs[n][n] : -lhs[n][n];
  for(size_t i=0;i!=n;++i)
  {
    const T abs_in =
       lhs[i][n]>T(0) ? lhs[i][n] : -lhs[i][n];
    if(abs_in>abs_nn)
    {
      abs_nn = abs_in;
      std::swap(lhs[i], lhs[n]);
      std::swap(rhs[i], rhs[n]);
    }
  }
  if(abs_nn>T(0))
  {
    for(size_t i=0;i!=n;++i)
    {
      for(size_t j=0;j!=n;++j)
      {
        lhs[i][j] -=
           lhs[i][n]*lhs[n][j]/lhs[n][n];
      }
      rhs[i] -= lhs[i][n]*rhs[n]/lhs[n][n];
    }
  }
  return n!=0 ? solve_impl(lhs, rhs, n-1) :
     rhs[0]/lhs[0][0];
}
```

Listing 5

Clearly, this is far better than our first attempt; the observed error is consistently smaller than expected. However, on performing the calculation using interval arithmetic, we find that we are still restricted by cancellation error, as shown in figure 5.

If we choose the point at which the observed cancellation error crosses the expected error, shown as a dashed line, we should choose a 7th order polynomial, corresponding again to 4 simultaneous equations.

These polynomial approximation algorithms are certainly an improvement upon finite differences since we have automated the annoying process of working out the formulae. However, we are still clearly constrained by the trade off between approximation and cancellation error.

This problem is fundamental to the basic approach of these algorithms; computing the coefficients of the Taylor series by solving simultaneous equations is inherently numerically unstable.

```
template<class F>
polynomial_derivative<F>::polynomial_derivative(
  const function_type &f,
  const unsigned long n)
: f_(f), n_(n),
  ef_(pow(argument_type(2*n+1)
      *eps<argument_type>(),
      argument_type(1)/argument_type(2*n+1))),
  lhs_(n, std::vector<result_type>(n)),
  rhs_(n)
{
  if(n==0) throw std::invalid_argument("");
}
template<class F>
typename polynomial_derivative<F>::result_type
polynomial_derivative<F>::operator()(
   const argument_type &x) const
{
  const argument_type abs_x =
     (x> result_type(0)) ? x : -x;
  for(size_t i=0;i!=n_;++i)
  {
    const argument_type y =
       abs_x + argument_type(i+1)*ef_*(
       abs_x+argument_type(1));
    const argument_type d = y-abs_x;
    result_type fac(1);
    for(size_t j=0;j!=n_;++j)
    {
      const result_type dn =
         pow(result_type(d), result_type(2*j+1));
      lhs_[i][j] = dn/fac;
      fac *=
         result_type(2*j+2) * result_type(2*j+3);
    }
    rhs_[i] = (f_(x+d)-f_(x-d)) / result_type(2);
  }
  return solve_impl(lhs_, rhs_, n_-1);
}
```

Listing 6

To understand why, it is illuminating to recast the problem as one of linear algebra.

A problem of linear algebra

By the rules of matrix and vector arithmetic we have, for a matrix **M** and vectors **v** and **w**

$$\mathbf{w} = \mathbf{M} \times \mathbf{v}$$
$$\mathbf{w}_i = \sum_j \mathbf{M}_{i,j} \times \mathbf{v}_j$$

where subscripts *i*,*j* of a matrix denote the *j*th column of the *i*th row, a subscript *i* of a vector denotes its *i*th element and the capital sigma denotes the sum of the expression to its right for every valid value of the symbol beneath it.

We can consequently represent the simultaneous equations of our improved algorithm with a single matrix-vector equation if we choose the elements of **M**, **v** and **w** such that

$$\mathbf{M}_{i,j} = \tfrac{1}{(2j-1)!}\left(i \times \delta\right)^{2j-1}$$
$$\mathbf{v}_j = f^{(2j-1)}(x)$$
$$\mathbf{w}_i = \tfrac{1}{2}\left(f\left(x + i \times \delta\right) - f\left(x - i \times \delta\right)\right)$$

for *i* and *j* from one up to and including *n*.

Those matrices with the same number of rows and columns, say *n*, whose elements are equal to one if the column index is equal to the row index and zero otherwise we call identity matrices, denoted by **I**. These have the property that for all vectors **v** with *n* elements

$$\mathbf{I} \times \mathbf{v} = \mathbf{v}$$

If we can find a matrix $\mathbf{M}^{-1}$ such that

$$\mathbf{M}^{-1} \times \mathbf{M} = \mathbf{I}$$

known as the inverse of **M**, we have

$$\mathbf{M}^{-1} \times \mathbf{w} = \mathbf{M}^{-1} \times \mathbf{M} \times \mathbf{v}$$
$$\mathbf{M}^{-1} \times \mathbf{w} = \mathbf{I} \times \mathbf{v}$$
$$\mathbf{M}^{-1} \times \mathbf{w} = \mathbf{v}$$

Our algorithm is therefore, in some sense, equivalent to inverting the matrix **M**.

That this can be a source of errors is best demonstrated by considering the geometric interpretation of linear algebra in which we treat a vector as Cartesian coordinates identifying a point.

For example, in two dimensions we have

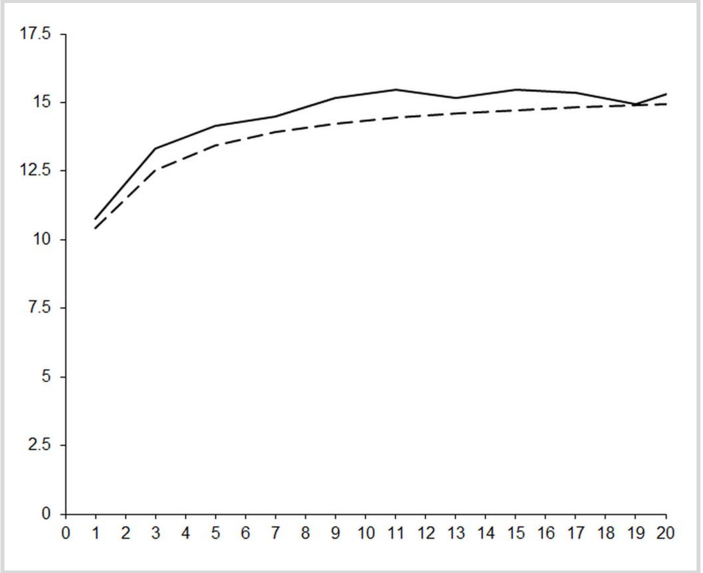


Figure 4

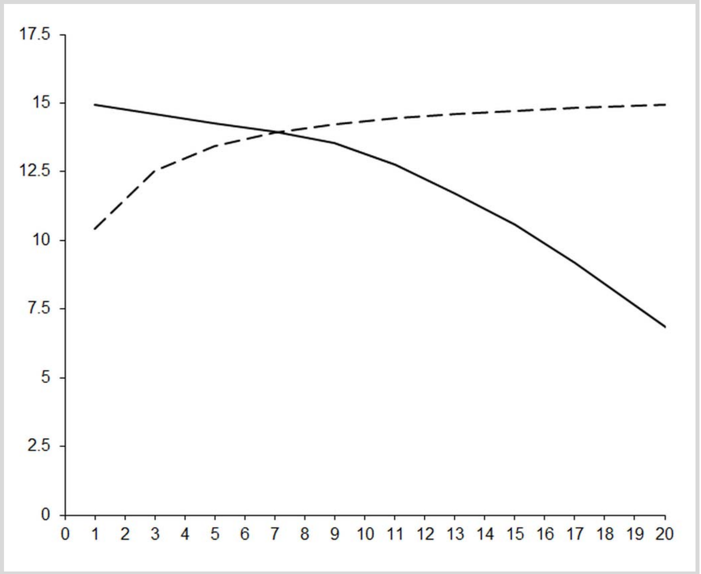


Figure 5

$$\mathbf{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\mathbf{M} \times \mathbf{v} = \begin{pmatrix} a \times x + b \times y \\ c \times x + d \times y \end{pmatrix}$$

Hence every two by two matrix represents a function that takes a point in the plane and returns another point in the plane.

If it so happens that *c÷a* equals *d÷b* then we are in trouble since the two dimensional plane is transformed into a one dimensional line.

For example, if

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 3 & 6 \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x + 4y \\ 3x + 6y \end{pmatrix}$$

then

$$y' = 1\tfrac{1}{2}x'$$

Given the result of such a function there is no possible way to identify which of the infinite number of inputs might have yielded it.

Such matrices are known as singular matrices and attempting to invert them is analogous to dividing by zero. They are not, however, the cause of our problems. Rather, we are troubled by matrices that are *nearly* singular.

To understand what this means, consider the two dimensional matrix

$$\mathbf{M} = \begin{pmatrix} 2 & 4 \\ 3 & 5\tfrac{1}{2} \end{pmatrix}$$

Figure 6 shows the result of multiplying a set of equally spaced points on the unit square by this matrix. The points clearly end up very close to a straight line rather than exactly upon one.

To a mathematician, who is equipped with the infinitely precise real numbers, this isn't really a problem. To a computer programmer, who is not, it causes no end of trouble. One dimension of the original data has been compressed to small deviations from the line. These deviations are represented by the least significant digits in the coordinates of the points and much information about the original positions is necessarily rounded off and lost. Any attempt to reverse the process cannot recover the original positions of the points with very much accuracy; cancellation error is the inevitable consequence.

That our approximations should lead to such matrices becomes clear when you consider the right hand side of the equation. We choose δ as small as possible in order to minimise approximation error with the result that the function is evaluated at a set of relatively close points. These evaluations will generally lie close to a straight line since the behaviour of the function will largely be governed by the first two terms in its Taylor series expansion.

Unfortunately, the approach we have used to mitigate cancellation error is horribly inefficient since we must solve a new system of simultaneous equations each time we increase the order of the approximating polynomials.

The question is whether there is there a better way?

Ridders' algorithm

The trick is to turn the problem on its head; rather than approximate the function with a polynomial and use its coefficients to approximate the derivative we treat the symmetric finite difference itself as a function of δ and approximate *it* with a polynomial.

Specifically, if we define

$$g_{f,x}(\delta) = \frac{f(x+\delta) - f(x-\delta)}{2\delta}$$

and we vigorously wave our hands, we have

$$f'(x) = g_{f,x}(0)$$

Now, if we approximate $g_{f,x}$ with an n^{th} order polynomial $\hat{g}^n_{f,x}$ then we can approximate the derivative of *f* by evaluating it at 0

$$f'(x) \approx \hat{g}^n_{f,x}(0)$$

We could try to find an explicit formula for $\hat{g}^n_{f,x}$ by evaluating $g_{f,x}$ at *n* non-zero values of δ and solving the resulting set of linear equations, as we did in our first polynomial algorithm, but we would end up with the same inefficiency as before when seeking the optimal order.

Ridders' algorithm [Ridders82] avoids this problem by evaluating the polynomial at zero without explicitly computing its coefficients. That this is even possible might seem a little unlikely, but nevertheless there is more than one way to do so.

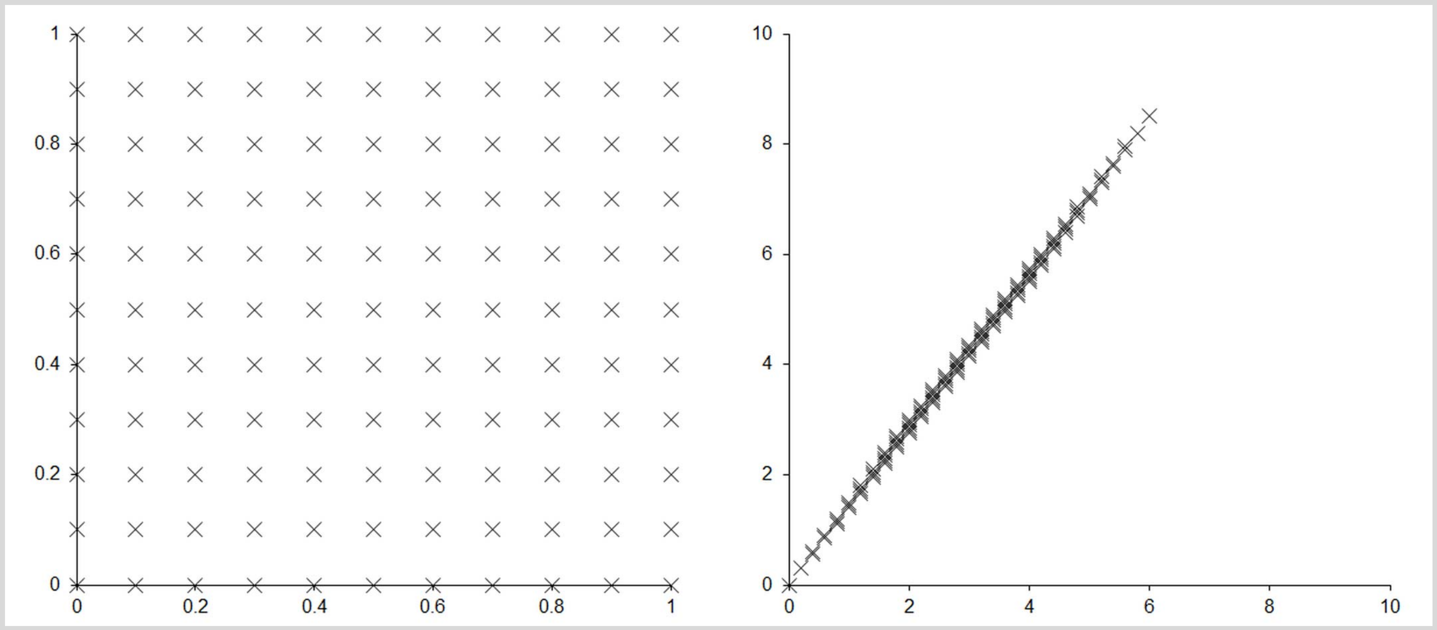


Figure 6

Neville's algorithm

We can easily compute the value of the polynomial that passes through a pair of points, trivially a straight line, without explicitly computing its coefficients with a kind of pro-rata formula.

Given a pair of points (x_0,y_0) and (x_1,y_1), the line that passes through them consists of the points (x,y) where

$$\frac{y-y_0}{x-x_0}=\frac{y_1-y_0}{x_1-x_0}$$

This can be rearranged to yield an equation for the line

$$y=y_0+\frac{y_1-y_0}{x_1-x_0}\times\left(x-x_0\right)$$

Neville's algorithm, described in *Numerical Recipes in C* [Press92], generalises this approach to any number of points, or equivalently any order of polynomial, and takes the form of a recursive set of formulae.

Given a set of n points (x_i,y_i) and a value x at which we wish to evaluate the polynomial that passes through them, we define the formulae

$$p_{i,i}\left(x\right)=y_i \qquad 1\leq i\leq n$$

$$p_{i,j}\left(x\right)=\frac{\left(x-x_j\right)\times p_{i,j-1}\left(x\right)+\left(x_i-x\right)\times p_{i+1,j}\left(x\right)}{x_i-x_j} \qquad 1\leq i<j\leq n$$

from which we retrieve the value of the polynomial at x with $p_{1,n}(x)$.

If we arrange the formulae in a triangular tableau, we can see that as the calculation progresses each value is derived from the pair above and below to its left.

$$\begin{array}{cccc}
p_{1,1}(x) & & & \\
 & p_{1,2}(x) & & \\
p_{2,2}(x) & & p_{1,3}(x) & \\
 & p_{2,3}(x) & & p_{1,4}(x) \\
p_{3,3}(x) & & p_{2,4}(x) & \\
 & p_{3,4}(x) & & \\
p_{4,4}(x) & & &
\end{array}$$

This shows why this algorithm is so useful; adding a new point introduces a new set of values without affecting those already calculated. We can consequently increase the order of the polynomial approximation without having to recalculate the entire tableau.

Further noting that we actually only really need the lower diagonal values when we do so allows us to implement Neville's algorithm with minimal memory usage, as shown in listing 7.

```
template<class T>
class neville_interpolate
{
public:
  typedef T value_type;
  explicit neville_interpolate(
     const value_type &xc);
  value_type refine(
     const value_type &x, value_type y);
private:
  value_type xc_;
  std::vector<value_type> x_;
  std::vector<value_type> p_;
};
```

Listing 7

The constructor simply initialises the point at which the polynomial will be evaluated, **xc_**, and pre-allocates some space for the **x_** and **p_** buffers, as shown in listing 8.

```
template<class T>
neville_interpolate<T>::neville_interpolate(
   const value_type &xc)
: xc_(xc)
{
  x_.reserve(8);
  p_.reserve(8);
}
```

Listing 8

The body of the algorithm is in the **refine** member function, given in listing 9.

It might not be immediately obvious that this function correctly applies Neville's algorithm due to its rather terse implementation. I must confess that once I realised that I didn't need to keep the entire tableau in memory I couldn't resist rewriting it a few times to improve its efficiency still further.

The key to understanding its operation is the fact that the x values are iterated over in reverse order.

The result of Neville's algorithm for n points is trivially an n-1th order polynomial since each value in the tableau is one order higher in x than those to its left used to calculate it and the leftmost values are constants and hence of order 0.

All that remains is to prove that it passes through the set of points (x_i,y_i).

Consider a single point (x_k,y_k), some i less than k and some j greater than k

$$\begin{aligned}
&p_{k,k}\left(x_k\right)=y_k\\
&p_{i,k}\left(x_k\right)\\
&\quad=\frac{\left(x_k-x_k\right)\times p_{i,k-1}\left(x_k\right)+\left(x_i-x_k\right)\times p_{i+1,k}\left(x_k\right)}{x_i-x_k}\\
&\quad=\frac{\left(x_i-x_k\right)\times p_{i+1,k}\left(x_k\right)}{x_i-x_k}=p_{i+1,k}\left(x_k\right)\\
&p_{k,j}\left(x_k\right)\\
&\quad=\frac{\left(x_k-x_j\right)\times p_{k,j-1}\left(x_k\right)+\left(x_k-x_k\right)\times p_{k+1,k}\left(x_k\right)}{x_k-x_j}\\
&\quad=\frac{\left(x_k-x_j\right)\times p_{k,j-1}\left(x_k\right)}{x_k-x_j}=p_{k,j-1}\left(x_k\right)
\end{aligned}$$

If we apply these equalities recursively, we have

$$\begin{aligned}
&p_{i,k}\left(x_k\right)=y_k\\
&p_{k,j}\left(x_k\right)=y_k
\end{aligned}$$

so that there are two diagonal lines of y_k's running up and down from $p_{k,k}(x_k)$ on the left of the tableau, albeit not necessarily of the same length or of more than the leftmost value.

If the final value lies on one of these diagonals, we are done. If not, then consider the value calculated from the second on each of these diagonals

$$\begin{aligned}
p_{k-1,k+1}\left(x_k\right)&=\frac{\left(x_k-x_{k+1}\right)\times p_{k-1,k}\left(x_k\right)+\left(x_{k-1}-x_k\right)\times p_{k,k+1}\left(x_k\right)}{x_{k-1}-x_{k+1}}\\
&=\frac{\left(x_k-x_{k+1}\right)\times y_k+\left(x_{k-1}-x_k\right)\times y_k}{x_{k-1}-x_{k+1}}\\
&=\frac{-x_{k+1}\times y_k+x_{k-1}\times y_k}{x_{k-1}-x_{k+1}}=y_k
\end{aligned}$$

If we continue to fill in the tableau we find that every value lying between the diagonals in the tableau, including $p_{1,n}(x_k)$, must therefore be equal to y_k.

Derivation 1

```
template<class T>
typename neville_interpolate<T>::value_type
neville_interpolate<T>::refine(
   const value_type &x, value_type y)
{
  std::vector<value_type>::reverse_iterator
     xi = x_.rbegin();
  std::vector<value_type>::reverse_iterator
     x0 = x_.rend();
  std::vector<value_type>::iterator
     pi = p_.begin();
  while(xi!=x0)
  {
    std::swap(*pi, y);
    y = (y*(xc_-x) + *pi*(*xi-xc_)) / (*xi-x);
    ++xi;
    ++pi;
  }
  x_.push_back(x);
  p_.push_back(y);
  return y;
}
```

Listing 9

The tableau representation of the algorithm is also helpful in demonstrating why Neville's algorithm works, as shown in derivation 1.

A better polynomial approximation

We could use interval arithmetic again to decide when to stop increasing the order of the approximating polynomial. Ridders' algorithm takes a different approach, however.

As the order of the polynomial increases we expect the result to get progressively closer to the correct value until the point at which cancellation error takes over. The absolute differences between successive approximations should consequently form a more or less decreasing sequence up to that point.

The algorithm therefore begins with a relatively large δ and, shrinking it at each step, computes the symmetric finite difference and refines the polynomial approximation for δ equal to 0. The value with the smallest absolute difference from that of the previous step is used as the approximation of the derivative with the algorithm terminating when the step starts to grow.

Now there is almost certainly going to be some numerical noise in the value of the polynomial at each step, so rather than stop as soon as the difference increases we shall terminate when the difference is twice the smallest found so far.

Note that the smallest absolute difference provides a rough estimate of the error in the approximation.

The rate at which we shrink δ and the termination criterion are based upon the implementation of this algorithm given in *Numerical Recipes in C*. There are a couple of important differences however.

Firstly, they use the penultimate value in the final row of the Neville's algorithm tableau as a second approximation to the derivative, having the same polynomial order as the previous iteration's approximation but with a smaller δ. This is used to improve the termination criterion for the algorithm.

Secondly, we exploit the fact that the symmetric finite difference doesn't depend upon the sign of δ. At each step we can refine the polynomial twice with the same symmetric difference; once with the positive δ, once with its negation. This doubles the order of the approximating polynomial and forces it to be symmetric about 0 with no additional evaluations of the function but may lead to increased cost in the application of Neville's algorithm. This is a reasonable trade-off if the cost of the function evaluations is the primary concern.

```
template<class F>
class ridders_derivative
{
public:
  typedef F function_type;
  typedef typename F::argument_type argument_type;
  typedef typename F::result_type result_type;
  explicit ridders_derivative(
     const function_type &f)
  ridders_derivative(const function_type &f,
                     const argument_type &d);
  result_type apply(const argument_type &x,
                    result_type &err) const;
  result_type operator()(
     const argument_type &x) const;
private:
  result_type refine(
     neville_interpolate<result_type> &f,
     const argument_type &x,
     const argument_type &dx) const;
  function_type f_;
  argument_type d_;
};
```

Listing 10

Listing 10 shows the class definition for our implementation of Ridders' algorithm.

The constructors are responsible for determining the initial δ. To avoid rounding error we shall again use a multiple of this and 1 plus the absolute value of the point at which we wish to compute the derivative.

If the function is reasonably well behaved from the perspective of the symmetric finite difference, a choice of 10% is fairly reasonable and is used in the first constructor as given in listing 11.

Note that if the approximate error is large it is worth reconsidering the choice of the initial step size.

The **`refine`** member function is used to refine a **`neville_interpolate`** approximation of the derivative with the pair of equivalent symmetric finite difference approximations, as shown in listing 12.

The **`apply`** member function implements the body of the algorithm as described with the function call operator simply returning its value without providing an error estimate, as illustrated in listing 13.

Figure 7 shows the actual (solid line) and estimated (dashed line) decimal digits of accuracy in the approximate derivative of the exponential function using our implementation of Ridders' algorithm.

This represents a vast improvement in both accuracy and error estimation over every algorithm we have studied thus far. Figure 8 demonstrates the former with a comparison of the results of Ridders' algorithm (solid line)

```
template<class F>
ridders_derivative<F>::ridders_derivative(
   const function_type &f) :
   f_(f),
   d_(argument_type(1)/argument_type(10))
{
}
template<class F>
ridders_derivative<F>::ridders_derivative(
   const function_type &f,
   const argument_type &d)
: f_(f), d_(d)
{
}
```

Listing 11

```
template<class F>
typename ridders_derivative<F>::result_type
ridders_derivative<F>::refine(
   neville_interpolate<result_type> &f,
   const argument_type &x,
   const argument_type &dx) const
{
  const argument_type xa = x+dx;
  const argument_type xb = x-dx;
  const result_type df =
    (f_(xa)-f_(xb))/result_type(xa-xb);
  f.refine(result_type(xb-xa), df);
  return f.refine(result_type(xa-xb), df);
}
```

Listing 12

```
template<class F>
typename ridders_derivative<F>::result_type
ridders_derivative<F>::apply(
   const argument_type &x,
   result_type &err) const
{
  static const argument_type c1 =
     argument_type(7)/argument_type(5);
  static const argument_type c2 = c1*c1;
  neville_interpolate<result_type>
     f(result_type(0));
  argument_type abs_x =
     (x>argument_type(0)) ? x : -x;
  argument_type dx = d_ *
     (abs_x+argument_type(1));
  result_type y0 = refine(f, x, dx);
  result_type y1 = refine(f, x, dx/=c1);
  result_type y = y1;
  err = (y0<y1) ? (y1-y0) : (y0-y1);
  result_type new_err = err;
  while(new_err<err+err)
  {
    y0 = y1;
    y1 = refine(f, x, dx/=c2);
    new_err = (y0<y1) ? (y1-y0) : (y0-y1);
    if(new_err<err)
    {
      err = new_err;
      y = y1;
    }
  }
  return y;
}
template<class F>
typename ridders_derivative<F>::result_type
ridders_derivative<F>::operator()(
   const argument_type &x) const
{
  result_type err;
  return apply(x, err);
}
```

Listing 13

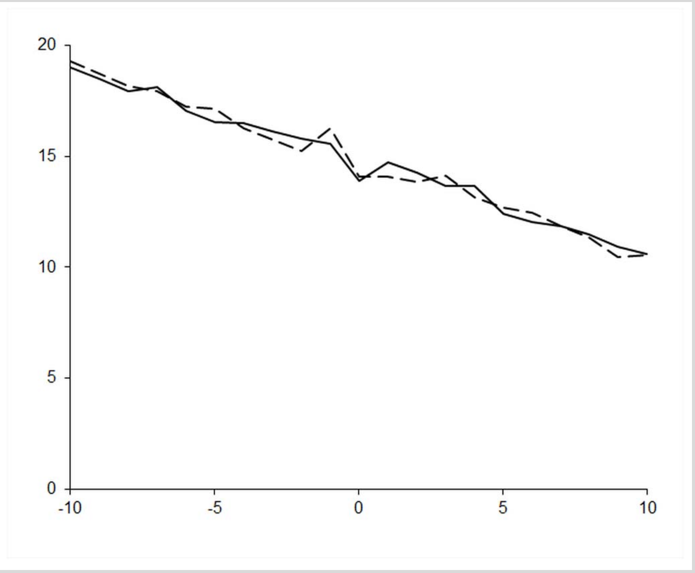


Figure 7

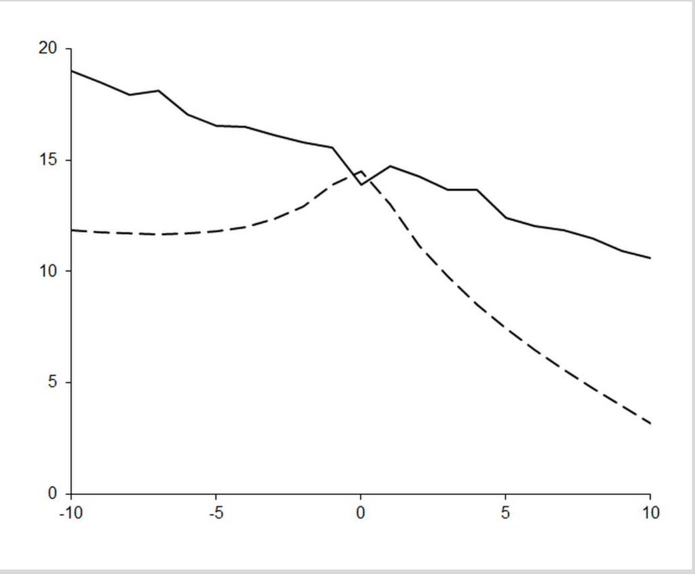


Figure 8

and a 7th order polynomial approximation (dashed line) using the improved version of our original algorithm based on the truncated Taylor series.

The relative error in these results is on average roughly 2E-15, just one decimal order of magnitude worse than the theoretical minimum.

We have thus very nearly escaped the clutches of numerical error and I therefore declare polynomial approximation of the derivative, in this its final and most effective form, a giant amongst ducks; QUAAAACK! ■

References and further reading

[Press92] Press, W.H. et al, *Numerical Recipes in C* (2nd ed.), Cambridge University Press, 1992.

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

Concurrent Programming with Go

Concurrency is becoming ever more important. Mark Summerfield looks at the approach of the new language Go.

The Go programming language is in some respects a radical departure from existing compiled languages. Its syntax, although C-ish, is much cleaner and simpler than C or C++'s, and it supports object-orientation through embedding (delegation) and aggregation, rather than by using inheritance. Go has a built-in garbage collector so we never have to worry about deleting/freeing memory – something that can be fiendishly complicated in a multithreaded context. In this article we will focus on another area where Go breaks new ground (at least, compared with other mainstream programming languages): concurrency.

Go has the usual concurrency primitives, such as mutexes, read–write mutexes, and wait conditions, as well as low-level primitives such as atomic adds, loads, and compare and swaps. But Go programmers are encouraged to avoid using any of these and instead to use Go's high-level *goroutines* and *channels*.

A goroutine is a very lightweight thread of execution that shares the same address space as the rest of the program. The *gc* compiler multiplexes one or more goroutines per operating system thread and can realistically support hundreds, thousands, or more goroutines.

A channel is a two-way (or one-way, at our option) communications pipeline. Channels are type safe, and when they are used to pass immutable values (**`bool`**s, **`int`**s, **`float64`**s, **`string`**s, and **`struct`**s composed of immutable values), they can be used in multiple goroutines without formality. When it comes to passing pointers or references, we must, of course, ensure that our accesses are synchronized.

Incidentally, goroutines and channels are an implentation of a form of CSP (Communicating Sequential Processes), based on the ideas of computer scientist C. A. R. Hoare.

Go's mantra for concurrency is:

> Do not communicate by sharing memory;
> instead, share memory by communicating.

In this article we will review a simple concurrent program called **`headcheck`**, that, given a list of URLs, performs an HTTP HEAD request on each one and reports its results. We will look at a few different ways the program can implement concurrency using Go's goroutines and channels, to give a flavour of the possibilities.

Listing 1 shows the **`struct`**s the program will operate on. We made **`Job`** a **`struct`** because this is syntactically more convenient when giving it methods.

Listing 2 shows the **`main()`** function. The built-in **`make()`** command is used to create channels (as well as values of the built in map and slice collection types).

In Listing 2, both **`nWorkers`** and **`bufferSize`** are constants (6 and 24; not shown).

The **`main()`** function begins by creating three channels, one for passing jobs to worker goroutines, one for receiving all the results, and another to keep track of when each worker goroutine has finished.

By default channels are unbuffered (their size is 0) which means that a receive will block until there is a send and a send will block if there's a sent item that hasn't been received. By buffering we allow a channel to accept as many sends as the size of the buffer, before sends are blocked. Similarly, we can do as many receives as there are items in the buffer, only blocking when the buffer is empty. The purpose of buffering is to improve throughput by minimizing the time goroutines spend being blocked.

```
type Result struct {
  url          string
  status       int
  lastModified string
}
type Job struct {
  url string
}
```

Listing 1

In this example we have buffered all the channels by giving **`make()`** a second buffer-size argument. We have made the **`jobs`** channel large enough to accept (an average of) two jobs per worker goroutine and made the results channel big enough to accept plenty of results without blocking the workers. The **`done`** channel's buffer's size is the same as the number of workers since, as we will see, each worker sends to that channel exactly once.

To execute code in a separate goroutine we use the **`go`** keyword. This keyword must be followed by a function call (which could be a call on a function literal – which is also a closure). The go statement completes 'immediately' and the called function is executed in a newly created goroutine. When the function finishes the Go runtime system automatically gets rid of its goroutine and reclaims the memory it used.

```
func main() {
  jobs := make(chan Job, nWorkers * 2)
  results := make(chan Result, bufferSize)
  done := make(chan bool, nWorkers)

  go addJobs(jobs)
  for i := 0; i < nWorkers; i++ {
    go doJobs(jobs, results, done)
  }
  go wait(results, done)
  process(results)
}
```

Listing 2

Mark Summerfield is an independent programmer, author, and trainer. His book, *Programming in Go* ISBN 0-321-77463-9, is due to be published Q1 2012. He can be contacted at www.qtrac.eu.

Once the main goroutine has finished, the program will terminate – even if there are other goroutines still executing

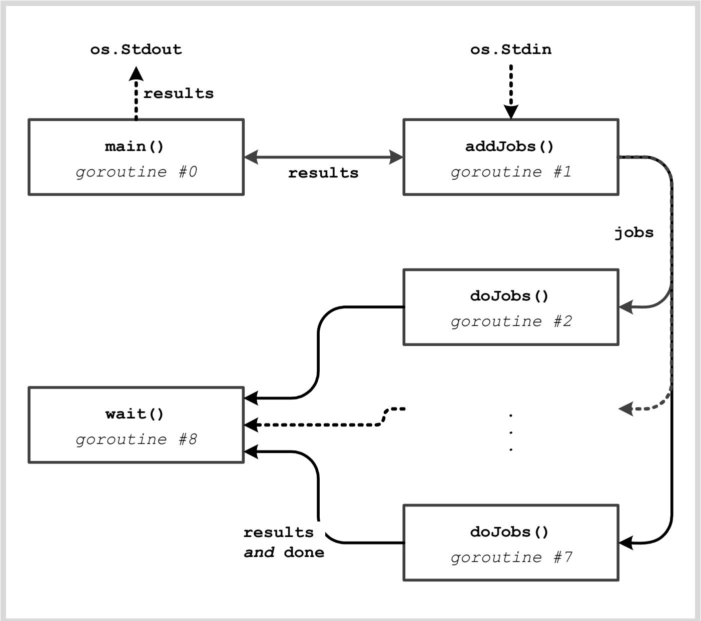


Figure 1

Here, the **`main()`** function executes the **`addJobs()`** function in its own separate goroutine, so execution continues immediately to the **`for`** loop. In the **`for`** loop six separate goroutines are created, each one executing an instance of the **`doJobs()`** function. All the newly created goroutines share the *same* **`jobs`** channel and the *same* **`results`** channel. The **`for`** loop completes as soon as the goroutines have been created and started and then another function is called, **`wait()`**, again in its own separate goroutine. And finally, we call the **`process()`** function in the current (**`main`**) goroutine.

Figure 1 schematically illustrates the relationships between the program's goroutines and channels.

Once the **`main`** goroutine has finished, the program will terminate – even if there are other goroutines still executing. So, we must ensure that all the other goroutines finish their work before we leave **`main()`**.

The **`addJobs()`** function is used to populate the **`jobs`** channel and is shown in Listing 3, but with the code for reading in the URLs elided.

Each job simply consists of a URL to check. URLs are read from **`os.Stdin`** (e.g., by using redirection on the command line). At each iteration we read both a line and an **`error`** value; if the error is **`io.EOF`** we have finished and break out of the **`for`** loop. (All of this has been elided.)

Once all the jobs have been added the **`jobs`** channel is closed to signify that there are no more jobs to be added. Sending to a channel is done using the syntax **`channel <- item`**. Items can be received from a channel that is non-empty, even if it is closed, so no jobs will be lost. When the

```
func addJobs(jobs chan Job) {
  reader := bufio.NewReader(os.Stdin)
  for {
    ... // Read in a URL
    url = strings.TrimSpace(url)
    if len(url) > 0 {
      jobs <- Job{url}
    }
  }
  close(jobs)
}
```

Listing 3

`addJobs()` function has finished the Go runtime system will take care of removing the goroutine in which it ran and reclaiming its memory.

The **`doJobs()`** function is shown in Listing 4. It is simple because it passes all its work on to a method of the **`Job`** type (not shown). The **`Job.Do()`** method sends one result of type **`Result`** to the **`results`** channel using the statement **`results <- result`**.

Go's **`for ... range`** statement can iterate over maps (data dictionaries like C++11's **`unordered_map`**), slices (in effect, variable length arrays), and channels. If the channel has an item it is received and assigned to the **`for`** loop's variable (here, **`job`**); if the channel has no item but isn't closed the loop *blocks*. Of course, this does not hold up the rest of the program, only the goroutine in which the loop is executing is blocked. The loop terminates when the channel is empty and closed.

Once all the jobs are done the function sends a **`bool`** to the **`done`** channel. Whether **`true`** or **`false`** is sent doesn't matter, since the **`done`** channel is used purely to keep the program alive until all the jobs are done.

The **`headcheck`** program has one goroutine adding jobs to the **`jobs`** channel and six goroutines reading and processing jobs from the same channel, all of them working concurrently. Yet, we don't have to worry about locking – Go handles all the synchronization for us.

Listing 5 shows the **`wait()`** function which was executed by **`main()`** in its own goroutine. This function has a regular **`for`** loop that iterates for as many worker goroutines as were created, and at each iteration it does a *blocking* receive using the syntax **`item <- channel`**. Notice that it doesn't matter whether **`true`** or **`false`** was sent – we only care that *something* was sent – since we discard the channel's items. Once all the

```
func doJobs(jobs chan Job,
  results chan Result, done chan bool) {
  for job := range jobs {
      job.Do(results)
  }
  done <- true
}
```

Listing 4

```
func wait(results chan Result,
  done chan bool) {
  for i := 0; i < nWorkers; i++ {
    <-done
  }
  close(results)
}
```

Listing 5

workers have sent to the **done** channel we know that there can be no more results added to the **results** channel, so we close that channel.

Listing 6 shows the **process()** function which is executed in the **main** goroutine. This function iterates over the **results** channel and blocks if no result is available. The **for** loop terminates when the **results** channels is empty and closed, which will only happen when the **wait()** function finishes. This ensures that this function blocks the **main** goroutine until every result has been received and output.

We could replace the **wait()** and **process()** functions with a single **waitAndProcess()** function executed in the main goroutine, as Listing 7 illustrates.

This function begins with a **while** loop that iterates so long as there is at least one worker still working. The **select** statement is structurally like a **switch** statement, only it works in terms of channel communications. A **select** with no **default** case is *blocking*. So, here, the first **select** blocks until it receives either a result or an empty **struct**.

Since we don't care what's sent to the **done** channel, only whether something's sent, we have defined the channel to be of type **chan struct{}**. This channel's value type specifies a **struct** with no fields; there is only one possible value of such a type and this is specified using **struct{}{}** which means create a (zero) value of type **struct{}**. Since such values have no data they are more expressive of our semantics than sending **bool**s whose value we would then ignore.

After each receive the **select** is broken out of and the loop condition is checked. This causes the **main** goroutine to block until all the workers have finished sending their results (because they only send to the **done** channel after they have finished all their jobs).

```
func process(results chan result) {
  for result := range results {
    result.Report(os.Stdout)
  }
}
```

Listing 6

```
func waitAndProcess(results <-chan Result,
  done <-chan struct{}) {
  for working := nWorkers; working > 0; {
    select { // Blocking
    case result := <-results:
      result.Report(os.Stdout)
    case <-done:
      working--
    }
  }
  for {
    select { // Non-blocking
    case result := <-results:
      result.Report(os.Stdout)
    default:
      return
    }
  }
}
```

Listing 7

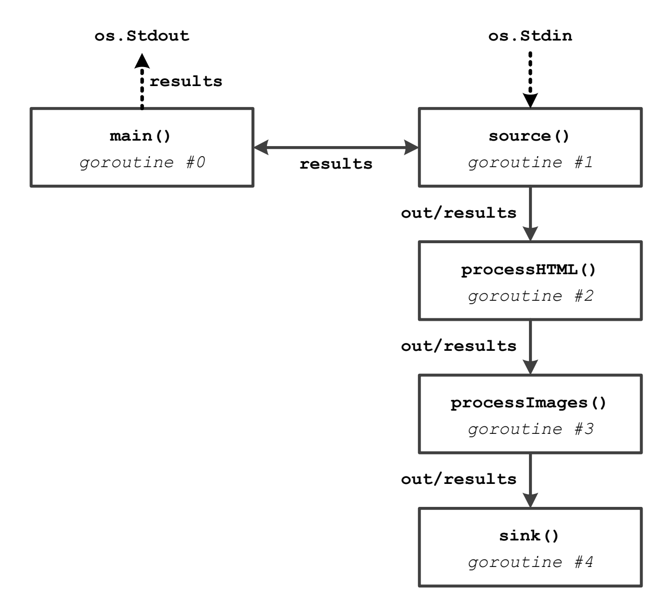


Figure 2

It is quite possible that after all the worker goroutines have finished there are still unprocessed results in the **results** channel (after all, we buffered the channel when we created it). So we execute a second **for** loop (an infinite loop) that uses a non-blocking **select**. So long as there are results in the **results** channel the **select** will receive each one and finish, and the **for** loop will iterate again. But once the **results** channel is empty and closed the **default** case will be executed, and the function returned from. At this point all the results have been output and the program will terminate.

A pipelining approach

Goroutines and channels are very flexible and versatile, to the extent that we can take some quite different approaches to concurrency. Listing 8 illustrates an alternative **headcheck** implementation's **main()** function.

Unlike the previous versions, this **headcheck** program only reports on URLs for HTML files and images, and ignores anything else. We could always add another pipeline component, say, **processRemainder()**, if we didn't want to ignore any URLs.

Figure 2 schematically illustrates the relationships between the program's goroutines and channels.

The function begins by creating a buffered **results** channel and then it executes a pipeline in a separate goroutine. (And as we will see, each component of the pipeline itself executes in a separate goroutine.) Control immediately passes to the **for ... range** loop which blocks waiting for results and finishes when the **results** channel is empty and closed. (The **Result** type was shown in Listing 1.) The **source()** function shown in Listing 9 is where the pipeline begins.

A Go function's return value is specified after the closing parenthesis that follows the function's arguments. Multiple values can be returned simply by enclosing them in parentheses. Here we return a receive-only channel and a send–receive channel.

```
func main() {
  results := make(chan Result, bufferSize)
  go sink(processImages(processHTML(
    source(results))))
  for result := range results {
    result.Report(os.Stdout)
  }
}
```

Listing 8

```
func source(results chan Result) (
  <-chan string, chan Result) {
  out := make(chan string, bufferSize)
  go func() {
    reader := bufio.NewReader(os.Stdin)
    for {
      ... // Read in a URL
      out <- url
    }
    close(out)
  }()
  return out, results
}
```

Listing 9

The **source()** function is passed the **results** channel, which it simply returns to its caller. All the pipeline functions share the same **results** channel. The function starts by creating a buffered output channel which is initially bi-directional. It then creates a goroutine to populate the **out** channel with jobs (in this case URL strings), after which the goroutine closes the channel. By closing the channel, future receivers (e.g., using a **for ... range** loop) will know when to finish. The **go func() { ... }()** creates a function literal (which is a closure) and executes it in a separate goroutine. So processing immediately continues to the **source()** function's last statement which simply returns the **out** channel as a receive-only channel thanks to the way the function's return value is declared, as well as the **results** channel.

The **processHTML()** function shown in Listing 10 has the same structure as all the pipeline component functions except for the **source()** function. It is passed two channels, a receive-only jobs channel (of type **chan string**) which it calls **in** (and which was the previous pipeline component's **out** channel), and the shared **results** channel. The function creates a new buffered bi-directional **out** channel with the same buffer size as the capacity of the **in** channel the function has been passed. It then creates a goroutine which executes a new function literal. The goroutine reads jobs from the **in** channel. This channel is closed by the previous pipeline component when it has been fully populated, so this goroutine's **for ... range** loop is guaranteed to terminate. For each job (URL) received, for those that this function can process it performs its processing (in this case a call to a **Check()** function), and sends the result to the **results** channel. And those jobs the function isn't concerned with are simply sent to the (new) **out** channel. At the end the function returns its **out** channel and the shared **results** channel.

The **processImages()** function has the same signature and uses the same logic.

The **sink()** function takes a receive-only **jobs** channel and a receive-only **results** channel. It is shown in Listing 11.

The **sink()** function is the last one in the pipeline. It iterates over the penultimate pipeline component's jobs channel, draining it until it is empty. (It might be empty in the first place if every job has been done by one or other pipeline component.)

At the end the function closes the **results** channel. Closing the **results** channel is essential to ensure that the **for ... range** loop in **main()** terminates. But we must be careful not to close the channel when one or more goroutines might still want to send to it. Looking back at the implementations of the **source()** and **processHTML()** functions we

```
func processHTML(in <-chan string,
  results chan Result) (<-chan string,
  chan Result) {
  out := make(chan string, cap(in))
  go func() {
    for url := range in {
      suffix := strings.ToLower(
        filepath.Ext(url))
      if suffix == ".htm" ||
        suffix == ".html" {
        results <- Check(url)
      } else {
        out <- url
      }
    }
    close(out)
  }()
  return out, results
}
```

Listing 10

can see that each creates its own **out** channel which it ultimately closes when it has finished processing. The last of these **out** channels is passed to the **sink()** function as its **in** channel – and this channel isn't closed until all the previous pipeline components have finished reading their **in** channels and closed their **out** channels. In view of this we know that once the **for ... range** loop in the **sink()** function has finished, all of the pipeline's preceding components have finished processing and no more results could be sent to the **results** channel, and hence the channel is safe to close.

The pipeline-based **headcheck** program ends up with five goroutines: one executing **sink()**, one started by **source()**, one started by **processHTML()**, one started by **processImages()**, and the main goroutine that **main()** executes in. All of these goroutines operate concurrently, passing type-safe values via channels, only terminating when their work is done, and with no explicit locks.

Thinking in terms of goroutines and channels is very different from thinking in terms of threads and locks, and may take some getting used to! Also, keep in mind that in this article we have seen just *some* of the possible ways of using goroutines and channels – many other approaches are possible. Go is a fascinating language, not just for its innovative approach to concurrency, but also for its clean syntax and very different approach to object orientation, and is well worth getting to know. ■

Further information

The Go Programming Language: http://golang.org/

```
func sink(in <-chan string,
  results chan Result) {
  for _ = range in {
    // Drain unprocessed URLs
  }
  close(results)
}
```

Listing 11