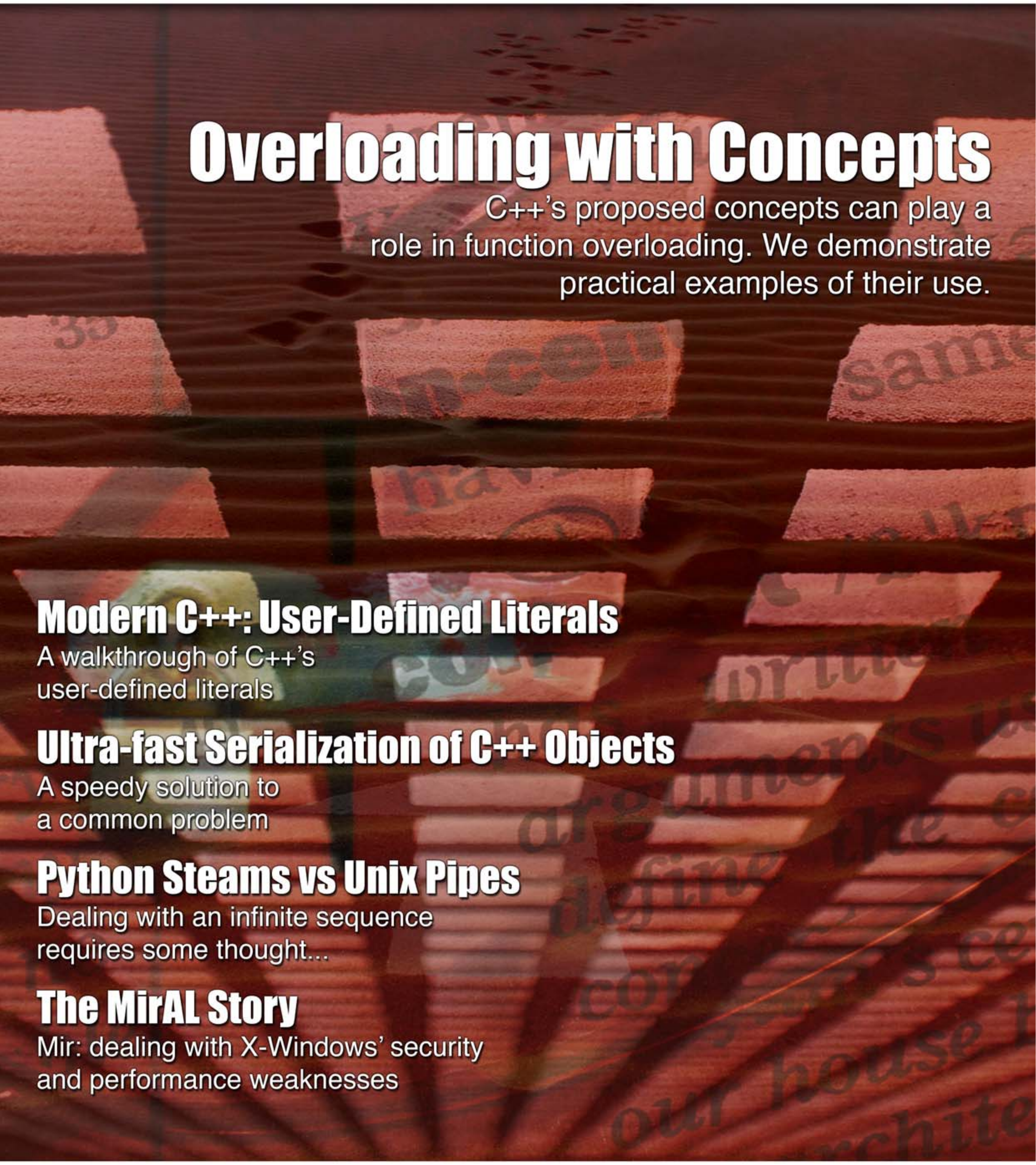
overload 136

DECEMBER 2016 £3



Overloading with Concepts

C++'s proposed concepts can play a role in function overloading. We demonstrate practical examples of their use.

Modern C++: User-Defined Literals

A walkthrough of C++'s user-defined literals

Ultra-fast Serialization of C++ Objects

A speedy solution to a common problem

Python Steams vs Unix Pipes

Dealing with an infinite sequence requires some thought...

The MirAL Story

Mir: dealing with X-Windows' security and performance weaknesses

A magazine of ACCU

ISSN: 1354-3172

JET
BRAINS

A Power Language Needs Power Tools

Smart editor with full language support
Support for C++03/C++11, Boost and libc++, C++ templates and macros.

Reliable refactorings
Rename, Extract Function / Constant / Variable, Change Signature, & more

Code generation and navigation
Generate menu, Find context usages, Go to Symbol, and more

Profound code analysis
On-the-fly analysis with Quick-fixes & dozens of smart checks

GET A C++ DEVELOPMENT TOOL THAT YOU DESERVE

R++

ReSharper C++
Visual Studio Extension for C++ developers

AC

AppCode
IDE for iOS and OS X development

CL

CLion
Cross-platform IDE for C and C++ developers

Start a free 30-day trial
jb.gg/cpp-accu

OVERLOAD 136

December 2016

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 137 should be submitted by 1st January 2017 and those for Overload 138 by 1st March 2017.

Overload is a publication of the ACCU

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

The ACCU

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

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

The Font of Wisdom

The choice of typeface can convey a tone. Frances Buontempo considers the plethora of dilemmas this presents.

Having just bought a new laptop, I have spent time installing things and as ever therefore failed to write an editorial. I noticed once I finally managed to find a command prompt in this new-fangled version of Windows, it seemed very unfamiliar. The first thing that stuck me was the font size, and in fact font itself. In conjunction with a recent outburst of font related puns on twitter[1], I was left thinking about fonts. First, why is it called a font? It is possible this comes from original letters for pamphlets and books being cast (think foundry → found → fount → font) in metal? The roots of this word may relate to pouring or gushing [found] and we can then find a split between fountains (founts) and cast containers. For example, I observe a trend [EnglishSE] towards using the phrase 'font of wisdom' (or knowledge) in place of 'fount of wisdom'. In the older form, fount, the idea is that of a fountain, or source, of something. The latter means a container, like a font or baptismal basin in a church. Wikipedia reminds me that font and typeface are often confused [Font]; the font is a specific size, weight and style of a typeface. Which font or typeface do you use to write your programs and scripts in? Has this been a conscious decision, is it just by habit, or IDE's default choice or even company diktat?

A couple of years ago at the ACCU conference, Kevlin Henny [Henney] observed that certain fonts, for example on book covers, pin down objects to specific points in history. How do fonts become trendy? Who invents them? Why are some people trying out using Comic Sans for technical presentations? Has anyone ever used it for a CV? I saw a CV in Courier, or at least some fixed width font, which rather impressed my boss at the time. Each new version of Microsoft Office seems to come with a new font as the default. This will date a document, to some extent. I recently saw a PowerPoint presentation and had a brief moment of shock at the choice of arrow-heads for bullet point. I haven't seen that used for years. Of course, all the trendy people have ceased to use PowerPoint anyway. Right?! Presentation formats have trends falling in and out of fashion.

We could suggest fonts start with the printing press, though having seen pictures of some very old hand-written books with very uniform calligraphy I could muse on an earlier history. Prescriptions of the exact height of hand-written letters such as uncial and half-uncial [Uncials] would distract us all. I presume having a specific format would make such documents and books easier to read. I am always surprised by how dreadful my handwriting tends to be, since I usually type at a computer and am therefore out of practice. Handwriting is slow, if you try to make sure it's readable. My typing is fast, as long as I ignore typos. I'm not clear how old moveable type is, and I thought Gutenberg invented the printing press but the internet [Internet] tells me I am wrong. 'He is not the inventor of the printing press, or of printing ink, or even of moveable type. Gutenberg's Bible was neither the first book printed using moveable type, nor the first book he printed using moveable type.' The same webpage claims he probably didn't have a beard either.

1. https://twitter.com/chrisoldwood/status/787400476354605056 'I shot the serif' being a 'Capital crime' apparently

Early computer programs would be holes in punched card, as we know. Once we started using letters – say I, J, K – to represent variables or try to write documents in word processors, we arrive at raster or bitmap fonts. When these are scaled up it becomes clear they are made of bits, and have squared steps where they apparently started with a curved line. This contrasts with a scalable graphic, or font, which is limited by the device displaying it. Instead of just having pixels in a grid set to a specific colour, the letter, glyph or whatever is stored as a curve which can then be scaled as desired. Actually, the common composite Bézier curve will be stored as control points and are defined by quadratic, cubic or even higher order equations. Storing the curve itself would require either a bitmap or something recursive, since we are trying to find a way to store a curve. Certainly computer screens have led to an explosion of fonts, from Verdana, Metafont to TrueType and beyond. It would seem that various fonts are supposed to be easier to read on different media. A whole slew of fonts designed for web-browsers seem to exist. In fact, Google Fonts [Google] offers an 'intuitive and robust directory of open-source designer web fonts'. I do like my directories to be robust. Wikipedia has a whole page devoted to web typography [Web typography]. Some specific issues crop up on webpages, in particular the need to be fast when rendering and the chance things will be in a variety of languages and even alphabets. I am still sure alphabet is not the generic word for glyphs, syllabaries, logographies or graphemes, and should perhaps just be reversed for the Greek alpha-beta. I digress.

It would be inappropriate to talk of fonts without briefly mentioning Comic Sans. We are, I hope, all aware that Comic Sans should be reserved for the appropriate time and place. That could be a child's party, or comic, again aimed at children, or a live code demonstration to ACCU London. (It was by Jon Skeet, so he can do what he wants.) A brief internet search for Comic Sans turns up many disparaging pages. For example, http://www.comicsanscriminal.com/ reminds us, 'All fonts have a personality and purpose.' Furthermore, http://bancomicsans.com/ quotes,

> There are bad types and good types, and the whole science and art of typography begins after the first category has been set aside. ~ Beatrice Warde

Category theory and types are another matter; the way you present your output sets a tone. The choice of font is just one part of the larger whole, which includes colour, layout and much more. Comics choose specific typefaces for different characters, perhaps as a form of leitmotif. The

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

visual layout brings the story alive. I wonder how webpages are perceived by visually-impaired visitors. If an audio description is read, does it take into account the font? Do websites even have audio descriptions?

Having touched on Comic Sans, this takes us to emojis. A recent 'news' story reported the development of a new set of emojis for older people [BBC], dubbed 'emoldjis'. I hadn't realised there is a formal process to create an emoji. With a bit of web-surfing I found a technical report [TR51] providing guidelines for interoperability of such characters across platforms. It appears the first Unicode emoji characters were added to Unicode 5.2 in 2009. Are there any programmer specific emojis I wonder? I observe an attempt at emoji domain names. A blog [Domain names] observes they can be difficult to navigate to without an emoji-dedicated keyboard. Websites do exist to generate them for an existing site, and you can then use the proffered hyperlink. Would you want to code in emojis? Once or twice I have seen proper mathematics symbols in comments in code, which is right and proper and doesn't show up properly if the Unicode used isn't supported in your editor of choice. If you find an editor that can cope, you could code in emoji. Or emojicode in fact; http://www.emojicode.org/

> Emojicode is a static strongly typed programming language…Emojicode is an open source, high-level, multi-paradigm, object-oriented programming language consisting of emojis, that allows you to build fast cross-platform applications while having a lot of fun. And it's 100% real.

The webpage gives a fine demonstration of Fibonacci number generation, but first doesn't seem to have any tests and second I can feel the pain of trying to correctly get that in print for *Overload* without even trying.

Aside from the character set, or perhaps together with it, we should consider punctuation. Early writing had no breaks between the words, let alone full stops to end sentences. This would make writing hard to parse, and indeed many programming languages tend to use punctuation to make the parser's life easier. Gradually we see spaces being introduced between words, and then various marks introduced to indicate pauses in the reading, such as a comma for a shorter pause. Or full stop for greater emphasis. I have seen many older texts tending to use a semicolon where a comma or perhaps period would be preferred nowadays. Often a sentence will last an entire paragraph. Modern writing seems to favour shorter sentences. In addition, some of our punctuation is used to disambiguate meanings. An example would be the difference between

> Charles the First walked and talked half an hour after his head was cut off.

versus

> Charles the First walked and talked; half an hour after his head was cut off.

Many other examples exist. Perhaps this just means English is deficient. I have a sense that Latin, for example, has many precise tenses and verb declensions which would make the meaning clearer, possibly reducing the need for punctuation. I wonder if there is an analogy between programming languages, such as C++, which require a lot of punctuation contrasted with ones which do not, such as Python, and human languages? Perhaps. Will we continue to get new punctuation marks, or do we have sufficient? Really?! Ah yes, interrobang (Unicode character U+203D, apparently). Developed by one Martin K. Speckter in 1962 [Interrobang], it hasn't fully caught on since it was rather difficult to type on a typewriter. It still begs the question, 'How would you punctuate a sentence like this?!' Do we need more punctuation, or should we return to simpler times, a lá New Roman?

Look at some code you have written recently. What font are you using to view it? Is that included in version control? I presume not. Are you using a fixed width font? There are times when you do want columns to line up, but many people are using proportional fonts to code instead of fixed-width ones. There is a brief discussion on Slashdot [Slashdot] about the use of proportional fonts, noting some claim they are quicker to read. Being Slashdot, the comments are worth a read, though one CdBee notes, '/. is meant to be the font of all knowledge, not the knowledge of all fonts.' People have fought wars (almost) over layout of braces and whitespace placement. This is all related to how easy something is to 'grok'. Some people cannot cope without colour coding. I suspect any UX style studies of the 'one true way' to present code will fail to take into account diversity and individual preference. You need something that works for you, but if you pair, or even mob, program, a setup that works for everyone is ideal, even if it's not possible. There is no one true way, or font of all wisdom on how to present things well. And there's no editorial either, but what did you expect? Before I finish, I must share a web article that considers the use of fonts in Blade Runner [Blade Runner]. It mentions 'Blade Runner's only reported instance of Eurostile Bold Extended' and that the details of the replicant Leon are almost in Caslon font, but, and I quote, 'despite everything else matching nicely, the top of the 6 is just wrong. I can only apologize for the discrepancy.' The analysis of the serial numbers and date formats that follow is highly recommended. I suspect I have found the source of the knowledge of all fonts. The fount of all things font?!

References

[BBC] http://www.bbc.co.uk/news/uk-england-coventry-warwickshire-37789947

[Blade Runner] https://typesetinthefuture.com/2016/06/19/bladerunner/

[Domain names] https://blog.uniteddomains.com/it-s-2016-where-are-our-emoji-domain-names-26215215fbd2#.i44gzyjz5

[EnglishSE] http://english.stackexchange.com/questions/95864/font-fount-of-information

[Font] https://en.wikipedia.org/wiki/Font

[found] http://www.etymonline.com/index.php?term=found

[Google] https://fonts.google.com/about

[Henney] https://www.infoq.com/presentations/unit-testing-tips-tricks

[Internet] https://www.fonts.com/content/learning/fontology/level-4/influential-personalities/gutenbergs-invention

[Interrobang] http://www.shadycharacters.co.uk/2011/04/the-interrobang-part-1/

[Slashdot] https://developers.slashdot.org/story/10/01/17/0715219/programming-with-proportional-fonts

[TR51] http://www.unicode.org/reports/tr51/

[Uncials] http://www.designhistory.org/Handwriting_pages/Uncials.html

[Web typography] https://en.wikipedia.org/wiki/Web_typography

The MirAL Story

The X-Windows system is all-pervasive but struggles with security and performance graphics. Alan Griffiths introduces Mir Abstraction Layer to deal with these issues.

What is Mir?

The project I'm working on is Mir. This is a replacement for the venerable X Windows (see 'We need a new windowing system'). Mir is a set of libraries providing the facilities for the participants in window management: the 'servers' that manage organising the windows onscreen and the 'clients' (or applications) that provide the content of the windows. In addition there are facilities to load plugin modules for different 'graphics platforms', 'input platforms' and renderers. At the time of writing, the supported platforms are the Mesa and Android graphics stacks and running the 'server' either directly on these drivers or as a 'guest' of either Mir itself or of X11.

There are multiple downstream projects using the 'client' side of Mir: there's Mir support available in GTK, Qt and SDL2. In addition, there's an X11 server that runs on Mir: Xmir, this allows X based applications to run on Mir servers.

On the server side, Mir hasn't had a range of projects using it: it has been in use by the Unity8 window manager used on phones and tablets and recently made available as a 'preview' on the Ubuntu 16.10 desktop.

While having a downstream project that uses Mir on real devices that ship to real customers has been a big benefit, the close association of the two projects has had some downsides.

We need a new windowing system

The X-Windows system has been, and remains, immensely successful in providing a way to interact with computers. It underlies many desktop environments and graphical user interface toolkits and lets them work together. But it comes from an era when computers were very different from now, and there are real concerns that are hard to meet.

In 1980 computers were big things managed by specialists and connecting them to one another was 'bleeding edge'. In that era, the cost of developing software was such that any benefit to be gained by 'listening in' on what was happening was negligible: there were few computers, they were isolated, and the work they did was not open to exploitation.

X-Windows developed in this environment and, through a series of extensions, has adapted to many changes. But it is inherently insecure: any application can find out what happening on the display (and affect it). You can write applications like Xeyes that tracks the cursor with its 'eyes' or 'Tickeys' that listens to the keyboard to generate typewriter noises.

X-Windows is poorly adapted to a world with millions of computers connected to the Internet, being used for credit card transactions and online banking, and managed by non-experts who willingly install programs from complete strangers.

There has been a growing realization that adapting X-Windows to the new requirements of security and graphics performance isn't feasible. There are at least two projects aimed at providing a replacement: Mir and Wayland/Weston. While some see these as competing, there are a lot of areas where they have common interests: they both need to interact with other software that previously assumed X11, and much of the work needed to introduce support an alternatives benefits both projects.

The itch

Because the 'client' side of Mir has multiple projects using it, the importance of a stable API and ABI has been both appreciated *and acted upon*. On the server side things haven't been so disciplined. The API has evolved gradually and the ABI has broken on almost every release.

The slow evolution of the server API has been managed by maintaining a 'compatibility branch' of Unity8 and releasing that at the same time as every Mir release. (I'm keeping the explanation simple: there are actually additional projects involved in this dance, which makes it even more involved and expensive.)

Maintaining and releasing these compatibility branches increases the effort involved in releasing Mir significantly. Changes need to made and tested across a family of projects belonging to separate teams. Because the pain involved increased gradually, this didn't get the attention I felt it deserved.

The unstable server API has another, indirect, cost: it makes it impractical for anyone outside of the few Canonical teams we work with to write and maintain a Mir server.

This seemed unlikely to change in the normal course of events. The API was not designed with ABI stability in mind and the development focus was on delivering more features and not reducing these costs. While some efforts have been made by the team which reduced the API 'churn', they didn't affect the underlying issue.

Elevating a system from chaos to order takes energy and conscious effort. And energy was constantly being drained by managing the release process.

Scratching the itch

Canonical has a policy of allowing staff to work on projects they choose for half a day each week (subject to some reasonable conditions). And, having checked with management, I elected to work on providing an alternative, stable API and ABI for writing of window managers.

I began by setting up a separate 'Mir Abstraction Layer' project (MirAL) and populating it with a copy of the code from the Mir 'example' servers. This immediately identified a series of bugs in the Mir packaging that had gone undetected. Nothing hard to fix, but obstacles to using Mir: headers referenced but not installed, overlooked dependencies on other projects, incorrect pkg-conf files, and so forth. I filed the Mir bugs and fixed them in my 'day job'.

I then started separating out the generic window management logic from the specifics in the examples and building an API between them. Thus emerged the three principles interfaces of this library: a 'window

Alan Griffiths Alan has 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

basic **window management functionality** like the placement of menus could be **shared between different approaches** to window management

management policy', a 'basic window manager' into which a policy slotted and 'window management tools' which provides functionality.

This meant that the basic window management functionality like the placement of menus could be shared between different approaches to window management: a 'normal' desktop, a 'tiling' version and 'kiosk' that could support embedded uses while not allowing the user to manipulate windows.

Having got these in place, along with some other meaningful concepts, I started rework to protect against the types of ABI breakage that were all too common with the existing server APIs. Data structures and virtual function tables in particular are fragile with respect to changes. One technique I used a lot was the 'Cheshire Cat' idiom as that avoids ABI breaking changes to virtual function tables.

Repurposing MirAL

I'd got to the point where I'd proved to myself that this approach could work when a new priority arrived in my 'day job'. This was to provide window management support for Unity8 on the desktop. Until this point, Unity8 had only had to deal with the 'one screen, one active application, one window' use case of the phone and then an extension of this to support 'sidestage' on tablets. And in Mir much of the 'window management support' was example and test code.

What was needed was a place to consolidate the existing window management support and iterate quickly towards a more integrated approach. Also, to support additional projects beyond Unity8 (both from within Canonical or from outside) this also needed to be a place where other shells and desktop environments can leverage this functionality.

It sounded a lot like what I'd started with MirAL. So MirAL switched from being my hobby to being my 'day job' and gained Unity8, a 'real world' shell, as a prospective downstream.

A colleague (Gerry Boland) who had done much of the work integrating Unity8 with Mir and I started working to make it possible to migrate the QtMir project (which did the integration) to use the new API and exploit the window management support it provided.

We copied the code from the 'QtMir' project into the MirAL source tree and started work on joining things together. This proved very helpful in refining the concepts in the MirAL API, identifying gaps in the functionality it provides and soon gave us confidence that this was a workable approach.

The effectiveness of this work has been established and recently work was started on joining things up the all the way into Unity8 and integrating with the more sophisticated window management it implements.

Working with applications

Toolkits with Mir backends

Another piece of work happening around the same time was the effort to get third party applications to work correctly with Mir shells. Most applications are not written directly against X-Windows but use 'toolkits' that provide higher level concepts. And these have requirements on window management (like putting tooltips in the right place).

Two toolkits of immediate significance are GTK+ (on which gnome applications are built) and Qt (which is widely used in Canonical) both of which already have optional Mir 'backends'. But the amount of testing these had got was limited whilst Mir work was focussed on the phone.

The window management work I was doing in MirAL and especially the sample 'miral-shell' became a testing ground how window management features interact with the Mir support in gtk-mir and qtubuntu. (And other client libraries such as SDL2.)

To aid with testing, MirAL comes with a handy script (miral-run) to set up the environment needed by these toolkits and run them against a Mir server.

X11 applications

Not all applications use these toolkits and supporting X11 applications by running the Xmir an X-server based on Mir is a bit fiddly. To facilitate testing this approach with MirAL there's another script 'miral-xrun' that finds a free port, starts an X server, runs the application and then closes the X server when the application exits.

Debugging window management

While working to track down problems in the interaction between toolkits and MirAL's window management I found the time to introduce an immensely helpful logging facility that logs all calls into the window management policy and all the calls made to the window management tools. This has been in discovering why what we see happens, happens. It has helped diagnose bugs in both MirAL and the toolkit backends. This can be used with any server based on MirAL by adding `--window-management-trace` to the command-line.

The state of MirAL right now

MirAL is available either from the Ubuntu archives, or from Launchpad [Launchpad] for building from source.

Using MirAL, it is very easy to create Mir based window managers. As an experiment colleague created this (rather silly) shell in under an hour: https://github.com/BrandonSchaefer/bad-shell.

Work is continuing to improve the window management capabilities and offer the 'missing' Mir facilities used by QtMir that are not currently supported by the new API. These will be presented in a form suitable for consumption by other projects.

For the latest information visit my 'Canonical Voices' blog [voices]. ■

References

[Launchpad] https://launchpad.net/miral

[voices] http://voices.canonical.com/alan.griffiths/category/miral/

Overloading with Concepts

Concepts can play a role in function overloading. Andrew Sutton shows us how.

This is the third, and long overdue, article in my series on C++ concepts. The first two articles focused on how concepts are used to constrain generic algorithms [Sutton15] and how concepts are defined [Sutton16]. This article describes a sometimes overlooked, frequently misunderstood, and yet extraordinarily powerful feature of concepts: their role in function overloading. Concepts are useful for more than just improving error messages and precise specification of interfaces. They also increase expressiveness. Concepts can be used to shorten code, make it more generic, and increase performance.

Before diving in, it's worth noting a few things that have happened since the publication of my last article. First, concepts were not included in C++17. Some committee members felt that there hasn't been sufficient time since the publication of the TS [N4549] to be confident that the design is appropriate, and many were undecided.

Second, GCC 6.2 was released in late August. This version includes a major update to two components of the concepts implementation. The diagnostics generator has been significantly revamped to provide precise diagnostics about the failure of a concept to be satisfied. The support for overloading on constraints (the topic discussed in this article), was completely rewritten to provide dramatic performance gains. In GCC, concepts can now be used for projects of significant size and complexity.

Finally, since concepts was not accepted into C++17, I have seen an increase in online content promoting concept emulation techniques over language support and even claims that expression SFINAE, constexpr if, **`static_assert`**, and clever metaprogramming techniques are sufficient for our needs. That's analogous to claiming that given **`if`** and **`goto`**, we don't need **`while`**, **`for`**, and range-**`for`**. Yes, it's logically correct, but in both cases we drag down the level of abstraction to specifying how things are to be done, rather than what should be done. The result is more work for the programmer, more bugs, and fewer optimization opportunities. C++ is not meant to be just an assembly language for template metaprogramming. Concepts allows us to raise the level of programming and simplify the code, without adding run-time overhead.

Recap

In my previous articles [Sutton15, Sutton16], I discussed a simple generic algorithm, **`in()`**, which determines whether an element can be found in a range of iterators. Here is an alternative version of the **`in()`** algorithm from the previous article. I've modified its constraints for the purpose of this article and also updated it to match some current naming trends in the C++ Standard Library (see Listing 1).

Andrew Sutton is an assistant professor at the University of Akron in Ohio where he teaches and researches programming software, programming languages, and computer networking. He is also project editor for the ISO Technical Specification, 'C++ Extensions for Concepts'. You can contact Andrew at asutton@uakron.edu.

```
template<Sequence S, Equality_comparable T>
  requires Same_as<T, value_type_t<S>>
bool in(const S& seq, const T& value) {
  for (const auto& x : range)
    if (x == value)
      return true;
  return false;
}
```

Listing 1

This rendition of **`in()`** takes a sequence instead of a range as its first argument, and an equality comparable value for its second. The algorithm has three constraints:

- the type of the **`seq`** must be a **`Sequence`**,
- the type of **`value`** must be **`Equality_comparable`**, and
- the **`value`** type must be the same as the element type of **`seq`**.

Here, **`value_type_t`** is a type alias that refers to the declared or deduced value type of **`R`**. The definitions of the **`Sequence`** and **`Range`** concepts needed for this algorithm look like Listing 2.

```
template<typename R>
concept bool Range() {
  return requires (R range) {
    typename value_type_t<R>;
    typename iterator_t<R>;
    { begin(range) } -> iterator_t<R>;
    { end(range) } -> iterator_t<R>;
    requires Input_iterator<iterator_t<R>>();
    requires Same_as<value_type_t<R>,
      value_type_t<iterator_t<R>>>();
  };
}

template<typename S>
concept bool Sequence() {
  return Range<R>() && requires (S seq) {
    { seq.front() } -> const value_type<S>&;
    { seq.back() } -> const value_type<S>&;
  };
}
```

Listing 2

This specification requires all **`Range`**s to have:

- two associated types named by **`value_type_t`** and **`iterator_t`**
- two valid operations **`begin()`** and **`end()`** that return input iterators,
- and that the value type of the range match that of the iterator.

Most **`Sequence`**s have the operations **`front()`** and **`back()`**, which return the first and last elements of the range. This isn't a fully developed

specifying how things are to be done, rather than what should be done ... more work for the programmer, more bugs, and fewer optimization opportunities

specification of a sequence, but it is sufficient for the discussion in this paper.

This seems reasonable. We can use the algorithm to determine if an element is contained within any sequence. Unfortunately, it no longer works for some collections:

```
std::set<int> answers { ... };
if (in(answers, 42)) // error: no front()
                     // or back()
...
```

This is unfortunate. We should clearly be able to determine if a key is contained within a set. But how do we do this?

Extending algorithms

For someone who knows concepts and the standard library, the solution in this case is obvious: just add another overload that accepts associative containers.

```
template<Associative_container A,
  Same_as<key_type_t<T>> T>
bool in(const A& assoc, const T& value) {
  return assoc.find(value) != s.end();
}
```

This version of `in()` has only two constraints: `A` must be an `Associative_container`, and `T` must be the same as key type of `A` (`key_type_t<A>`). For associative containers, we simply look up `value` using `find()` and then see if we found it by comparing to `end()`. That's likely to be much faster than a sequential search.

Note that, unlike the `Sequence` version, `T` is not required to be equality comparable. This is because the precise requirements of `T` are determined by the associative container, and those requirements are usually determined by a separate comparator or hash function.

The concept `Associative_container` is defined like Listing 3.

```
template<typename S>
concept bool Associative_container() {
  return Regular<S> && Range<S>() &&
    requires {
      typename key_type_t<S>;
      requires Object_type<key_type_t<S>>;
    } &&
    requires (S s, key_type_t<S> k) {
      { s.empty() } -> bool;
      { s.size() } -> int;
      { s.find(k) } -> iterator_t<S>;
      { s.count(k) } -> int;
    };
}
```

Listing 3

That is, an associative container is `Regular`, defines a `Range` of elements, has a `key_type` (which may differ from the `value_type`), and a set of operations including `find()`, etc.

As with `Sequence` before, this is clearly not an exhaustive list of requirements for an associative container. It doesn't address insertion and removal, and excludes specific requirements for `const` iterators. Moreover, we haven't really described how we expect `size()`, `empty()`, `find()` and `count()` to behave. For now, we'll just rely on our existing knowledge of the Standard Library.

This concept includes all of the associative containers in the C++ Standard Library (`set`, `map`, `unordered_multiset`, etc.). It also includes non-standard-library associative containers, assuming that they expose this interface. For example, this overload would work for all of Qt's associative containers (`QSet<T>`, `QHash<T>`, etc.) [Qt].

To use concepts to extend algorithms, we need to understand how the compiler can tell a plain `Sequence` from an `Associative_container`. In other words, what happens when we call `in()`?

```
std::vector<int> v { ... };
std::set<int> s { ... };

if (in(v, 42))// Calls the `Sequence` overload
  std::cout << "found the answer...";
if (in(s, 42))// Calls the
              // `Associative_container` overload
std::cout << "found the answer...";
```

For each call to `in()`, the compiler determines which function is called based on the arguments given. This is called *overload resolution*. This is an algorithm that attempts to find a single best function (amongst one or more candidates) to call based on the arguments given.

Both calls of `in()` refer to templates so the compiler performs template argument deduction and then form function declaration specializations based on the results. In both cases, deduction and substitution succeed in the usual and predictable way, so we have to choose amongst two specializations at each call site. This is where the constraints enter into the equation. Only functions whose constraints are satisfied can be selected by overload resolution.

In order to determine if a function's constraints are satisfied, we substitute the deduced template arguments into the associated constraints of the function's template declaration, and then we evaluate the resulting expression. The constraints are satisfied when substitution succeeds, and the expression evaluates to `true`.

In the first call to `in()`, the deduced template arguments are `std::vector<int>` and `int`. These arguments satisfy the constraints of `Sequence` but not those of the `Associative_container` because a `std::vector` does not have `find()` or `count()`. Therefore, the `Associative_container` candidate is rejected, leaving only the `Sequence` candidate.

we can continue extending the definition of a generic algorithm by adding overloads that differ only in their constraints

In the second call to `in()`, the deduced arguments are `std::set<int>` and `int`. The resolution is the opposite of the one before: a `std::set` is never a `Sequence` because it lacks `front()` and `back()`, so that candidate is rejected, and overload resolution selects the `Associative_container` candidate.

In this cases the resolution is straightforward, and many readers will readily recognize the similarity to the `enable_if` technique used today. This works because the constraints on both overloads are sufficiently exclusive to ensure that a container satisfies the constraints of one template or the other, but not both.

The situation gets a bit more interesting if we want to add more overloads of this algorithm. We could extend the algorithm for specific types or templates like we might have done without concepts. Essentially, we could enumerate the valid definitions of `in()` for those types. For example, extending `in()` for WinRT's `Map` class [WinRT] might require a declaration like this:

```
template<typename K, typename V, typename C>
bool in(const Platform::Collections
              ::Map<K, V, C>& map, const K& k) {
  return map.HasKey(k);
}
```

When there are many such viable definitions, this quickly becomes tedious and unmanageable. For any data structure that might represent a set of keys, we need a new overload. This simply does not scale.

If we're lucky, many of those new enumerated overloads will have identical definitions. That would certainly be the case for WinRT's `Map` and `MapView` classes; both would return `map.HasKey(k)`. In that case, we can unify their definitions into a single, more general template with appropriate constraints. For example:

```
template<WinRtMap M>
bool in(const M& map, const key_type_t<M>& k) {
  return map.HasKey(k);
}
```

Here, `WinRtMap` would be a concept requiring members common to both `Map` and `MapView`. This would also accept any other map implementations that the library accrues over time, assuming they satisfied the `WinRtMap` constraints.

In general, we can continue extending the definition of a generic algorithm by adding overloads that differ only in their constraints. There are exactly three cases that we need to consider when overloading with concepts:

1. Extend a definition by providing an overload that works for a completely different set of types. The constraints of these new overloads would either be mutually exclusive or have some minimal amount of overlap with existing constraints.
2. Provide an optimized version of an existing overload by specializing it for a subset of its arguments. This entails creating a new overload that has stronger constraints than it's more general form.
3. Provide a generalized version that is defined in terms of constraints shared by one or more existing overloads.

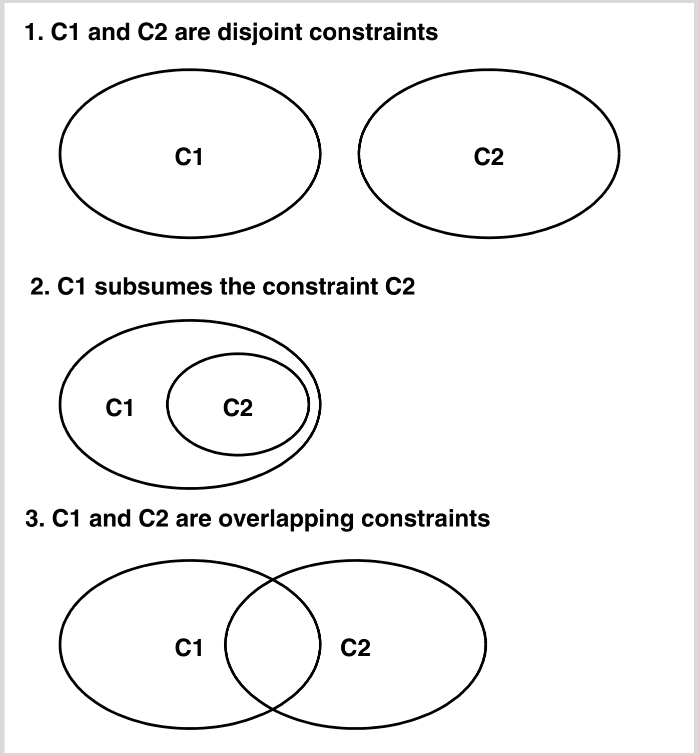

1. C1 and C2 are disjoint constraints
C1
C2
2. C1 subsumes the constraint C2
C1
C2
3. C1 and C2 are overlapping constraints
C1
C2

Figure 1

The three cases can easily be thought of in terms of the Venn Diagrams shown in Figure 1, which shows the possible relationships between the constraints of overloaded functions.

Each case in the Figure 1 corresponds to one of the cases above. This section is primarily an example the first case because we generally expect `Sequence`s and `Associative_container`s to be fundamentally different data abstractions.

When constraints are not (mostly) disjoint multiple candidates can be viable, the compiler must determine the best possible candidate for the call. I explain this process in the next section. However, if the compiler can't determine a best candidate, the resolution is ambiguous. In fact, this is the reason I changed the first `in()` algorithm to require `Sequence`s instead of just `Range`s. It minimizes the amount of overlap and therefore likelihood of ambiguity.

Having disjoint constraints does not guarantee that a call will be unambiguous. We could, for example, try to define a container that satisfies the requirements of both `Sequence` and `Associative_container`. In this case, both overloads would be viable, but neither overload is inherently better than the other. Unless we

Constraint subsumption allows us to optimize generic algorithms based on the interfaces provided by their arguments

```
template<typename I>
concept bool Forward_iterator() {
  return Regular<I>() && requires (I i) {
    typename value_type_t<I>;
    { *i } -> const value_type_t<I>&;
    { ++i } -> I&;
  };
}
```

Listing 4

added new overloads to accommodate this kind of data structure, the result would be an ambiguous resolution.

That said, **Sequence** and **Associative_container** actually do have overlapping constraints; they both require the **Range** concept. We could consider these overloads as being an instance of the third case. This hints that there may be an algorithm that can be defined in terms of the intersecting requirements. But it's not quite so simple. I intend to discuss these issues in a future article.

The second case is an important feature of generic programming in C++ and is the basis of type-based optimizations in generic libraries. Constraint subsumption allows us to optimize generic algorithms based on the interfaces provided by their arguments. This is the topic of the next section.

Specializing algorithms

For this section, we will leave behind our familiar **in()** algorithm and focus on the C++ iterator hierarchy because it naturally lends itself to this discussion.

In some cases, we can define data structures with an extended set of properties or operations that can be used to define more permissive or more efficient versions of an algorithm. This idea is are embodied by the standard library's iterator hierarchy of iterators.

Forward iterators can be used to traverse a sequence in only one direction (forward) by advancing one element at a time using **++**. Listing 4 is a simple concept for forward iterators.

Based on this concept, we can define two useful algorithms: one that advances multiple steps using a loop and one that computes the number of steps between two iterators (see Listing 5).

For **advance()**, **n** must be non-negative because forward iterators cannot go backwards. Bidirectional iterators, however, can be used to traverse a sequence in both directions (forward and backward) by advancing one element at a time using **++** or **--**. Here is that concept.

```
template<typename I>
concept bool Bidirectional_iterator() {
  return Forward_iterator<I>() && requires (I i)
  {
    { --i } -> I&;
  };
}
```

```
template<Forward_iterator I>
void advance(I& iter, int n) {
  // precondition: n >= 0
  while (n != 0) { ++iter; --n; }
}
template<Forward_iterator I>
int distance(I first, I limit) {
  // precondition: limit is reachable from first
  for (int n = 0; first != limit; ++first, ++n)
    ;
  return n;
}
```

Listing 5

Bidirectional_iterator is defined in terms of **Forward_iterator**. In other words, a bidirectional iterator is a forward iterator that can also move backwards.

Bidirectional_iterator's set of requirements completely subsumes that of **Forward_iterator** (case 2 in Figure 1). As a result, whenever **Bidirectional_iterator<X>** is true (for all **X**), **Forward_iterator<X>** must also be true. In this case we say that **Bidirectional_iterator** *refines* **Forward_iterator**.

This refinement lets us define a new version of **advance()** that can move in both directions.

```
template<Bidirectional_iterator I>
void advance(I& iter, int n) {
  if (n > 0)
    while (n != 0) { ++iter; --n; }
  else if (n < 0)
    while (n != 0) { --iter; ++n; }
}
```

The **Bidirectional_iterator** concept allows us to relax the precondition of **advance()**, so that we can used negative values of **n**. On the other hand, **Bidirectional_iterator** provides no new information that could help us improve **distance()**.

We can, however, provide optimizations of both **advance()** and **distance()** for random access iterators. These iterators can be used to traverse a sequence in two directions but can advance multiple elements in one 'step' using **+=** or **-=**. We can also count the distance between two iterators by subtracting them. A simplified version of that concept can be defined like this:

```
template<typename I>
concept bool Random_access_iterator() {
  return Bidirectional_iterator<I>() &&
    requires (I i, int n) {
    { i += n } -> I&;
    { i -= n } -> I&;
    { i - i } -> int;
  };
}
```

The **Random_access_iterator** concept refines **Bidirectional_iterator**; it adds three new required operations. By providing these operations, we can construct a optimized versions of **advance()** and **distance()** that do not require loops.

```
template<Random_access_iterator I>
void advance(I& iter, int n) {
  iter += n;
}

template<Random_access_iterator I>
int distance(I first, I limit) {
  return limit - first;
}
```

We can use these algorithms to define a large number of useful operations. For example, Listing 6 is an implementation of **binary_search**.

The algorithm is defined for forward iterators, but of course it can also be used for bidirectional and random access iterators too. The versions of **advance()** and **distance()** that are used depend on the kind of iterator passed to the algorithm. When used with forward and bidirectional iterators, the algorithm is linear in the size of the input range. For random access iterators, the algorithm is much faster since **distance()** and **advance()** don't require extra traversals of the input sequence.

The ability to specialize algorithms by constraints and by types is critical for the performance of C++ generic libraries. Simply put, this is the killer app of templates and generic programming. Concepts make it much easier to define and use these specializations. But how does the compiler know which overload to choose?

In the previous examples using sequences and associative containers, only one overload of **in()** was ever viable since the arguments were either one or the other, but not both. However, if we call **binary_search()** with random access iterators, say pointers into an array, all three overloads of **advance()** and both overloads of **distance()** will be viable. This makes sense. Every implementation of those functions are perfectly well defined for pointers.

In this case, the compiler must choose the best amongst the viable candidates. Roughly speaking, C++ considers one function to be 'better' than another using the following rules:

1. Functions requiring fewer or 'cheaper' conversions of the arguments are better than those requiring more or costlier conversions.
2. Non-template functions are better than function template specializations.
3. One function template specialization is better than another its parameter types are more specialized. For example, **T*** is more specialized than **T**, and so is **vector<T>**, but **T*** is not more specialized than **vector<T>**, nor is the opposite true.

The Concepts TS adds one more rule:

```
template<Forward_iterator I, Ordered T>
  requires Same_as<T, value_type_t<I>>()
bool binary_search(I first, I limit,
                   T const& value) {
  if (first == limit)
    return false;
  auto mid = first;
  advance(mid, distance(first, limit) / 2);
  if (value < *mid)
    return search(first, mid, value);
  else if (*mid < value)
    return search(++mid, limit, value);
  else
    return true;
}
```

Listing 6

4. If two functions cannot be ranked because they have equivalent conversions or are function template specializations with equivalent parameter types, then the better one is *more constrained*. Also, unconstrained functions are the least constrained.

In other words constraints act as a tie-breaker for the usual overloading rules in C++. The ordering of constraints (more constrained) is essentially determined by the comparing sets of requirements for each template to determine if one is a strict superset of another.

In order to compare constraints, the compiler first analyzes the associated constraints of the function in order to build a set of so-called atomic constraints. They are 'atomic' because they cannot be broken down into smaller bits. Atomic constraints includes C++ constant expressions (e.g., type traits) and requirements in a *requires-expression*.

For example, in the resolution of **advance()**, when called with a random access iterator, the set of constraints for each overload are:

Concept	Atomic requirements

`Forward_iterator`	`value_type_t<I>`
	`{ *i } -> value_type_t<I> const&`
	`{ ++i } -> I&`

`Bidirectional_iterator`	`value_type_t<I>`
	`{ *i } -> value_type_t<I> const&`
	`{ ++i } -> I&`
	`{ --i } -> I&`

`Random_access_iterator`	`value_type_t<I>`
	`{ *i } -> value_type_t<I> const&`
	`{ ++i } -> I&`
	`{ --i } -> I&`
	`{ i += n } -> I&`
	`{ i -= n } -> I&`
	`{ i - j } -> int`

For brevity, I excluded the **Regular<I>** constraint appearing in **Forward_iterator** since it (and its requirements) are common to all iterator concepts. Comparing these, we find that **Bidirectional_iterator** has a strict superset of the requirements of **Forward_iterator**, and **Random_access_iterator** has a strict superset of the requirements of **Bidirectional_iterator**. Therefore, **Random_access_iterator** is the most constrained, and that overload is selected.

The new overloading rule does not guarantee that overload resolution will succeed. In particular if the two viable candidates have overlapping or logically equivalent constraints, the resolution will be ambiguous. There are a few of reasons this might happen.

Semantic refinement

In some cases, refinements are purely semantic. They do not not provide operations that the compiler can use to differentiate overloads. In fact, this problem appears in the standard iterator hierarchy: input and forward iterators share exactly the same set of operations.

Conceptually an input iterator is an iterator that represents a position in an input stream. As an input iterator is incremented, previous elements are consumed. That is, previously accessed elements are no longer reachable through that iterator or any copy of that iterator. In contrast, forward iterator do not consume their elements when incremented. Previously accessed elements can be accessed by copies. This is typically referred to as the multipass property. This is a purely semantic property. Listing 7 is a concept for input iterators and an revised concept for forward iterators.

All of the syntactic requirements are defined in the **Input_iterator** concept. The **Forward_iterator** concept just includes **Input_iterator**s. In other words, **Forward_iterator**'s set of requirements is exactly the same as that of **Input_iterator**'s. If we tried to define overloads requiring these concepts, the result would always be ambiguous (neither is better than the other).

```
template<typename I>
concept bool Input_iterator() {
  return Regular<I>() && requires (I i) {
    typename value_type_t<I>;
    { *i } -> value_type_t<I> const&;
    { ++i } -> I&;
  };
}

template<typename I>
concept bool Forward_iterator() {
  return Input_iterator<I>();
}
```

Listing 7

Differentiating between these concepts is actually useful. For example, one of `vector`'s constructors (see Listing 8) has a more efficient implementation for forward iterators than for input iterators.

This doesn't work if the compiler can't tell a `Forward_iterator` from an `Input_iterator`.

We can fix this by adding new syntactic requirements to the `Forward_iterator` concept that relate to its rank in the iterator hierarchy. This has traditionally been done using *tag dispatch*: associating a tag class (an empty class in an inheritance hierarchy) with iterator type for the purpose of selecting appropriate overloads. That associated type is its `iterator_category`. A revised `Forward_iterator` might look like this:

```
template<typename I>
concept bool Forward_iterator() {
  return Input_iterator<I>() && requires {
    typename iterator_category_t<I>;
    requires Derived_from<I,
      forward_iterator_tag>();
  };
}
```

With this definition, `Forward_iterator`'s requirements now subsume those of `Input_iterator`, and the compiler can now differentiate the overloads above. As an added benefit, using random access iterators will be even more efficient since `distance()` requires only a single integer operation.

As another example, C++17 adds a new iterator category: contiguous iterators. A contiguous iterator is a random access iterator whose referenced objects are allocated in adjacent regions of memory whose addresses increase with each increment of the iterator. This opens the door to a number of low-level memory optimizations. It's also quite obviously a purely semantic property. So if we want to define a new concept, we'll need to differentiate it from `Random_access_iterator`. Fortunately, we just defined the machinery to do so.

```
template<Object_type T, Allocator_of<T> A>
class vector {
  template<Input_iterator I>
    requires Same_as<T, value_type_t<I>>()
  vector(I first, I limit) {
    for ( ; first != limit; ++first)
      push_back(*first); // O(log n) allocations
  }

  template<Forward_iterator I>
    requires Same_as<T, value_type_t<I>>()
  vector::vector(I first, I limit) {
    reserve(distance(first, limit));
      // 1 allocation
    insert(begin(), first, limit);
  }
  // ...
```

Listing 8

```
template<typename I>
concept bool Contiguous_iterator() {
  return Random_access_iterator<I>() && requires {
    requires Derived_from<I,
    contiguous_iterator_tag>();
  };
}
```

Listing 9

Tag classes are not the only way to solve this problem. I'm just reusing existing standard library infrastructure to solve the problem in this paper. In fact, the only concepts that require these tag classes are `Forward_iterator` and `Contiguous_iterator`. We don't actually need any of the other tag classes. We could simply use an associated type trait, variable template, or even an extra operation. In other words, we could do something like Listing 10 for forward iterators.

All of these approaches would yield the same result; the ability for the compiler to differentiate overloads requiring these concepts.

As a general rule, this technique should only be used to differentiate concepts that vary only in their semantics. Prefer to define concepts so that their interfaces reflect their different semantics.

Conclusions

In this article, we took a first look at how concepts can be used to extend and specialize algorithms through overloading. These ideas have been around for a long time, and in that time, we've developed some fairly advanced type-based techniques to manipulate overload outcomes. With concepts, these ideas are codified as part of the language; you simply write appropriate constraints for your algorithms and overload resolution does all the hard work for you.

At least, that's the ideal. Sometimes it can be a bit tricky to avoid ambiguities when there are overlapping constraints. Case in point, I've had to massage the constraints on the `in()` algorithm to make these examples work, but they do work. The next article in this series will be dedicated to the generalization of constrained algorithms and techniques for untangling ambiguous resolutions. ■

Acknowledgements

Bjarne Stroustrup provided a number of suggestions to improve the structure and content of this article.

References

[N4549] Sutton, Andrew (ed). ISO/IEC Technical Specification 19217. Programming Languages – C++ Extensions for Concepts, Aug 2015.

[Qt] Qt. 'Qt Documentation: Container Classes'. http://doc.qt.io/qt-4.8/containers.html.

[Sutton15] 'Introducing concepts'. ACCU *Overload*. Vol 129. Oct 2015.

[Sutton16] 'Defining concepts'. ACCU *Overload*. Vol 131. Oct 2105.

[WinRt] 'WinRT Documentation: Platforms::Collections::Map Class'. https://msdn.microsoft.com/en-us/library/hh441508.aspx.

```
template<typename T>
constexpr bool is_forward_iterator_v = false;

template<typename T>
constexpr bool is_forward_iterator_v<T*> = true;

template<typename I>
concept bool Forward_iterator() {
  return Input_iterator<I>() &&
    is_forward_iterator_v<T*>;
}
```

Listing 10

Ultra-fast Serialization of C++ Objects

Serialising and de-serialising is a common problem. Sergey Ignatchenko and Dmytro Ivanchykhin demonstrate one way to do this quickly.

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

Task definition

Recently, we were working on a system which required an extremely fast (ideally, the fastest possible) serialization of the state of the Reactor/Finite State Machine (FSM). In addition, we knew for sure that deserialization would happen with exactly the same executable; in other words, we didn't care at all about either (a) cross-platform issues or (b) extensibility.

Where it came from

The whole task comes from exploiting deterministic Reactors/FSMs. As discussed in [NoBugs15] and [NoBugs16], as soon as we have a deterministic Reactor/FSM, it is possible to use this determinism to achieve such things as production post-mortem analysis, and low-latency fault tolerance. For example, for post-mortem analysis, it is sufficient to write all the inputs of the deterministic Reactor/FSM, and in the case of a crash to replay it from the very beginning.

On the other hand, keeping the whole history of the Reactor inputs is usually impractical, so we need to resort to some kind of 'circular buffer' [NoBugs15]. To be able to observe the last *N* seconds of the life of the Reactor/FSM *before* the crash, the 'circular buffer' needs to contain (a) a snapshot of the current state of the Reactor/FSM, and (b) all the inputs received after this snapshot is taken. To achieve low-latency determinism-based fault tolerance, the logic is more complicated, but the snapshot of the current state is still required.

And as soon as we've said 'we need to make a snapshot', we need to serialize our state one way or another. Moreover, we need to do it Damn Fast – otherwise this debugging/fault tolerance feature would become too expensive. On the positive side – in practice, serialization will happen to memory (and in case of a production post-mortem – it won't even be used in any way until program crashes), so we'll be dealing with purely serialization code, with very little overhead to mask any of our performance blunders.

```
struct Y {
  int yy;
};
struct X {
  int xx;
  struct Y* y; // allocated via
               // malloc(); 'owning' pointer
  int z;
};
```

Listing 1

Note that in both these cases we can be 100% sure that we'll be deserializing this state on the executable which is *identical* to the executable which serialized the state. In other words, all the usual serialization/marshalling problems such as different alignments, endianness, etc. – do NOT apply here. ☺

One more case when we know for sure that it is *exactly* the same executable is when we're serializing data for inter-thread transfers within the same process; as a result, techniques discussed below will work in this case too. However, whether our serialization is optimal in such scenarios is not that obvious. In some cases – specifically, if you do NOT need to reconstruct a modifiable state on receiving side and are just passing messages around – flattening techniques such as those by FlatBuffers, MAY still happen to be faster (on the deserialization side, that is).

The fastest way to serialize – C

Now, as we have our task defined as 'the fastest possible serialization for in-memory structure, assuming that it will be deserialized by exactly the same executable', we can start thinking about implementing it.

First, let's consider serializing a state in a C program.

Usually, FSM/Reactor state can be described as a kind of generalized tree, with each of the nodes being a C struct, and containing 'owning' pointers to other allocated C structs. As a simple example, see Listing 1.

And the fastest way to serialize struct X, will be something along the lines of Listing 2. Unless we're resorting so some trickery with allocators or 'flattening' of our original structure, it is extremely difficult to beat this code performance-wise.

Deserialization would work along the lines of Listing 3. Deserialization is inevitably slower than serialization (there is an expensive **`malloc()`** within, ouch) – but it is pretty much inevitable for the kind of data structure we're working with. Also, for our use cases described above, deserialization will happen MUCH more rarely than serialization (on program crash or on hardware catastrophic failure), so we don't really care too much about the performance of deserialization – we just need it to work.

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko and Dmytro Ivanchykhin using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

Dmytro Ivanchykhin has 10+ years of development experience, and has a strong mathematical background (in the past, he taught maths at NDSU in the United States). Dmytro can be contacted at d_ivanchykhin@yahoo.com

as soon as we've said '**we need to make a snapshot**', we need to **serialize our state** one way or another

```
struct OutMemStream {
  uint8_t* pp;
  uint8_t* ppEnd;
};
inline void writeToStream( OutMemStream* dst,
    void* p, size_t sz ) {
  assert( dst->pp + sz < ppEnd );
    //in the real-world, think what to do here
  memcpy( dst->pp, p, sz );
  dst->pp += sz;
}
void serializeX( OutMemStream* dst, X* x ) {
  writeToStream( dst, x, sizeof(X) );
  writeToStream( dst, x->y, sizeof(Y) );
  //that's it!
}
```

Listing 2

From C to C++

C++ serialization

Ok, now let's try to rewrite it into C++ (where we're no longer restricted to Plain Old Data a.k.a. POD). To make things closer to reality, let's serialize the class X in Listing 4, which contains (directly or indirectly) two `std::string`s, a `std::vector`, and a `std::unique_ptr`.

Once again, it is very difficult to beat this serialization (that is, unless playing some dirty tricks with flattening or allocators). Nonetheless, it

```
struct InMemStream {
  uint8_t* pp;
  uint8_t* ppEnd;
};
inline void readFromStream( InMemStream* src,
    void* p, size_t sz ) {
  assert( src->pp + sz < ppEnd );
  memcpy( p, src->pp, sz );
  src->pp += sz;
}
void deserializeX( SomeMemStream* src, X* x ) {
  readFromStream( src, x, sizeof(X) );
    // x->y contains garbage at this point(!)
    // ok, not exactly garbage - but a pointer
    // which is utterly invalid in our current space
  x->y = malloc( sizeof(Y) );
    //phew, no garbage anymore
  assert( x->y );
  readFromStream( src, x->y, sizeof(Y) );
}
```

Listing 3

```
class OutMemStream {
  public:
  inline void write( const void* p, size_t sz );
    // implemented along the lines of the
    // writeToStream() above
  inline void writeString( const std::string& s )
  {
    size_t l = s.length();
    write( &l, sizeof(size_t) );
    write( s.c_str(), l );
  }

  template<class T>
  inline
  void writeVector( const std::vector<T>& v ) {
    // NB: can be further optimized by writing the
    // whole v.data() at once.
    size_t sz = v.size();
    write( &sz, sizeof(size_t) );
    for( auto it : v )
      it.serialize( this );
  }
};
class Y {
  public:
  int yy;
  std::string zz;
  std::string zz2;
  void serialize( OutMemStream* dst ) const;
  Y( const PreDeserializer& );
     //pre-deserializing constructor, see below
  Y( InMemStream* src );
     //deserializing constructor
};
class X {
  int xx;
  std::unique_ptr<Y> y;
  std::vector<Y> vy;
  void serialize( OutMemStream* dst ) const;
  X( const PreDeserializer& ) const;
     //pre-deserializing constructor
  X( InMemStream* src );
     //deserializing constructor
};
void Y::serialize( OutMemStream* dst ) const {
  dst->write( this, sizeof(Y) );
  dst->writeString( zz );
  dst->writeString( zz2 );
  // NB: we do NOT serialize POD members
  // such as 'yy' separately
}
```

Listing 4

When deserializing inherited/ polymorphic objects ... we cannot really overwrite the whole object without the risk of overwriting virtual table pointer(s)

```
void X::serialize( OutMemStream* dst ) const {
  dst->write( this, sizeof(X) );
  y->serialize( dst );
  dst->writeVector( vy );
  // NB: we do NOT serialize POD members
  // such as 'xx' separately
}
```

Listing 4 (cont'd)

contains all the necessary information (in fact, a little bit more than that) to deserialize our object when/if we need it.

C++ deserialization – Take 1

However, deserialization in C++ is not going to be that simple. The problem here is that as we didn't store data on a per-field basis, which means that unless we do something, on deserialization we'll be overwriting 'owning' pointers with their values in the old program (and rewriting this garbage with a pointer to allocated data later). While this was ok for POD types in C, in C++ it can cause all kinds of trouble (such as an attempt to free a non-allocated pointer) unless we're careful. The approach in Listing 5, however, is very clean in this regard.

Overall, deserialization of a class T goes as follows:

```
class PreDeserializer {
};  // just an empty class, to be used as a flag
    // to constructor

class InMemStream {
  uint8_t* pp;
  uint8_t* ppEnd;
  public:
  inline void read( void* p, size_t sz );
    // implemented along the lines
    // of the readFromStream() above
  inline void constructString( std::string* s ) {
    size_t l;
    read( &l, sizeof(size_t) );
    assert( pp+l < ppEnd );
    new( s ) std::string(
      reinterpret_cast<const char*>(pp), l );
    pp += l;
  }

  template<class T>
  inline void constructVector( std::vector<T>* v )
  {
    size_t sz;
    read( &sz, sizeof(size_t) );
    new( v ) std::vector<T>;
```

Listing 5

```
    for( size_t i=0; i < sz ; ++i ) {
      v->push_back( T( this ) );
    }
  }
};
Y::Y( const PreDeserializer& ) {
  // here we need to construct a valid object
  // just ANY valid object, preferably the
  // cheapest one to be constructed-destructed,
  // as it will be destructed right away :-)
}

Y::Y( InMemStream* src ) {
  // at this point 'zz' and 'zz2' are already
  // constructed we cannot call src->read(this) as
  // it will overwrite valid 'zz'/'zz2' causing all
  // kinds of trouble.
  zz.~basic_string<char>();
  //no idea why zz.~string() doesn't work
  zz2.~basic_string<char>();
  // now 'zz'/'zz2' are no longer constructed,
  // and we can overwrite them safely. On the
  // other hand, starting from this point, we're
  // NOT exception-safe
  src->read( this, sizeof(Y) );
  //at this point 'zz'/'zz2' contain garbage
  src->constructString( &zz );
  src->constructString( &zz2 );
  // phew, no garbage anymore,
  // 'this' is once again a valid object
  // and we're again exception-safe
}

X::X( const PreDeserializer& ){
  // nothing here; we do NOT really need
  // anything from here
}

X::X( InMemStream* src ) {
  // at this point 'y' is already constructed
  // we cannot call src->read(this) as it will
  // overwrite valid 'y' and 'vy' causing all
  // kinds of trouble.
  vy.~vector<Y>();
  y.~unique_ptr<Y>();
  // now 'y' and 'vy' are no longer constructed,
  // and we can overwrite them safely. On the other
  // hand, starting from this point, we're NOT
  // exception-safe
  src->read( this, sizeof(X) );
  //at this point 'y' and 'vy' contain garbage
```

Listing 5 (cont'd)

```
  new(&y) std::unique_ptr<Y>( new Y(src) );
  src->constructVector( &vy );
  // phew, no garbage anymore,
  // 'this' is once again a valid object
  // and we're again exception-safe
}
```

Listing 5 (cont'd)

- We construct an object of our class T, constructing all its non-POD members using Pre-Deserialization constructors (we don't need to construct the members at all, but there is no way to avoid it in C++)
- Within the object deserializing constructor, we have the following 'sandwich':
  - We destruct all non-POD members by explicitly calling their respective destructors. It gives us the right to overwrite them.
  - We overwrite the whole object T via **memcpy()**. At this point, non-POD members will contain garbage (more precisely, pointers which are invalid in our current space).
  - We re-construct all the non-POD members via their deserializing constructor. No garbage anymore, and we're ready to go. ☺

Our Take 1 approach will work well – that is, until we need to deal with base classes, and especially polymorphic classes. ☹ Polymorphic objects, among other things, contain a so-called 'Virtual Table Pointer', and overwriting it almost universally qualifies as a 'pretty bad idea'. ☹ Which leads us to the following...

C++ deserialization – Take 2, inheritance-friendly

Let's consider the same classes X and Y, with class X having a **unique_ptr<Y>**, but let's say that Y is a polymorphic base class, so **unique_ptr<Y>** can be either an instance of Y, or an instance of YY.

Strictly speaking, our original serialization already has all the information we need; however, extracting it can be quite cumbersome without knowing the exact class layout (and this is compiler-specific). So, we'll modify our serialization a bit (see Listing 6).

Here, we're sacrificing a tiny bit of performance on serialization (sigh) to keep things very cross-platform and not to depend on the exact class layout; on the other hand, the penalty here is pretty small (we're speaking at most about 1–2 CPU clocks plus a pipeline stall per **polymorphicSerialize()**, though in practice usually it will be much less than that due to branch predictions).

Now to deserialization. When deserializing inherited/polymorphic objects (and let's not forget about multiple inheritance and virtual bases) we cannot really overwrite the whole object without the risk of overwriting virtual table pointer(s)[1]. As a result, the best way we can see for deserializing such objects is on a per-field basis (see Listing 7).

Phew. This kind of code should be able to handle pretty much any kind of inheritance – and at extremely high serialization speeds too. Still, a virtual call to **serializationID()** is a slowdown (however minor it is), and apparently it can be avoided.

C++ deserialization – Take 2.1, deducing object type from VMT pointers

Strictly speaking, when we're writing the whole object it already contains everything we need to deserialize. In particular, it already contains a Virtual Method Table (VMT) pointer which is equivalent to **serializationID()**; in other words, it is not really necessary to invoke a rather expensive virtual **serializationID()** on serialization. The only problem is how to deduce object type from the VMT pointers (and that's without making too many assumptions about object layout, which is very platform- and compiler-dependent).

1. Yes, there can be more than one virtual pointer per object – at least, in the case of virtual base classes.

```
class Y { //polymorphic base
  public:
  int yy;
  std::string zz;
  std::string zz2;

  void
  polymorphicSerialize( OutMemStream* dst ) const;
  void serialize( OutMemStream* dst ) const {
    dst->write( this, sizeof(Y) );
  serializeAsBase( dst );
  }
  void serializeAsBase( OutMemStream* dst ) const
  {
  // non-POD ONLY for serializeAsBase()
    dst->writeString( zz );
    dst->writeString( zz2 );
  }
  explicit Y( InMemStream* src );
  explicit Y( const Y* that );
    // constructor from struct serialized by
    // child class
  void deserializeAsBase( InMemStream* src );
  static std::unique_ptr<Y>
    polymorphicCreateNew( InMemStream* src );

  virtual size_t serializationID() const
  { return 0; }
  virtual ~Y() {}
};
class YY : public Y {
  public:
  int yy2;

  void serialize( OutMemStream* dst ) const {
    dst->write( this, sizeof(YY) );
    Y::serializeAsBase(dst);
  }
  explicit YY( InMemStream* src );

  virtual size_t serializationID() { return 1; }
};

void Y::polymorphicSerialize( OutMemStream* dst )
{
  size_t id = serializationID();
  dst->write( &id, sizeof(size_t) );
  serialize( dst );
}

class X {
  int xx;
  std::unique_ptr<Y> y;
  std::vector<Y> vy;

  void serialize( OutMemStream* dst ) const;
  X( InMemStream* src );
    //deserializing constructor
};

void X::serialize( OutMemStream* dst ) const {
  dst->write( this, sizeof(X) );
  y->polymorphicSerialize( dst );
  dst->writeVector( vy );
  // we still do NOT serialize non-POD objects
  // explicitly in Take 2, we will deserialize
  // them explicitly though
}
```

Listing 6

```
class InMemStream {
  uint8_t* pp;
  uint8_t* ppEnd;

  public:
  inline void read( void* p, size_t sz );
    //same as before
  inline void* readInPlace( size_t sz ) {
    assert( pp + sz < ppEnd );
    void* ret = pp;
    pp += sz;
    return ret;
  }
  inline void* fetchInPlace( size_t sz ) const {
    assert( pp + sz < ppEnd );
    return pp;
  }
  inline std::string readString() {
    size_t l;
    read( &l, sizeof(size_t) );
    assert( pp+l < ppEnd );
    pp += l;
    return std::string(
      reinterpret_cast<const char*>(pp - l), l );
  }
  template< class T >
  inline void readVector( std::vector<T>& v ) {
    size_t sz;
    read( &sz, sizeof(size_t) );
    v.clear();//just in case
    for( size_t i=0; i < sz ; ++i ) {
      v.push_back( T( this ) );
    }
  }
};

Y::Y( InMemStream* src ) {
  Y* that = reinterpret_cast<Y*>(
            src->readInPlace(sizeof(Y)) );
  yy = that->yy;

  deserialiseAsBase( src );
}

std::unique_ptr<Y> Y::polymorphicCreateNew(
InMemStream* src ) const {
  size_t id;
  src->read( &id, sizeof(size_t) );
  switch( id ) {
    case 0:
      return std::unique_ptr<Y>( new Y(src) );
    case 1:
      return std::unique_ptr<Y>( new YY(src) );
    default:
      assert( false );
  }
}
Y::Y( Y* that ) {
  // NB: on non-x86/x64 CPUs, there may be a need
  // to memcpy 'that' into a temporary aligned
  // variable, along the lines of:
  //    alignas(Y) uint8_t tmp[sizeof(Y)];
  //    memcpy(tmp,that,sizeof(Y));
  //    and then use 'tmp' instead of 'that'
  //    this applies to ALL the cases where
  //    readInPlace()/fetchInPlace() are involved
  yy = that->yy;
}
```

Listing 7

```
void Y::deserializeAsBase( InMemStream* src ) {
  //non-POD ONLY for deserializeAsBase()
  zz = src->readString();
  zz2 = src->readString();
}
YY::YY( InMemStream* src ) : Y(
reinterpret_cast<YY*>(
             src->fetchInPlace(sizeof(YY)) )) {
  YY* that = reinterpret_cast<YY*>(
             src->readInPlace(sizeof(YY)) );
yy2 = that->yy2;
  Y::deserializeAsBase( src );
}
X::X( InMemStream* src ) {
  X* that = reinterpret_cast<X*>(
            src->readInPlace(sizeof(X)) );
  xx = that->xx;
  y = Y::polymorphicCreateNew( src );
  src->readVector( vy );
}
```

Listing 7 (cont'd)

One thing which seems to work (still to be double-checked) to deduce object type from VMT pointers is as follows:

- At some point (say, when our program starts), we're creating an instance of each polymorphic objects we're interested in
- To avoid dealing with garbage, we're creating these instances over zeroed memory (for example, using placement new over pre-zeroed buffer)
- When creating a child object, we're initializing the parent object within the child, in exactly the same manner as we're initializing standalone parent object
- We `memcpy()` each of such objects, creating an 'object dump' of each of the polymorphic objects
- Then, as soon as we have a child object and a parent object and their respective dumps, we can:
  - Cast child pointer to parent pointer to determine offset at which parent sits within the child
  - Now we can compare byte-by-byte dumps of the parent-within-child (using the offset mentioned above) and standalone-parent.

    Normally, the only different bytes within the parent-within-child, and standalone-parent (given that they were created as described above), are VMT pointers; moreover, the dumps should differ in at least one byte. Therefore, we can distinguish between a polymorphic child and polymorphic parent (by one of them having certain byte(s) at certain offset(s) as certain pre-defined value(s)).

    Bingo! These bytes are equivalent to the `serializationID()`.

Therefore, we can avoid writing the `serializationID()` during serialization, saving a few more CPU cycles (and bringing performance back to the C structure performance level) – all of that without any priorknowledge about class layout(s).

It should be noted that we didn't try this approach ourselves, but it still looks perfectly plausible ;-).

Other stuff

Of course, this is not really an exhaustive list of problems you can encounter during ultra-high-speed serialization. However, most of the other problems you'll run into are typical for any kind of C++ serialization. In particular, non-owning pointers (and abstract graphs) need to be handled in pretty much the same manner as for any other C++ serialization (see, for example, [ISOCPP] for a relevant discussion).

Performance

From what we've seen, this kind of Ultra-Fast Serialization is extremely fast; it is pretty much on par with C raw-structure-dump serialization, and is around 5–10 times faster than FlatBuffers (this is also consistent with the numbers provided by FlatBuffers themselves here: [FlatBuffers]). Even when comparing with home-grown code with per-field serialization, our Ultra-Fast Serialization still wins (up to 1.5x-2x) due to `memcpy()` over the whole struct having significant advantage over per-field copying.

However, comparing the performance of our Ultra-Fast Deserialization with FlatBuffers is neither very interesting nor really relevant. It is not very interesting because, for the use cases described above, we'll be doing serialization orders of magnitude more frequently than deserialization (as deserialization occurs only when something goes wrong). It is not really relevant, because (unlike FlatBuffers) we need to restore a data structure to exactly the same as its original state (which is usually built in the manner described above, and is not easily flattenable); as a result, we're bound to make all those expensive allocations (and they will eat most of the CPU clocks on deserialization).

On code generation

It is always a good idea to move all this mundane serialization code into some kind of code generator; as described in [NoBugs16a], writing a code generator which will generate code along the lines above is not rocket science.

Still, even in a manually-written form, this technique is actually usable in practice (that is, unless your data structures are very elaborate).

Limitations

One all-important caveat of our Ultra-Fast Serialization technique is the following:

> *DON'T even think of using it unless you can GUARANTEE that the serialized data will be deserialized by EXACTLY the same executable as the one which serialized it.*

This explicitly prohibits ALL of the following:

- Deserializing using the same code compiled by a different compiler/for different platform
- Deserializing using the same library within different executables (well, this MIGHT fly, but we'd rather not risk it). Exactly the same .so/.dll library is ok, however.
- Deserializing by different version of the same executable/shared library

If you do one of these things, most likely, Ultra-Fast Serialization will work for some time – but using it under these conditions is akin to sitting on a powder keg with a fuse already lit. Still, if you know for 100% sure that all the serialization/deserialization will happen in EXACTLY the same executable, it will be very difficult to beat this serialization technique performance-wise.

If the 'same executable' prerequisite doesn't apply to your case, use FlatBuffers (or any of their competitors) instead. As usual, there is no such thing as 'The Best Technique for Everything in Existence', so you DO need different tools for different types of job. And 'serialization for transfer over the network for client code compliant with the protocol' and 'serialization for exactly the same executable' are two rather different beasts. ■

References

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

[NoBugs15] Modular Architecture: Client-Side. On Debugging Distributed Systems, Deterministic Logic, and Finite State Machines, 'No Bugs' Hare, http://ithare.com/chapter-vc-modular-architecture-client-side-on-debugging-distributed-systems-deterministic-logic-and-finite-state-machines/

[NoBugs16] Deterministic Components for Distributed Systems, 'No Bugs' Hare, *Overload* #133

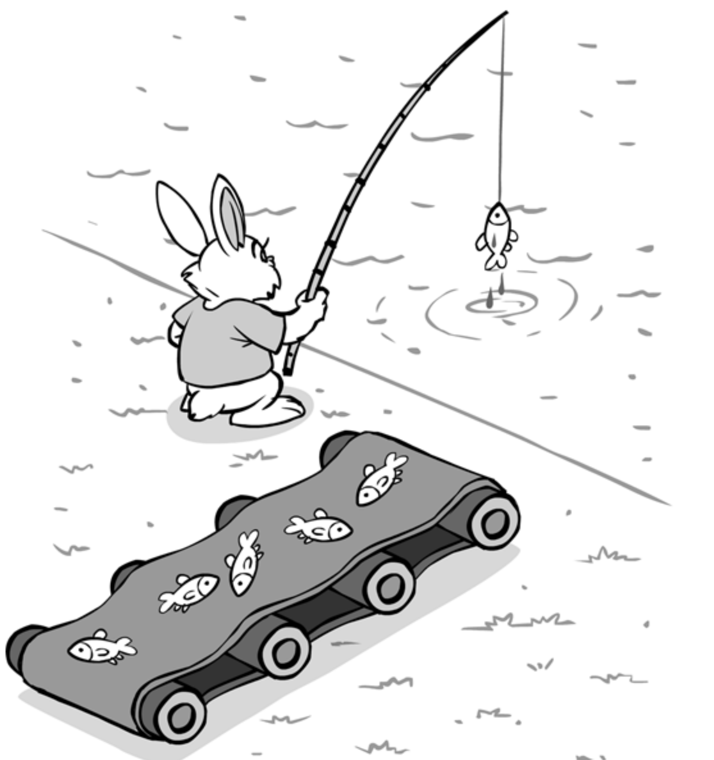
[ISOCPP] Serialization and Unserialization, https://isocpp.org/wiki/faq/serialization

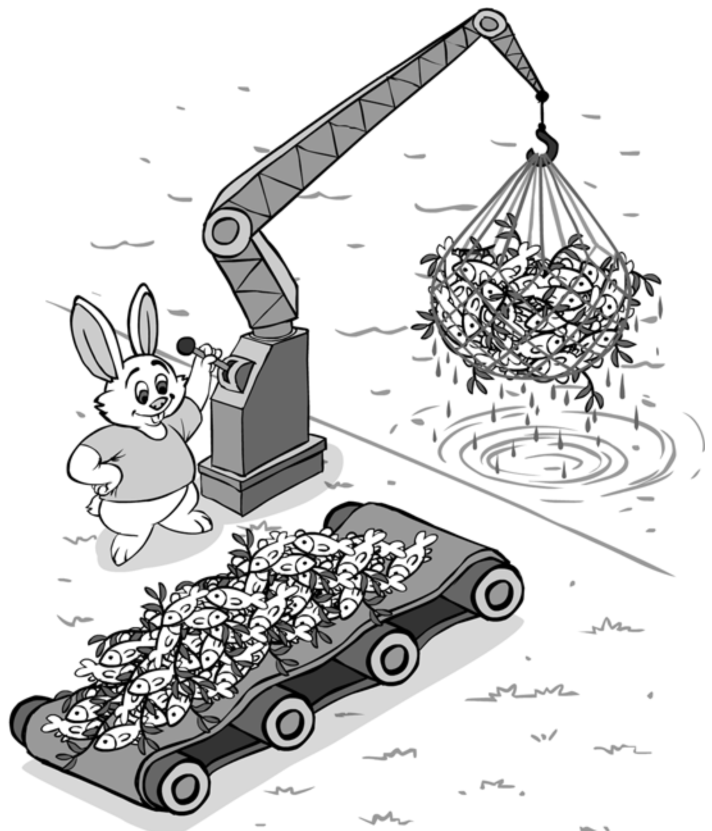
[FlatBuffers] Flatbuffers Benchmarks, https://google.github.io/flatbuffers/flatbuffers_benchmarks.html

[NoBugs16a] IDL: Encodings, Mappings, and Backward Compatibility, 'No Bugs' Hare, http://ithare.com/idl-encodings-mappings-and-backward-compatibility/

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.





Modern C++ Features: User-Defined Literals

User-defined literals were introduced in C++11. Arne Mertz walks us through their use.

User-defined literals are a convenient feature added in C++11.

C++ always had a number of built-in ways to write literals: pieces of source code that have a specific type and value. They are part of the basic building blocks of the language:

```
32 043 0x34 // integer literals, type int
4.27 5E1    // floating point literals,
            // type double
'f', '\n'   // character literals, type char
"foo"       // string literal, type const char[4]
true, false // boolean literals, type bool
```

These are only the most common ones. There are many more, including some newcomers in the newer standards. Other literals are **nullptr** and different kinds of prefixes for character and string literals. There also are suffixes we can use to change the type of a built-in numeric literal:

```
32u         // unsigned int
043l        // long
0x34ull     // unsigned long long
4.27f       // float
5E1l        // long double
```

Suffixes for user-defined literals

With C++11, we got the option of defining our own suffixes. They can be applied to integer, floating point, character and string literals of any flavor. The suffixes must be valid identifiers and start with an underscore – those without an underscore are reserved for future standards.

Using the literals

User-defined literals are basically normal function calls with a fancy syntax. I'll show you in a second how those functions are defined. First, let's see some examples of how they are used:

- user-defined integer literal with suffix **_km**

  45_km
- user-defined floating point literal with suffix **_mi**

  17.8e2_mi
- user-defined character literal with suffix **_c**

  'g'_c
- user-defined character literal (**char32_t**) with suffix **_c**

  U'%'_c
- user-defined string literal with suffix **_score**

  "under"_score
- user-defined string literal (raw, UTF8) with suffix **_stuff**

  u8R"##("(weird)")##"_stuff

Defining literal operators

The functions are called literal operators. Given an appropriate class for lengths, the definition of literal operators that match the first two examples above could look like this:

```
Length operator "" _km(unsigned long long n) {
  return Length{n, Length::KILOMETERS};
}

Length operator ""_mi(long double d) {
  return Length{d, Length::MILES};
}
```

More generally, the syntax for the function header is **<ReturnType> operator "" <Suffix> (<Parameters>)**. The return type can be anything, including **void**. As you see, there can be whitespace between the **""** and the suffix – unless the suffix standing alone would be a reserved identifier or keyword. That means, if we want our suffix to start with a capital letter after the underscore, e.g. **_KM**, there may be no white space. (Identifiers with underscores followed by capitals are reserved for the standard implementation.)

The allowed parameter lists are constrained: for a user-defined integral or floating point literal, you can already see an example above. The compiler first looks for an operator that takes an **unsigned long long** or **long double**, respectively. If such an operator can not be found, there has to be *either* one taking a **char const*** *or* a **template<char...>** operator taking no parameters.

In the case of the so-called raw literal operator taking a **const char**, the character sequence constituting the integral or floating point literal is passed as the parameter. In the case of the template, it is passed as the list of template arguments. E.g. for the **_mi** example above this would instantiate and call:

```
operator ""_mi<'1', '7', '.', '8', 'e', '2'>()
```

Use cases

The example with the units above is a pretty common one. You will have noted that both operators return a **Length**. The class would have an internal conversion for the different units, so with these user defined literals it would be easy to mix the units without crashing your spaceship [Wikipedia]:

```
auto length = 32_mi + 45.4_km;
std::cout << "It's " << length.miles()
          << " miles\n";          //60.21
std::cout << "or " << length.kilometers()
          << " kilometers.\n"; //96.899
```

The standard library also contains a bunch of these (and yes, they still are called 'user-defined' in standard speak). They are not directly in namespace **std** but in subnamespaces of **std::literals**:

Arne Mertz is a self-taught C++ and clean code enthusiast who has been using C++ for over a decade. He continues to deepen his understanding by writing and speaking about C++, e.g. in his weekly "Simplify C++!" blog. You can contact Arne via arne.mertz@zuehlke.com or @arne_mertz.

In theory, we could write literal operators that have side effects and do anything we want, like a normal function

- From `std::literals::complex_literals`, the suffixes `i`, `if` and `il` are for the imaginary part of `std::complex` numbers. So, `3.5if` is the same as `std::complex<float>{0, 3.5f}`
- From `std::literals::chrono_literals`, the suffixes `h`, `min`, `s`, `ms`, `us` and `ns` create durations in `std::chrono` for hours, minutes, seconds, milli-, micro- and nanoseconds, respectively.
- In `std::literals::string_literals`, we have the suffix `s` to finally create a `std::string` right from a string literal instead of tossing around `char const*`.

A word of caution

While user defined literals look very neat, they are not much more than syntactic sugar. There is not much difference between defining and calling a literal operator with `"foo"_bar` and doing the same with an ordinary function as `bar("foo")`. In theory, we could write literal operators that have side effects and do anything we want, like a normal function.

However, that is not what people would expect from something that does not look like 'it does something'. Therefore it is best to use user defined literals only as obvious shorthand for the construction of values.

Playing with other modern C++ features

A while ago I came across a case where I had to loop over a fixed list of `std::string`s defined at compile time. In the old days before C++11, the code would have looked like this:

```
static std::string const strings[] =
  {"foo", "bar", "baz"};
for (std::string const* pstr = strings;
  pstr != strings+3; ++pstr) {
    process(*pstr);
}
```

This is horrible. Dereferencing the pointer and the hard-coded `3` in the loop condition just don't seem right. I could have used an `std::vector<std::string>` here, but that would mean a separate function to prefill and initialize the `const` vector since there were no lambdas.

Today we have range based `for`, `initializer_list`, `auto` and user-defined literals for strings:

```
using namespace std::literals::string_literals;
//...
for (auto const& str : {"foo"s, "bar"s, "baz"s})
{
  process(str);
}
```

And the code looks just as simple as it should. ■

References

[Wikipedia] The Mars Climate Orbiter: Cause of Failure
https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure

Python Streams vs Unix Pipes

Dealing with an infinite sequence requires some thought. Thomas Guest presents various ways to approach such a problem.

I chanced upon an interesting puzzle:

> Find the smallest number that can be expressed as the sum of 5, 17, 563 and 641 consecutive prime numbers, and is itself a prime number.

Here, the prime numbers are an infinite steam:

2, 3, 5, 7, 11, 13 ...

and sums of *N* consecutive primes are similarly infinite. For example, the sum of 2 consecutive primes would be the stream:

2+3, 3+5, 5+7, 7+11, 11+13 ...

which is:

5, 8, 12, 18, 24 ...

and the sum of 3 consecutive primes is:

10 (=2+3+5), 15, 23, 31 ...

Had we been asked to find the smallest number which can be expressed as the sum of 3 consecutive primes *and* as the sum of 5 consecutive primes *and* is itself prime, the answer would be 83.

```
>>> 23 + 29 + 31
83
>>> 11 + 13 + 17 + 19 + 23
83
```

Infinite series and Python

My first thought was to tackle this puzzle using Python iterators and generators. Here's the outline of a strategy:

- starting with a stream of primes
- tee the stream to create 4 additional copies
- transform these copies into the consecutive sums of 5, 17, 563 and 641 primes
- merge these consecutive sums back with the original primes stream
- group the elements of this merged stream by value
- the first group which contains 5 elements must have occurred in every source, and is therefore a prime and representable as the consecutive sum of 5, 17, 563 and 641 primes
- which solves the puzzle!

Note that when we copy an infinite stream we cannot consume it first. We will have to be lazy or we'll get exhausted.

Courtesy of the *Python Cookbook*, I already had a couple of useful recipes to help implement this strategy (see Listing 1).

Both these functions merit a closer look for the cunning use they make of standard containers, but we'll defer this inspection until later. In passing, note that **`stream_merge()`**'s docstring suggests we might try using it as basis for **`primes()`**:

1. form the series of composite (non-prime) numbers by merging the streams formed by multiples of prime numbers;
2. the primes remain when you remove these composites from the series of natural numbers.

```
def primes():
  '''Generate the sequence of prime numbers: 2,
     3, 5 ... '''
  ....
def stream_merge(*ss):
  '''Merge a collection of sorted streams.
  Example: merge multiples of 2, 3, 5
  >>> from itertools import count, islice
  >>> def multiples(x):
    return (x * n for n in count(1))
  >>> s = stream_merge(multiples(2),
    multiples(3), multiples(5))
  >>> list(islice(s, 10))
  [2, 3, 4, 5, 6, 6, 8, 9, 10, 10]
  '''
  ....
```

Listing 1

This scheme is hardly original – it's a variant of Eratosthenes' sieve – but if you look carefully you'll notice the self-reference. Unfortunately recursive definitions of infinite series don't work well with Python[1], hence **`primes()`** requires a little more finesse. We'll take a look at it later.

Moving on, to solve the original puzzle we need a consecutive sum filter. Listing 2 will transform a stream of numbers into a stream of consecutive sums of these numbers.

Here we can think of the summed elements as lying within a sliding window: each time we slide the window an element gets added to the top and an element gets removed from the bottom, and we adjust **`csum`** accordingly.

So, now we have:

- the series of prime numbers, **`primes()`**
- a **`stream_merge()`** connector
- a **`consecutive_sum()`** filter

The remaining stream adaptors come from the standard itertools module. Note that the **`stream_merge()`** works here since all the consecutive sum series are strictly increasing. Note also that the stream of prime numbers can be treated as **`consecutive_sum(s=primes(), n=1)`**, handling the 'and is itself a prime number' requirement. (See Listing 3.)

1. CPython, more precisely — I don't think anything in the Python language itself prohibits tail recursion. Even using CPython, yet another recipe from the online Python Cookbook explores the idea of an @tail_recursion decorator

Thomas Guest is an experienced and enthusiastic software developer who likes puzzles, programming, running and noodles. His website is http://wordaligned.org. He can be contacted at tag@wordaligned.org

Haskell makes no compromises when it comes to functional programming. Its lazy evaluation and inductive recursion make it a perfect fit for this kind of puzzle

```
def consecutive_sum(s, n):
    '''Generate the series of sums of n
       consecutive elements of s
    Example: 0, 1, 2, 3, 4 ... => 0+1, 1+2,
       2+3, 3+4, ...
    >>> from itertools import count, islice
    >>> list(islice(consecutive_sum(count(), 2),
       10))
    [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
    '''
    lo, hi = itertools.tee(s)
    csum = sum(next(hi) for _ in range(n))
    while True:
        yield csum
        csum += next(hi) - next(lo)
```

Listing 2

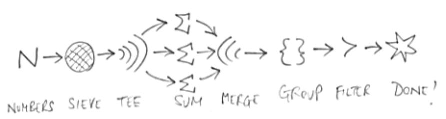


Figure 1

Here, **solns** is yet another stream, the result of merging the *N* input consecutive sum streams then filtering out the numbers which appear *N* times; that is, the numbers which can be expressed as sums of 1, 5, 17, 563 and 641 consecutive primes.

The first such number solves the original puzzle.

```
>>> next(solns)
7002221
```

Figure 1 is a picture of how these stream tools link up to solve this particular puzzle. The great thing is that we can reconnect these same tools to solve a wide range of puzzles, and indeed more practical processing tasks. To use the common analogy, we direct data streams along pipes.

```
>>> lens = 1, 5, 17, 563, 641
>>> N = len(lens)
>>> from itertools import tee, groupby
>>> ps = tee(primes(), N)
>>> csums = [consecutive_sum(p, n) for p, n in
zip(ps, lens)]
>>> solns = (n for n, g in
groupby(stream_merge(*csums))
            if len(list(g)) == N)
```

Listing 3

Infinite series in other languages

Python is the language I find most convenient most of the time, which explains why I reached for it first. It's an increasingly popular language, which helps explain why I didn't need to write the tricky parts of my solution from scratch: they'd already been done. Python is also a language which makes compromises. Having used Python to find a solution to the puzzle I wondered if there wasn't some other language better suited to this kind of problem.

Haskell makes no compromises when it comes to functional programming. Its lazy evaluation and inductive recursion make it a perfect fit for this kind of puzzle – but my approach of teeing, filtering and merging made me consider the Unix Shell. Now, I use Bash every day and page through its manual at least once a week. Scripting appeals and I'm comfortable at the command line. How hard could it be to solve this puzzle using Bash? After all, I already knew the answer!

Partial sums

Here's a simple shell function to generate partial sums. I've used awk, a little language I gave up on a long time ago in favour of more rounded scripting languages like Perl and then Python. Now I look at it again, it seems to fill a useful gap. Awk processes a file sequentially, applying pattern-action rules to each line, a processing template which I've reinvented less cleanly many times. Despite my rediscovery of awk, I'll be keeping its use strongly in check in what follows.

```
$ psum() { awk '{ print s += $1 }'; }
```

Much like Perl, awk guesses what you want to do. Here, it conjures the summation variable, **s**, into existence, assigning it a default initial value of 0. (Good guess!) Since we're doing arithmetic awk converts the first field of each input line into a number. We can test **psum** by using **jot** to generate the sequence 1, 2, 3, 4, 5 (this is on a Mac – on a Linux platform use **seq** instead of **jot**).

```
$ jot 5 | psum
1
3
6
10
15
```

Consecutive sums

You may be wondering why we've bothered creating this partial sum filter since it's the sums of consecutive elements we're after, rather than the sum of the series so far. Well, notice that if **P[i]** and **P[i+n]** are two elements from the series of partial sums of S, then their difference, **P[i+n] - P[i]**, is the sum of the **n** consecutive elements from S.

So to form an n-element consecutive sum series we can **tee** the partial sums streams, advance one of these by n, then zip through them in parallel finding their differences. An example makes things clear – see Listing 4.

Here, **jot 5** generates the sequence 1, 2, 3, 4, 5, which **psum** progressively accumulates to 1, 3, 6, 10, 15. We then **tee** this partial sum

```
$ mkfifo pipe
$ jot 5 | psum | tee pipe | tail -n +2 | paste - pipe
3       1
6       3
10      6
15      10
        15
```

Listing 4

```
$ jot 5 | psum | tee pipe | tail -n +2 | paste - pipe | awk '{print $1 - $2}'
  2
  3
  4
  5
  15
```

Listing 5

```
$ let "N=1<<32" && echo $N | tee >(awk '{print $1 * $1}')
  4294967296
  18446744073709551616
```

Listing 6

```
$ psum()  { awk 'BEGIN { print 0 }{print s += $1 }'; }
$ delay() { let "n = $1 + 1" && tail +$n; }
$ sdiff() { mkfifo p.$1 && tee p.$1 | delay $1 | paste - p.$1 | \
            awk 'NF == 2 {print $1 - $2 }'; }
```

Listing 7

```
$ psum()  { awk 'BEGIN { print 0 }{print s += $1 }'; }
$ delay() { let "n = $1 + 1" && stdbuf -o 0 tail +$n; }
$ sdiff() { mkfifo p.$1 && tee p.$1 | delay $1 | \
            stdbuf -o 0 paste - p.$1 | \
            awk 'NF == 2 {print $1 - $2 }'; }
```

Listing 8

series through two pipes: the first, **pipe**, is an explicitly created named pipe created by **mkfifo**, the second is implicitly created by the pipeline operator, **|**. The remainder of the command line delays one series by one (note that **tail** numbers lines from 1, not 0, so tail -n +1 is the identity filter) then pastes the two series back together.[2]

By appending a single **awk** action to the pipeline we get a consecutive sum series (see Listing 5).

The output 2, 3, 4, 5 is the series of consecutive sums of length 1 taken from the original series 1, 2, 3, 4, 5. The trailing 15 and the 1 missed from the start are edge case problems, and easily corrected.

Accumulating an increasing series of numbers in order to find the differences between elements lying a given distance apart on this series isn't a very smart idea on a computer with a fixed word-size, but it's good to know (e.g.) that awk doesn't stop counting at 32 bits. (See Listing 6.)

Exactly if and when awk stops counting, I'm not sure. The documentation doesn't say and I haven't looked at the source code.

Bug fixes

Let's capture these tiny functions and name them. In Listing 7, then, are revised **psum()** and **sdiff()** filters. The edge case problems should now be fixed.

A quick test:

```
$ jot 5 | psum | sdiff 3
6
9
12
```

The output is, as expected, the series of sums of consecutive triples taken from 1, 2, 3, 4, 5 (6=1+2+3, 9=2+3+4, 12=3+4+5).

There's a pernicious bug, though. These functions can't handle infinite series so they are of limited use as pipeline tools. For example, if we stream in the series 0, 1, 2, … (generated here as the partial sums of the series 1, 1, 1, …) nothing gets output and we have to interrupt the process.

```
# This command appears
# to hang
$ yes 1 | psum | sdiff 1
^C
```

To work around this is, we can use Gnu **stdbuf** to prohibit **tail** and **paste** from using output buffers (see Listing 8).

Now the data flows again:

```
 # Accumulate the stream 1
1 1 ... and print the
 # difference between
successive elements
 $ yes 1 | psum | sdiff 1
 1
 1
 1
 1
 ^C
```

Merging streams

The Unix shell merges streams rather more succinctly than Python. **Sort -m** does the job directly. Note that a standard **sort** cannot yield any output until all its inputs are exhausted, since the final input item might turn out to be the one which should appear first in the output. Merge sort, **sort -m**, can and does produce output without delay.[3]

```
$ yes | sort
^C
$ yes | sort -m
y
y
y
y
y
^C
```

Generating primes

No doubt it's possible to generate the infinite series of prime numbers using native Bash code, but I chose to reuse the *Python Cookbook* recipe (Listing 9) for the job.

This is a subtle little program which makes clever use of Python's native hashed array container, the dictionary. In this case dictionary values are the primes less than n and the keys are composite multiples of these primes. The loop invariant, roughly speaking, is that the dictionary values are the primes less than n, and the corresponding keys are the lowest

2. Tail is more commonly used to yield a fixed number of lines from the end of the file: by prefixing the line count argument with a + sign, it skips lines from the head of the file. The GNU version of head can similarly be used with a - prefix to skip lines at the tail of a file. The notation is {compact,powerful,subtle,implementation dependent}.

3. Sort -m is a sort which doesn't really sort — its inputs should already be sorted — rather like the +n option turning tail on its head.

```
primes
#!/usr/bin/env python
import itertools

def primes():
  '''Generate the prime number series: 2, 3, 5
  ... '''
  D = {}
  for n in itertools.count(2):
    p = D.pop(n, None)
    if p is None:
      yield n
      D[n * n] = n
    else:
      x = n + p
      while x in D:
        x += p
      D[x] = p

for p in primes():
  print(p)
```

Listing 9

multiples of these primes greater than or equal to n. It's a lazy, recursion-free take of Eratosthenes' sieve.

For the purposes of this article the important things about this program are:

- it generates an infinite series of numbers to standard output[4], making it a good source for a shell pipeline
- by making it executable and adding the usual shebang incantation, we can invoke this Python program seamlessly from the shell.

Pipe connection

Recall the original puzzle:

> Find the smallest number that can be expressed as the sum of 5, 17, 563 and 641 consecutive prime numbers, and is itself a prime number.

First, let's check the connections by solving a simpler problem which we can manually verify: to find prime numbers which are also the sum of 2 consecutive primes. As we noted before, this is the same as finding primes numbers which are the consecutive sums of 1 and 2 primes.

In one shell window we create a couple of named pipes, **`c.1`** and **`c.2`**, which we'll use to stream the consecutive sum series of 1 and 2 primes respectively. The results series comprises the duplicates when we merge these pipes.

```
$ mkfifo c.{1,2}
$ sort -mn c.{1,2} | uniq -d
```

In another shell window, stream data into **`c.1`** and **`c.2`**:

```
$ for i in 1 2; do (primes | psum | sdiff $i >
c.$i) & done
```

In the first window we see the single number 5, which is the first and only prime number equal to the sum of two consecutive primes.

Prime numbers equal to the sum of three consecutive primes are more interesting. In each shell window recall the previous commands and switch the 2s to 3s (a simple command history recall and edit, ^2^3^, does the trick). The merged output now looks like this:

```
$ sort -mn c.1 c.3 | uniq -d
23
31
41
...
```

4. The series is infinite in theory only: at time n the number of items in the **`has_prime_factors`** dictionary equals the number of primes less than n, and each key in this dictionary is larger than n. So resource use increases steadily as n increases.

I used a MacBook laptop used to run these scripts.

Model Name:	MacBook
Model Identifier:	MacBook1,1
Processor Name:	Intel Core Duo
Processor Speed:	2 GHz
Number Of Processors:	1
Total Number Of Cores:	2
L2 Cache (per processor):	2 MB
Memory:	2 GB
Bus Speed:	667 MHz

We can check the first few values:

```
23 = 5 + 7 + 11
31 = 7 + 11 + 13
41 = 11 + 13 + 17
```

At this point we're confident enough to give the actual puzzle a try. Start up the solutions stream.

```
$ mkfifo c.{1,5,17,563,641}
$ sort -mn c.{1,5,17,563,641} | uniq -c | grep "5 "
```

Here, we use a standard shell script set intersection recipe: **`uniq -c`** groups and counts repeated elements, and the **`grep`** pattern matches those numbers common to all five input streams.

Now we can kick off the processes which will feed into the consecutive sum streams, which sort is waiting on.

```
$ for i in 1 5 17 563 641; do (primes | psum |
sdiff $i > c.$i) & done
```

Sure enough, after about 15 seconds elapsed time, out pops the result:

```
$ sort -mn c.{1,5,17,563,641} | uniq -c | grep "5 "
  5 7002221
```

15 seconds seems an eternity for arithmetic on a modern computer (you could start up a word processor in less time!), and you might be inclined to blame the overhead of all those processes, files, large numbers, etc. In fact it took around 6 seconds for the Python program simply to generate prime numbers up to 7002221, and my pure Python solution ran in 9 seconds.

Portability

One of the most convenient things about Python is its portability. I don't mean 'portable so long as you conform to the language standard' or 'portable if you stick to a subset of the language'. I mean that a Python program works whatever platform I use without me having to worry about it.

Non-portability put me off the Unix shell when I first encountered it: there seemed too many details, too many platform differences – which shell are you using? which extensions? which implementation of the core utilities, etc, etc? Readily available and well-written documentation didn't help much here: generally I want the shell to just do what I mean, which is why I switched so happily to Perl when I discovered it.

Since then this situation has, for me, improved in many ways. Non-Unix platforms are declining as are the different flavours of Unix. Bash seems to have become the standard shell of choice and Cygwin gets better all the time. GNU coreutils predominate. As a consequence I've forgotten almost all the Perl I ever knew and am actively rediscovering the Unix shell.

Writing this article, though, I was reminded of the platform dependent behaviour which used to discourage me. On a Linux platform close to hand I had to use **`seq`** instead of **`jot`**, and **`awk`** formatted large integers in a scientific form with a loss of precision.

```
$ echo '10000000001 0' | awk '{print $1 - $2}'
1e+10
```

On OS X the same command outputs 10000000001. I couldn't tell you which implementation is more correct. The fix is to explicitly format these numbers as decimal integers, but the danger is that the shell silently

```
from heapq import heapify, heappop, heapreplace

def stream_merge(*ss):
  '''Merge a collection of sorted streams.'''
  pqueue = []
  for i in map(iter, ss):
    try:
      pqueue.append((i.next(), i.next))
    except StopIteration:
      pass
  heapify(pqueue)
  while pqueue:
    val, it = pqueue[0]
    yield val
    try:
      heapreplace(pqueue, (it(), it))
    except StopIteration:
      heappop(pqueue)
```

Listing 10

```
module Main where

import List

isPrime x = all (\i -> 0/=x`mod`i) $ takeWhile (\i
-> i*i <= x) primes

primes = 2:filter (\x -> isPrime x) [3..]

cplist n = map (sum . take n) (tails primes)

meet (x:xs) (y:ys) | x < y = meet xs (y:ys)
                   | y < x = meet (x:xs) ys
                   | x == y =  x:meet xs ys
main = print $ head $ \
(primes `meet` cplist 5) `meet` (cplist 17 `meet`
cplist 563) `meet` cplist 641
```

Listing 11

swallows these discrepancies and you've got a portability problem you don't even notice.

```
$ echo '10000000001 0' | awk '{printf "%d\n",
$1 - $2}'
10000000001
```

Stream merge

I mentioned **`stream_merge()`** at the start of this article, a general purpose function written by Raymond Hettinger which I originally found in the Python Cookbook. As with the prime number generator, you might imagine the merge algorithm to be recursively defined:

1. to merge a pair of streams, take items from the first which are less than the head of the second, then swap;
2. to merge N streams, merge the first stream with the merged (N-1) rest.

Again the Python solution does it differently, this time using a heap as a priority queue of items from the input streams. It's ingenious and efficient. Look how easy it is in Python (Listing 10) to shunt functions and pairs in and out of queues.

A more sophisticated version of this code has made it into the Python standard library, where it goes by the name of **`heapq.merge`** (I wonder why it wasn't filed in **`itertools`**?)

Alternative solutions

As usual Haskell wins the elegance award, so I'll quote in full a solution (Listing 11) built without resorting to cookbookery which produces the (correct!) answer in 20 seconds. ■

Letter

Silas S. Brown comments on Steve Love's recent article.

Hi Steve,

Just read your article 'A Lifetime in Python' in *Overload* 133 and the sentence 'It will (probably) be garbage collected at some indeterminate point in the future'. That's true of the Java implementation of Python, but the C implementation does reference counting, so it is able to delete objects as soon as the last reference to them falls out of scope; the garbage collector is used only as a backup in case of cyclic references. So in this case the db object will be deleted when **`addCustomerOrder`** returns, unless its internal structure contains references back to the parent object, in which case yes it will be garbage collected when the gc next runs (which is usually once every fixed number of bytecode instructions).

But that's only a small observation on an excellent article.

Thanks.

Silas S. Brown

And Steve replies:

Hi Silas,

Thanks for the feedback. I was trying to convey the idea that the object may or may not be garbage collected, but I guess it's not that clear that some platforms don't use a gc routinely.

But whether the object is ref counted, gc'd or anything else, it still won't have **`close`** called on it.

In any case, I think this is a nice clarification on the python lifetime management, and would be happy to see it as a letter, if you're happy with that.

Cheers,

Steve

If you read something that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

Hello World in Go

Go provides a way to write efficient concurrent programs in a C-like language. Eleanor McHugh shares a "Hello, world!" tutorial.

It's a tradition in programming books to start with a canonical 'Hello World' example and whilst I've never felt the usual presentation is particularly enlightening, I know we can spice things up a little to provide useful insights into how we write Go programs.

Let's begin with the simplest Go program that will output text to the console (Listing 1).

The first thing to note is that every Go source file belongs to a package, with the main package defining an executable program whilst all other packages represent libraries.

```
1 package main
```

For the main package to be executable it needs to include a **`main()`** function, which will be called following program initialisation.

```
2 func main() {
```

Notice that unlike C/C++, the **`main()`** function neither takes parameters nor has a return value. Whenever a program should interact with command-line parameters or return a value on termination, these tasks are handled using functions in the standard package library. We'll examine command-line parameters when developing **`Echo`**.

Finally let's look at our payload.

```
3 println("hello world")
```

The **`println()`** function is one of a small set of built-in generic functions defined in the language specification and which in this case is usually used to assist debugging, whilst **`"hello world"`** is a value comprising an immutable string of characters in utf-8 format.

We can now run our program from the command-line (Terminal on MacOS X or Command Prompt on Windows) with the command

```
$ go run 01.go
hello world
```

Packages

Now we're going to apply a technique which I plan to use throughout my book by taking this simple task and developing increasingly complex ways of expressing it in Go. This runs counter to how experienced programmers usually develop code but I feel this makes for a very effective way to introduce features of Go in rapid succession and have used it with some success during presentations and workshops.

There are a number of ways we can artificially complicate our hello world example and by the time we've finished I hope to have demonstrated all the features you can expect to see in the global scope of a Go package. Our first change is to remove the built-in **`println()`** function and replace it with something intended for production code (see Listing 2).

```
1 package main
2 func main() {
3   println("hello world")
4 }
```

Listing 1

```
1 package main
2 import "fmt"
3 func main() {
4   fmt.Println("hello world")
5 }
```

Listing 2

The structure of our program remains essentially the same, but we've introduced two new features.

```
2 import "fmt"
```

The **`import`** statement is a reference to the **`fmt`** package, one of many packages defined in Go's standard runtime library. A **`package`** is a library which provides a group of related functions and data types we can use in our programs. In this case, **`fmt`** provides functions and types associated with formatting text for printing and displaying it on a console or in the command shell.

```
4 fmt.Println("hello world")
```

One of the functions provided by **`fmt`** is **`Println()`**, which takes one or more parameters and prints them to the console with a carriage return appended. Go assumes that any identifier starting with a capital letter is part of the public interface of a package whilst identifiers starting with any other letter or symbol are private to the package.

In production code we might choose to simplify matters a little by importing the **`fmt`** namespace into the namespace of the current source file, which requires we change our **`import`** statement.

```
2 import . "fmt"
```

And this consequently allows the explicit package reference to be removed from the **`Println()`** function call.

```
4 Println("hello world")
```

In this case we notice little gain; however, in later examples we'll use this feature extensively to keep our code legible (Listing 3).

```
1 package main
2 import . "fmt"
3 func main() {
4   Println("hello world")
5 }
```

Listing 3

Eleanor McHugh London-based hacker Ellie has a passion for all things computational and has worked on mission critical systems ranging from cockpit avionics to banking security and digital trust arbitration. She's the co-founder of Innovative Identity Solutions Limited and author of *A Go Developer's Notebook* along with numerous esoteric talks on programming in Ruby and Go. As a responsible parent Ellie enjoys polyhedral dice, home brewing and gothic music.

its syntax allows a function to return more than one value and as such each function takes two sets of (), the first for parameters and the second for results

```
package main
import . "fmt"
const Hello = "hello"
const world = "world"
func main() {
  Println(Hello, world)
}
```

Listing 4

One aspect of imports that we've not yet looked at is Go's built-in support for code hosted on a variety of popular social code-sharing sites such as GitHub and Google Code. Don't worry, we'll get to this in later chapters of my book.

Constants

A significant proportion of Go codebases feature identifiers whose values will not change during the runtime execution of a program and our 'Hello World' example is no different (Listing 4), so we're going to factor these out.

Here we've introduced two constants: **`Hello`** and **`world`**. Each identifier is assigned its value during compilation, and that value cannot be changed at runtime. As the identifier **`Hello`** starts with a capital letter the associated constant is visible to other packages – though this isn't relevant in the context of a **`main`** package – whilst the identifier **`world`** starts with a lowercase letter and is only accessible within the **`main`** package.

We don't need to specify the type of these constants as the Go compiler identifies them both as strings.

Another neat trick in Go's armoury is multiple assignment so let's see how this looks (see Listing 5).

This is compact, but I personally find it too cluttered and prefer the more general form (Listing 6).

Because the **`Println()`** function is *variadic* (i.e. can take a varible number of parameters) we can pass it both constants and it will print them on the same line, separate by whitespace. **`fmt`** also provides the **`Printf()`** function which gives precise control over how its parameters are displayed using a format specifier which will be familiar to seasoned C/C++ programmers.

```
  Printf("%v %v\n", Hello, world)
```

`fmt` defines a number of % replacement terms which can be used to determine how a particular parameter will be displayed. Of these **`%v`** is the most generally used as it allows the formatting to be specified by the type of the parameter. We'll discuss this in depth when we look at user-defined types, but in this case it will simply replace a **`%v`** with the corresponding string.

```
package main
import . "fmt"
const Hello, world = "hello", "world"
func main() {
  Println(Hello, world)
}
```

Listing 5

```
package main
import . "fmt"
const (
  Hello = "hello"
  world =  "world"
)
func main() {
  Println(Hello, world)
}
```

Listing 6

```
package main
import . "fmt"
const (
  Hello = "hello"
  world =  "world"
)
func main() {
  Printf("%v %v\n", Hello, world)
}
```

Listing 7

When parsing strings the Go compiler recognises a number of *escape sequences* which are available to mark tabs, new lines and specific unicode characters. In this case we use **`\n`** to mark a new line (Listing 7).

Variables

Constants are useful for referring to values which shouldn't change at runtime; however, most of the time when we're referencing values in an imperative language like Go we need the freedom to change these values. We associate values which will change with variables. What follows is a simple variation of our Hello World program which allows the value of **`world`** to be changed at runtime by creating a new value and assigning it to the **`world`** variable (Listing 8).

```
package main
import . "fmt"
const Hello = "hello"
var world = "world"
func main() {
  world += "!"
  Println(Hello, world)
}
```

Listing 8

```
package main
import . "fmt"

const Hello = "hello"
var world = "world"

func main() {
  world := world + "!"
  Println(Hello, world)
}
```

Listing 9

There are two important changes here. Firstly we've introduced syntax for declaring a variable and assigning a value to it. Once more Go's ability to infer type allows us assign a **string** value to the variable **world** without explicitly specifying the type.

```
var world = "world"
```

However if we wish to be more explicit we can be.

```
var world string = "world"
```

Having defined **world** as a variable in the global scope we can modify its value in **main()**, and in this case we choose to append an exclamation mark. Strings in Go are immutable values so following the assignment **world** will reference a new value.

```
world += "!"
```

To add some extra interest, I've chosen to use an *augmented assignment* operator. These are a syntactic convenience popular in many languages which allow the value contained in a variable to be modified and the resulting value then assigned to the same variable.

I don't intend to expend much effort discussing scope in Go. The point of my book is to experiment and learn by playing with code, referring to the comprehensive language specification available from Google when you need to know the technicalities of a given point. However, to illustrate the difference between *global* and *local* scope we'll modify this program further (see Listing 9).

Here we've introduced a new *local* variable **world** within **main()** which takes its value from an operation concatenating the value of the *global* **world** variable with an exclamation mark. Within **main()**, any subsequent reference to **world** will always access the *local* version of the variable without affecting the *global* **world** variable. This is known as *shadowing*.

The **:=** operator marks an assignment declaration in which the type of the expression is inferred from the type of the value being assigned. If we chose to declare the local variable separately from the assignment we'd have to give it a different name to avoid a compilation error (Listing 10).

Another thing to note in this example is that when **w** is declared it's also initialised to the zero value, which in the case of **string** happens to be **""**. This is a **string** containing no characters.

In fact, all variables in Go are initialised to the zero value for their type when they're declared and this eliminates an entire category of initialisation bugs which could otherwise be difficult to identify.

```
package main
import . "fmt"
const Hello = "hello"
var world = "world"
func main() {
  var w string
  w = world + "!"
  Println(Hello, w)
}
```

Listing 10

```
package main
import . "fmt"
const Hello = "hello"
func main() {
  Println(Hello, world())
}
func world() string {
  return "world"
}
```

Listing 11

Functions

Having looked at how to reference values in Go and how to use the **Println()** function to display them, it's only natural to wonder how we can implement our own functions. Obviously we've already implemented **main()** which hints at what's involved, but **main()** is something of a special case as it exist to allow a Go program to execute and it neither requires any parameters nor produces any values to be used elsewhere in the program. (See Listing 11.)

In this example we've introduced **world()**, a function which to the outside world has the same operational purpose as the variable of the same name that we used in the previous section.

The empty brackets **()** indicate that there are no parameters passed into the function when it's called, whilst **string** tells us that a single value is returned and it's of type **string**. Anywhere that a valid Go program would expect a **string** value we can instead place a call to **world()** and the value returned will satisfy the compiler. The use of **return** is required by the language specification whenever a function specifies return values, and in this case it tells the compiler that the value of **world()** is the string **"world"**.

Go is unusual in that its syntax allows a function to return more than one value and as such each function takes two sets of **()**, the first for parameters and the second for results. We could therefore write our function in long form as

```
func world() (string) {
  return "world"
}
```

In this next example we use a somewhat richer function signature, passing the parameter **name** which is a string value into the function **message()**, and assigning the function's return value to **message** which is a variable declared and available throughout the function. (See Listing 12.)

As with **world()**, the **message()** function can be used anywhere that the Go compiler expects to find a string value. However, where **world()** simply returned a predetermined value, **message()** performs a calculation using the **Sprintf()** function and returns its result.

Sprintf() is similar to **Printf()** which we met when discussing constants, only rather than create a string according to a format and displaying it in the terminal it instead returns this string as a value which we can assign to a variable or use as a parameter in another function call such as **Println()**.

```
package main
import "fmt"
func main() {
  fmt.Println(message("world"))
}
func message(name string) (message string) {
  message = fmt.Sprintf("hello %v", name)
  return message
}
```

Listing 12

```
package main
import . "fmt"
func main() {
  Println(message("world"))
}
func message(name string) (message string) {
  message = Sprintf("hello %v", name)
  return
}
```

Listing 13

```
package main
import . "fmt"
func main() {
  greet("world")
}
func greet(name string) {
  Println("hello", name)
}
```

Listing 14

Because we've explicitly named the return value, we don't need to reference it in the return statement as each of the named return values is implied. (See Listing 13.)

If we compare the **`main()`** and **`message()`** functions (Listing 14), we notice that **`main()`** doesn't have a **`return`** statement. Likewise if we define our own functions without return values we can omit the **`return`** statement, though later we'll meet examples where we'd still use a **`return`** statement to prematurely exit a function.

In Listing 15, we'll see what a function which uses multiple return values looks like.

Because **`message()`** returns two values we can use it in any context where at least two parameters can be consumed. **`Println()`** happens to be a *variadic* function, which we'll explain in a moment, and takes zero or more parameters so it happily consumes both of the values **`message()`** returns.

For our final example (Listing 16) we're going to implement our own *variadic* function.

We have three interesting things going on here which need explaining. Firstly I've introduced a new type, **`interface{}`**, which acts as a proxy for any other type in a Go program. We'll discuss the details of this shortly but for now it's enough to know that anywhere an **`interface{}`** is accepted we can provide a **`string`**.

```
package main
import . "fmt"
func main() {
  Println(message())
}
func message() (string, string) {
  return "hello", "world"
}
```

Listing 15

```
package main
import . "fmt"
func main() {
  print("Hello", "world")
}
func print(v ...interface{}) {
  Println(v...)
}
```

Listing 16

In the function signature we use **`v ...interface{}`** to declare a parameter **`v`** which takes any number of values. These are received by **`print()`** as a sequence of values and the subsequent call to **`Println(v...)`** uses this same sequence as this is the sequence expected by **`Println()`**.

So why did we use **`...interface{}`** in defining our parameters instead of the more obvious **`...string`**? The **`Println()`** function is itself defined as **`Println(...interface{})`** so to provide a sequence of values en masse we likewise need to use **`...interface{}`** in the type signature of our function. Otherwise we'd have to create a **`[]interface{}`** (a **`slice`** of **`interface{}`** values, a concept we'll cover in detail in a later chapter of my book) and copy each individual element into it before passing it into **`Println()`**.

Encapsulation

In this tutorial, we'll for the most part be using Go's primitive types and types defined in various standard packages without any comment on their structure; however, a key aspect of modern programming languages is the encapsulation of related data into structured types and Go supports this via the **`struct`** type. A **`struct`** describes an area of allocated memory which is subdivided into slots for holding named values, where each named value has its own type. A typical example of a **`struct`** in action would be Listing 17, which gives:

```
$ go run 17.go
Hello world
```

Here we've defined a struct **`Message`** which contains two values: **`X`** and **`y`**. Go uses a very simple rule for deciding if an identifier is visible outside of the package in which it's defined which applies to both package-level constants and variables, and **`type`** names, methods and fields. If the identifier starts with a capital letter it's visible outside the package, otherwise it's private to the package.

The Go language spec guarantees that all variables will be initialised to the zero value for their type. For a **`struct`** type this means that every field will be initialised to an appropriate zero value. Therefore when we declare a value of type **`Message`** the Go runtime will initialise all of its elements to their zero value (in this case a zero-length string and a nil pointer respectively), and likewise if we create a **`Message`** value using a literal

```
19   m := &Message{}
```

```
package main
import "fmt"
type Message struct {
  X string
  y *string
}
func (v Message) Print() {
  if v.y != nil {
    fmt.Println(v.X, *v.y)
  } else {
    fmt.Println(v.X)
  }
}
func (v *Message) Store(x, y string) {
  v.X = x
  v.y = &y
}
func main() {
  m := &Message{}
  m.Print()
  m.Store("Hello", "world")
  m.Print()
}
```

Listing 17

```go
package main
import "fmt"
type HelloWorld struct {}
func (h HelloWorld) String() string {
  return "Hello world"
}
type Message struct {
  HelloWorld
}
func main() {
  m := &Message{}
  fmt.Println(m.HelloWorld.String())
  fmt.Println(m.String())
  fmt.Println(m)
}
```

Listing 18

Having declared a **struct** type we can declare any number of **method** functions which will operate on this type. In this case we've introduced **Print()** which is called on a **Message** value to display it in the terminal, and **Store()** which is called on a pointer to a **Message** value to change its contents. The reason **Store()** applies to a pointer is that we want to be able to change the contents of the **Message** and have these changes persist. If we define the method to work directly on the value these changes won't be propagated outside the method's scope. To test this for yourself, make the following change to the program:

```
14 func (v Message) Store(x, y string) {
```

If you're familiar with functional programming then the ability to use values immutably this way will doubtless spark all kinds of interesting ideas.

There's another **struct** trick I want to show off before we move on and that's *type embedding* using an anonymous field. Go's design has upset quite a few people with an inheritance-based view of object orientation because it lacks inheritance; however, thanks to *type embedding* we're able to compose types which act as proxies to the *methods* provided by anonymous fields. As with most things, an example (Listing 18) will make this much clearer.

```
$ go run 18.go
Hello world
Hello world
Hello world
```

Here we're declaring a type **HelloWorld** which in this case is just an empty **struct**, but which in reality could be any declared type. **HelloWorld** defines a **String()** method which can be called on any **HelloWorld** value. We then declare a type **Message** which embeds the **HelloWorld** type by defining an anonymous field of the **HelloWorld** type. Wherever we encounter a value of type **Message** and wish to call **String()** on its embedded **HelloWorld** value we can do so by calling **String()** directly on the value, calling **String()** on the **Message** value, or in this case by allowing **fmt.Println()** to match it with the **fmt.Stringer** interface.

Any declared type can be embedded, so in Listing 19, we're going to base **HelloWorld** on the primitive **bool** boolean type to prove the point.

In our final example (Listing 20) we've declared the **Hello** type and embedded it in **Message**, then we've implemented a new **String()** method which allows a **Message** value more control over how it's printed.

```
$ go run 20.go
Hello
Hello world
```

In all these examples we've made liberal use of the ***** and **&** operators. An explanation is in order.

Go is a systems programming language, and this means that a Go program has direct access to the memory of the platform it's running on. This requires that Go has a means of referring to specific addresses in memory and of accessing their contents indirectly. The **&** operator is prepended to the name of a variable or to a value literal when we wish to discover its address in memory, which we refer to as a pointer. To do anything with the pointer returned by the **&** operator we need to be able to declare a pointer variable which we do by prepending a type name with the ***** operator. An example (Listing 21) will probably make this description somewhat clearer, and we get:

```
$ go run 21.go
name = Ellie stored at 0x208178170
pointer_to_name references Ellie
```

Go allows user-defined types to declare methods on either a *value type* or a *pointer to a value type*. When methods operate on a *value type* the value manipulated remains immutable to the rest of the program (essentially the method operates on a copy of the value) whilst with a *pointer to a value type* any changes to the value are apparent throughout the program. This has far-reaching implications which we'll explore in later chapters.

```go
package main
import "fmt"
type HelloWorld bool
func (h HelloWorld) String() (r string) {
  if h {
    r = "Hello world"
  }
  return
}
type Message struct {
  HelloWorld
}
func main() {
  m := &Message{ HelloWorld: true }
  fmt.Println(m)
  m.HelloWorld = false
  fmt.Println(m)
  m.HelloWorld = true
  fmt.Println(m)
}
```

Listing 19

```go
package main
import "fmt"
type Hello struct {}
func (h Hello) String() string {
  return "Hello"
}
type Message struct {
  *Hello
  World string
}
func (v Message) String() (r string) {
  if v.Hello == nil {
    r = v.World
  } else {
    r = fmt.Sprintf("%v %v", v.Hello, v.World)
  }
  return
}
func main() {
  m := &Message{}
  fmt.Println(m)
  m.Hello = new(Hello)
  fmt.Println(m)
  m.World = "world"
  fmt.Println(m)
}
```

Listing 20

```
package main
import . "fmt"
type Text string
func main() {
  var name Text = "Ellie"
  var pointer_to_name *Text
  pointer_to_name = &name
  Printf("name = %v stored at %v\n", name, pointer_to_name)
  Printf("pointer_to_name references %v\n", *pointer_to_name)
}
```

Listing 21

Generalisation

Encapsulation is of huge benefit when writing complex programs and it also enables one of the more powerful features of Go's type system, the **`interface`**. An **`interface`** is similar to a **`struct`** in that it combines one or more elements but rather than defining a type in terms of the data items it contains, an **`interface`** defines it in terms of a set of method signatures which it must implement.

As none of the primitive types (**`int`**, **`string`**, etc.) have methods they match the empty **`interface (interface{})`** as do all other types, a property used frequently in Go programs to create generic containers.

Once declared, an interface can be used just like any other declared type, allowing functions and variables to operate with unknown types based solely on their required behaviour. Go's type inference system will then recognise compliant values as instances of the interface, allowing us to write generalised code with little fuss.

In Listing 22, we're going to introduce a simple **`interface`** (by far the most common kind) which matches any type with a **`func String() string`** method signature.

```
package main
import "fmt"
type Stringer interface {
  String() string
}
type Hello struct {}
func (h Hello) String() string {
  return "Hello"
}
type World struct {}
func (w *World) String() string {
  return "world"
}
type Message struct {
  X Stringer
  Y Stringer
}
func (v Message) String() (r string) {
  switch {
  case v.X == nil && v.Y == nil:
  case v.X == nil:
    r = v.Y.String()
  case v.Y == nil:
    r = v.X.String()
  default:
    r = fmt.Sprintf("%v %v", v.X, v.Y)
  }
  return
}
```

Listing 22

```
func main() {
  m := &Message{}
  fmt.Println(m)
  m.X = new(Hello)
  fmt.Println(m)
  m.Y = new(World)
  fmt.Println(m)
  m.Y = m.X
  fmt.Println(m)
  m = &Message{ X: new(World), Y: new(Hello) }
  fmt.Println(m)
  m.X, m.Y = m.Y, m.X
  fmt.Println(m)
}
```

Listing 22 (cont'd)

```
$ go run 22.go
Hello
Hello world
Hello Hello
world Hello
Hello world
```

This **`interface`** is copied directly from **`fmt.Stringer`**, so we can simplify our code a little by using that interface instead:

```
type Message struct {
  X fmt.Stringer
  Y fmt.Stringer
}
```

As Go is strongly typed **`interface`** values contain both a pointer to the value contained in the **`interface`**, and the *concrete* type of the stored value. This allows us to perform type assertions to confirm that the value inside an **`interface`** matches a particular concrete type (see Listing 23).

```
package main
import "fmt"
type Hello struct {}
func (h Hello) String() string {
  return "Hello"
}
type World struct {}
func (w *World) String() string {
  return "world"
}
type Message struct {
  X fmt.Stringer
  Y fmt.Stringer
}
func (v Message) IsGreeting() (ok bool) {
  if _, ok = v.X.(*Hello); !ok {
    _, ok = v.Y.(*Hello)
  }
  return
}
func main() {
  m := &Message{}
  fmt.Println(m.IsGreeting())
  m.X = new(Hello)
  fmt.Println(m.IsGreeting())
  m.Y = new(World)
  fmt.Println(m.IsGreeting())
  m.Y = m.X
  fmt.Println(m.IsGreeting())
  m = &Message{ X: new(World), Y: new(Hello) }
  fmt.Println(m.IsGreeting())
  m.X, m.Y = m.Y, m.X
  fmt.Println(m.IsGreeting())
}
```

Listing 23

```
29 func main() {
30   m := &Message{}
31   fmt.Println(m.IsGreeting())
32   m.X = Hello{}
33   fmt.Println(m.IsGreeting())
34   m.X = new(Hello)
35   fmt.Println(m.IsGreeting())
36   m.X = World{}
37 }
```

Listing 24

```
go run 23.go
false
true
true
true
true
true
```

Here we've replaced **`Message`**'s **`String()`** method with **`IsGreeting()`**, a predicate which uses a pair of *type assertions* to tell us whether or not one of **`Message`**'s data fields contains a value of concrete type **`Hello`**.

So far in these examples we've been using pointers to **`Hello`** and **`World`** so the **`interface`** variables are storing pointers to pointers to these values (i.e. **`**Hello`** and **`**World`**) rather than pointers to the values themselves (i.e. **`*Hello`** and **`*World`**). In the case of **`World`** we have to do this to comply with the **`fmt.Stringer`** interface because **`String()`** is defined for **`*World`** and if we modify **`main`** to assign a **`World`** value to either field (see Listing 24) we'll get a compile-time error:

```
$ go run 24.go
# command-line-arguments
./24.go:36: cannot use World literal (type World)
as type fmt.Stringer in assignment:
World does not implement fmt.Stringer (String
method has pointer receiver)
```

The final thing to mention about **`interface`**s is that they support embedding of other **`interface`**s. This allows us to compose a new, more restrictive **`interface`** based on one or more existing **`interface`**s. Rather than demonstrate this with an example, we're going to look at code lifted directly from the standard **io** package which does this (Listing 25).

Here **io** is declaring three interfaces, the **`Reader`** and **`Writer`**, which are independent of each other, and the **`ReadWriter`** which combines both. Any time we declare a variable, field or function parameter in terms of a **`ReaderWriter`**, we know we can use both the **`Read()`** and **`Write()`** methods to manipulate it.

Startup

One of the less-discussed aspects of computer programs is the need to initialise many of them to a pre-determined state before they begin executing. Whilst this is probably the worst place to start discussing what to many people may appear to be advanced topics, one of my goals in this chapter is to cover all of the structural elements that we'll meet when we examine more complex programs.

```
67 type Reader interface {
68   Read(p []byte) (n int, err error)
69 }
78 type Writer interface {
79   Write(p []byte) (n int, err error)
80 }
106 type ReadWriter interface {
107   Reader
108   Writer
109 }
```

Listing 25

```
package main
import . "fmt"
const Hello = "hello"
var world  string
func init() {
  world = "world"
}
func main() {
  Println(Hello, world)
}
```

Listing 26

Every Go package may contain one or more **`init()`** functions specifying actions that should be taken during program initialisation. This is the one case I'm aware of where multiple declarations of the same identifier can occur without either resulting in a compilation error or the shadowing of a variable. In the following example we use the **`init()`** function to assign a value to our **`world`** variable (Listing 26).

However, the **`init()`** function can contain any valid Go code, allowing us to place the whole of our program in **`init()`** and leaving **`main()`** as a stub to convince the compiler that this is indeed a valid Go program (Listing 27).

When there are multiple **`init()`** functions, the order in which they're executed is indeterminate so in general it's best not to do this unless you can be certain the **`init()`** functions don't interact in any way. Listing 28 happens to work as expected on my development computer but an implementation of Go could just as easily arrange it to run in reverse order or even leave deciding the order of execution until runtime.

HTTP

So far our treatment of Hello World has followed the traditional route of printing a preset message to the console. Anyone would think we were living in the fuddy-duddy mainframe era of the 1970s instead of the shiny 21st Century, when web and mobile applications rule the world.

Turning Hello World into a web application is surprisingly simple, as Listing 29 demonstrates.

```
package main
import . "fmt"

const Hello = "hello"
var world  string

func init() {
  world = "world"
  Println(Hello, world)
}

func main() {}
```

Listing 27

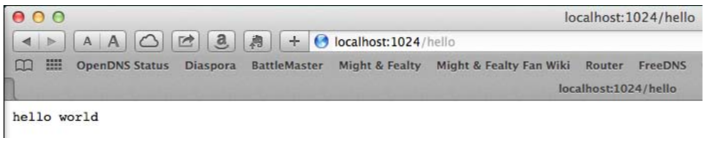
```
package main
import . "fmt"
const Hello = "hello"
var world  string
func init() {
  Print(Hello, " ")
  world = "world"
}
func init() {
  Printf("%v\n", world)
}
func main() {}
```

Listing 28

```
package main
import (
  . "fmt"
  "net/http"
)
const MESSAGE = "hello world"
const ADDRESS = ":1024"
func main() {
  http.HandleFunc("/hello", Hello)
  if e := http.ListenAndServe(ADDRESS, nil); e != nil {
    Println(e)
  }
}
func Hello(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/plain")
  Fprintf(w, MESSAGE)
}
```

Listing 29

Our web server is now listening on localhost port 1024 (usually the first non-privileged port on most Unix-like operating systems) and if we visit the url http://localhost:1024/hello with a web browser our server will return Hello World in the response body.



```
hello world
```

The first thing to note is that the **net/http** package provides a fully-functional web server which requires very little configuration. All we have to do to get our content to the browser is define a **handler**, which in this case is a function to call whenever an **http.Request** is received, and then launch a server to listen on the desired address with **http.ListenAndServe()**. **http.ListenAndServe** returns an error if it's unable to launch the server for some reason, which in this case we print to the console.

We're going to import the **net/http** package into the current namespace and assume our code won't encounter any runtime errors to make the simplicity even more apparent (Listing 30). If you run into any problems whilst trying the examples which follow, reinserting the **if** statement will allow you to figure out what's going on.

HandleFunc() registers a URL in the web server as the trigger for a function, so when a web request targets the URL the associated function will be executed to generate the result. The specified handler function is passed both a **ResponseWriter** to send output to the web client and the **Request** which is being replied to. The **ResponseWriter** is a file handle so we can use the **fmt.Fprint()** family of file-writing functions to create the response body.

```
package main
import (
  . "fmt"
  . "net/http"
)
const MESSAGE = "hello world"
const ADDRESS = ":1024"
func main() {
  HandleFunc("/hello", Hello)
  ListenAndServe(ADDRESS, nil)
}
func Hello(w ResponseWriter, r *Request) {
  w.Header().Set("Content-Type", "text/plain")
  Fprintf(w, MESSAGE)
}
```

Listing 30

```
package main
import (
  . "fmt"
  . "net/http"
)
const MESSAGE = "hello world"
const ADDRESS = ":1024"
func main() {
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, MESSAGE)
  })
  ListenAndServe(ADDRESS, nil)
}
```

Listing 31

Finally we launch the server using **ListenAndServe()**, which will block for as long as the server is active, returning an error if there is one to report.

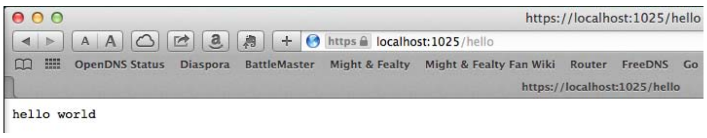
In this example (Listing 30) I've declared a function **Hello** and by referring to this in the call to **HandleFunc()** this becomes the function which is registered. However, Go also allows us to define functions anonymously where we wish to use a function value, as demonstrated in the following variation on our theme.

Functions are first-class values in Go and in Listing 31 **HandleFunc()** is passed an anonymous function value which is created at runtime. This value is a closure so it can also access variables in the lexical scope in which it's defined. We'll treat closures in greater depth later in my book, but for now Listing 32 is an example which demonstrates their basic premise by defining a variable **messages** in **main()** and then accessing it from within the anonymous function.

This is only a very brief taster of what's possible using **net/http** so we'll conclude by serving our hello world web application over an SSL connection (see Listing 33).

Before we run this program we first need to generate a certificate and a public key, which we can do using **crypto/tls/generate_cert.go** in the standard package library.

```
$ go run $GOROOT/src/pkg/crypto/tls/
generate_cert.go -ca=true -host="localhost"
2014/05/16 20:41:53 written cert.pem
2014/05/16 20:41:53 written key.pem
$ go run 33.go
```



```
hello world
```

```
package main
import (
  . "fmt"
  . "net/http"
)
const ADDRESS = ":1024"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
  ListenAndServe(ADDRESS, nil)
}
```

Listing 32

```
package main
import (
  . "fmt"
  . "net/http"
)
const SECURE_ADDRESS = ":1025"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter,
  r *Request) {
    w.Header().Set("Content-Type",
    "text/plain")
    Fprintf(w, message)
  })
  ListenAndServeTLS(SECURE_ADDRESS, "cert.pem",
  "key.pem", nil)
```

Listing 33

This is a self-signed certificate, and not all modern web browsers like these. Firefox will refuse to connect on the grounds the certificate is inadequate and not being a Firefox user I've not devoted much effort to solving this. Meanwhile both Chrome and Safari will prompt the user to confirm the certificate is trusted. I have no idea how Internet Explorer behaves. For production applications you'll need a certificate from a recognised Certificate Authority. Traditionally this would be purchased from a company such as Thawte for a fixed period but with the increasing emphasis on securing the web a number of major networking companies have banded together to launch Let's Encrypt. It's a free CA issuing short-duration certificates for SSL/TLS with support for automated renewal.

If you're anything like me (and you have my sympathy if you are) then the next thought to idle through your mind will be a fairly obvious question: given that we can serve our content over both HTTP and HTTPS connections, how do we do both from the same program?

To answer this we have to know a little – but not a lot – about how to model concurrency in a Go program. The go keyword marks a goroutine which is a lightweight thread scheduled by the Go runtime. How this is implemented under the hood doesn't matter, all we need to know is that when a goroutine is launched it takes a function call and creates a separate thread of execution for it. In Listing 34, we're going to launch a goroutine to run the HTTP server then run the HTTPS server in the main flow of execution.

When I first wrote this code it actually used two goroutines, one for each server. Unfortunately no matter how busy any particular goroutine is, when the **`main()`** function returns our program will exit and our web servers will terminate. So I tried the primitive approach we all know and love from C (see Listing 35).

```
package main
import (
  . "fmt"
  . "net/http"
)
const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter,
  r *Request) {
    w.Header().Set("Content-Type",
    "text/plain")
    Fprintf(w, message)
  })
  go func() {
    ListenAndServe(ADDRESS, nil)
  }()
  ListenAndServeTLS(SECURE_ADDRESS, "cert.pem",
  "key.pem", nil)
}
```

Listing 34

```
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter,
  r *Request) {
    w.Header().Set("Content-Type",
    "text/plain")
    Fprintf(w, message)
  })
  go func() {
    ListenAndServe(ADDRESS, nil)
  }()
  go func() {
    ListenAndServeTLS(SECURE_ADDRESS,
    "cert.pem", "key.pem", nil)
  }()
  for {}
}
```

Listing 35

Here we're using an infinite **`for`** loop to prevent program termination: it's inelegant, but this is a small program and dirty hacks have their appeal. Whilst semantically correct this unfortunately doesn't work either because of the way goroutines are scheduled: the infinite loop can potentially starve the thread scheduler and prevent the other goroutines from running.

```
$ go version
go version go1.3 darwin/amd64
```

In any event an infinite loop is a nasty, unnecessary hack as Go allows concurrent elements of a program to communicate with each other via *channels*, allowing us to rewrite our code as in Listing 36.

For the next pair of examples we're going to use two separate goroutines to run our HTTP and HTTPS servers, yet again coordinating program termination with a shared channel. In Listing 37, we'll launch both of the goroutines from the **`main()`** function, which is a fairly typical code pattern.

For our second deviation (Listing 38), we're going to launch a goroutine from **`main()`** which will run our HTTPS server and this will launch the second goroutine which manages our HTTP server.

```
package main
import (
  . "fmt"
  . "net/http"
)
const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter,
  r *Request) {
    w.Header().Set("Content-Type",
    "text/plain")
    Fprintf(w, message)
  })
  done := make(chan bool)
  go func() {
    ListenAndServe(ADDRESS, nil)
    done <- true
  }()
  ListenAndServeTLS(SECURE_ADDRESS, "cert.pem",
  "key.pem", nil)
  <- done
}
```

Listing 36

```
package main
import (
  . "fmt"
  . "net/http"
)
const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter,
  r *Request) {
    w.Header().Set("Content-Type",
    "text/plain")
    Fprintf(w, message)
  })
  done := make(chan bool)
  go func() {
    ListenAndServe(ADDRESS, nil)
    done <- true
  }()
  go func () {
    ListenAndServeTLS(SECURE_ADDRESS,
    "cert.pem", "key.pem", nil)
    done <- true
  }()
  <- done
  <- done
}
```

Listing 37

There's a certain amount of fragile repetition in this code as we have to remember to explicitly create a channel, and then to send and receive on it multiple times to coordinate execution. As Go provides first-order functions (i.e. allows us to refer to functions the same way we refer to data, assigning instances of them to variables and passing them around as parameters to other functions), we can refactor the server launch code as in Listing 39.

However, this doesn't work as expected, so let's see if we can get any further insight

```
package main
import (
  . "fmt"
  . "net/http"
)
const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter,
  r *Request) {
    w.Header().Set("Content-Type",
    "text/plain")
    Fprintf(w, message)
  })
  done := make(chan bool)
  go func () {
    go func() {
      ListenAndServe(ADDRESS, nil)
      done <- true
    }()
    ListenAndServeTLS(SECURE_ADDRESS,
    "cert.pem", "key.pem", nil)
    done <- true
  }()
  <- done
  <- done
}
```

Listing 38

```
 1 package main
 2 import (
 3   . "fmt"
 4   . "net/http"
 5 )
 6 const ADDRESS = ":1024"
 7 const SECURE_ADDRESS = ":1025"
 8 func main() {
 9   message := "hello world"
10   HandleFunc("/hello", func(w ResponseWriter,
     r *Request) {
11     w.Header().Set("Content-Type",
       "text/plain")
12     Fprintf(w, message)
13   })
14   Spawn(
15     func() { ListenAndServeTLS(SECURE_ADDRESS,
       "cert.pem", "key.pem", nil) },
16   func() { ListenAndServe(ADDRESS, nil) },
17     )
18 }
19 func Spawn(f ...func()) {
20   done := make(chan bool)
21   for _, s := range f {
22     go func() {
23       s()
24       done <- true
25     }()
26   }
27   for l := len(f); l > 0; l-- {
28     <- done
29   }
30 }
```

Listing 39

```
$ go vet 39.go
39.go:23: range variable s captured by func literal
exit status 1
```

Running **go** with the **vet** command runs a set of heuristics against our source code to check for common errors which wouldn't be caught during compilation. In this case we're being warned about this code

```
21 for _, s := range f {
22   go func() {
23     s()
24     done <- true
25   }()
26 }
```

Here we're using a closure so it refers to the variable **s** in the **for**...**range** statement, and as the value of **s** changes on each successive iteration, so this is reflected in the call **s()**.

To demonstrate this, we'll try a variant where we introduce a delay on each loop iteration much greater than the time taken to launch the goroutine (see Listing 40).

When we run this we get the behaviour we expect with both HTTP and HTTPS servers running on their respective ports and responding to browser traffic. However, this is hardly an elegant or practical solution and there's a much better way of achieving the same effect (Listing 41).

By accepting the parameter **server** to the goroutine's closure we can pass in the value of **s** and capture it so that on successive iterations of the range our goroutines use the correct value.

Spawn() is an example of how powerful Go's support for first-class functions can be, allowing us to run any arbitrary piece of code and wait for it to signal completion. It's also a *variadic* function, taking as many or as few functions as desired and setting each of them up correctly.

If we now reach for the standard library we discover that another alternative is to use a **sync.WaitGroup** to keep track of how many active goroutines we have in our program and only terminate the program

```go
package main
import (
  . "fmt"
  . "net/http"
  "time"
)
const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
  Spawn(
    func() { ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil) },
    func() { ListenAndServe(ADDRESS, nil) },
  )
}
func Spawn(f ...func()) {
  done := make(chan bool)
  for _, s := range f {
    go func() {
      s()
      done <- true
    }()
    time.Sleep(time.Second)
  }
  for l := len(f); l > 0; l-- {
    <- done
  }
}
```

Listing 40

when they've all completed their work. Yet again this allows us to run both servers in separate goroutines and manage termination correctly. (See Listing 42.)

As there's a certain amount of redundancy in this, let's refactor a little by packaging server initiation into a new **`Launch()`** function. **`Launch()`** takes a parameter-less function and wraps this in a **`closure`** which will be launched as a goroutine in a separate thread of execution. Our **`sync.WaitGroup`** variable servers has been turned into a global variable to simplify the function signature of **`Launch()`**. When we call **`Launch()`** we're freed from the need to manually increment servers prior to goroutine startup, and we use a defer statement to automatically call **`servers.Done()`** when the goroutine terminates even in the event that the goroutine crashes. See Listing 43. ■

```go
  for _, s := range f {
    go func(server func()) {
      server()
      done <- true
    }(s)
  }
```

Listing 41

This is an extract from *A Go Developer's Notebook*, a living eBook about Go and programming.

Living eBooks are purchased once and freely updated when the author has something new to say.

You can purchase your copy at http://leanpub.com/GoNotebook

```go
package main
import (
  . "fmt"
  . "net/http"
  "sync"
)
const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
  var servers sync.WaitGroup
  servers.Add(1)
  go func() {
    defer servers.Done()
    ListenAndServe(ADDRESS, nil)
  }()
  servers.Add(1)
  go func() {
    defer servers.Done()
    ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
  }()
  servers.Wait()
}
```

Listing 42

```go
package main
import (
  . "fmt"
  . "net/http"
  "sync"
)
const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"

var servers sync.WaitGroup
func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
  Launch(func() {
    ListenAndServe(ADDRESS, nil)
  })
  Launch(func() {
    ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
  })
  servers.Wait()
}
func Launch(f func()) {
  servers.Add(1)
  go func() {
    defer servers.Done()
    f()
  }()
}
```

Listing 43

Afterwood

One JavaScript module was removed and every Node.js build was knocked for six. Chris Oldwood fictionalises the tale.

The alarm punctuated the morning silence and brought Norman to an abrupt state of consciousness. After the disturbing buzzer was itself silenced the room was then gently brought to life with the sound of gurgling as the teasmade continued the morning ritual by making a fresh cuppa. Norman always liked to start the day with a fresh brew.

The rest of the day began much like any other with the usual bowl of high-fibre cereal, an invigorating shower and the precision buttering of bread for his home-made packed lunch. As he cycled to work he thought about the impending cricket match at the weekend. The team had played well all season and only lost a single game so far and Norman, as team manager, felt he had been instrumental in their success by giving the players plenty of freedom.

As Norman approached the office his thoughts began to switch from the cricket pitch to the day ahead. He made a mental note to check the cricket kitbag during lunchtime and then his personal life faded-out and his working life came into focus as the bicycle shed came closer into view. He reached into his rucksack and pulled out a couple of different locks, both fairly chunky, and with a padlock that Fort Knox would be proud of. With the bike firmly restrained he reached into his pocket to find a security pass which he proudly presented to the guard.

Norman took security very seriously. Prior to taking his current position at SeaPan Ltd he had himself worked as a security guard at a museum in the city. Initially he'd only been entrusted with the less valuable works of art such as the Roman pottery and textiles, but his exemplary work ethic and timekeeping had quickly earned him promotion. Within only a few months he found himself responsible for the safe-keeping of the museum's prized collection of precious stones. Whilst he could appreciate the workmanship of the ceramics there was always something a little more special about the way the light would sparkle from the gems.

Alas the hours were unsociable and this interfered with his desire to play in, and eventually manage, the local cricket team. While recounting his tale of woe to another member of the team it transpired there was an opening at SeaPan Ltd which might be of interest. Norman quickly impressed in the interview and by the evening the job was already his.

The route to the department took in various twists and turns and like many postal rooms the décor was drab and functional. It was a far cry from the beautiful galleries of the museum but the job had its benefits and best of all he had his own office. He paused for a moment to admire his name on the door, 'Normal P. Marshall, Package Manager', then gave a wry smile and stepped inside prepared for the challenges of another day.

The morning largely passed by without undue concern. There was the usual assortment of queries from various members of staff complaining about the intolerable lack of service despite the fact that the problem was one of their own making. One rather irate salesman called Conan was becoming increasingly agitated over the delay of his parcel. He descended all five floors to the postal room demanding to know what imbecile was in charge, and what they were going to do about finding his missing packet. Norman nonchalantly stepped over to the large basket in the corner and pulled out a parcel they received a few days earlier which had been badly addressed. He asked Conan whether it was the one he had been expecting and, after grudgingly receiving a confirmation, took a moment to calmly re-educate his colleague on the most effective technique for ensuring packages are correctly identified.

Lunchtime finally arrived and Norman took his rucksack down from the top of the filing cabinet. He pulled out the make-shift lunchbox recycled from an old margarine tub and started to munch away whilst flicking through the discarded newspaper Conan had brought along in a rather pathetic attempt to look a little more menacing.

One further benefit of his new role was that the office was only a few minutes away from the cricket pavilion where the team played. This made popping over there to check the kit was in order for the weekend match easy to squeeze within his lunch break; after all he wasn't expecting any surprises, just check the items off his mentally stored list and get back to the office for the afternoon shift.

He opened the pavilion door, rested his cycle up against the bar and headed into the home team dressing room. The long leather bag was already open and there had definitely been signs of some rummaging around. Norman knelt down and began to sift through the bag checking off the balls, gloves, bails, etc. Reaching the bottom his face slowly turned pale as he realised he was an item short – one of the left pads was missing!

Norman quickly rationalised that this wasn't an opportunistic thief at work, but probably just one of the other players taking a few liberties. He quickly searched the pavilion to be sure it hadn't been haphazardly returned, but found no trace. With his lunch hour about to expire he locked the pavilion back up and headed over to the office.

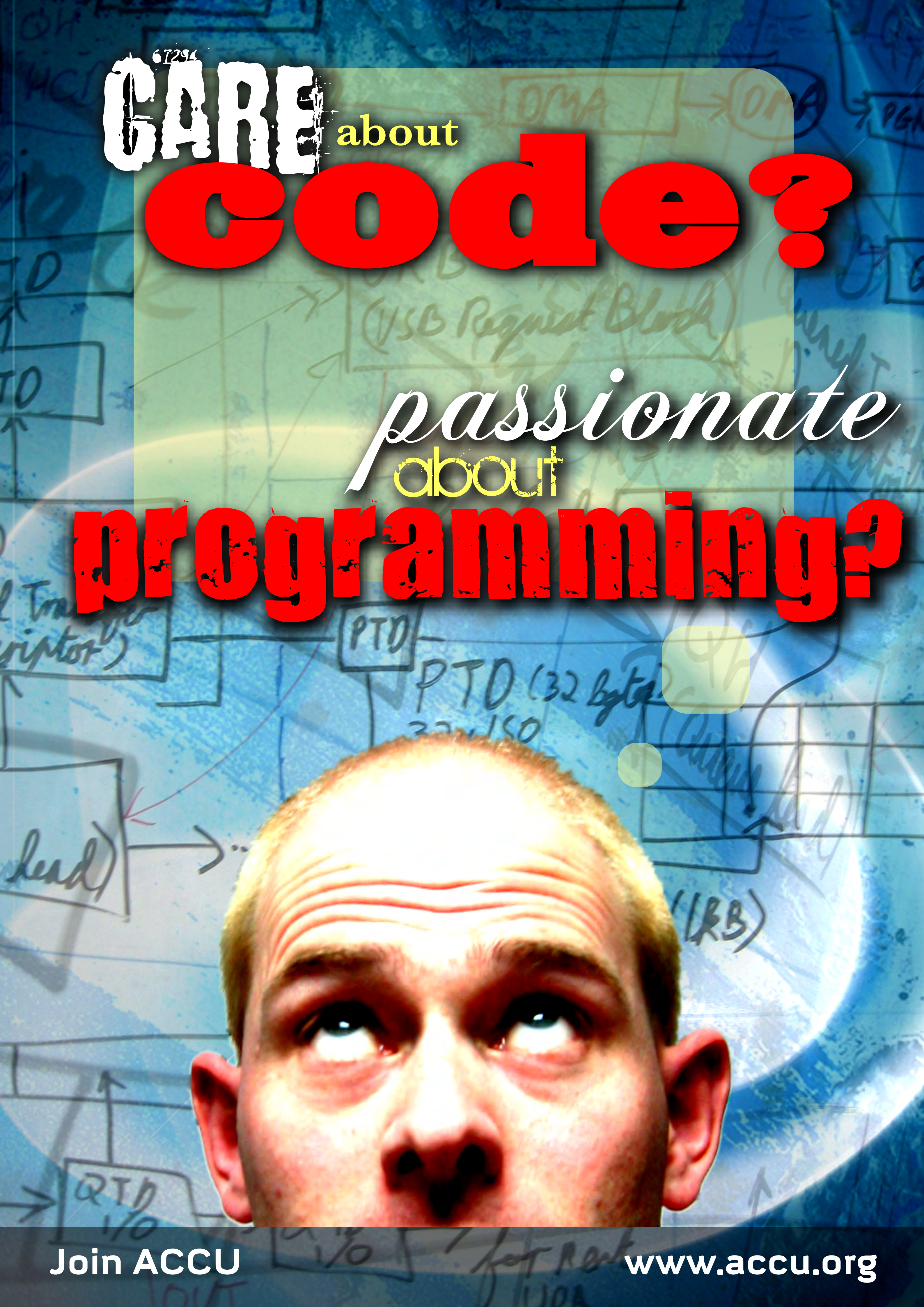
He knew he didn't have time to personally ring everyone until the following evening so decided to broadcast his plight via social media in the hope that someone would own up, or at least take up the mantle on his behalf. Various replies came in during the afternoon with a few offering support, but the majority were just sarcastic comments. Norman had pretty thick skin but when people who didn't even play in the team began trolling it started to grate on him.

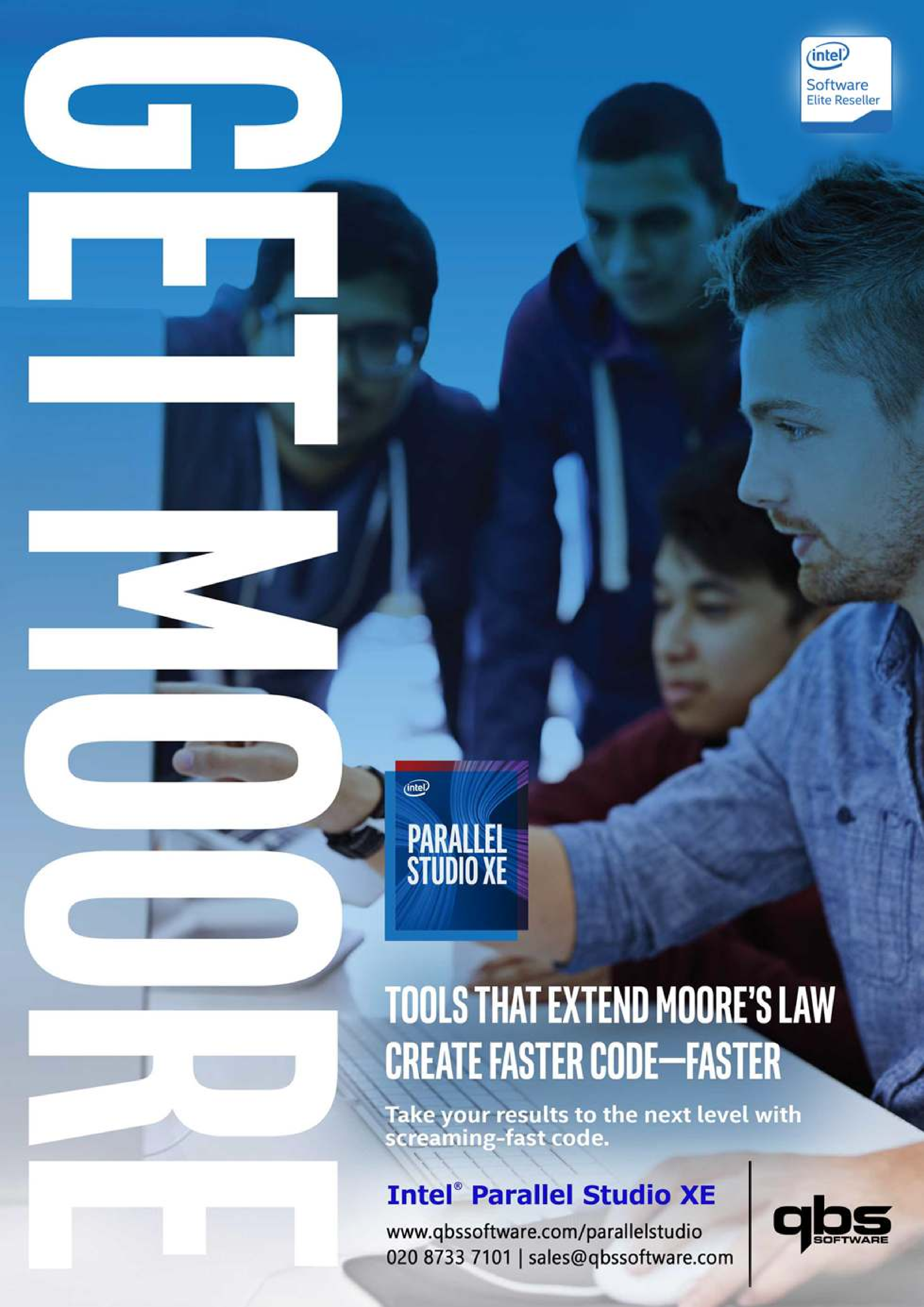
Fortunately the mystery was soon resolved. The team captain had decided to get some extra batting practice in and, finding the groundsman had temporarily unlocked the pavilion, had helped himself to the pad. (It takes ages to put them on so decided just to protect his leading leg.) Having donated them to the club in the first place he felt he was probably within his right to borrow them whenever he pleased. The groundsman unexpectedly locked up soon after and so he couldn't return it to the kitbag later.

Norman swung by the captain's house on the way home to give him a piece of his mind. He didn't want to have to store the cricket kit under lock-and-key, but he felt if some members couldn't be trusted to look after what belonged to the entire team, then that's what he'd have to resort to.

Another win for the team at the weekend tempered Norman's animosity towards the captain and his knee jerk reaction to regulate access quickly began to fade. Instead he crafted a polite notice for the wall in the home team dressing room which reminded the players to consider the needs of the club before their own. The incident had clearly struck a chord as some of them pooled together and bought spares to ensure they would never again be short on match day. ■

Chris Oldwood Chris is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood


CARE about
code?
passionate
about
programming?
Join ACCU
www.accu.org



TOOLS THAT EXTEND MOORE'S LAW
CREATE FASTER CODE—FASTER

Take your results to the next level with screaming-fast code.

Intel® Parallel Studio XE

www.qbssoftware.com/parallelstudio
020 8733 7101 | sales@qbssoftware.com

qbs SOFTWARE