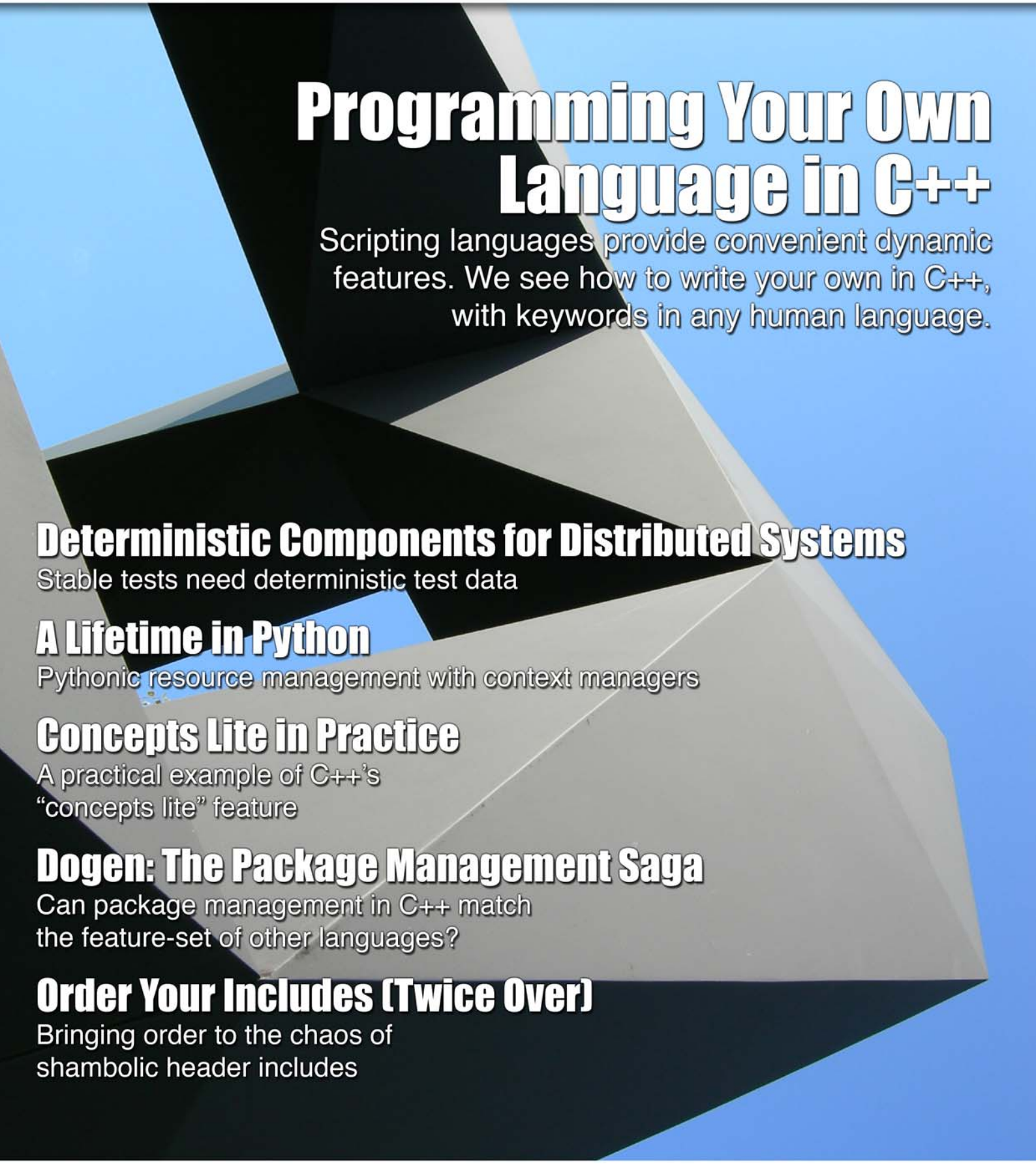
overload 133
JUNE 2016 £3



Programming Your Own Language in C++

Scripting languages provide convenient dynamic features. We see how to write your own in C++, with keywords in any human language.

Deterministic Components for Distributed Systems

Stable tests need deterministic test data

A Lifetime in Python

Pythonic resource management with context managers

Concepts Lite in Practice

A practical example of C++'s "concepts lite" feature

Dogen: The Package Management Saga

Can package management in C++ match the feature-set of other languages?

Order Your Includes (Twice Over)

Bringing order to the chaos of shambolic header includes

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 133

June 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 134 should be submitted by 1st July 2016 and those for Overload 135 by 1st September 2016.

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.

Overload is a publication of the ACCU

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

Metrics and Imperialism

Measuring accurately underpins science. Frances Buontempo considers what constitutes sensible metrics.

Having previously observed that time tends to fly by [Buontempo16a], I considered starting to measure how I am actually spending my time. This naturally led on to consideration of all the things we actually measure and why, causing the usual lack of time to write an editorial. Those regular readers who have been tracking my progress so far will find this means I have written a total of zero editorials to date. We have previously considered how many different authors *Overload* has had [Buontempo15] and how many articles they have written. We have had a few new authors since then, and of course would welcome even more. Simple counts like these can be illuminating.

Measuring can be a powerful tool. It helps you to observe changes, which you may have not otherwise noticed. It allows us to verify, or perhaps invalidate, a hypothesis. You can track your weight, or heart rate, or time taken to run a given distance, in order to monitor your health. You can track when you go to bed, and how long you sleep for, to find patterns in order to form better habits. Are you getting enough sleep? Do you usually go to bed at the same time? Some measurements can just report back a pass/fail, for example a build, with tests of course, on a continuous integration server. With this simple initial setup you can take it further to get more nuanced information, for example code coverage, number of warnings in the build (obviously zero, even if you had to `#pragma` turn off warnings to achieve this [King16]). There are various metrics for code quality we can use. Some are based on possibly arbitrary, though intuitively appealing, ideas such as McCabe's cyclomatic complexity, and cohesion [McCabe76]. Recently, many organisations have been adopting agile, and therefore started to measure how a team is doing, frequently through velocity. Jim Coplien spoke to the Bath Scrum User Group [BSUG] just before this year's ACCU conference about Scrum. In particular, he polled the audience, another good way of measuring things, to see if they knew why we have the Daily Scrum (clues: it's not to answer three questions, though that may be the format it takes), and what a team's sprint failure rate will be. Coplien constantly encouraged us to ask why we were doing things a certain way and not just do something for the sake of performing a ritual because 'the literature' said it would work. In fact, part of his mission is to build a body of pattern literature for the agile community to share. He did encourage us to just measure three main things: velocity, without correcting for meetings, holidays and the like, return on investment and finally happiness. Who starts a retrospective by asking if the last sprint made people happy or sad? Does your velocity have a direction? Stepping beyond scrum and process management, he pointed out another measure for that evening. His quick on the spot head-count indicated very few women were present, which is an issue Anna-Jayne Metcalfe dug into in her closing keynote, and lightning talk. The keynote, 'Comfort Zones', is now on the newly created ACCU conference youtube channel [ACCU channel].

This closing keynote, 'Comfort Zones', touched on the topic of anger, and aggression as a chosen response, in particular if people feel threatened or are frightened by change. Most of us will have heard the suggestion to 'count to ten' before responding to something we find annoying. Simply counting can make a difference. Getting to ten before responding gives you a chance to choose your reaction, rather than just responding in anger. In the *Hitch Hiker's Guide to the Galaxy*, Ford "carried on counting quietly" to annoy computers [Adams79]. Before a team I previously worked with started trying to measure test coverage, some of us just counted how many tests we had – seeing the number go up was motivating. Even setting up an empty test project on Jenkins tended to sprout some actual tests relatively quickly. Several people I know have managed to lose a lot of weight recently. Just weighing themselves, or counting calories, or steps taken every day provided 'weigh points' on the journey and were motivating. OK, bad joke, way points are reference points in space, though beacons, "Nodnol, 871 selim" [Red Dwarf] and so on show any units can be used, including kilograms or pounds. Just counting things can be informative. Sometimes the numbers 'fall into the wrong hands' though, or form the basis of a macho competition. I have previously worked in companies where people proudly announced how many hours they had worked yesterday or this week. Putting in long hours is not always heroic. It can mean you are on the path to a heart attack, a marriage breakup, or just being very inefficient. If two people achieve the same goals and one was at work for 7 hours while the other pulled a 12 hour day, which would you rather have on your team? Working hard doesn't mean working longer. As I said previously, being busy isn't the same as being productive [Buontempo16a]. In dysfunctional teams, and organisations, there's a danger metrics can be used as a weapon. If a scrum team's velocity goes down, the solution is not making them work over the weekend. Neither is it questioning the team members one at a time to find a scapegoat, or the person to blame. The team's velocity, is, after all the **team's** velocity. To slightly misquote a Japanese proverb, we are measuring to fix the problem, not the blame.

When we measure we typically have units, though not always. Agile encourages us to use story or estimation points. There will be no linear mapping to the number of hours (or days) an item of work will take. They capture the perceived effort involved. If you use planning poker cards they tend to follow a Fibonacci sequence, to break the team out of a linear mind-set. Various resources, for example [ScrumPlop1] emphasise they are **unitless** numbers. Furthermore, bear in mind that these are an

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.

estimate or guess. Larger stories will tend to be less clear, so many teams and resources strive towards smaller stories, for example 'Small Items' [ScrumPlop2]. I have worked on a couple of teams now where we seem to manage seven stories each sprint, regardless of the capacity, tinkered with for holidays and meetings, or even the perceived effort for each item of work. Just counting stories done, sometimes in the midst of many hours playing planning poker or the like never ceases to amuse me. I recommend having two counts at least – the official way, and any other team metric that works for you. How many items does your team manage each day, or week or sprint? If you don't work on an agile team, or even a team, do you know what you can achieve? Do you know what's holding you up? Are you happy? Are you productive? If not, what can you do to change things?

Having considered some areas where a unitless approach can be appropriate, such as counts of completed work or story points, most metrics have units. The United Kingdom uses the so-called metric system now, though informally many older people still measure their height in feet or weight in stones. Prior to this we had the 'imperial' system. This appears to date back to the 1824 Weights and Measures act [Weights], which the internet assures me wasn't implemented until a couple of years later. What's in a name? This was an attempt to provide uniform weight and measures, so that throughout the Empire you knew what you would get if you asked for a gallon, or a slug. Many of the original measures were based on quite natural ideas – a foot is the length of a foot, and so on. If you need to be accurate this could be problematic, hence the attempt to standardise, however older measures would give a ball-park figure that might be good enough for many situations. The drive to standardisation spread over several centuries [Britannica], meaning that American measures forked off from the 1824 act, leaving us with differences between various units: a US gallon is smaller than a British imperial gallon. Pints differ. The list goes on. Standards change over time. C++ has had many reformulations. This is a good thing. An interesting point to note is that the precise definitions of various measures usually involve another measure. A weight might be given at a specific temperature. The benchmarks and references used vary over time. Indeed, how we accurately measure time has changed. How to measure time could fill an article or even a book. Instead consider the history of the 'metre', or 'meter' if you will. The French Academy of Sciences chose a definition for the metre based on one ten-millionth of the earth's meridian along a quadrant [NIST]. This allowed more precision than another suggestion to use the length of a pendulum with a half period of one second. Since gravity varies you would have to be precise about where the pendulum was, which may have required precise coordinates or distance in metres from somewhere, which could prove awkward, being slightly recursive in definition. Having spent time conducting a lot of pendulum experiments in 'A' Level physics at school I suspect it's easy to miss the exact moment it reaches the extrema of the swings. Since the 18th century the definition of a metre has been refined several times. By 1983 it became "the length of a path travelled by light in a vacuum during a time interval of 1/299,792,458 of a second" [NIST]. I would be even more likely to blink at the wrong moment using this, than for my pendulum experiments. Where imperial measurements had been based on some natural ideas, like a foot, and tended to be in multiples of 12 or 16 allowing easy division into various ratios, the French Revolution pushed towards multiples of ten. In fact, it seems the French Revolution experimented with a ten day week, admittedly in a twelve month year [Calendar]. Apart from the drive to decimalise everything, the renaming of days and months was part of a drive to remove religious names from the days and months. For whatever reason, this didn't stick. People seem to prefer a seven day week, and long weekends. The attempt to impose a new system of measures doesn't seem to work. An imperious imposition of the metric system over an imperial system isn't always successful. Some British people cling to the old imperial measures out of a sense of patriotism. If you trace the units' origins though, you quickly find that they were imposed by invaders, such as a Roman 'mile', or 'millia' being a thousand paces. Yard is from Anglo-Saxon 'gyrd' for a stick [Metrics]. The list is long.

As I finish writing, on May 4th, my musings on the imperial death march [Imperial March] conjures a picture of intimidation and fear. Metrics are a way to communicate. They need to be incredibly accurate in order to be scientific. History shows us that as our knowledge increases so our measures change. As we discover new things we will need to continually improve our 'yardsticks'. Sometimes we don't need accurate numbers. If you measure code coverage by your tests you can get bogged down in whether you include empty lines, comments, deciding if each part of a logical disjunction (`or`) is evaluated before a branch is taken or just one by lazy evaluation and so on. It might be sufficient to just look at the names of the tests with a customer or product owner to get a sense of coverage of the requirements, rather than the code. Just because you can measure something doesn't mean you should. Steve Elliot spoke at the Pipeline conference in London this year. He shared many metrics that can be useful from a DevOps perspective; measure ALL the things (starting in a development environment) but encouraged us to make sure they weren't used to be Orwellian [Buontempo16b]. Rewarding people who write the most lines of code is asking for trouble. Catching a process that is failing before it hits QA or the customers, however, is better. Constantly reassessing the way you measure is important. The numbers, graphs and dashboards are a way to communicate between the whole team, rather than enable a witch-hunt. Simply talking can solve problems, but it should be based on a mixture of feelings and science.

Change is good. Change is constant. Embrace it, but watch and measure where things are heading. Ask yourself if you are happy. Ask yourself how you can improve. Ask yourself if I will ever write an editorial.

References

[ACCU channel] See Day 4 for the closing keynote. https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw/featured

[Adams79] *The Hitch Hikers Guide to the Galaxy*, Douglas Adams, 1979.

[Britannica] http://www.britannica.com/science/British-Imperial-System

[BSUG] Bath Scrum User Group http://www.meetup.com/Bath-Scrum-User-Group/events/228943766/

[Buontempo15] 'How to write an article' *Overload* 125, Feb 2015 http://accu.org/index.php/journals/2061

[Buontempo16a] 'Where does all the time go?' *Overload* 132, April 2016 *http://accu.org/var/uploads/journals/Overload132.pdf*

[Buontempo16b] http://buontempoconsulting.blogspot.co.uk/2016/03/pipeline-2016.html

[Calendar] https://en.wikipedia.org/wiki/French_Republican_Calendar

[Imperial March] https://www.youtube.com/watch?v=-bzWSJG93P8 See what I did there?

[King16] Guy Bolton King, ACCU Conference, 2016 http://accu.org/index.php/conferences/accu_conference_2016/accu2016_sessions#Without_Warning:_Keeping_the_Noise_Down_in_Legacy_Code_Builds

[McCabe76] McCabe, 'A Complexity Measure' *IEEE Transactions of Software Engineering* 2(4), 1976

[Metrics] http://www.metric.org.uk/myths/imperial#imperial-was-invented-in-britain

[NIST] http://physics.nist.gov/cuu/Units/meter.html

[Red Dwarf] http://reddwarf.wikia.com/wiki/RD:_Backwards

[ScrumPlop1] https://sites.google.com/a/scrumplop.org/published-patterns/value-stream/estimation-points

[ScrumPlop2] https://sites.google.com/a/scrumplop.org/published-patterns/value-stream/small-items

[Weights] https://en.wikipedia.org/wiki/Imperial_units

Dogen: The Package Management Saga

How do you manage packages in C++? Marco Craveiro eventually discovered Conan after some frustrating experiences.

Over the last few years I've had a little project on the side called Dogen. Dogen is a code generator designed to target domain models, with the lofty ambition of automating the modelling process as much as possible: users create domain models using a supported UML tool and respecting a set of predefined restrictions; Dogen uses the tool's diagram files to generate the source code representation. The generated code contains most of the services required from a typical domain object such as serialisation, hashing, streaming and so on. Dogen is written in C++ 14 and generates C++ 14 too, but other languages will eventually be supported as well. You can find our repo at https://github.com/DomainDrivenConsulting/dogen.

Now, for a four-year-old project, I guess it's fair to say that Dogen hasn't exactly set the Open Source world on fire. Nevertheless, it has proven to be a personal fountain of lessons and experiences on software development; one such lesson was package management and that's what I shall reminisce about in this article.

The conundrum

Like any other part-time C++ developer whose professional mainstay is C# and Java, I have keenly felt the need for a package manager when in C++-land. The problem is less visible when you are working with mature libraries and dealing with just Linux, due to the huge size of the package repositories and the great tooling built around them. However, things get messier when you start to go cross-platform, and messier still when you are coding on the bleeding edge of C++: either the package you need is not available in the distro's repos or even PPA's; or, when it is, its rarely at the version you require. Alas, for all our sins, that's exactly where we were when Dogen got started.

A spoonful of Dogen history

Dogen sprung to life just a tad after C++-0x became C++-11, so we experienced first hand the highs of a quasi-new-language followed by the lows of feeling the brunt of the bleeding edge pain. For starters, nothing we ever wanted was available out of the box, on any of the platforms we were interested in. Even Debian Testing was a fair bit behind the curve – probably stalled due to a compiler transition or other. In those days, Real Programmers were Real Programmers and mice were mice: we had to build and install the C++ compilers ourselves and, even then, C++-11 support was new, a bit flaky and limited. We then had to use those compilers to compile all of the dependencies in C++-11 mode.

Marco Craveiro has been programming professionally for the best part of twenty years and carefully following the trials and tribulations of his favourite technologies: Linux and C++. Recently, he became interested in the field of Computational Neuroscience and is now attempting to use Linux and C++ to model exceedingly small parts of the brain. In his copious spare time, he writes a haphazard blog with a haphazard programming column called Nerd Food.

The PFH days

After doing this manually once or twice, it soon stopped being fun. And so we solved this problem by creating the PFH – the Private Filesystem Hierarchy – a gloriously over-ambitious name to describe a set of wrapper scripts that helped with the process of downloading tarballs, unpacking, building and finally installing them into well-defined locations. It worked well enough in the confines of its remit, but we were often outside those, having to apply out-of-tree patches, adding new dependencies and so on. We also didn't use Travis then; not even sure it existed, but if it did, the rigmarole of the bleeding edge experience would certainly put a stop to any ideas of using it. So we used a local install of CDash with a number of build agents on OSX, Windows (MinGW) and Linux (32-bit and 64-bit). Things worked beautifully when nothing changed and the setup was stable; but every time a new version of a library – or God forbid, of a compiler – was released, one had that sense of dread: do I really need to upgrade? And yet we often did, because we needed the features.

Since one of the main objectives of Dogen was to learn about C++-11, one has to say that the pain was worth it. But all of the moving parts described above were not ideal and they were not the thing you want to be wasting your precious time on when it is very scarce. They were certainly not scalable.

The good days and the bad days

Things improved slightly for a year or two when distros started shipping C++-11 compliant compilers and recent boost versions. This led to an attack of pragmatism, during which we ditched all platforms except for Linux, got rid of almost all our private infrastructure and moved over to Travis. For a while things looked really good. However, due to Travis' Ubuntu LTS policy, we were stuck with a rapidly ageing Boost version. At first PPAs were a good solution, but over time these became stale too.

Soon we were stuck, unable to afford to revert back to the bad old days of the PFH but also unable to freeze all dependencies in time, as it would provide a worse development experience. So it was that the only route left was to break the build on Travis and hope that a solution would manifest itself. The red build painfully lingered on, commit after commit, whilst alternatives such as Drone.io and GitLab were unsuccessfully tried.

Finally, there was nothing else for it. We simply needed a package manager to manage the development dependencies.

Nuget hopes dashed

Having used Nuget in anger for both C# and C++ projects – and given Microsoft's recent change of heart with regards to open source – I was secretly hoping that Nuget would get some traction in the wider C++ world. Nuget works well enough in Mono, and C++ support for Windows was added fairly early on. It was somewhat limited and a bit quirky at the start but it kept on getting better, to the point of being actually usable; we now use Nuget to manage our C++ dependencies at work – a Windows shop in the main – and it has improved our quality of life dramatically.

Some crazy-cool Spaniards had decided to create a stand alone package manager. Being from the same peninsula, I felt compelled to use their wares

Unfortunately, the troubles begun on closer inspection. The truth is that Microsoft's current Nuget focus is C# and Visual Studio, not Linux and C++. Also, it seems that outside Microsoft and Xamarin, there just isn't enough traction for this tool at present.

However, there have been a couple of recent announcements from Microsoft that give me hope things may change in the future:

- Clang with Microsoft CodeGen in VS 2015 Update 1
- Support for Android CMake projects in Visual Studio

Surely the logical consequence is to be able to manage packages in a consistent way across platforms? We can but hope.

Biicode comes to the rescue?

Nuget did not pan out but what did happen was even more unlikely: some crazy-cool Spaniards had decided to create a stand alone package manager. Being from the same peninsula, I felt compelled to use their wares, and was joyful as they went from strength to strength – including the success of their open source campaign. And I loved the fact that it integrated really well with CMake, and that CLion provided Biicode integration very early on.

However, my biggest problem with Biicode was that it was just too complicated. I don't mean to say the creators of the product didn't have very good reasons for their technical choices – Lord knows creating a product is hard enough, so I have nothing but praise to anyone who tries. However, for me personally, I never had the time to understand why Biicode needed its own version of CMake, nor did I want to modify my CMake files too much in order to fit properly with Biicode and so on. Basically, I needed a solution that worked well and required minimal changes at my end. Having been brought up with Maven and Nuget, I just could not understand why there wasn't a simple `packages.xml` file that specified the dependencies and then some non-intrusive CMake support to expose those into the CMake files. As you can see from some of my posts, it just seemed it required 'getting' Biicode in order to make use of it, which for me was not an option.

Another thing that annoyed me was the difficulty on knowing what the 'real' version of a library was. I wrote, at the time:

> One slightly confusing thing about the process of adding dependencies is that there may be more than one page for a given dependency and it is not clear which one is the 'best' one. For RapidJson there are three options, presumably from three different Biicode users:
>
> - fenix: authored on 2015-Apr-28, v1.0.1.
> - hithwen: authored 2014-Jul-30
> - denis: authored 2014-Oct-09
>
> The 'fenix' option appeared to be the most up-to-date so I went with that one. However, this illustrates a deeper issue: how do you know you can trust a package? In the ideal setup, the project owners would add Biicode support and that would then be the one true version. However, like any other project, Biicode faces the initial adoption conundrum: people are not going to be willing to spend time adding support for Biicode if there aren't a lot of users of Biicode out there already, but without a large library of dependencies there is nothing to draw users in. In this light, one can understand that it makes sense for Biicode to allow anyone to add new packages as a way to bootstrap their user base; but sooner or later they will face the same issues as all distributions face.
>
> A few features would be helpful in the mean time:
>
> - popularity/number of downloads
> - user ratings
>
> These metrics would help in deciding which package to depend on.

For all these reasons, I never found the time to get Biicode setup and these stories lingered in Dogen's backlog. And the build continued to be red.

Sadly Biicode the company didn't make it either. I feel very sad for the guys behind it, because their heart was on the right place.

Which brings us right up to date.

Enter Conan

When I was a kid, we were all big fans of Conan. No, not the barbarian, the Japanese Manga Future Boy Conan. For me the name Conan will always bring back great memories of this show, which we watched in the original Japanese with Portuguese subtitles. So I was secretly pleased when I found conan.io, a new package management system for C++. The guy behind it seems to be one of the original Biicode developers, so a lot of lessons from Biicode were learned.

To cut a short story short, the great news is I managed to add Conan support to Dogen in roughly 3 hours and with very minimal knowledge about Conan. This to me was a litmus test of sorts, because I have very little interest in package management – creating my own product has proven to be challenging enough, so the last thing I need is to divert my energy further. The other interesting thing is that roughly half of that time was taken by trying to get Travis to behave, so its not quite fair to impute it to Conan.

Setting up Dogen for Conan

So, what changes did I do to get it all working? It was a very simple 3-step process. First I installed Conan using a Debian package from their site.

I then created a `conanfile.txt` on my top-level directory:

```
[requires]
Boost/1.60.0@lasote/stable
[generators]
cmake
```

Finally I modified my top-level `CMakeLists.txt`:

```
# conan support
if(EXISTS
  "${CMAKE_BINARY_DIR}/   conanbuildinfo.cmake")
  message(STATUS "Setting up Conan support.")
  include
    ("${CMAKE_BINARY_DIR}/conanbuildinfo.cmake")
  CONAN_BASIC_SETUP()
```

```
  else()
    message(STATUS "Conan build file not found,
      skipping include")
  endif()
```

This means that it is entirely possible to build Dogen without Conan, but if it is present, it will be used. With these two changes, all that was left to do was to build:

```
$ cd dogen/build/output
$ mkdir gcc-5-conan
$ cd gcc-5-conan
$ conan install ../../..
$ make -j5 run_all_specs
```

Et voilà, I had a brand spanking new build of Dogen using Conan. Well, actually, not quite. I've omitted a couple of problems that are a bit of a distraction on the Conan success story. Let's look at them now.

Problems and their solutions

The first problem was that Boost 1.59 does not appear to have an overridden FindBoost, which means that I was not able to link. I moved to Boost 1.60 – which I wanted to do anyway – and it worked out of the box.

The second problem was that Conan seems to get confused with Ninja, my build system of choice. For whatever reason, when I use the Ninja generator, it fails like so:

```
$ cmake ../../../ -G Ninja
$ ninja -j5
$ ninja: error: '~/.conan/data/Boost/1.60.0/
lasote/stable/package/
ebdc9c0c0164b54c29125127c75297f6607946c5/lib/
libboost_system.so', needed by 'stage/bin/
dogen_utility_spec', missing and no known rule to
make it
```

This is very strange because Boost System is clearly available in the Conan download folder. Going back to **make** solved this problem. I've opened an issue in Conan (#56) and its currently under investigation.

The third problem is more Boost related than anything else. Boost Graph has not been as well maintained as it should, really. Thus users now find themselves carrying patches, and all because no one seems to be able to apply them upstream. Dogen is in this situation as we've hit the issue described at Stack Overflow: 'Compile error with boost.graph 1.56.0 and g++ 4.6.4'. Sadly this is still present on Boost 1.60; the patch exists in Trac but remains unapplied (#10382). This is a tad worrying as we make a lot of use of Boost Graph and intend to increase the usage in the future.

At any rate, as you can see, none of the problems were showstoppers, nor can they all be attributed to Conan.

Getting Travis to behave

Once I got Dogen building locally, I then went on a mission to convince Travis to use it. It was painful, but mainly because of the lag between commits and hitting an error. The core of the changes to my YML file were as in Listing 1.

I probably should have a bash script by now, given the size of the YML, but hey – if it works. The changes above deal with installation of the package, applying the boost patch and using Make instead of Ninja. Quite trivial in the end, even though it required a lot of iterations to get there.

Conclusions

Having a red build is a very distressful event for a developer, so you can imagine how painful it has been to have red builds for *several months*. So it is with unmitigated pleasure that I got to see build #628 in a shiny emerald green. As far as that goes, it has been an unmitigated success.

In a broader sense though, what can we say about Conan? There are many positives to take home, even at this early stage of Dogen usage:

- it is a lot less intrusive than Biicode and easier to setup. Biicode was very well documented, but it was easy to stray from the beaten track and that then required reading a lot of different wiki pages. It seems easier to stay on the beaten track with Conan.
- as with Biicode, it seems to provide solutions to Debug/Release, multi-platforms and multiple compilers. We shall be testing it on Windows soon and reporting back.
- hopefully, since it started Open Source from the beginning, it will form a community of developers around the source with the know-how required to maintain it. It would also be great to see if a business forms around it, since someone will have to pay the cloud bill. It certainly is gaining popularity, as the recent CppCast attests.

In terms of negatives:

- I still believe the most scalable approach would have been to extend Nuget for the C++ Linux use case, since Microsoft is willing to take patches and since they foot the bill for the public repo. However, I can understand why one would prefer to have total control over the solution rather than depend on the whims of some middle-manager in order to commit.
- it seems publishing packages requires getting down into Python. Haven't tried it yet, but I'm hoping it will be made as easy as importing packages with a simple text file. The more complexity around these flows the tool adds, the less likely they are to be used.
- there still are no 'official builds' from projects. As explained above, this is a chicken and egg problem, because people are only willing to dedicate time to it once there are enough users complaining. Having said that, since Conan is easy to setup, one hopes to see some adoption in the near future.
- even when using a GitHub profile, one still has to define a Conan specific password. This was not required with Biicode. Minor pain, but still, if they want to increase traction, this is probably an unnecessary stumbling block. It was sufficient to make me think twice about setting up a login, for one.

In truth, these are all very minor negative points, but still worth making them. All and all, I am quite pleased with Conan thus far. ■

```
install:
<snip>
  # conan
  - wget https://s3-eu-west-1.amazonaws.com/conanio-production/downloads/
conan-ubuntu-64_0_5_0.deb -O conan.deb
  - sudo dpkg -i conan.deb
  - rm conan.deb
<snip>
script:
  - export GIT_REPO="`pwd`"
  - cd ${GIT_REPO}/build
  - mkdir output
  - cd output
  - conan install ${GIT_REPO}
  - hash=`ls ~/.conan/data/Boost/1.60.0/lasote/stable/package/`
  - cd ~/.conan/data/Boost/1.60.0/lasote/stable/package/${hash}/include/
  - sudo patch -p0 < ${GIT_REPO}/patches/boost_1_59_graph.patch
  - cmake ${GIT_REPO} -DWITH_MINIMAL_PACKAGING=on
  - make -j2 run_all_specs
<snip>
```

Listing 1

QM Bites – Order Your Includes (Twice Over)

Header includes can be a shambles. Matthew Wilson encourages us to bring some order to the chaos.

TL;DR

Order includes in groups of descending specificity and lexicographically within groups

Bite

Consider the following example of **#include**s in source file `Cutter.cpp`, containing the implementation of a class **Cutter** for a fictional organisation AcmeSoftware with a product Blade. In this case, the class's implementation and header files are located in the same source directory; this need not always be so, but the discussion to follow still applies.

```
#include "stdafx.h"
#include <vector>
#include <string>
#include <acmecmn/string_util2.h>
#include <acmecmn/string_util1.h>
#include <Blade/Sharpener.hpp>
#include <Blade/Protector.hpp>
#include "Cutter.hpp"
#include <stdlib.h>
#include <map>
```

This is quite wrong. Here's the right way to do it:

```
#include "stdafx.h"

#include "Cutter.hpp"

#include <Blade/Protector.hpp>
#include <Blade/Sharpener.hpp>

#include <acmecmn/string_util1.h>
#include <acmecmn/string_util2.h>

#include <map>
#include <string>
#include <vector>

#include <stdlib.h>
```

This has been ordered according to descending order of specificity of groups, and then lexicographically within groups. The drivers are, respectively, *modularity* and *transparency*.

The reason for descending order of specificity of groups is to expose hidden dependencies – *coupling!* – in any of the header files. Unlike languages such as Java and C#, the order of 'imports', in the form of **#include**s, has significance in C and C++. For example, if `Cutter.hpp` makes use of **std::vector** but does not itself include that file then compilation units that include it are at risk of compile error; the original order masks that. The same rationale applies to the files in other groups: if `Blade/Sharpener.hpp` requires a definition in `acmecmn/string_util1.h` then this would also be exposed.

The reason for lexicographical ordering with groups is to make it easier to comprehend each group's contents. In the real world there can be many tens of included files, and if not in some readily comprehensible order then duplicates can more easily occur, which is then obviously a problem when trying to remove unnecessary includes. (Admittedly, by having a strict lexicographical ordering within groups there is a slightly increased possibility of hiding interheader coupling between files in a group, but unless you want to go to some extreme such as reverse lexicographical ordering for headers and forward for implementation files – which I do *not* advise – you'll have to wear this slight risk. The grouping will take care of the vast majority.)

The reason for a blank line between groups is obvious: to delineate one group from another to further aid transparency.

Usually, the most specific group – the *Level-1* group in an implementation file – would be its declaring header(s), containing declarations of its API functions and/or defining its class: in this case `Cutter.hpp`. Note that it makes no difference whether the declaring header is in the same directory, i.e. **#include "Cutter.hpp"**, or in another directory, e.g. **#include <AcmeSoft/Blade/Cutter.hpp>**: its pre-eminence is unchanged.

Sometimes, for implementation reasons, we have to have a *Level-0* group – in this case this is the *precompiled header include file* `stdafx.h`. (In a soontobecooked Bite I will discuss why and how you should get rid of the presence of these things from your source.)

Similarly, for very rare implementation reasons, we have to have a *Level-N* group. I have not shown such in this case, but if you've chomped on enough C++ in your time you'll have experienced such things, perhaps to conduct some shameful but necessary preprocessor surgery after all includes but before any implementation code is translated.

Hence, the rule is to order includes:

1. Order all files in groups of descending order of specificity, with each group separated by a blank line, including:
   a. Explicit Level-0 includes (if required);
   b. Level-1 **include**(s): the declaring header(s) file (for implementation files only);
   c. Other include groups for the given application component;
   d. Other include groups for the organisation;
   e. Other include groups for 3rdpartysoftware;
   f. Standard C++ headers;
   g. Standard C headers;
   h. Explicit *Level-N* includes (if required).
2. Lexicographically order all includes within groups;

Further Reading

C++ Coding Standards, Herb Sutter & Andrei Alexandrescu, AddisonWesley, 2004

Large Scale C++ Software Design, John Lakos, AddisonWesley, 1996

Matthew Wilson Matthew is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.

A Lifetime In Python

Resource management is important in any language. Steve Love demonstrates how to use context managers in Python.

Variables in Python generally have a lifetime of their own. Or rather, the Python runtime interpreter handles object lifetime with automated garbage collection, leaving you to concentrate on more important things. Like **Resource** lifetime, which is much more interesting.

Python provides some facilities for handling the deterministic clean-up of certain objects, because sometimes it's necessary to know that it has happened at a specific point in a program. Things like closing file handles, releasing sockets, committing database changes – the usual suspects.

In this article I will explore Python's tools for managing resources in a deterministic way, and demonstrate why it's easier and better to use them than to roll your own.

Why you need it

Python, like many other languages, indicates runtime errors with exceptions, which introduces *interesting* requirements on state. Exceptions are not necessarily visible directly in your code, either. You just have to know they might occur. Listing 1 shows a very basic (didactic) example.

```
def addCustomerOrder( dbname, customer, order ):
  db = sqlite3.connect( dbname )                    (1)
  db.execute( 'INSERT OR REPLACE INTO customers \
    (id, name) VAlUES (?, ?)', customer )           (2)
  db.execute( 'INSERT INTO orders (date, custid,\
    itemid, qty) VALUES (?, ?, ?, ?)', order )      (3)
  db.commit()                                       (4)
  db.close()                                        (5)
```

Listing 1

If an exception occurs between lines (1) and (3), the data won't get committed to the database, the connection to the database will not be closed, and will therefore 'leak'. This could be a *big* problem if this function or other functions like it get called frequently, say as the backend to a large web application. This wouldn't be the best way to implement this in any case, but the point is that the `db.execute()` statement can throw all kinds of exceptions.

You might then try to explicitly handle the exceptions, as shown in Listing 2, which ensures the database connection is closed even in the event of an exception. Closing a connection without explicitly committing changes will cause them to be rolled back.

It is a bit messy, and introduces some other questions such as: what happens if the `sqlite3.connect` method throws an exception? Do we need *another* outer-try block for that? Or expect clients of this function to wrap it in an exception handler?

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He can be contacted at steve@arventech.com

```
def addCustomerOrder( dbname, customer, order ):
  db = sqlite3.connect( dbname )
  try:
    db.execute( 'INSERT OR REPLACE \
      INTO customers \
      (id, name) VAlUES (?, ?)', customer )
    db.execute( 'INSERT INTO orders \
      (date, custid, itemid, qty) \
      VALUES (?, ?, ?, ?)', order )
    db.commit()
  finally:
    db.close()
```

Listing 2

Fortunately, Python has already asked, and answered some of these questions, with the Context Manager. This allows you to write the code shown in Listing 3.

The connection object from the `sqlite3` module implements the Context Manager protocol, which is invoked using the `with` statement. This introduces a block scope, and the Context Manager protocol gives objects that implement it a way of defining what happens when that scope is exited.

In the case of the connection object, that behaviour is to commit the (implicit, in this case) transaction if no errors occurred, or roll it back if an exception was raised in the block.

Note the explicit call to `db.close()` *outside* of the `with` statement's scope. The *only* behaviour defined for the connection object as Context Manager is to commit or roll back the transaction when the scope is exited. This construct doesn't say anything at all about the lifetime of the `db` object itself. It will (probably) be garbage collected at some indeterminate point in the future.

You can do it too

This customer database library might have several functions associated with it, perhaps including facilities to retrieve or update customer details, report orders and so on. Perhaps it's better represented as a type, exposing an interface that captures those needs. See Listing 4 for an example.

```
def addCustomerOrder( dbname, customer, order ):
  with  sqlite3.connect( dbname ) as db:
    db.execute( 'INSERT OR REPLACE \
      INTO customers \
      (id, name) VAlUES (?, ?)', customer )
    db.execute( 'INSERT INTO orders \
      (date, custid, itemid, qty) \
      VALUES (?, ?, ?, ?)', order )
  db.close()
```

Listing 3

The only behaviour defined for the connection object as Context Manager is to commit or roll back the transaction when the scope is exited

```
class Customers( object ):
  def __init__( self, dbname ):
    self.db = sqlite3.connect( dbname )

  def close( self ):
    self.db.close()

  def addCustomerOrder( self, customer, order ):
    self.db.execute( 'INSERT OR REPLACE \
      INTO customers (id, name) \
      VAlUES (?, ?)', customer )
    self.db.execute( 'INSERT INTO orders \
      (date, custid, itemid, qty) \
      VALUES (?, ?, ?, ?)', order )

  # Other methods...

with Customers( dbname ) as db:
    db.addCustomerOrder( customer, order )
db.close()
```

Listing 4

Unfortunately, the line containing the **with** statement provokes an error similar to this:

```
    File "customerdb.py", line 21, in <module>
      with Customers( dbname ) as db:
AttributeError: __exit__
```

You can't use **with** on just any type you create. It's not a magic wand, either: the changes won't get committed to the database if **commit()** is not called! However, the Context Manager facility isn't limited to just those types in the Python Standard Library. It's implemented quite simply, as seen in Listing 5.

The **__init__()** method is still there, but just saves the name away for later use. When the **with** statement is executed, it calls the object's **__enter__()** method, and binds the return to the **as** clause if there is one: in this case, the **db** variable. The main content of the original construction method has been moved to the **__enter__()** method. Lastly, when the **with** statement block scope is exited, the **__exit__()** method of the managed object is called. If no exceptions occurred in the block, then the three arguments to **__exit__()** will be **None**. If an exception *did* occur, then they are populated with the type, value and stack trace object associated with the exception. This implementation essentially mimics the behaviour of the sqlite3 connection object, and rolls back if an exception occurred.

Returning a false-value indicates to the calling code that any exception that occurred inside the **with** block should be re-raised. Returning **None** counts – and is only explicitly specified here for the purposes of explaining it. A Python function with no return statement is implicitly **None**. Returning a true-value indicates that any such exception should be suppressed.

```
class Customers( object ):
  def __init__( self, dbname ):
    self.dbname = dbname

  def __enter__( self ):
    self.db = sqlite3.connect( self.dbname )
    return self

  def __exit__( self, exc, val, trace ):
    if exc:
      self.db.rollback()
    else:
      self.db.commit()
    return None

  def close( self ):
    self.db.close()

  def addCustomerOrder( self, customer, order ):
    self.db.execute( 'INSERT OR REPLACE \
      INTO customers (id, name) \
      VAlUES (?, ?)', customer )
    self.db.execute( 'INSERT INTO \
      orders (date, custid, itemid, qty) \
      VALUES (?, ?, ?, ?)', order )

  # Other methods...

with Customers( dbname ) as db:
  db.addCustomerOrder( customer, order )
db.close()
```

Listing 5

Consistent convenience

Having to explicitly close the connection *after* the block has exited is a bit of a wart. We could decide that our own implementation of the **__exit__()** method invokes **close()** on the connection object having either committed or rolled back the changes, but there is a better way.

The **contextlib** module in the Python Standard Library provides some convenient utilities to help with exactly this, including the **closing** function, used like this:

```
from contextlib import closing
with closing( Customers( dbname ) ) as db:
  db.addCustomerOrder( customer, order )
```

It will automatically call **close()** on the object to which it's bound when the block scope is exited.

Python File objects also have a Context Manager interface, and can be used in a **with** statement too. However, *their* behaviour on exit *is* to close the file, so you don't need to use the closing utility for file objects in Python.

It's a little odd having to know the internal behaviour of a given type's Context Manager implementation, but sometimes the price of convenience is a little loss of consistency

Object vs. Resource Lifetime

These two concepts are frequently, and mistakenly, used interchangeably. The lifetime of an object is the time between its creation and its destruction, which is usually the point at which its memory is freed. The lifetime of the resource is tied to neither of those things, although it's very often sensible to associate the object with its resource when the object is created (i.e. in its `__init__()` method). You cannot know, for all intents and purposes, when the *object* lifetime ends, but you *can* know – and can control – when the *resource* lifetime ends. Python's Context Manager types and the associated `with` statement give you that control.

You may have heard that Python objects can have a destructor – the `__del__()` method. This special method is called when the object is garbage collected, and it allows you to perform a limited amount of last-chance cleanup. A common misapprehension is that invoking del thing will call the `__del__()` method on `thing` if it's defined. It won't.

```
with open( filename ) as f:
  contents = f.read()
```

So much for consistency! It's a little odd having to know the internal behaviour of a given type's Context Manager implementation (and the documentation isn't *always* clear on which types in the Standard Library *are* Context Managers), but sometimes the price of convenience is a little loss of consistency.

To reiterate the point about lifetime, even though the connection and file objects in the previous two examples have been closed, the lifetimes of the *objects* has not been affected.

When one isn't enough

Sometimes it's useful to associate several resources with a single Context Manager block. Suppose we want to be able to import a load of customer order data from a file into the database using the facility we've already made.

In Python 3.1 and later, this can be achieved like this:

```
with closing( Customers( dbname ) ) as db,  \
  open( 'orders.csv' ) as data:
  for line in data:
    db.addCustomerOrder( parseOrderData( line ) )
```

If you're stuck using a version of Python earlier than that, you have to nest the blocks like this:

```
with closing( Customers( dbname ) ) as db:
  with open( 'orders.csv' ) as data:
    for line in data:
      db.addCustomerOrder( parseOrderData( line ) )
```

Either syntax gets unwieldy very quickly with more than two or three managed objects. One approach to this is to create a new type that implements the Context Manager protocol, and wraps up multiple resources, leaving the calling code with a single with statement on the wrapping type, as shown in Listing 6.

```
class WrappedResources( object ):
  def __init__( self, dbname, filename ):
    self.dbname = dbname
    self.filename = filename

  def __enter__( self ):
    self.db = sqlite3.connect( self.dbname )
    self.data = open( self.filename )

  def __exit__( self, *exceptions ):
    if not any( exceptions ): self.db.commit()

  def close( self ):
    self.data.close()
    self.db.close()

  def addCustomerOrder( customer, order ):
    pass # do the right thing here

with closing( WrappedResource( dbname, fname ) ) \
  as res:
  for line in res.data:
    res.addCustomerOrder( parseOrderData( line ) )
```

Listing 6

That really is a little clunky, however you look at it, since it's fairly obvious that the class has multiple responsibilities, and exposes the managed objects publicly, amongst other things. There are better ways to achieve this, and we will return to this shortly.

Common cause

Having implemented a (basic) facility to import data from a file to our database, we might like to extend the idea and optionally read from the standard input stream. A simple protocol for this might be to read `sys.stdin` if no filename is given, leading to code like this:

```
with options.filename and \
  open( options.filename ) or sys.stdin as input:
  # do something with the data
```

That's all very well, but is a little arcane, and *closing* the standard input handle when it completes might be considered bad manners. You could go to all the bother of reinstating the standard input handle, or redirecting it some other way, but that too seems more complicated than what is required.

Python's `contextlib` module has another handy utility to allow you to use a generator function as a Context Manager, without going to the trouble of creating a custom class to implement the protocol. It is used to **decorate** a function, which must `yield` exactly one value to be bound to the `as` clause of a `with` statement. Actions to perform when the block is entered are put before the `yield`, actions to perform when the block is

Exception safety facilities like the Python Context Manager are common to many languages that feature the use of exceptions to indicate errors

```
import contextlib

@contextlib.contextmanager
def simpleContext():
  doPreActionsHere()          (1)
  yield managed_object
  doPostActionsHere()         (2)
```

Listing 7

exited are put after the **yield**. It follows the basic pattern shown in Listing 7:

(1) will be called when the **with** statement is entered. It's the equivalent of the **__enter__()** method

(2) will be called when the block is exited. It's the equivalent of the **__exit__()** method

This allows us to define a couple of factory functions for our inputs, as shown in Listing 8.

Since opening a 'real' file returns an object that is already a Context Manager, the function for that isn't decorated. Likewise, since we do *not* want to perform any action on the **sys.stdin** object on exit, that function has no behaviour after the **yield**.

It should be clear that the Context Manager protocol is more general purpose than just for performing some clean-up action when leaving a scope. Exception safety is the primary purpose of the Context Managers, but the **__enter__()** and **__exit__()** methods can contain any arbitrary behaviour, just as the decorated function can perform any actions before and after the **yield** statement. Examples include tracking function entry and exit, and logging contexts such as those Chris Oldwood shows in C# [Oldwood].

Many and varied

As previously mentioned, it's sometimes necessary to manage multiple resources within a single block. Python 3.1 and later support this by

```
import contextlib

def openFilename():
  return open( options.filename )

@contextlib.contextmanager
def openStdIn():
  yield sys.stdin

opener = options.filename and openFilename \
  or openStdIn
with opener() as f:
  pass    # Use f
```

Listing 8

```
with contextlib.ExitStack() as stack:
  f = stack.enter_context( open( \
    options.filename ) )
  db = stack.enter_context( sqlite3.connect( \
    options.dbname ) )
```

Listing 9

allowing multiple Context Manager objects to be declared in a single **with** statement, but this becomes cluttered and unmanageable quickly. You can, as we demonstrated, create your own Context Manager type, but that too can be less than ideal. Once again, Python 3.3 answers the question with another **contextlib** utility, the **ExitStack**.

It manages multiple Context Manager objects, and allows you to declare them in a tidy (and indentation-saving) manner. See Listing 9.

Objects have their **__exit__()** method called, in the reverse order to which they were added, when the block is exited.

The **ExitStack** can manage a runtime-defined collection of context managers, such as this example taken directly from the Python 3.4 documentation [Python]:

```
with ExitStack() as stack:
  files = [ stack.enter_context( open( fname ) ) \
  for fname in filenames ]
  # All opened files will automatically be closed
  # at the end of the with statement, even if
  # attempts to open files later in the list raise
  # an exception
```

Conclusion

Python's Context Managers are a convenient and easy-to-use way of managing Resource Lifetimes, but their utility goes beyond that, due to the flexible way they are provided. The basic idea is not a new one – even in Python, where it was first introduced in version 2.5 – but some of these facilities are only available in later versions of the language. The examples given here were tested using Python 3.4.

Exception safety facilities like the Python Context Manager are common to many languages that feature the use of exceptions to indicate errors, because this introduces the need for some local clean-up in the presence of what is (in effect) a non-local jump in the code. They are, however, useful for things beyond this need, and Python provides several useful utilities to help manage the complexity this brings. ■

References

[Python] Python 3.4 Documentation. https://docs.python.org/3.4/library/contextlib.html

[Oldwood] Oldwood, Chris. Causality, http://chrisoldwood.com/articles/causality.html

Deterministic Components for Distributed Systems

Non-deterministic data leads to unstable tests. Sergey Ignatchenko considers when this happens and how to avoid it.

Over the last 10 years, significant improvements have been made in program testing. I don't want to go into a lengthy argument whether TDD is good or bad here – this is not the point of this article. However, one thing I want to note is a rather obvious (so obvious that it is often taken as granted) point that

> *For test results to have any meaning, the test needs to be reproducible*

In theory, we can speak of 'statistical testing' when the program is run for 1000 times (and if it fails not more than *x* times out of these 1000 runs, it is considered fine), but for most of the programs out there it would be a Really Bad Way to test them.

So far so obvious, but what does this really mean in practice? Let's consider the relationship between two subtly different properties: the 'test being reproducible' and the 'program being deterministic' (defined as 'the program has all its outputs completely defined by its inputs'). For the time being, we'll refrain from answering the question 'what qualifies as an input'; we'll come to discussing it a bit later.

First, let's note that for a 100% deterministic program, all the tests are always reproducible. In other words, a program being deterministic is *sufficient* to make all the tests against this program reproducible. On the other hand, if all the possible tests for our program are always reproducible, it means that our program is deterministic.

However, there are tests out there which are reproducible even when the program is not entirely deterministic. For example, if our program X prints the current time when we specify the `-t` option, and it emits concatenation of the input string with 'beyond any repair' otherwise, we can say (assuming that we don't consider *current time* as one of the program inputs) that:

- The program is not entirely deterministic
- the test with the `-t` option is not reproducible
- all the other tests are reproducible

Let's see what we have observed up to now:

- for deterministic programs, ***all*** the tests are reproducible
- for non-deterministic programs, there can be both reproducible tests and non-reproducible ones

The second statement can be seen as an indication that all is not lost – we ***can*** have meaningful tests for non-deterministic programs. And this does stand; on the other hand, a much more unpleasant statement stands too:

> *For a real-world non-deterministic program, we are bound to leave some functionality untested*

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

Let's see this on an example of our program X. We certainly can have a set of reproducible tests over our program – the ones which don't use the `-t` option. However, this will leave us with a significant part of our functionality untested.

To avoid this, we can try to test our program with the `-t` option to emit the right format, but wait – without meddling with the system time we won't even be able to test it on Feb 29th etc. Ok, we can add manipulating system time to our test, but even then we'll be testing only the time format (and not correctness of the time which was printed). We may try to measure *current time* before calling our program, and to compare this value with that printed by our program X; it might seem a good solution, but apparently, even if the program prints time with minute precision and runs for only 0.1 second, *from time to time* the test will fail. More specifically, such a test will fail whenever this 0.1 second (between the moment when we've measured *current time* and the moment our program has made its own measurement) happens to occur exactly across the boundary between two different minutes. Ok, we could add tolerance (which BTW already makes our test imprecise) and consider the program valid if the printed output *printed_time* satisfies an equation *measured_time* < *printed_time* < *measured_time + 0.1s*, but even this is not sufficient, as we haven't accounted for potential system time adjustments (including automated ones).

As we can see, even for an absolutely trivial non-deterministic program, accurate testing becomes a very non-trivial task. In practice, for a non-trivial program writing a set of tests which cover a significant part of the functionality quickly becomes a daunting task (or, more often, leads to significant functionality left untested). And if our program is traditionally multithreaded (defined as 'multithreaded with explicit thread sync using mutexes or equivalent') testing very quickly becomes pretty much pointless exactly because of the lack of determinism: the program which works perfectly on your test rig can easily fail on some other computer (or fail on the same computer from time to time) – just because whoever runs it was unlucky. Bummer.

Distributed programs and unit-tests: false sense of security

Let's come back from our so-far-theoretical clouds to the earth of programming. Let me tell you one real-world story in this regard (let's call it 'Unit Test Horror Story' for future reference).

Once upon a time, there was a company which successfully ran a multi-million-dollar online business. Moreover, they enjoyed very little unplanned downtime by their industry standards (1 hour per year or so). Then, on a nice sunny day (or a dark cloudy one – it won't change anything), a new developer came to the company. It just so happened that he was a strong proponent of TDD, so he started writing a unit-test for a new feature, and the test failed; he implemented the feature, and the test succeeded. And then the new developer read a lecture to all those non-TDD developers, saying 'Now you see how easy it is to write error-free programs!' Next day, his changes went to production and caused one of those once-per-year downtimes.

deterministic components can provide a **Holy Grail of production post-mortem**: an ability to **reproduce the bug exactly** as it has happened in production

The moral of this story is certainly ***not*** along the lines of 'see, TDD is useless' (IMHO, TDD, when taken in moderation, is quite useful, though it is far from being a silver bullet). The point here is to understand what has caused the downtime; and apparently, it was *an unusual sequence of otherwise perfectly valid messages* (sometimes such unusual sequences are referred to as 'races').

Moreover, from my experience, these

> *unusual sequences of otherwise perfectly valid messages are by far The Most Difficult To Find Bugs in pretty much any real-world distributed system.*

As a consequence (and combined with an observation that such unusual sequences tend to be among the most unexpected for the developer), it means that

> *Unit tests, while important, are insufficient to ensure the validity of a distributed system*

This happens because even if every component of your program is deterministic, when you make a distributed system out of these components, your distributed system (taken as a whole) is very unlikely to be deterministic. However, as we'll see below, even per-component determinism helps ***a lot*** for debugging such distributed systems. This includes things such as automated replay-based regression testing, low-latency fault-tolerance and relocation for components, and the holy grail of production post-mortem analysis.

Benefits of deterministic components

Let's discuss the major benefits of deterministic components in a distributed system.

Replay-based regression testing

As we've seen in the 'Unit Test Horror Story' above, unit tests are not sufficient to test components of the distributed system. On the other hand, if our component is entirely deterministic, we can do the following:

- while a previous version of the component is running in production, we can write down all the inputs for this specific component as an *input log*. Ideally, we should be able to do this for all the components in the system, but it can be done on per-component basis too.
- then, we can run the same input log against the new version of the component
- if our new version of the component doesn't have any *changes* to the logic (and has only *extensions* which are not activated until new inputs are used) – then the output ***should*** be exactly the same as for the previous version of the component
- while there is no strict guarantee that this kind of testing would help with the 'Unit Test Horror Story', the chances are that it would. Moreover, for heavily loaded systems, experience shows that the vast majority of those unusual sequences of otherwise valid inputs ***do*** happen every few hours (YMMV, batteries not included). In any case, this kind of testing, while not being a guarantee against bugs (nothing is), is a very valuable addition to the arsenal of regression tests which can and should be run.

Of course, you've already noticed the weak point of this kind of testing: as noted above, it will work only as long as you have made *no changes to existing functionality*. This indeed is one of the reasons why this kind of testing is not a silver bullet. However, two things help in this regard:

- most of the time, for a real-world system, functionality is *added* rather than modified
- provided that you make rather small and functionally oriented git-style independent commits (as opposed to one commit for everything which happened this week), it is often still possible to separate those changes which are ***not*** supposed to change any existing behavior, to re-apply these changes on top of the previous version, and to run replay-based regression tests against these changes. While it is admittedly an imperfect way of testing, it still does help quite a bit in practice.

Deterministic production post-mortem

However thoroughly we test our systems, from time to time they do fail ☹. In such cases, it is of utmost importance to identify the problem ASAP, and to fix it, so it doesn't happen again.

In this regard, deterministic components can provide a Holy Grail of production post-mortem: an ability to reproduce the bug *exactly* as it has happened in production. If our component is deterministic, then:

- we're able to write all the inputs of the component into an *input log*
- after the program fails in production, we can get the *input log* and run it in the comfort of a developer's machine, under a debugger, as many times as we want, and get exactly the same variables at exactly the same points as happened in production. Note that, strictly speaking, to get exactly the same variable values, we'll need each line of our component to be deterministic, which is a stronger property than the component being deterministic as a whole. However, in practice the latter is normally achieved via the former, so we can ignore the difference between the two for most practical purposes.

In reality, of course, it is not that simple. If our system runs for months, storing and replaying *all* the inputs which have happened during those months is rarely feasible. However, we can avoid this problem by running our *input log* in a circular manner.

As mentioned above, running *input log* for the entire history of our component rarely qualifies as a viable option. On the other hand, if our component is deterministic, *and* we can get *current state* of our component in some kind of serialized form, then we can build our *input log* as follows:

- *input log* is written in a circular manner, so that it stores only the last *t* seconds of the component inputs
- on each wrap-around of our circular storage, we store the *current state* of our component in the same *input log*

if all our Reactor does is calculate a pure function, it stays deterministic just by its very nature

- if we need to perform a post-mortem, we use a deserialized *current state* to initialize our deterministic component and then reapply the remaining inputs from the *input log* to get a complete picture of the last *t* seconds of the component's life before the crash

Such circular *input logs* are highly practical at least in some deployment scenarios, and provide an ability to see the last *t* seconds of life of a component before it failed. While it might be that the problem occurred before these last *t* seconds, and was only manifested later, this technique still happens to be quite useful in practice.

Low-latency fault tolerance and component relocation

Yet another major benefit of having your components both deterministic and serializable is that it is possible to gain some fault tolerance from them (without any help from application level code). In particular, the following schema will work:

- circular *input log* runs on a physical server X, and our deterministic component runs on a physical server Y
- all the outputs from our deterministic component are sequentially numbered, and go through physical server X, where the last number of the output is kept as a variable `LastOutputNumber`

Then you can recover from any single server failure in a perfectly transparent manner:

- if server X has failed, then server Y has all the information we need, and replacement server X' can start writing *input log* from scratch (starting from storing serialized *current state*)
- if server Y has failed, then we can follow the following procedure:
  - start replacement server Y' with a replacement instance of our deterministic component (initialized with serialized *current state* from *input log*)
  - variable `LastSkipNumber` is set to `LastOutputNumber`
  - all the records in *input log* after serialized *current state* are replayed
  - during replay, all the outputs of our deterministic component are skipped until we reach `LastSkipNumber`. This is possible because all the outputs are deterministic, and they exactly repeat those outputs which have already been sent back to the other components/clients/whoever-else

This kind of determinism-based fault tolerance provides fault-tolerance with very little additional latency. In fact, it is ideologically similar to the *virtual lockstep* way of achieving fault tolerance (and provides significantly better latency than so-called *fast checkpoints*).

In a somewhat similar manner, it is possible to achieve low-latency relocation of the components from one physical server to another one. The idea revolves around running two instances of the component for some time to reduce latency during serialization/deserialization/state-transfer. The second instance of the component would have its outputs skipped until it has caught up with the first one, and then the first one can be dropped.

Making components deterministic

Ok, I hope that by now I've managed to convince you that determinism is a Good Thing™. Now, let's discuss how to write those deterministic components.

Obvious stuff

First of all, let's note that there are obvious things to keep in mind when aiming for determinism. In particular:

- ***don't*** use any uninitialized memory in your program
- more generally, ***don't*** use anything which causes the dreaded 'undefined behavior' in C/C++
- ***don't*** use pointers for any operations except for dereferencing.
  - In particular, using pointer values (as opposed to data pointed to by pointers) as a key for sorting is especially dreadful (as usually allocators are not required to be deterministic, especially in a multithreaded environment, so such sorting can easily lead to non-deterministic results)

NB: for the purposes of this article, we won't aim for cross-platform determinism; while cross-platform determinism is an interesting beast and has its additional benefits, for now we will set it aside to avoid complications. This will save us from quite a few problems, such as iterations over unordered collections and even more nasty different-order-of-floating-point-operations stuff.

Relation to Reactors

Now, let's note that deterministic components tend to go very well alongside with the Reactor event-driven pattern (as described in [Wikipedia]). This more or less includes such things as Erlang message passing, Node.js, and to certain extent – Akka Actors.

In summary, the Reactor pattern takes incoming inputs (a.k.a. 'service requests' or 'events') and processes them synchronously, one by one. It also serves as a very convenient point to write all these incoming events into our *input log*.

So far so good, and if all our Reactor does is calculate a pure function, it stays deterministic just by its very nature. However, as soon as you do as little as call `get_current_time()` within your component, it ceases to be deterministic ☹. Therefore, we need to discuss ways how to deal with non-determinism.

While the determinism-assuring techniques described below are, strictly speaking, not limited to Reactor and Reactor-like patterns, it is easier to describe some of them in terms of Reactors (and they're likely to be used in the context of Reactors too).

Considering call output as a program input

As we've seen above, even a very simple program which prints current time is not deterministic. However, there is a trick which allows to make it deterministic. More specifically,

all we need to do to make our program deterministic, is to replace a call to a system-level function with a call to our own function

If we make current time an input of our program, related non-determinism will go away

In other words, all we need to do to make our program deterministic, is to replace a call to a system-level function `get_current_time()` with a call to our own function `get_our_own_current_time()`, where `get_our_own_current_time()` goes along the following lines:

```
time_t get_our_own_current_time() {
  switch( deterministic_mode ) {
    case DETERMINISTIC_REPLAY:
      return read_time_t_from_input_log();
    case DETERMINISTIC_RECORDING:
      time_t ret = get_current_time();
        //NON-DETERMINISTIC!
      write_time_t_to_input_log(ret);
        //wrote to input-log
        //to ensure determinism
      return ret;
    case DETERMINISTIC_OFF:
      return get_current_time();
      //NON-DETERMINISTIC!
  }
}
```

Bingo! We can have our determinism and eat it too!

This trick of declaring something non-deterministic as an input and writing it into *input log* is pretty much universal in the sense that in theory it can be applied to pretty much any non-deterministic function call including, but not limited to, reading of real random data from /dev/urandom. Even mutexes can be handled this way (though in this case *all* the data protected by mutex needs to be written to the *input log*, ouch). On the other hand, in practice there are two significant caveats:

- when large data chunks are read, it is often infeasible to handle them this way (if you read 1Gbyte from your file/DB, throwing it into *input log* is rarely a good idea); however, there are many practical cases when it can be avoided:
  - if the data is expected to be constant (like 'read from a constant file') – then there is no need to record it to the *input log* (as it can be reproduced); in extreme cases, if you suspect that data potentially can be corrupted, you can write some kind of hash into *input log* instead of the data itself
  - if the data is some kind of cache (which can be re-obtained from the authoritative source instead) – it doesn't need to be logged either.
  - If we can say that data on disk is a part of our *current state*, then there is no need to log such accesses either. Note though that this option would usually make serializing your *current state* significantly more expensive
- For frequently called functions such as obtaining *current time*, using this trick makes replay more fragile than it is necessary. For example, if you use this trick and then add another call to `get_current_time()` somewhere within your implementation, it makes your *input logs* incompatible (and usually this is not intended)

This trick doesn't go well with fault-tolerant implementations (which need to write all the inputs to a separate physical box)

Guessing game

A different technique to ensure determinism can be described as follows. If we know in advance what will be necessary for the processing of our input event, then we can supply it to our component (logging it to *input log* in advance, so that it is no longer a problem for fault-tolerant implementations). In particular, as ***lots*** of input events need *current time*, we can say that we'll provide it to all of them. In this case:

- Code outside of the Reactor will call the system level `get_current_time()` before processing each input event.
- Code outside of the Reactor will store the time read (say, to TLS variable `event_time`)

  Note that global/per-process singleton won't be able to ensure determinism if you have more than one thread running your deterministic objects within your process.
- It will call Reactor's `process_event()` or equivalent

Reactor app-level code still needs to call:

```
get_our_own_current_time()
```

instead of:

```
get_current_time()
```

but implementation of `get_our_own_current_time()` becomes much simpler in this case:

```
thread_local time_t event_time;
  //event_time is pre-populated by caller
time_t get_our_own_current_time() {
  return event_time;
}
```

Note that with this implementation, it is ***not*** possible to measure execution times *within* the event handler (as all the calls to `get_current_time()` for the same event will return the same value). On the other hand, as any kind of execution times measurements would make our program inherently non-deterministic (at the very least when we're running it on a modern CPU), it is not ***that*** big deal. And if you need to make some performance analysis, you still can use something along the lines of Node.js-style `console.time()/console.timeEnd();` as these functions do ***not*** return the measured time interval value to the program, but rather print it to a log file – then, as long as we do ***not*** consider log file as one of program outputs (and the program itself doesn't read these values from the log file), we're fine from determinism point of view.

Unfortunately, not all the required inputs can be pre-guessed successfully. However, in quite a few cases, the following technique does help:

- We're starting to `process_event()` with or without the data that may be needed

If we know in advance what will be necessary for the processing of our input event, then we can supply it to our component

- If the data is needed but is not provided, we throw a special exception requesting the data from the caller
  - This exception ***must*** be thrown before any changes to Reactor's state are made. This fits very well into a typical Validate-Calculate-Modify Reactor pattern described in [ITHare16].
  - It is also important to discard all the outputs coming from the processing of this `process_event()` (or to avoid emitting them in the first place)
- If a caller receives this `ThisKindOfDataRequested` exception, it simply provides the requested data and repeats the same call

This exception-as-a-request-for-non-deterministic-data does help in quite a few scenarios. However, as with anything else, it is not a silver bullet ☹, and chances are that you will need to look for your own ways to ensure determinism.

Conclusions

We've discussed the impact of per-component determinism on programming. In particular, we've found several major benefits of making your components deterministic (from replay-based regression testing to determinism-based fault tolerance). Also, we've discussed some ways of achieving this holy grail of determinism; while techniques discussed here won't ensure determinism for all programs, they have been seen to allow determinism for quite a few real-world systems (ranging from stock exchanges to online games). ■

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

References

[ITHare16] 'No Bugs' Hare, 'Asynchronous Processing for Finite State Machines/Actors: from plain event processing to Futures (with OO and Lambda Call Pyramids in between)', http://ithare.com/asynchronous-processing-for-finite-state-machines-actors-from-plain-events-to-futures-with-oo-and-lambda-call-pyramids-in-between/

[Wikipedia] Reactor pattern https://en.wikipedia.org/wiki/Reactor_pattern

Programming Your Own Language in C++

Scripting languages allow dynamic features easily. Vassili Kaplan writes his own in C++ allowing keywords in any human language.

To iterate is human, to recurse divine.
~ L. Peter Deutsch

Why is yet another scripting language needed? Not only are there already quite a few scripting languages present but there are also a few frameworks to build new languages and parsers, notably ANTLR Framework [ANTLR] and the Boost Spirit library [Spirit]. Nevertheless, I see two main advantages in writing a language like the one presented in this article: the possibility of adding a new function, a control flow statement or a new data structure on the fly and to have keywords translated to any language with just a few configuration changes (or to add synonyms to existing keywords, so that, for instance, **`ls`** and **`dir`** could be used interchangeably). Also, as you will see from this article, the learning curve is much shorter to get started adding your own code in C++.

In *C Vu* [Kaplan15a, Kaplan15b] I published the split-and-merge algorithm to parse a string containing a mathematical expression. Later, in *MSDN Magazine* [Kaplan15c, Kaplan16] I implemented this algorithm in C# and showed how one can implement a scripting language in C# based on the split-and-merge algorithm.

In this article, I am going to extend the split-and-merge algorithm presented earlier and show how one can use it to write a scripting language in C++. In *MSDN Magazine* [Kaplan16] the scripting language was called CSCS (Customized Scripting in C#). For simplicity, I will continue calling the language I discuss here CSCS (Customized Scripting in C++ using the split-and-merge algorithm) as well.

The CSCS language, as described in *MSDN Magazine* [Kaplan16], was still not very mature. In particular, there is a section towards the end of the article mentioning some of the important features usually present in a programming language that CSCS was still missing. In this article I'll generalize the CSCS language and show how to implement most of those missing features and a few others in C++.

All the code discussed here is available for download [Downloads].

The split-and-merge algorithm to parse a language statement

Here we'll generalize the split-and-merge algorithm to parse not only a mathematical expression but any CSCS language statement. A separation character must separate all CSCS statements. It is defined in the `Constants.h` file as **`const char END_STATEMENT = ';'`**.

The algorithm consists of two steps.

In the first step, we split a given string into a list of objects, called 'Variables'. Each **`Variable`** consists of an intermediate result (a number, a string, or an array of other **`Variable`**s) and an 'action' that must be applied to this **`Variable`**. Previously we called this **`Variable`** a 'Cell' and it could hold only a numerical value.

The last element of the created list of **`Variable`**s has so called 'null action', which, for convenience, we denote by the character **`")"`**. It has the lowest priority of 0.

```
class Variable {
public:
  Variable(): type(Constants::NONE) {}
  Variable(double val): numValue(val),
    type(Constants::NUMBER) {}
  Variable(string str): strValue(str),
    type(Constants::STRING) {}

  string toString() const;
  bool canMergeWith(const Variable& right);
  void merge(const Variable& right);
  void mergeNumbers(const Variable& right);
  void mergeStrings(const Variable& right);

private:
  double numValue;
  string strValue;
  vector<Variable> tuple;
  string action;
  string varname;
  Constants::Type type;
};
```

Listing 1

For numbers, an action can be any of **`+`**, **`-`**, **`*`**, **`/`**, **`%`**, **`&&`**, **`||`**, and so on. For strings, only **`+`** (concatenation), and logical operators **`<`**, **`<=`**, **`>`**, **`>=`**, **`==`**, **`!=`** are defined.

Listing 1 contains an excerpt from the **`Variable`** class definition.

The separation criteria for splitting a string into a list of **`Variable`**s are: an action, an expression in parentheses, or a function (including a variable, which is also treated as a function), previously registered with the parser. We are going to talk how to register a function with the parser in the next section. In Listing 2 you can see all of the actions defined for numbers and their priorities.

In the case of an expression in parentheses or a function, we apply recursively the whole split-and-merge algorithm to that expression in parentheses or the function argument in order to get a **`Variable`** object as a result. At the end of the first step we are going to have a list of **`Variable`**s, each one having an action to be applied to the next **`Variable`** in the list. Thanks to the recursion there will be no functions left after step 1, just numbers, strings or lists of numbers, strings or lists of numbers, strings, … and so on recursively. We call these lists 'tuples'. Internally they are implemented as vectors (see Listing 1).

Vassili Kaplan has been a Software Developer for over 15 years, working in different countries and with different languages (including C++, C#, Python, and lately Objective-C. His latest baby is iLanguage app for iPhone). He has a Masters in Math from Purdue University. He currently resides in Switzerland and can be contacted at vassilik@gmail.com

have keywords translated to any language
with just a few configuration changes

```
unordered_map<string, int> prio;
prio["++"] = 10;
prio["--"] = 10;
prio["^"]  = 9;
prio["%"]  = 8;
prio["*"]  = 8;
prio["/"]  = 8;
prio["+"]  = 7;
prio["-"]  = 7;
prio["<"]  = 6;
prio[">"]  = 6;
prio["<="] = 6;
prio[">="] = 6;
prio["=="] = 5;
prio["!="] = 5;
prio["&&"] = 4;
prio["||"] = 3;
prio["+="] = 2;
prio["-="] = 2;
prio["*="] = 2;
prio["/="] = 2;
prio["%="] = 2;
prio["="]  = 2;
```

Listing 2

The data structure holding the string with the expression to parse and a pointer to the character currently being parsed is **`ParsingScript`**. An excerpt from its definition is shown in Listing 3.

The main parsing cycle of the first part of the algorithm is shown in Listing 4.

The second step consists in merging the list of **`Variable`**s created in the first step according to their priorities. Two **`Variable`** objects can be merged together only if the priority of the action of the **`Variable`** on the left is greater or equal than the priority of the action of the **`Variable`** on the right. If not, we jump to the **`Variable`** on the right and merge it with the **`Variable`** on its right first, and so on, recursively. As soon as the **`Variable`** on the right has been merged with the **`Variable`** next to it, we return back to the original **`Variable`**, the one we weren't able to merge before, and try to remerge it with the newly created **`Variable`** on its right. Note that eventually we will be able to merge the list since the last **`Variable`** in this list has a null action with zero priority.

Check out the implementation of the second step of the algorithm in Listing 5. The function **`merge()`** is called from outside with the **`mergeOneOnly`** parameter set to **`false`**. That parameter is set to **`true`** when the function calls itself recursively, because we need to merge the **`Variable`** on the right with the **`Variable`** on its right just once, and return back in order to remerge the current **`Variable`**.

Let's see an example of applying the split-and-merge algorithm to the following CSCS language statements:

```
class ParsingScript
{
public:
  ParsingScript(const string& d): data(d),
    from(0) {}
  inline char operator()(size_t i) const {
    return data[i]; }
  inline size_t size() const             {
    return data.size(); }
  inline bool stillValid() const         {
    return from < data.size(); }
  inline size_t getPointer() const       {
    return from; }
  inline char current() const            {
    return data[from]; }
  inline char currentAndForward()        {
    return data[from++]; }
  inline char tryNext() const            {
    return from+1 < data.size() ?
           data[from+1] : Constants::NULL_CHAR; }
  inline void forward(size_t delta  = 1) {
    from += delta; }
private:
  string data; // contains the whole script
  size_t from; // a pointer to the
               // script data above
};
```

Listing 3

```
a = "blah"; b = 10; c = a == "blue" || b == 1;
print(c);
```

We have four statements above, separated by the `;` character.

The parsing of the first two statements is analogous – as soon as the parser gets the **`=`** token, it will register the parameters on the left (**`a`** and **`b`**), mapping them to their corresponding values (**`"blah"`** and **`10`**). We will see how to register variables and functions with the parser in the next section (actually for the parser it doesn't matter whether it is a variable or a function – they are all treated the same). Note that there is no need to declare any variables – they are all deduced from the context.

More interesting is the third statement:

```
c = a == "blue" || b == 1;
```

When parsing the right part of this statement (after the assignment), as soon as the parser gets a function or a variable (this can be anything that is not in quotes and is not a number) it will try to find if there is already a corresponding variable or a function registered with the parser. If not, a **`ParsingException`** will be thrown, but in our case we have already defined **`a`** and **`b`** in the previous two statements, so their values will be substituted, transforming the right side of the statement above to:

```
"blah" == "blue" || 10 == 1;
```

there will be no need to remember that one must use dir, copy, move, findstr on Windows, but ls, cp, mv, and grep on Unix

```
do // Main processing cycle of the first part.
{
  char ch = script.currentAndForward ();
  // get the next character and move one forward

  bool keepCollecting = stillCollecting(script,
    parsingItem, to, action);
  if (keepCollecting)
  { // The char still belongs to the
    // previous operand.
    parsingItem += ch;
    // Parse until the next extracted character is
    // in the "to" string.
    bool goForMore = script.stillValid() &&
      !Utils::contains(to, script.current()));
    if (goForMore) {
      continue;
    }
  }
  // Done getting the next token. The getValue()
  // call below may recursively call this method if
  // extracted item is a function or is starting
  // with a '('.
  ParserFunction func(script, parsingItem, ch,
    action);
  Variable current = func.getValue(script);

  if (action.empty()) { // find the next "action"
                        // token or a null action '('
    action = updateAction(script, to);
  }

  char next = script.current(); // we've already
                                // moved forward
  bool done = listToMerge.empty() &&
    (next == END_STATEMENT ||
    (action == Constants::NULL_ACTION &&
     current.getType() != NUMBER);
  if (done) {
    // If there is no numerical result, we are not
    // in a math expression.
    listToMerge.push_back(current);
    return listToMerge;
  }
  current.action = action;
  listToMerge.push_back(current);
  parsingItem.clear();
} while (script.stillValid() &&
    !Utils::contains(to, script.current());
```

Listing 4

The first step of the split-and-merge algorithm produces the following list of **`Variable`**s from the above statements:

```
Variable Parser::merge(Variable& current,
  size_t& index, vector<Variable>& listToMerge,
  bool mergeOneOnly)
{
  while (index < listToMerge.size()) {
    Variable& next = listToMerge[index++];
    while (!current.canMergeWith(next)) {
      merge(next, index, listToMerge,
            true/*mergeOneOnly*/);
    }
    current.merge(next);
    if (mergeOneOnly) {
      break;
    }
  }
  return current;
}
```

Listing 5

```
Split("blah" == "blue" || 10 == 1) →
```

1. `Variable(strValue = "blah", action = "==")`
2. `Variable(strValue = "blue", action = "||")`
3. `Variable(numValue = 10, action = "==")`
4. `Variable(numValue = 1, action = ")")`

In the second part of the algorithm we merge the list of **`Variable`**s above. The **`Variable`**s 1 and 2 can be merged on the first pass since the priority of the `"=="` action is higher than the priority of the `"||"` action according to the Listing 2. The merging of **`Variable`**s 1 and 2 consists in applying the action `"=="` of the first variable to the values of both variables, leading to a new **`Variable`** having the action of the **`Variable`** on its right:

```
Merge(Variable(strValue = "blah", action = "=="),
Variable(strValue = "blue", action = "||")) =
Variable("blah" == "blue",
  action = "||") = Variable(numValue = 0,
  action = "||")
```

Next, we merge `Variable(numValue = 0, action = "||")` with the next **`Variable`** in the list, `Variable(numValue = 10, action = "==")`. But now the action `= "||"` has a lower priority than the action `= "=="`, therefore we need to merge first the `Variable(numValue = 10, action = "==")` with the next `Variable(numValue = 1, action = ")")`. Since the null action `")"` has the lowest priority, the merge can be done, producing a new **`Variable`**:

```
Merge(Variable(numValue = 10,
  action = "=="),Variable(numValue = 1,
  action = ")")) =
Variable(10 == 1,
  action = ")") = Variable(numValue = 0,
  action = ")")
```

Since this merge was successful we must now get back to the `Variable(numValue = 0, action = "||")` and remerge it with the newly created `Variable(numValue = 0, action = ")")`, producing:

```
Merge(Variable(numValue = 0,
  action = "||"),Variable(numValue = 0,
  action = ")")) =
Variable(0 || 0,
  action = ")") = Variable(numValue = 0,
  action = ")")
```

Since there are no more `Variable`s left in the list, the result of merging is 0. This value will be assigned to the variable `"c"` and registered with the parser.

The last statement is `"print(c);"`. After extracting the token `print` the parser will look if there is a function named `print` already registered with the parser. Since there is one, the parser will recursively call the whole split-and-merge algorithm on the argument of the `print()` function, `"c"`. Since `"c"` was registered with the parser in the previous step, the parser will return back its value, 0, to the print function. Let's see more closely how variables and functions can be implemented and registered with the parser taking as an example the `print()` function.

Registering functions and variables with the parser

All the functions that can be added to the parser must derive from the `ParserFunction` class.

The `Identity` is a special function which is called when we have an argument in parentheses. It just calls the main entry method of the split-and-merge algorithm to load the whole expression in parentheses:

```
class IdentityFunction : public ParserFunction
{
public:
  virtual Variable evaluate(ParsingScript& script)
  {
    return Parser::loadAndCalculate(script);
  }
};
```

All split-and-merge functions and variables are implemented similarly.

There are three basic steps to register a function with the parser:

- Define a function keyword token, i.e. the name of the function in the scripting language, CSCS, e.g.:

```
static const string PRINT; // in Constants.h
const string Constants::PRINT = "print";
                              // in Constants.cpp
```

- Implement the class to be mapped to the keyword from the previous step. Basically just the `evaluate()` method must be overridden. E.g. for the `print()` function:

```
Variable
  PrintFunction::evaluate(ParsingScript&
    script)
{
  vector<Variable> args =
  Utils::getArgs(script, Constants::START_ARG,
  Constants::END_ARG);
  for (size_t i = 0; i < args.size(); i++) {
    cout << args[i].toString();
  }
  cout << endl;
  return Variable::emptyInstance;
}
```

- Map an object of the class implemented in the previous step with the previously defined keyword as follows:

```
ParserFunction::addGlobalFunction
      (Constants::PRINT, new  PrintFunction());
```

`Utils::getArgs()` auxiliary function parses the arguments of the print function and first gets a list of strings, separated by commas from `print(string1, string2, ..., stringN)`. Then it recursively applies the split-and-merge algorithm to each of these `string1, string2, ..., stringN`. Finally, it prints them out to the terminal.

The `addGlobalFunction()` method (see Listing 6) just adds a new entry to the global dictionary `s_functions` (implemented as an `unordered_map`) used by the parser to map keywords to functions.

```
void ParserFunction::addGlobalFunction
  (const string& name, ParserFunction* function)
{
  auto tryInsert =
    s_functions.insert({name, function});
  if (!tryInsert.second) {
    // The variable or function already exists.
    // Delete it and replace with the new one.
    delete tryInsert.first->second;
    tryInsert.first->second = function;
  }
}
```

As we've mentioned before, we treat a variable as a function: as soon as the parser gets an expression like `"a = something"`, it will register a function with the keyword `"a"` and map it to the `Variable`

```
Variable IfStatement::evaluate(ParsingScript&
script)
{
  size_t startIfCondition = script.getPointer();

  Variable result = Parser::loadAndCalculate
    (script, Constants::END_ARG_STR);
  bool isTrue = result.numValue != 0;

  if (isTrue) {
    result = processBlock(script);

    if (result.type ==
          Constants::BREAK_STATEMENT ||
        result.type ==
          Constants::CONTINUE_STATEMENT) {
      // Got here from the middle of the if-block.
      // Skip it.
        script.setPointer(startIfCondition);
      skipBlock(script);
    }
    skipRestBlocks(script);
    return result;
  }

  // We are in Else. Skip everything in the
  // If statement.
  skipBlock(script);

  ParsingScript nextData(script.getData(),
    script.getPointer());
  string nextToken =
    Utils::getNextToken(nextData);

  if (Constants::ELSE_IF_LIST.find(nextToken) !=
      Constants::ELSE_IF_LIST.end()) {
    script.setPointer(nextData.getPointer() + 1);
    result = processIf(script);
  }
  if (Constants::ELSE_LIST.find(nextToken) !=
      Constants::ELSE_LIST.end()) {
    script.setPointer(nextData.getPointer() + 1);
    result = processBlock(script);
  }
  return Variable::emptyInstance;
}
```

Listing 6

`"something"` (which will be calculated applying recursively the split-and-merge algorithm). In C++ the code for this is:

```
ParserFunction::addGlobalFunction("a",
  something /*Variable*/);
```

Implementing the if – else if – else control flow statements

Let's see how to implement the `if()` – `else if()` – `else()` functionality in CSCS.

The first and the third steps are clear: define the `if` constant and register a class implementing the `if` control flow statement with the parser, the same way we registered the `print()` function above.

The second step, the implementation of the `if` statement, is shown in Listing 6.

First we evaluate the condition inside of `if()` by recursively calling the split-and-merge algorithm on that condition. If the condition is true we process the whole `if()` block, recursively calling the split-and-merge algorithm on each statement inside of the `processBlock()` method. If the condition is false we first skip the whole `if()` block in the `skipBlock()` method. Then we evaluate the `else()` and `else if()` statements. The evaluation of `else if()` is basically same as the evaluation of the `if()` statement itself, so for `else if()` we recursively call the `if()` statement evaluation.

Note that we enhanced the execution of the `if`-statement here – as soon as there is a `break` or a `continue` statement, we get out of the `if ()` block – same way we get out from the `while()` block. This can be useful in case of nested `if`s.

Similarly, we can register any function with the parser, e.g. `while()`, `for()`, `try()`, `throw()`, `include()`, etc. We can also define local or global variables in the same way. In the next section we are going to see how to define functions in CSCS and add passed arguments as local variables to CSCS.

Implementing custom functions in CSCS

To write a custom function in the scripting language, let's introduce two functions in C++, `FunctionCreator` and `CustomFunction`, both deriving from the `ParserFunction` base class. A `FunctionCreator` object is registered with the parser in the same way we registered `if()` and `print()` functions above.

As soon as the parser gets a token with the `"function"` keyword, an instance of the `FunctionCreator` will be executed, namely, its `evaluate()` method, see Listing 7.

Basically, it just creates a new object, an instance of the `CustomFunction`, and initializes it with the extracted function body and the list of parameters. It also registers the name of the custom function with the parser, so the parser maps that name with the new `CustomFunction` object which will be called as soon as the parser encounters the function name keyword.

```
Variable FunctionCreator::evaluate(ParsingScript&
    script) {
  string funcName = Utils::getToken(script,
    TOKEN_SEPARATION);
  vector<string> args =
    Utils::getFunctionSignature(script);
  string body = Utils::getBody(script, '{', '}');

  CustomFunction* custFunc =
    new CustomFunction(funcName, body, args);
  ParserFunction::addGlobalFunction(funcName,
    custFunc);

  return Variable(funcName);
}
```

Listing 7

```
Variable CustomFunction::evaluate(ParsingScript&
script)
{
  // 0. Extract function arguments.
  vector<Variable> args = Utils::getArgs(script);
  // 1. Add passed arguments as local variables.
  StackLevel stackLevel(m_name);
  for (size_t i = 0; i < m_args.size(); i++) {
    stackLevel.variables[m_args[i]] = args[i];
  }
  ParserFunction::addLocalVariables(stackLevel);
  // 2. Execute the body of the function.
  Variable result;
  ParsingScript funcScript(m_body);
  while (funcScript.getPointer() <
     funcScript.size()-1 && !result.isReturn) {
    result =
       Parser::loadAndCalculate(funcScript);
    Utils::goToNextStatement(funcScript);
  }
  // 3. Return the last result of the execution.
  ParserFunction::popLocalVariables();
  return result;
}
```

Listing 8

So all of the functions that we implement in the CSCS code correspond to different instances of the `CustomFunction` class. The custom function does primarily two things, see Listing 8. First, it extracts the function arguments and adds them as local variables to the parser (they will be removed from the parser as soon as the function execution is finished or an exception is thrown). It also checks that the number of actual parameters is equal to the number of the registered ones (this part is skipped for brevity).

Second, the body of the function is evaluated, using the main parser entry point, the `loadAndCalculate()` method.

If the function body contains calls to other functions, or to itself, the calls to the `CustomFunction` can be recursive.

Let's see this with an example in CSCS. It calculates recursively the Fibonacci numbers, see Listing 9.

```
cache["fib"] = 0;
function fibonacci(n) {
  if (!isInteger(n)) {
    exc = "Fibonacci is for integers only"
      "(n="+ n +")";
    throw (exc);
  }
  if (n < 0) {
    exc = "Negative number (n="+ n +") supplied";
    throw (exc);
  }
  if (n <= 1) {
    return n;
  }
  if (contains(cache["fib"], n)) {
    result = cache["fib"][n];
    print("  found in cache fib(", n, ")=",
      result);
    return result;
  }
  result = fibonacci(n - 2) + fibonacci(n - 1);
  cache["fib"][n] = result;
  return result;
}
```

Listing 9

```
Calculating Fibonacci(10)...
  found in cache fib(2)=1
  found in cache fib(3)=2
  found in cache fib(4)=3
  found in cache fib(5)=5
  found in cache fib(6)=8
  found in cache fib(7)=13
  found in cache fib(8)=21
Fibonacci(10)=55

Calculating Fibonacci(-10)...
Caught: Negative number (n=-10) supplied at
  fibonacci()

Calculating Fibonacci(1.500000)...
Caught: Fibonacci is for integers only
(n=1.500000) at
  fibonacci()
```

Figure 1

The Fibonacci function above uses an auxiliary **isInteger()** function, also implemented in CSCS:

```
function isInteger(candidate) {
  return candidate == round(candidate);
}
```

The **isInteger()** function calls yet another, **round()** function. The implementation of the **round()** function is already in the C++ code and is analogous to the implementation of the **print()** function that we saw in the previous section.

To execute the Fibonacci function with different arguments we can use the following CSCS code:

```
n = ...;
print("Calculating Fibonacci(", n, ")...");
try {
  f = fibonacci(n);
  print("fibonacci(", n, ")=", f);
} catch(exc) {
  print("Caught: " + exc);
}
```

We get the output in Figure 1 for different values of **n**.

Since the exceptions happened at the global level, the exception stacks printed consisted only of the **fibonacci()** function itself. To keep track of the execution stack, i.e. CSCS functions being called, internally we use a C++ stack data structure, where we add every executing **CustomFunction** object as soon as we start function execution and remove it as soon as the execution is over. In case of an exception we just print out the contents of this stack. Then we clear the execution stack up to the level where the exception was caught. The implementation of the execution stack and of the **try()** and **throw()** functions can be consulted in [Downloads].

The implementation of the Fibonacci function in Listing 9 uses caching of already calculated Fibonacci numbers by using dictionaries – we will see how one can implement dictionaries next.

Arrays and other data structures

To declare an array and initialize it with some data we use the same CSCS language statement. The array elements for the initialization are declared between the curly braces. Here is an example in CSCS:

```
a = 10;
arr = {++a-a--, ++a*exp(0)/a--, -2*(--a - ++a)};
i = 0;
while(i < size(arr)) {
  print("a[", i, "]=", arr[i], ", expecting ",
    i);
  i++;
}
```

The number of elements in the array is not explicitly declared since it can be deduced from the assignment.

The function **size()** is implemented in C++. It returns the number of elements in an array. In case the passed argument is not an array, it will return the number of characters in it.

Internally an array is implemented as a vector, so you can add elements to it on the fly. In CSCS we access elements of an array, or modify them, by using the squared brackets. As soon as the parser gets a token containing an opening squared bracket, it knows that it is for an array index, so it applies recursively the whole split-and-merge algorithm to the string between the squared brackets to extract the index value. There can be unlimited number of dimensions of an array (well, limited by the system resources) because the array is implemented as a **vector<Variable>**: the **Variable** class has a member called **"tuple"** and declared as **vector<Variable>**. For instance, accessing **a[i][j][k]** in the CSCS code means **a.tuple[i].tuple[j].tuple[k]** in the C++ underlying code (**"a"** is a **Variable**, see Listing 1 for **Variable** definition).

In other words, for each consequent index in squared brackets the parser will create a new **Variable** of type array or use an existing **"tuple"** member.

If we access an element of an array and that element has not been initialized yet, an exception will be thrown by the parser. However, it's possible to assign a value to just one element of an array, even if the index used is greater than the number of elements in the array and even if the array has not been initialized yet. In this case the non-existing elements of the array will be initialized with the empty values. The CSCS code below is legal, even if the array has not been initialized before:

```
i = 10;
while(--i > 0) {
  array[i] = 2*i;
}
print("array[9]=", array[9]);       // prints 18
print("size(array)=", size(array)); // prints 10
```

We can also add other data structures to the CSCS language. Let's see an example of adding a dictionary, implemented internally as an **unordered_map**. We add the following member to the **Variable** class:

```
unordered_map<string, size_t> dictionary;
```

This is the mapping between a dictionary key and an index of already existing member **vector<Variable>** tuple, where the actual dictionary value will be stored. So every time a new key-value pair is added to the dictionary the following code is executed:

```
tuple.emplace_back(var);
dictionary[key] = tuple.size() - 1;
```

Every time an existing key is accessed the following code is executed (a check for existence is skipped):

```
auto it = dictionary.find(key);
size_t ptr = it->second;
return tuple[ptr];
```

With a few changes one can use not only strings, but anything else as a dictionary key. Similarly, we can add other data structures to CSCS – as long as a data structure exists, or can be implemented in C++, it can be added to CSCS as well.

In Listing 6 we saw the implementation of the **if()-else if()-else** control flow statements. Towards the end of the listing you might have scratched your head, asking why we didn't compare the extracted token with the **"else"** string, but we did a comparison with the **ELSE_LIST**? The reason is that the **ELSE_LIST** contains all possible synonyms and translations of the **"else"** keyword to any of the languages that the user might have supplied in the configuration file. How is a keyword translation added to the parser?

```
function     = función
include      = incluir
if           = si
else         = sino
elif         = sino_si
return       = regresar
print        = imprimir
size         = tamaño
while        = mientras
```

Figure 2

```
ls           = dir
cp           = copy
mv           = move
rm           = rmdir
grep         = findstr
```

Figure 3

How to add keyword synonyms and language translations

One of the advantages of writing a custom programming language is a possibility to have the keywords in any language (besides the 'base' language, understandably chosen to be English). Also we can create our language in such a way, that there will be no need to remember that one must use **dir**, **copy**, **move**, **findstr** on Windows, but **ls**, **cp**, **mv**, and **grep** on Unix; and that a very nice shortcut **cd..** works only on Windows: in our language we can have both! And with just a few configuration changes.

Here is how we can add keyword synonyms and translations to the CSCS language.

First, we define them in a configuration file; Figure 2 is an incomplete example of a configuration file with Spanish translations. The same configuration file may contain an arbitrary number of languages. For example, we can also include the keyword synonyms in the same file (see Figure 3).

After reading the keyword translations we add them to the parser one by one (see Listing 10).

First, we try to add a translation to one of the registered functions (like **print()**, **sin()**, **cos()**, **round()**, **if()**, **while()**, **try()**, **throw()**, etc.). Basically, we map the new keyword to the **ParserFunction**, corresponding to the original keyword. Therefore, as soon as the parser extracts either the original keyword (say **"cp"**), or the one added from the configuration file (e.g. **"copy"**), the same C++ function **CopyFunction**, deriving from the **ParserFunction** class, will be called.

Then we try to add new keywords to the sets of additional keywords, that are not functions (e.g. a **"catch"** is processed only together with the try-block, **"else"** and **"else if"** are processed together with the **if**-block, etc.) The **tryAddToSet()** is an auxiliary template function that adds a translation to a set, in case the original keyword name belongs to that set (e.g. **CATCH = "catch"** belongs to the **CATCH_LIST**).

```
void addTranslation(const string& origName,
                    const string& translation)
{
  ParserFunction* origFunction =
    ParserFunction::getFunction(origName);
  if (origFunction != 0) {
    ParserFunction::addGlobalFunction
       (translation, origFunction);
  }
  tryAddToSet(originalName, translation, CATCH,
    CATCH_LIST);
  tryAddToSet(originalName, translation, ELSE,
    ELSE_LIST);
  //… other sets
}
```

Listing 10

```
palabras = {"Este", "sentido", "es", "en",
 "Español"};
tam = tamaño(palabras);
i = 0;
mientras(i < tam) {
  si (i % 2 == 0) {
    imprimir(palabras[i]);
  }
  i++;
}
```

Listing 11

Listing 11 is an example of the CSCS code using Spanish keywords and functions.

Conclusions

Using the techniques presented in this article and consulting the source code in [Downloads] you can develop your own fully customized language using your own keywords and functions. The resulting language will be interpreted at runtime directly, statement by statement.

We saw that implementing a printing function and a control flow statement is basically the same: one needs to write a new class, deriving from the **ParserFunction** class and override its **evaluate()** method. Then one needs to register that function with the parser, mapping it to a keyword. The **evaluate()** method will be called by the parser as soon as the parser extracts the keyword corresponding to this function. For the lack of space we didn't show how to implement the **while()**, **try**, **throw**, **break**, **continue**, and **return** control flow statements but they are all implemented analogously. The same applies to the prefix and postfix **++** and **--** operators that we did not have space to show but you can consult in [Downloads].

Using the above approach of adding a new function to the parser, anything can be added to the CSCS language as long as it can be implemented in C++.

You can also use this custom language as a shell language (like bash on Unix or PowerShell on Windows) to perform different file or operating system commands (find files, list directories or running processes, kill or start a new process from the command line, and so on). Stay tuned to see how to do that in our next article. ■

References

[ANTLR] ANTLR Framework, http://www.antlr.org

[Downloads] Implementation of the CSCS language in C++, http://www.ilanguage.ch/p/downloads.html

[Kaplan15a] V. Kaplan, 'Split and Merge Algorithm for Parsing Mathematical Expressions', ACCU *CVu*, 27-2, May 2015, http://accu.org/var/uploads/journals/CVu272.pdf

[Kaplan15b] V. Kaplan, 'Split and Merge Revisited', ACCU *CVu*, 27-3, July 2015, http://accu.org/var/uploads/journals/CVu273.pdf

[Kaplan15c] V. Kaplan, 'A Split-and-Merge Expression Parser in C#', *MSDN Magazine*, October 2015, https://msdn.microsoft.com/en-us/magazine/mt573716.aspx

[Kaplan16] V. Kaplan, 'Customizable Scripting in C#', *MSDN Magazine*, February 2016, https://msdn.microsoft.com/en-us/magazine/mt632273.aspx

[Spirit] Boost Spirit Library, http://boost-spirit.com

Acknowledgements

I'd like to thank Frances Buontempo and the *Overload* review team for providing their feedback which enabled me to elevate the content presented in this article.

Concepts Lite in Practice

Concepts should make templates easier to use and write. Roger Orr gives a practical example to show this.

The Concepts TS has been published and it has been implemented in gcc 6.1, released April 2016. Using concepts with gcc is as simple as adding the `-fconcepts` flag.

What does using concepts look like *in practice*: do we get what we hoped for? To answer that question properly we need to ask how, and why, we got here.

C++ is a rich language and supports polymorphic behaviour both at run-time and at compile-time. At run-time C++ uses a class hierarchy and virtual function calls to support object oriented practices where the function called depends on the run-time type of the target; at compile time templates support generic programming where the function called depends on the compile-time static type of the template arguments.

One major difference between these two is that the first one is tightly constrained by the inheritance hierarchy of the objects involved but the second one can be applied to *unrelated* types.

Run-time polymorphism is a key component in object oriented design. The function **signature** to use is decided at compile time based on the static type of the target object; the **implementation** used is based on the run-time type of the object. The rules of C++ guarantee that any object satisfying the static type will provide an implementation for the target function. This has been a fundamental part of C++ for a very long time (about as long as it has been called C++) and has been very stable – it has been essentially unchanged for over 30 years

Compile time polymorphism has also been in the language for a very long time. The principle is to provide a 'template' which enables the compiler to generate code at compile time. One of the main motivations for this was for type-safe containers; but when coupled with non-type template arguments, overloading, and tag-dispatching the result in modern C++ is very expressive. There is, I know, a range of opinions over the utility of template meta-programming and a lot of this is because it can be very hard to **debug**.

The reason for this is that templated code is fragile – whether the code is valid depends on the template arguments provided by the user when the template is instantiated. While the writer of a template has (usually) tested at least one instantiation of their code, they may have made, possibly unconscious, assumptions about the template arguments that can be used with their template.

(Note that if there is no valid instantiation the program is ill-formed, but a diagnostic is not required – it is a 'hard problem' to solve in general so compilers are not required to identify this in every case.)

Library writers currently rely on documenting their assumptions about template parameters but it is hard to get this right. Additionally, compilers don't read documentation and so the diagnosis of a compilation failure can be painful (for the user).

Roger Orr Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

For example, consider this simple piece of code:

```
class X{};
std::set<X> x;
x.insert(X{});
```

I get 50–100 lines of error text from this, depending on which compiler I use.

The fundamental problem is that I'm missing the `<` operator for `x` so it does not satisfy the LessThanComparable requirement documented in the C++ standard:

Table 18 – LessThanComparable requirements [lessthancomparable]

Expression	Return type	Requirement
a < b	convertible to bool	< is a strict weak ordering relation

(In a future version of C++ this simple example may become valid – generation of default comparison operators may make it into the language.)

Enter concepts

One of the early papers on Concepts was Bjarne's paper 'Concept Checking – A more abstract complement to type checking' from Oct 2003 [N1510]. He points out that the fundamental problem that makes it hard to provide good checking for templates in C++ is that templates are not constrained, so by default *any* possible type may be used to instantiate a template.

As a later paper puts it ([N2081], Sep 2006) "Concepts introduce a type system for templates that makes templates easier to use and easier to write."

Currently type checking occurs when some part of the instantiation fails, rather than when checking against the signature of the function being called. Concepts allow the programmer to provide constraints against the types of template arguments which become part of the function signature and hence should be easier to check and provide clearer diagnostics to the user.

The proposals went through a number of changes and enhancements during the development of C++11 (then known as C++0x) but it eventually became clear that the system being proposed was too complex and could not be nailed down in time for a delivery of C++11 in a realistic timeframe. The proposal at that time included, among other things, concept *checking* which ensured the implementation complied with the concepts.

After much discussion and a number of meetings agreement was reached to provide a simplified form of concepts, so-called 'Concepts Lite', and to deliver this as a free-standing Technical Specification rather than initially putting it into the C++ standard itself.

This would give a common standard for those implementing concepts and allow time for experimentation with the feature and feedback from this

the calling code does not need access to the implementation at compile time and the implementation can be changed without invalidating the call site

would allow refinement of the proposal before it was adopted into the C++ standard itself.

Building up an example

Let's build up a simple example to see what can be done with the current language rules and then show what can be done with Concepts Lite.

Consider this simple function declaration:

```
bool check(int lhs, int rhs);
```

The function takes two `int` values and the validity of calling code is decided *without* any need to see the implementation of the function.

```
int main()
{
  int i = 1;
  int j = 2;
  return check(i, j); // Valid!
}
```

A possible function *definition* could be:

```
bool check(int lhs, int rhs)
{ return lhs == rhs; }
```

but the calling code does not need access to the implementation at compile time and the implementation can be changed without invalidating the call site.

Unfortunately using a single function means the following code compiles (possibly with a warning, but the code is valid C++) but is probably not doing what is expected:

```
int main()
{
  double root10 = sqrt(10.0);// approx 3.162278
  double pi = 3.14159;       // near enough
  return check(root10, pi);  // implicit
}                            // conversion to int
```

If we want to check the original `double` values rather than the (truncated) `int` values, we need to generalise the code. One way is to add an overload:

```
bool check(int lhs, int rhs);
bool check(double lhs, double rhs);
{
  bool b1 = check(1, 2);  // works with int
  bool b2 = check(e, pi); // works with double
}
```

We have generalised our function to a *predefined* set of types – but we need to duplicate the implementation.

Another approach is to use a template:

```
template <typename T>
bool check(T lhs, T rhs);
```

We now have only one function definition:

```
template <typename T>
bool check(T lhs, T rhs) { return lhs == rhs; }
```

```
Basic_Template_Failure.cpp: In instantiation of
 'bool check(T&&, U&&) [with T = int&; U = int*]':
... comparison between pointer and integer [-
fpermissive]
 bool check(T && lhs, U && rhs) { return lhs ==
rhs; }                                    ~~~~^~~~
```

Listing 1

and we have produced a template for a set of functions for an *unbounded* number of types.

```
{
  bool b1 = check(1, 2);       // works with int
  bool b2 = check(root10, pi); // works with
}                              // double
```

We can easily generalise further to take two *different* types:

```
template <typename T, typename U>
bool check(T lhs, U rhs);
```

Now, without ***any*** further changes to the implementation, the function can deal with any two types for which equality is defined.

As we mentioned earlier though, compilation errors are reported during the *instantiation* of the function template; for example, if we pass an `int` and the address of an `int` we see Listing 1.

We could do better ...

'SFINAE'

When substituting possible values for template arguments the result can be invalid – in this case the program itself is not invalid, but the troublesome substitution is simply removed from the overload set. This is known as 'substitution failure is not an error' which abbreviates to SFINAE.

Use of this rule can be used for `enable_if` and other similar techniques. Listing 2 is a simple example.

```
template <typename T>
void f(T t, typename T::value_type u);

template <typename T>
void f(T t, T u);

int main()
{
   f(1, 2);  // first f() non-viable when
             // substituting int
   std::vector<int> v;
   f(v, 2);  // second f() not a match as
             //types differ
}
```

Listing 2

it can be rather challenging to find an equivalent SFINAE check, and sometimes there can be subtle differences

```
Basic_Enable_Failure.cpp:40:21: error: no
matching function for call to 'check(int&, int*)'
   return check(i, &j);
                    ^
Basic_Enable_Failure.cpp:34:6: note: candidate:
template<class T, class U, class> bool check(T&&,
U&&)
 bool check(T && lhs, U && rhs) { return lhs ==
rhs; }^~~~~
Basic_Enable_Failure.cpp:34:6: note:   template
argument deduction/substitution failed:
```

Listing 3

In the first call to `f()` the compiler tries to substitute `int` into the first template, which involves trying to form the type `int::value_type`. This is not valid, so the first template is skipped and the second one is used.

We can use SFINAE in our `check()` function to ensure that the equality check will be valid:

```
template <typename T, typename U,
  typename = std::enable_if_t<
    is_equality_comparable<T, U>::value>>
bool check(T && lhs, U && rhs);
```

This has the desired effect of removing the function from the overload set if the two types are not equality comparable (see Listing 3).

Notice what we do here, which is very common with this sort of technique. We have added a *third* template argument to the template, which doesn't have a name as it is not used anywhere, and given it a default value specified in terms of the arguments that we wish to constrain.

We do of course need to provide `is_equality_comparable`; Listing 4 is one possible implementation.

While writing this is not for the faint hearted – and nor is reading it – this is the sort of construct that can be written once and provided in a common header. It can therefore be used without needing to investigate the implementation. Note however that, even apart from the complexity, there is a problem with the *disjoint* between the check and the expression being checked

```
decltype(std::declval<T&>() == std::declval<U&>())
```

The expression we want to detect is `lhs == rhs` but we have to write a more complicated expression, subject to the various rules for SFINAE, which acts as a proxy for the check we actually wanted.

For some expressions it can be rather challenging to find an equivalent SFINAE check, and sometimes there can be subtle differences.

Concepts to the rescue

There are several ways we can constrain our check function with concepts. The first, and most basic way, is by adding a `requires` clause:

```
template <typename T, typename U>
requires requires(T t, U u) { t == u; }
bool check(T && lhs, U && rhs);
```

The function *declaration* shows the constraint using normal C++ syntax (that matches the code used in the definition.) Calling `check()` with `int` and `char*` gives this error:

```
Basic_Requires.cpp:19:6: note: constraints not
satisfied
bool check(T && lhs, U && rhs) { return lhs ==
rhs; }
```

(I'm using the gcc 6.1 release for these examples; there are some additional changes from Andrew Sutton still to come in gcc that should further improve the error messages.)

Alternatively, we could define a *named* concept and use that:

```
template<typename T, typename U>
concept bool Equality_comparable() {
  return requires(T t, U u) {
    { t == u } -> bool;
  };
}
```

Naming a concept has two main benefits.

- The concept can be shared by multiple template declarations
- Naming allows us to express some of the semantic constraints on the type.

We use it like this by supplying it with appropriate arguments in the declaration:

```
template <typename T, typename U>
requires Equality_comparable<T, U>()
bool check(T && lhs, U && rhs);
```

```
#include <type_traits>

template<typename T, typename U, typename = void>
struct is_equality_comparable : std::false_type
{ };

template<typename T, typename U>
struct is_equality_comparable<T,U,
  typename std::enable_if<
    true,
    decltype(std::declval<T&>() ==
      std::declval<U&>(), (void)0)
    >::type
  > : std::true_type
{
};
```

Listing 4

we now have the reverse problem: you can argue our template is now over-constrained

```
Basic_Concept.cpp:33:29: error: cannot call
function
'bool check(T&&, U&&) [with T = int&; U = char*&]'
   return check(argc, argv[0]);

Basic_Concept.cpp:29:6: note:   concept
  'Equality_comparable<int&, char*&>()' was not
satisfied
```

Listing 5

Calling `check()` with `int` and `char*` now gives us Listing 5.

As another alternative the concepts TS allows us to write a concept using a variable-like syntax instead of the function-like syntax used above:

```
template<typename T, typename U>
concept bool Equality_comparable =
  requires(T t, U u) {
    { t == u } -> bool;
  };

template <typename T, typename U>
requires Equality_comparable<T, U>
bool check(T && lhs, U && rhs);
```

The syntax is quite similar to the function template form, except for the reduction in the number of brackets. One restriction though is that you cannot overload variable concepts. The error handling is pretty much the same:

```
Basic_Variable_Concept.cpp:29:6: note:   concept
  'Equality_comparable<int&, char*&>' was not
satisfied
```

Making consistent concepts

The `Equality_comparable` concept I introduced above is asymmetric as it only checks that the expression `t == u` is valid. This matches our (only!) use case, but is not suitable for use as a *general* equality concept.

The ranges TS defines something like this:

```
template<typename T, typename U>
concept bool EqualityComparable() {
  return requires(T t, U u) {
    { t == u } -> bool;
    { u == t } -> bool;
    { t != u } -> bool;
    { u != t } -> bool;
  };
}
```

However, if we use this concept in our example we now have the reverse problem: you can argue our template is now *over*-constrained. Consider this simple example of trying to use `check()` with a simple user-defined class (Listing 6).

```
struct Test{
  bool operator==(Test);
};

bool foo(Test v, Test v2)
{
  return check(v, v2);
}

Basic_Concept_Failure.cpp: In function 'int
main()'
Basic_Concept_Failure.cpp:39:20: error: cannot
call function 'bool check(T&&, U&&)'
  return check(v, v2);
... concept 'Equality_comparable<Test&, Test&>()'
```

Listing 6

In order to call our `check()` function we have to declare, but not necessarily define, an extra operator.

```
struct Test{
  bool operator==(Test);
  bool operator!=(Test);
};
```

or

```
bool operator!=(Test, Test);

bool foo(Test v, Test v2)
{
  return check(v, v2); // Now Ok
}
```

This was a simple example as we were instantiating the template with the same type for both `T` and `U`. If the arguments to `check()` differ in type we have a little more work to do (Listing 7).

```
struct Test{
  bool operator==(int);
};

bool foo(Test v)
{
  return check(v, 0);
}

Basic_Concept_Failure2.cpp: In function 'int
main()'
Basic_Concept_Failure2.cpp:39:20: error:
cannot call function 'bool check(T&&, U&&)'
   return check(v, 0);
... concept 'Equality_comparable<Test&, int>()'
```

Listing 7

When the constraints for a function change, the type provided may now silently fail to satisfy the constraints

```
struct Test{
  bool operator==(int);
  bool operator!=(int);
};

// These two cannot be defined in-class
bool operator==(int, Test);
bool operator!=(int, Test);

bool foo(Test v)
{
  return check(v, 0); // Ok
}
```

Listing 8

In order to call our `check()` function we now have to declare ***three*** extra functions (see Listing 8).

Is this good or bad? It's both, unfortunately.

Defining a smallish set of well-defined and 'complete' concepts is arguably a good idea as it encourages/enforces more consistent operator declarations for types. Stepanov's original design for the STL was based on his work in *Elements of Programming* and he defined various type attributes such as Regular and Constructible. This type of classification is a natural fit for this sort of use of concepts.

However, it can increase the work needed to adapt a class to comply with a concept. C++ programmers are used to providing the minimal set of methods to use a template and have a reluctance to provide more than this. Whether this is something to be encouraged is arguable.

The '`requires requires`' use is less affected by this issue as each function template has its own clause – but it raises the complexity of reading *each* function declaration, and breaks DRY.

Behaviour under change

When using run-time polymorphism, the derived type must implement all the pure virtual methods in the base class. If a new method is added, or a signature is changed, errors are reported when the derived class is compiled and/or instantiated (depending on exactly what's changed).

When the constraints for a function change, the type provided may now *silently* fail to satisfy the constraints. This may result in a compile time failure, or may result in a *different* overload of the function being selected.

More complex constraints

If we look at an example with a more complicated constraint we see some additional benefits of using concepts over the existing technology that uses `enable_if` or other SFINAE techniques.

While `enable_if` and similar techniques do provide ways to constrain function templates it can get quite complicated. One recurring problem is the ambiguity of template argument types ***solely*** constrained by `enable_if` – for example, see Listing 9.

```
struct V {
  enum { int_t, float_t } m_type;

  // Constructor from 'Int' values
  template <typename Int,
    typename = std::enable_if_t<
      std::is_integral<Int>::value>>
  V(Int) : m_type(int_t) { /* … */ }

  // Constructor from 'Float' values
  template <typename Float,
    typename = std::enable_if_t<
      std::is_floating_point<Float>::value>>
  V(Float) : m_type(float_t) { /* ... */ }
};
```

Listing 9

Unfortunately this example does ***not*** compile – as we have two overloads of the same constructor with the same type (that of the first template argument) – see Listing 10.

The compiler never gets to try overload resolution as declaring the two overloads is a syntax error. If we add a *second* argument we can resolve the ambiguity and allow overload resolution to take place; SFINAE will then remove the case(s) we do not want.

We can give the extra argument a default value to avoid the caller needing to be concerned with it. (Listing 11)

The additional dummy arguments are of two *different* types, `dummy<0>` and `dummy<1>`, and so we no longer have ambiguity and both functions participate in overload resolution.

However, this addition of dummy arguments that take no other part in the function call adds needless complexity for both the compiler and for the readers and writer of the template. We are also relying on the optimiser removing the dummy arguments.

Concepts are a language feature and so the solution using them is much clearer, as can be seen in Listing 12.

```
ambiguity_with_enable_if.cpp:25:5: error:
   'template<class Float,
   class> V::V(Float)' cannot be overloaded
  V(Float) : m_type(float_t) {}
  ^
ambiguity_with_enable_if.cpp:20:5: error:
   with 'template<class Int, class> V::V(Int)'
  V(Int) : m_type(int_t) {}
  ^
```

Listing 10

The concepts TS supports a 'concept introducer' syntax and an abbreviated syntax

```
enum { int_t, float_t } m_type;

template <int> struct dummy { dummy(int) {} };

// Constructor from 'Int' values
template <typename Int,
  typename = std::enable_if_t<
    std::is_integral<Int>::value>
>
V(Int, dummy<0> = 0) : m_type(int_t) {/* ... */}

// Constructor from 'Float' values
template <typename Float,
  typename = std::enable_if_t<
    std::is_floating_point<Float>::value>
>
V(Float, dummy<1> = 0) : m_type(float_t) {/*...*/}
```

Listing 11

There is no ambiguity here as the constraints are part of the overload resolution rules themselves and so there is no need for dummy arguments.

Concept introducer syntax

The concepts TS supports a 'concept introducer' syntax and an abbreviated syntax which I have not yet demonstrated, so let's modify the example to do so.

A lot of the complexity in the wording of the Concepts TS is to ensure that the specification of a constrained function using this additional syntax is equivalent to the **`requires`** form (Listing 13).

This is syntactic sugar to avoid needing to use **`typename`**:

```
template <Int T>
bool check(T value);
```

```
struct T {
  enum { int_t, float_t } m_type;

  // Constructor from 'Int' values
  template <typename Int>
    requires std::is_integral<Int>::value
  V(Int) : m_type(int_t) { /* ... */ }

  // Constructor from 'Float' values
  template <typename Float>
    requires std::is_floating_point<Float>::value
  V(Float) : m_type(float_t) { /* ... */ }
};
```

Listing 12

```
template <typename T>
concept bool Int = std::is_integral_v<T>;

template <typename T>
concept bool Float = std::is_floating_point_v<T>;

struct V {
  enum { int_t, float_t } m_type;

  // Constructor from 'Int' values
  template <Int T>
  V(T) : m_type(int_t) { /* ... */ }

  // Constructor from 'Float' values
  template <Float F>
  V(F) : m_type(float_t) { /* ... */ }
};
```

Listing 13

is equivalent to:

```
template <typename T>
requires Int<T>
bool check(T value);
```

The translation between the introducer syntax and that using **`requires`** is fairly simple and unlikely to cause confusion. Note though that, as they are equivalent, both forms can occur in the same translation unit – and that the name **`T`** could be different.

It does save characters: in this case 37 vs 58. How much of a benefit this is seems to depend who you ask.

You can further abbreviate the syntax:

```
struct V {
  enum { int_t, float_t } m_type;

  // Constructor from 'Int' values
  V(Int) : m_type(int_t) { /* ... */ }

  // Constructor from 'Float' values
  V(Float) : m_type(float_t) { /* ... */ }
};
```

This syntax allows the declaration of templates without needing to use **`<>`**.

Is this a good thing? There seem to be three main answers:

- Yes
- No
- Maybe

One reason why it might be troublesome is the difference that a function being a template or not makes to the lookup and argument matching rules (see Listing 14).

```
bool check(Int value);

void test(Float f)
{
  if (check(f))
  {
    // do something
  }
}
```

Listing 14

```
bool check(auto value);

void test(Float f)
{
  if (check(f))
  {
    // do something
  }
}
```

Listing 15

Will `check()` get called? If `Int` is a type we look for conversions on argument types that we will not look for if `Int` is a concept.

Additionally, whether we need access to the definition of `check()` depends on whether it is a template or not.

Equivalent or functionally equivalent?

What does it all mean, anyway?

```
bool check(Int value);
```

This is equivalent to:

```
template <Int A>
bool check(A value);
```

which is itself equivalent to:

```
template <typename C>
requires Int<C>
bool check(C value);
```

and all of these are *functionally* equivalent to:

```
template <typename D>
requires std::is_integral_v<D>
bool check(D value);
```

The difference is that a program *can* contain two declarations of a function template that are equivalent – but it is *not* valid to contain two declarations of a function template that are only functionally equivalent. There are additional rules defining the finer details of equivalence in the concepts TS (or [N4335]).

Restrictions on the concept introducer syntax

```
bool check(Int value, Int other);
```

This is equivalent to:

```
template <Int T>
bool check(T value, T other);
```

Note that the two variables will always have the ***same*** type – we cannot use the short form syntax (there were some very tentative proposals, but nothing was accepted) to replace:

```
template <Int T, Int U>
bool check(T value, U other);
```

auto templates

The concepts TS also allows using `auto` to introduce an unconstrained template parameter (Listing 15).

I see this as less problematic than the constrained case – it mirrors the existing use of `auto` for declaring a polymorphic lambda. There is no ambiguity in the mind of the reader about whether or not the function so declared is a template, the use of `auto` means it must be a template and so no further context is needed.

While 'Concepts Lite' is not part of C++17 (see, for example, [Honermann] for more on this...) this part of the TS does stand a little apart from the rest and I for one would have been happy to accept this into C++17 as a separate proposal; sadly this hasn't been proposed yet and the departure gate for C++17 is probably closed for that option.

Summary

Concepts do seem to be significantly simpler to read and write than the current alternatives.

It does seem to me that it delivers better compiler errors for users; while I'm sure it is possible to find counter-examples I expect them to be rarer.

It is easier to constrain functions, for writers, without requiring complex and sometimes fragile meta-programming trickery. Concepts also express intent in code rather than by inference, in documentation, or in comments.

Using concepts does have the potential of under or over constraining. If we under-constrain types that match, the concepts fail when the template is instantiated, and if we over-constrain, we restrict the set of allowable types further than we needed.

The syntax is slightly awkward – this is my biggest issue with the current wording in the TS.

- `template <typename T> requires requires(T t) {…}`
  '`requires requires`' – can we be serious?
- `concept bool x = requires(...) {}`
  the type *must* be specified and *must* be `bool`
- supporting the variable form, in addition to the function form, seems to be unnecessary

I am not persuaded that we need the concept introducer syntax. While it reduces the number of symbols in the code I am not sure that this actually *simplifies* things or merely hides away complexity that we do need to know about.

Now is the time for C++ programmers to ***use*** concepts and to provide feedback to the standards body. With the recent release of gcc 6.1 doing this has become easier. ■

References

[Honermann] http://honermann.net/blog/?p=3 ('Why concepts didn't make it into C++17')

[N1510] http://www.stroustrup.com/n1510-concept-checking.pdf

[N2018] http://open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2081.pdf

[N4335] http://open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4335.html (pre-publication concepts working paper)

Afterwood

Magazines sometimes use the back page for adverts or summaries. Chris Oldwood has decided to provide us with his afterwords, or 'afterwood'.

Back when I started my professional programming career, the world was awash with paper-based programming journals. At the first company I worked for they had some of the more popular ones on circulation, such as *Dr Dobbs Journal* (DDJ), *Microsoft Systems Journal* (MSJ), *Windows Developer Magazine* (WDM) and *C/C++ Users Journal* (CUJ). On my travels as a journeyman I've also ended up discovering a number of other journals that I subscribed to in an attempt to help quench my thirst for programming knowledge: *C++ Report*, *Java Report*, *TechNet Magazine* and *Application Development Advisor*. Sadly it wasn't until the demise of many of these that I happened across ACCU and therefore *C Vu* and *Overload*. (I had asked on a company C++ forum about what printed journals were still around now that CUJ was going the way of the Dodo.)

The format was largely the same for each publication – there was a beginning, middle, and an end. The start of the magazine usually had some kind of editorial, which might be a simple summary of the content, perhaps calling out the most exciting contributions, or it could be more like a conference keynote – just some musing on the IT industry in general. If our esteemed editor thinks she's doing a good job at avoiding writing a proper editorial then she should re-read some older journals to see how much an editor can really get away with...

Naturally the meat of the sandwich was the articles, with a mixture of regular columnists like Ed Nisley and Al Williams (DDJ), and one-off submissions from other people like ACCU's own Matthew Wilson. Occasionally you had a little garnish, such as a 'Letters Page' where readers would email (or even write, like with an actual pen and paper, or perhaps a typewriter) to the magazine to comment on some previous article from months ago. Maybe there was a typo in the code, or (more likely) a disagreement about the approach taken or conclusion drawn. Another common filling was the 'New Products' section which was often a listing of 'recently' released software tools provided by the manufacturers themselves. Luckily the cadence of software releases was so much greater than that of the printed publication so there was always something different each month.

In contrast, the final piece, like some editorials, was more of an art form in its own right. Where the editorial perhaps appeared to have some constraints around it being a fairly straight-laced affair, the closing remarks seemed to be pretty much a free-for-all sometimes. That said they were related, though occasionally only very tangentially, to the IT industry. With just a single page and less chance of putting off punters browsing in the shops (the contents page being nearer the front than the back) there was more latitude.

I'll be honest and admit that I didn't really have the foggiest idea about what a few of the authors were on about some of the time. That will likely be due to a series of in-jokes and references to people and technology that probably pre-dates my birth, let alone my entry into the profession. But that didn't stop me loving them all the same. In fact I felt uncomfortable being on the outside, not knowing what those inside were savouring, and so in a way it drove me to take more of an interest in the history of the industry and its people. My own comparisons around the Kardashians and JavaScript frameworks will no doubt soon become an anachronism too due to the pace of technology.

Not every journal followed this format for its conclusion though, or perhaps not for every issue. As an example, this very publication (*Overload*) has, as far as my cursory sampling suggests, never had an afterword to match every foreword. I was originally going to say 'regular afterword' but there is a pattern, albeit only once a year, where the April issue sees Teedy Deigh pass on her own brand of programming wisdom. *C++ Report* utilised its final page to carry on an earlier tradition by hosting a showcase of Obfuscated C++ which was curated by Rob Murray. Whilst meant to tax your C++ skills, or in jest to show how perverse you could be with C++, it often felt scarily close to the codebase I was working on during the day.

Some of those that were given a platform at the tail end of the magazine used it to have a good old-fashioned rant. David S. Platt, the incumbent for *MSDN Magazine* (the modern successor to MSJ), currently has a column titled 'Don't Get Me Started' where he gets to let off a little steam. Another notable grumpy voice from the past was Gary Barnett who penned 'Angry Young Man' for *Application Development Advisor*. I seem to remember that he occasionally tag-teamed with Martin Banks who took up the mantle under the subtly different guise of an 'Angry Old Man'. Luckily the arguments were a little more coherent than some of the rants you see during the lightning talks at a conference, and in the end they soon mellow out into 'Mildly Displeased Columnists'. Well, the British ones did.

Michael Swaine, with his 'Swaine's Flames' for *Dr Dobbs* Journal, was a more sedate affair. Often his musings were played out as a scene between various techies that frequented a fictitious 'watering hole' in Silicon Valley that went by the cute name of Foo Bar. I didn't know nearly enough about what was happening in that part of the world to truly grasp whatever point he was making (if any) but I enjoyed the relaxed attitude and the writing style made a real change from the usual dry technical prose. Not content with propping up the back end of DDJ he also wrote 'The Final Page' for one of DDJ's sister publications – *Web Techniques* – and I wouldn't be at all surprised to bump into him in other parts of CMP Media's vast publishing estate. Michael Swaine, now editor for *PragPub*, still appears to be going strong in the digital age under the banner of Swaine's World.

Of all the columns that I probably understood the least at the time, but still revered the most, was the bi-monthly 'Post-Mortem Debunker' in the *C/C++ Users Journal* written by Stan Kelly-Bootle. I had always assumed it was a pen name which, given the style, might have been attributable to someone such as Douglas Adams if it wasn't a physical impossibility by then. It was also somewhat reminiscent of the writings of

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

Back in those youthful days, when the printed page ruled the roost, I wondered if the final page was a form of 'pasture' for old programmers

John Gall (*The Systems Bible*) which I had only briefly glimpsed back then. With the birth of *Wikipedia* I discovered that he really was an accomplished author with a distinguished past-life in computer science. Oh, and he was a singer and song-writer too, if the man wasn't already talented enough. What made his writing particularly entertaining was his word play against a backdrop of modern computing fused with some interesting tales about what it was like programming right back at its dawn (on the EDSAC). I seem to remember his footnotes were often a goldmine of little one liners.

The one end page that I turned to as the first thing I hastily wanted to read was Raymond Chen's 'Windows Confidential' column in Microsoft's *TechNet Magazine*. His blog 'The Old New Thing' (an extended version of his column) became the first one I read every day, and it eventually turned into one of my favourite technical books too. Its name provides the (somewhat obvious) inspiration for my own blog – *The OldWood Thing*.

Each month for his column he would pick something curious about the way the Windows operating system or Win32 API behaved and would provide some background material that would bring a semblance of sanity to what was often seemingly perverse. Regularly the answer would reach right back into the dim-and-distant past of 16-bit Windows, DOS or its predecessors. Other times it might involve a backwards compatibility issue with a product that was just far too popular with customers to alienate. To balance things out there were also a few misguided design decisions thrown in there too, but ultimately it's helped instil in me a perspective of what software engineering at a mammoth scale is all about.

Back in those youthful days, when the printed page ruled the roost, I wondered if the final page was a form of 'pasture' for old programmers. Hipsters hadn't started sporting beards back then and the only data point I had was of Michael Swaine; I hope they'll forgive my statistical error. Now finding myself in this very position I can speculatively report that the end of the magazine does not appear to be correlated with the end of one's programming career. If anything it may provide the catalyst for indulging in a more elaborate style of writing. ■

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

Visual Studio

Upgrade from Visual Studio Professional with MSDN subscriptions to Visual Studio Enterprise 2015 with MSDN and get 25% off!

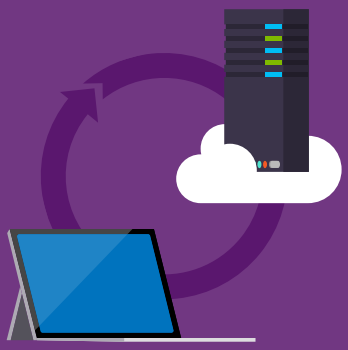
Offer expires 30th June 2016

Visual Studio Enterprise is an integrated, end-to-end solution for teams of any size with demanding quality and scale needs. Take advantage of comprehensive tools and services for designing, building and managing complex enterprise solutions.

Greater productivity for enterprise application development and delivery

Plan, execute and monitor your entire testing effort, continuously

Manage complexity and close the loop between Dev and Ops



Microsoft Partner
Gold Volume Licensing

qbs SOFTWARE

For more information contact QBS Software : +44 (0) 20 8733 7101 | sales@qbs.co.uk
www.qbssoftware.com/visualstudio2015