overload 116

AUGUST 2013 £3

Hard Upper Limit on Memory Latency

How the speed of our code is determined by physical constraints

Portable String Literals in C++

Tactics for accessing files whose paths contain international text literals

Simple Instrumentation

How to diagnose programs that run out of memory or grind to a halt

Auto - A Necessary Evil?

When is it appropriate to define something `auto` in C++11?

Dynamic C++

We continue our exploration of how to add dynamic facilities to C++. This time we investigate some of the more powerful tools.

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 116

August 2013

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 117 should be submitted by 1st September 2013 and those for Overload 118 by 1st November 2013.

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

Copyrights and Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

Learning Fantasy Languages

The last Overload editorial described a Fantasy Language. Now, how do you learn one?

I would like to thank Ric Parkin for stepping in and writing an editorial for the previous *Overload*. His presentation of a fantasy language (FL) got me thinking, thereby stopping me from getting round to writing an editorial yet again. How would I learn FL? I would not be able to search stack overflow [SO] for answers. Perhaps this is a good thing. I would not be able to buy any books on the subject. This may or may not be a good thing. Books are frequently time consuming and, as previously discussed, heavy [*Overload* 111]. I have not bought any books so far this year, provided that comics, a book for completing a survey, my Dad buying the 4th edition of *The C++ Programming Language* [C++PL] for the latest ACCU study group and somehow ending up with a book after going near the charity book stool at this year's ACCU conference do not count. Similarly, any books in the form of PDFs that have ended up on my Kindle do not count. So, how would I learn FL? In this extreme example, I would have to implement it myself in order to learn it. I did start learning C++ by writing a parser for C++, which appears to be unconventional. To be fair, it just parsed C with classes to expose constructors and public functions to a rapid application development tool. Nonetheless, this was a good way to start learning a language. It certainly made using it seem a doddle by comparison. Before progressing on to using the Standard Template Library, I was encouraged to write several container classes myself to get a solid grounding in how they might work and what sort of interface they should provide. Though it may have been quicker to just use the tools that already existed, rather than re-invent the wheel, I appreciated the chance to think deeply about what is going on under the hood.

Since my book ban, I have been trying to read some of the un-read books I own. My dream is to buy more when I have caught up. I am making some progress. I have eleven books on the go at the moment, and have just finished *The Structure and Interpretation of Computer Programs* [SICP]. The latter part of the book considers various approaches to implementing lisp, which makes the reader think deeply about how a programming language works. It poses a variety of problems ranging from considering the effect of trying various orders of evaluation of expressions, through to lazy evaluation with streams, non-deterministic programming, creating register machines, writing a debugger, implementing garbage collection and considering what tail recursion means. Many of these are complicated ideas that someone who just programs for a day job would never really have to know about. Again we have an example of a deep – dive into a topic to learn it, which while not being strictly needed, can help with a rigorous understanding of the subject. Richard Feynman apparently once said "What I cannot create I do not understand." There is clearly merit in learning how to write a lisp evaluator in lisp. Or working on a C++ compiler. Or joining the standards committee.

If the final step in learning FL, or indeed any language, is to implement it, what are the first steps? There may be some overlap with how humans learn anything. I was introduced to Piaget's theory of cognitive development [Piaget] at teacher training college. At the time, I failed to realise that his ideas are about how small children learn while they grow up, rather than how adults learn. Nonetheless, some of the stages and processes still possibly happen in adults. He talks about assimilating or taking on board new information and how this gets accommodated into our current world view. The accommodation step may shift our perspective slightly. If a person is learning a programming language from scratch the problems encountered will differ from those of another who is learning a second language. I keep writing

```
for ( auto & item in stuff ) { …}
```

in C++11, since I know python. A person who is new to programming is likely to get the "`:`" instead of "`in`".[1]

Aside from the syntax differences, idioms differ between languages. Can you tell if someone knows functional programming and tries to use it in another language? Quite probably. Piaget describes other stages, such as a sensorimotor stage, from birth until the age of two, which may not be relevant to learning programming, apart from the discussions about hand-eye coordination which may relate to typing, which we touched on in a previous *Overload* [*Overload* 114]. In conjunction with the various stages through life, he identifies various processes that occur. These include conversation, decentering (shifting your perspective to view something from various angles), reversibility (breaking something to piece it back together), serriation (sorting by characteristics), transitivity, classification, elimination of egocentrism (seeing things from another's perspective), inductive logic, testing for concrete operations (talking to others about ideas) and finally at the formal operational stage ideas are abstracted and deductive reasoning can happen. Sketching over the details, the high level view I was given at college was that pupils learn by reading, writing, talking then doing or possibly being able to explain to a peer. A similar path may happen as you learn a programming language.

You will almost certainly read code in a language you are trying to learn before trying to write any, FL excepted since it doesn't exist yet. This provides a unique problem for language innovators. Having read some code you will eventually start trying to write some code. If you wish to continue learning the language you will need to start talking about it to others, maybe describing what you have done, perhaps via some tests, possibly getting a code review, or possibly even writing an article about it. The next step would be mentoring someone, thereby forcing you to

[1] Many thanks to Jon Wakely for helpfully suggesting I `#define in :`, though I suspect that could cause other problems.

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

clearly communicate what you have understood. This will probably require an 'elimination of egocentrism' touched on earlier. It is frequently never enough to just state how something is and expect another to repeat it back parrot fashion – there needs to be engagement and you need to understand the mentee's perspective. Just repeating the same words over and over when trying to teach someone a new idea will almost never help. Understanding someone else's confusion makes you understand more deeply yourself. When teaching someone it is all too easy to just give rules to follow, maybe by giving hard and fast coding standard to adhere to, or by insisting on a 'rigid agile' processes. As we know, Captain Hector Barbossa said, "The code is more what you'd call guidelines than actual rules." Scott Meyers frequently gives his advice in the form of guidelines, for example "Prefer **`const`** and **`inline`** to **`#define`**." [EC++]

I find this way of presenting wisdom and understanding beautiful. Most people are rebels at heart and will follow the letter of the law rather than its spirit. If you can give guidelines that encourage people to code with their brains engaged you are doing well.

The last step which Piaget omitted is to explain all this to a computer. Even when you think you understand an algorithm, or what your customers want, coding it up frequently throws up issues you did not initially spot. Like writing a recipe, you need to be precise and explicit in all the steps.

> *Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it.* ~ [Knuth]

Clearly, if I can write a C++ compiler, debugger, refactoring tool, and present proposals to the standards committee, I probably have a deep knowledge and understanding of C++. Knowledge covers rattling off edge cases, or shouting out "20.3.2" to someone speaking at the ACCU conference (you know who you are). Understanding is a different matter. 'Understand' is a strange word. The internet suggests its etymology is related to 'stand' together with 'under' though more with a sense of 'inter', 'between' or 'among', giving a hint of Greek 'entera' or intestines [etymonline]. If I understand something, I am right in its guts. I stand within it. I can see the world from a new perspective. The word 'understand' also carries flavours of investigations and beginnings. Understanding is the beginning. What is it the beginning of? It can be a step towards wisdom. Though wisdom is about knowing, it does not mean being able to rattle of a list of facts parrot fashion. Though wisdom might involve understanding, it digs deeper. "A wise man has no extensive knowledge; He who has extensive knowledge is not a wise man." [Lao-tzu] In words reminiscent of Fight Club, the Bible tells us, "The beginning of wisdom is this: Get wisdom. Though it cost all you have, get understanding." [Proverbs 4: 7]

Having said all this, is understanding really important? When people started a quest for artificial intelligence their aim was to create computers that mimic the way humans did things, hoping to make machines that could think. Leaving aside the question of what think actually means, this has raised several further points. Most of us are aware of the Turing test, where a human tries to decide if they are communicating with a human or a machine [Turing]. This was introduced as a way of deciding if machines can think, without having to define 'think'. It has also spawned excellent science fiction stories. John Searle suggested if a machine could not be disambiguated from a human in this test, it could not really be seen as thinking. He introduced a thought experiment called the Chinese Room argument [Chinese room].He imagines being locked in a room and given paper with Chinese symbols on. Since he doesn't understand these, they just looks like meaningless squiggles. Helpfully, he is also given precise instructions in English of how to respond to the Chinese, which can be regarded as a program. People outside the room pass him questions in Chinese, and by following the instructions he manages to answer the questions by writing appropriate squiggles. This may well give the appearance of understanding Chinese to those outside the room who receive his written answers, though it is clear to us he does not. If I do not understand Ric's FL, but can program in it, does this matter? I would like to suggest it does. I will miss opportunities to improve the language and design my own. I will probably never be able to develop wisdom while using the language, guiding me to a stage of unconscious competence, where I can seemingly use the language with no thought, or at least effort, at all [Stages of competence] and I suspect I will not be able to explain it well to others.

Thanks again to Ric for stepping in last time. I am sorry this has distracted me from writing an editorial yet again. I did consider implementing FL, but have been so busy avoiding writing an editorial I didn't have time to do that either. It remains to be seen what excuses I can come out with for next time.

References

[C++PL] *The C++ Programming Language*, Bjarne Stroustrup, Addison Wesley 2013

[Chinese room] Searle, John. R. (1980) 'Minds, brains, and programs', *Behavioral and Brain Sciences* 3 (3): 417-457 http://cogprints.org/7150/1/10.1.1.83.5248.pdf

[EC++] Scott Meyers *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* (Addison-Wesley Professional Computing Series) 2nd Edition 1997

[etymonline] http://etymonline.com/?term=understand

[Knuth] http://en.wikiquote.org/wiki/Donald_Knuth

[Lao-tzu] *Tao te Ching*, c.550 B.C.E.

[Overload 111] *Overload* 111, October 2012

[Overload 114] *Overload* 114, April 2013

[Piaget] https://en.wikipedia.org/wiki/Piaget's_theory_of_cognitive_development

[Proverbs 4: 7] New International Version

[SO] http://stackoverflow.com/

[SICP] http://mitpress.mit.edu/sicp/full-text/book/book.html

[Stages of competence] http://en.wikipedia.org/wiki/Four_stages_of_competence

[Turing] http://en.wikipedia.org/wiki/Turing_test

Hard Upper Limit on Memory Latency

Achieving very low latency is important. Sergey Ignatchenko asks how low can we go.

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

In [Bunny12], we discussed upper limits on the feasible possible memory size. It was found that even if every single atom of silicon implements one bit of memory, then implementing 2^{128} bytes will take 54 cubic kilometers of silicon, and to implement 2^{256} bytes there won't be enough atoms in the observable universe. Now, we will proceed to analyse the upper limit of speed for huge amounts of memory (which are lower than the absolute limits mentioned above, but are still much higher than anything in use now). In other words, we'll try to provide an answer to questions like, "Is it realistic to expect 2^{100}-byte RAM to have access times which are typical for modern DDR3 SDRAM?"

Assumptions

We need to agree on the assumptions on which we will rely during our analysis. First, let's assume that RAM is still made out of silicon, and that each bit of RAM requires at least one atom of silicon to implement it. This is an extremely generous assumption (current implementations use several orders of magnitude more atoms than that). Second, let's rely on the assumption that nothing (including information) can possibly travel faster than speed of light in vacuum. This is a well-known consequence from invariance of speed of light and causality (many will name it a scientific fact rather than assumption or hypothesis, but we won't argue about the terms here).

Analysis

Let's consider memory which has *B* bytes. Let's assume that each bit is implemented by one single atom of silicon. Then, this memory will take minimum a possible volume of

$$V_{\min} = \frac{V_{mSi}}{N_A} \times B \times 8$$

where N_A is Avogadro's number (6.02×10^{23} mol^{-1}), and V_{mSi} is molar volume of silicon (12×10^{-6} m^3/mol). Therefore, for our 2^{100}-byte (which is approximately equal to 1.27×10^{30} bytes) RAM it will take at least 200 cubic meters of silicon. Now let's assume that whatever device which needs access to our RAM has dimensions which are negligible compared to the size of RAM silicon, so we can consider access to our RAM coming from a point (let's name this point an 'access point'). Now let's arrange our RAM around the access point in a sphere (a sphere being the most optimal shape for our purposes). Such a sphere will have radius of

$$R_{\min} = \sqrt[3]{\frac{V}{\frac{4}{3}\pi}} = \sqrt[3]{\frac{6}{\pi} \times \frac{V_{mSi}}{N_A} \times B}$$

Therefore, for our 2^{100}-byte RAM, a silicon sphere implementing it will have radius of at least 3.7 meters. Now, let's find out how long it will take an electromagnetic wave to go through R_{min} (back and forth, to account for the time it takes a request to go to the location where the data is stored, and data to come back):

$$T_{\min} = \frac{2 \times R_{\min}}{c} = \frac{2}{c} \times \sqrt[3]{\frac{6}{\pi} \times \frac{V_{mSi}}{N_A} \times B} \qquad (*)$$

where *c* is a speed of light (strictly speaking, we should take speed of electromagnetic waves in silicon, but as we're speaking about lower bounds, and the difference is relatively small for our purposes, we can safely use the speed of light in vacuum, or 3×10^8 m/s). Substituting, we find that for our example 2^{100}-byte RAM, equals approximately 25 nanoseconds.

It means that (given our assumptions above) there is a hard limit of 25 nanoseconds on the minimum possible guaranteed latency of 2^{100}-byte RAM. While the number may look low, we need to realize that modern typical RAM latencies are more than an order of magnitude lower than that: for example, typical latency for DDR3 SDRAM is 1–1.5 ns (approximately 20 times less than our theoretical limit for 2^{100}-byte RAM).

Now we can ask another question – "What is the maximum memory size for which we can realistically expect latencies typical to modern DDR3 SDRAM?" Using formula (*), we can calculate it as approximately 2^{90} bytes. That is, even if each bit is implemented by single atom of silicon, 2^{90}-byte RAM is the largest RAM which can possibly have latencies comparable to modern DDR3 SDRAM.

Generalization

If (as is currently the case) each bit is implemented with *N* atoms of silicon, our formula (*) will become

$$T_{\min} = \frac{2}{c} \times \sqrt[3]{\frac{6}{\pi} \times \frac{V_{mSi}}{N_A} \times B \times N}$$

allowing the calculation of latency limits depending on the technology in use. For example, if for our 2^{100}-byte RAM every bit is represented with 1000 atoms of silicon (which is comparable – by the order of magnitude – to technologies used in modern RAM), the best possible latency will

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

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

become 250 ns. As for the largest memory which can have latencies comparable to modern DDR3 SDRAM (given 1000 atoms per bit implementation), it is approximately 2^{80} bits.

Further considerations

It should be mentioned that latency is not the only parameter which determines memory performance; another very important parameter is memory bandwidth (and memory bandwidth is not affected by our analysis). Also it should be mentioned that T_{min} is in fact the minimum latency we can guarantee for all the bits stored (bits stored closer to the access point will have lower latencies than T_{min}). Another practical consideration is caching – our analysis did not take caching into account, and for most common access patterns caching will improve average latencies greatly.

Conclusions

One interesting consequence which comes out of our analysis is that currently silicon technology has already got very close to the hard physical limits (as opposed to technological limits which dominated electronics for decades), and that even relativistic effects may come into play when trying to improve things further along the lines of Moore's law. While this is a known thing for those dealing with bleeding-edge electronics, it is usually ignored by people in the software industry, where it is quite common to extrapolate Moore's law to last for centuries. On the other hand, approaching such hard physical limits may signal a close of the usual every-year expansion of number of cores/RAM/HDD size/..., and such an end may have very significant effects on the future of the software industry. While it is unclear if it will be a Good Thing or Bad Thing for people in the industry, what is clear is that such an end would be quite a drastic change for the software development industry as a whole. ■

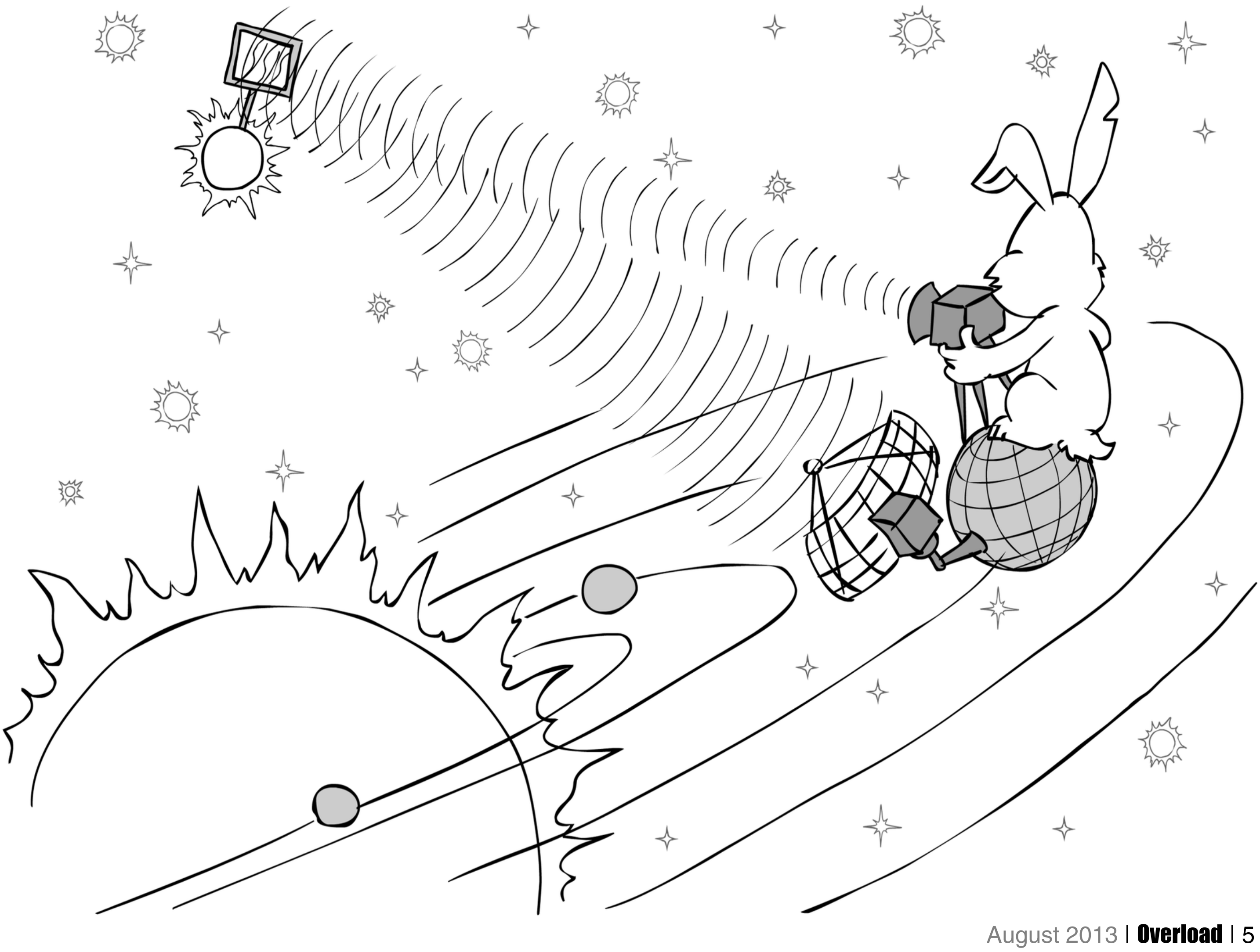
References

[Bunny12] "No Bugs" Bunny, '~~640K~~ 2^{256} Bytes of Memory is More than Anyone Would Ever ~~Need~~ Get', *Overload* #112, December 2012

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

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.



Simple Instrumentation

Programs often run out of memory or grind to a halt. Chris Oldwood demonstrates how to add instrumentation to your code to track its performance.

The days when an application was a simple process that ran entirely locally are long gone. Most systems these days are complex beasts that reach out to a variety of services which can be hosted both locally and remotely. As the number of links in the chain grows, so does the potential for a performance problem to surface. And as we delegate more and more work to 3rd party services instead of owning them, sometimes the best we can hope for is to keep a very close eye on how they're behaving.

The internal components we also rely on will continually get upgraded with new features and bug fixes so we need to make sure that any change in their performance characteristics are still within our expected tolerances. Even during day-to-day operation there will be spikes in volumes and anomalous behaviour in both our own code and those of our upstream and downstream dependencies that will need to be investigated.

If you've ever had a support query from a user that just says "the system is running like a dog" and you have no performance data captured from within the bowels of your application then you've got an uphill battle. This article explores what it takes to add some basic instrumentation to your code so that you can get a good idea of what is happening in and around your system at any time.

System-scale monitoring

Your operations team will no doubt already be monitoring the infrastructure, so why isn't that enough? Well, for starters they are almost certainly *only* monitoring your production real estate – your development and test environments are likely to be of little importance to them. And yet those are the environments where you want to start seeing any performance impact from your changes so that they can be dealt with swiftly before reaching production.

The second reason is that they only have a 1,000 foot view of the system. They can tell you about the peaks and troughs of any server, but when an anomaly occurs they can't tell you about which service calls might have invoked that behaviour. Databases in particular have a habit of behaving weirdly as the load spikes and so knowing which query is being blocked can give you clues about what's causing the spike.

What to measure

Sir Tony Hoare tells us that "premature optimisation is the root of all evil" [Hoare] and therefore the first rule of thumb is to measure everything we can afford to. In a modern system where there are umpteen network calls and file I/O there is very little for which the act of measuring will in any way *significantly* degrade the performance of the operation being measured.

Chris Oldwood is a freelance developer who started out as a bedroom coder in the 80s, writing assembler on 8-bit micros. These days it's C++ and C# on Windows in big plush corporate offices. He is the commentator for the Godmanchester Gala Day Duck Race and can be contacted via gort@cix.co.uk or @chrisoldwood

Of course if you write an instrumentation message to a log file for every disk I/O request you'll probably spend more time measuring than doing useful work and so the level of measurement needs to be granular enough to capture something useful without being too intrusive or verbose. This may itself depend on how and where you are capturing your data as you probably don't want to be contending with the resource that you're trying to measure.

As a starting point I would expect to measure every call out to a remote service, whether that is a web service, database query, etc. Every remote call has so many other moving parts in play that it's almost certainly going to act up at some point. Reading and writing files to disk, especially when that disk is based remotely on a NAS/SAN/DFS is also a must, although the granularity would be to read/write the entire file. On top of those any intensive memory or CPU hogs should also get instrumented as a matter of course.

There is one scenario when I would consider discarding the measurements for a task and that's when an exception has been thrown. If an error has occurred you cannot tell how much of the operation was performed, which for a lost connection could be none, and so having lots of dubious tiny points in your data will only succeed in distorting it.

Types of instrument

You only need a few basic instruments to help you capture most of the key metrics. Some of those instruments are 'real', whereas others may need to be synthesized by using API's from the OS or language runtime.

Wall clock

By far the simplest instrument is the good old fashioned clock. The term 'wall clock' is often used to distinguish it from other internal clocks as it measures elapsed time as perceived by the user (or operator). These are probably the easiest to implement as virtually every runtime has the ability to report the current date and time, and so therefore you can calculate the difference.

The main thing you need to watch out for is the difference between the precision and the accuracy of the clock. Although you may be able to format the date and time down to 1 nanosecond that doesn't mean it's captured at that resolution. For example the .Net `System.DateTime` type has a precision measured in nanoseconds but is only accurate to around 16 ms [Lippert]. As things stand today I have found this good enough to capture the vast majority of 'big picture' measurements.

There are often other platform specific alternatives to the classic wall clock, some are old fashioned like the original `GetTickCount()` API in Windows which also only has a resolution of 10–16 ms on the NT lineage (it was a whopping 50 ms under 3.x/95). The Windows `QueryPerformanceCounter()` API in contrast has a much, much higher resolution, due to the hardware used to derive it, and it's from this that the .Net `StopWatch` type gains its sub-microsecond precision *and* accuracy [Lippert]. However, be warned that this impressive feat is heavily dependent on the hardware and is therefore more usable server-side than client-side.

One of the problems with any measurement taken in isolation is that you often need a little more knowledge about what the ‘size’ of the task actually was

Specialised clocks

Internally operating systems often track metrics around process and thread scheduling. One such bit of data Windows provides is the amount of time spent in the OS kernel as opposed to user space. Coupled with the elapsed time you can discover some useful facts about your process.

If it’s a high-performance computing (HPC) style engine that is expecting to be CPU bound, you might expect all its time to be spent in user space, but if it’s in neither you’re probably doing more I/O than you expect. I have observed a high amount of time in the kernel when there are a large number of exceptions bouncing around (and being ignored) or when the process is doing an excessive amount of logging. The converse can also be true about an I/O bound task that seems to be consuming too much CPU, such as during marshalling.

Network latency

Prior to Windows 2000 the clocks on Windows servers were notoriously bad, but with the addition of support for Kerberos the situation got much better (it had to) as servers now synchronised their clocks with a domain controller. With servers having a much better idea of what the current time is, you can begin to capture the network latency incurred by a request – this is the time it takes for the request to be sent and for the response to the received.

You can measure the latency by timing the operation from the client’s perspective (the remote end, server-wise) and also within the service handling the request (the local end, server-wise) and then calculating the difference. For example, if the client sees the request taking 10 seconds and the service knows it only spent 8 seconds processing it, the other 2 seconds must be spent in and around the infrastructure used to transport the request and response. Although you might not be able to correctly apportion the time to the request or response sides (as that relies on the absolute times) you can at least calculate the total latency which only relies on the relative differences.

This kind of metric is particularly useful anywhere you have a queue (or non-trivial buffering) in place as it will tell you how long a message has been stuck in the pending state. For example in a grid computing scenario there is usually a lot of infrastructure between your client code and your service code (which is executed by a remote engine process) that will queue your work and marshal any data to and from the client. However, if the queue has the ability to retry operations you will have to allow for the fact that the latency might also include numerous attempts. This is one reason why handling failures and retries client-side can be beneficial.

Memory footprint

Aside from the use or idleness of your CPUs, you’ll also want to keep track of the memory consumed by your processes. Sadly this is probably not tracked at a thread level like CPU times because a global heap is in use. However, single-threaded processes are often used for simplicity and resilience and so you might be in a position to track this accurately in some parts of the system.

On Windows you generally have to worry about two factors – virtual address consumption and committed pages. Due to the architecture of Windows’ heaps you can end up in the mysterious position of appearing to have oodles of free memory and yet still receive an out-of-memory condition. See my previous *Overload* article on breaking the 4 GB limit with 32-bit Windows processes for more details [Memory]. Suffice to say that process recycling is a common technique used with both managed *and native* code to work around a variety of heap problems.

For services with some form of caching built-in memory footprint metrics will help you discover if your cache eviction policy is working efficiently or not.

Custom measurements

One of the problems with any measurement taken in isolation is that you often need a little more knowledge about what the ‘size’ of the task actually was. If your workload varies greatly then you’ll not know if 1 second is fast, acceptable or excessive unless you also know that it had thousands of items to process. Then when you’re analysing your data you can scale it appropriately to account for the workload.

To help provide some context to the instrument measurements it’s worth accumulating some custom measurements about the task. These might include the number of items in the collection you’re reading/writing or the number of bytes you’re sending/receiving.

Causality

Every similar set of measurements you capture need to be related in some way to an operation and so you also need to capture something about what that is and also, if possible, some additional context about the *instance* of that operation. Once you start slicing and dicing your data you’ll want to group together the measurements for the same operation so that you can see how it’s behaving over time.

Additionally when you start to detect anomalies in your systems’ behaviour you’ll likely want to investigate them further to see what led to the scenario. Whilst the date and time the data was captured is a good start, it helps if have some kind of ‘transaction identifier’ that you can use to correlate events across the whole system. I described one such approach myself in a previous edition of *Overload* [Causality].

Output sinks

Once you’ve started accumulating all this data you’ll want to persist it so that you can chew over it later. There are various options, none of which are mutually exclusive, and so it might make sense to publish to multiple targets for different reasons – usually based around active monitoring or passive offline investigation.

One thing you’ll definitely want to bear in mind is how you handle errors when writing to an output sink fails. You almost certainly don’t want the failure to cause the underlying operation to also fail. If the sink is remote you could try buffering the data for a short while, but then you run the risk of destabilising the process, and eventually the machine [Memory], if you buffer too much for too long.

Log file

The humble (text based) log file is still with us, and for good reason. Many of us have honed our skills at manipulating these kinds of files and as long as you keep the format simple, they can provide a very effective medium.

The single biggest benefit I see in storing your instrumentation data in your log files is the locality of reference – the data about the operation's performance is right there alongside the diagnostic data about what it was doing. When it comes to support queries you've already got lots of useful information in one place.

One method to help with singling out the data is to use a custom severity level, such as 'PRF' (for performance), as this will use one of the key fields to reduce the need for any complex pattern matching in the initial filtering. The operation itself will need a unique name, which could be derived from the calling method's name, so that you can aggregate values. And then you'll need to the data encoded in a simple way, such as key value pairs. Here is an example log message using that format:

```
2013-06-26 17:54:05.345 PRF [Fetch Customers]
Customer:Count=1000;WallClock:DurationInMs=123;
```

Here we have the operation name – `Fetch Customers` – and two measurements – the number of items returned followed by the elapsed time in milliseconds. Although the simpler key names `Count` and `Duration` may appear to suffice at first, it's wise to qualify them with a namespace as you may have more than one value captured with the same name. Including the units also helps if you capture the same value in two different formats (e.g. milliseconds and `TimeSpan`). This might seem redundant, but when reading a log for support reasons it helps if you don't have to keep doing arithmetic to work out whether the values are significant or not.

Classic database

Where better to store data than a database? Your system may already capture some significant timings in your main database about the work it's doing, such as when a job was submitted, processed and completed. And so you might also want your instrumentation data captured alongside it to be in one place. Some people are more comfortable analysing data using SQL or by pulling portions of data into a spreadsheet as not everyone is blessed with elite command line skills and intricate knowledge of the Gnu toolset.

Of course when I say 'classic' database I don't just mean your classic Enterprise-y RDBMS – the vast array of NOSQL options available today are equally valid and quite possibly a better fit.

Round Robin Database

For analysis of long term trends the previous two options probably provide the best storage, but when it comes to the system's dashboard you'll only be interested in a much shorter time window, the last 10 minutes for example. Here you'll want something akin to a fixed-size circular buffer where older data, which is of less interest, is either overwritten or pruned to make room for newer values.

There are a few variations on this idea; one in particular is the Round Robin Database (or RRDtool [RRDtool]). This prunes data by using a consolidation function to produce aggregate values that lower the resolution of older data whilst still maintaining some useful data points.

Performance counters

Another variation of the circular buffer idea is to expose counters from the process itself. Here the process is responsible for capturing and aggregating data on a variety of topics and remote processes collect them directly instead. This has the advantage of not requiring storage when you don't need it as you then log only what you need via a remote collector. However, back at the beginning I suggested you probably want to capture everything you can afford to, so this benefit seems marginal.

```
public Customers FetchCustomers()
{
  const string operation = "Fetch Customers";
  var stopwatch = new Stopwatch();
  var cpuMonitor = new CpuMonitor();
  // Do the thing that fetches the customers.
  cpuMonitor.Stop();
  stopwatch.Stop();
  var measurements = new Measurements
  {
    stopwatch.Measurements,
    cpuMonitor.Measurements
  };
  sink.Write(operation, measurements);
  return customers;
}
```

Listing 1

One word of caution about the Windows performance counter API – it's not for the faint of heart. It is possible that it's got better with recent editions (or via a 3rd party framework) but there are some interesting 'issues' that lie in wait for the unwary. That said the concept still remains valid, even if it's implemented another way, such as using a message bus to broadcast a status packet.

A simple instrumentation framework

Anyway, enough about the theory what does this look like in code. You shouldn't be surprised to learn it's not rocket science. Basically you just want to start one or more instruments before a piece of functionality is invoked, and stop them afterwards. Then write the results to one or more sinks.

If you're working with C++ then RAII will no doubt have already featured in your thoughts, and for C# developers the Dispose pattern can give a similar low noise solution. The effect we want to achieve looks something like Listing 1 (I'm going to use C# for my examples).

This code, which is largely boilerplate, adds a lot of noise to the proceedings that obscures the business logic and so we should be able to introduce some sort of façade to hide it. What I'd be looking for is something a little more like Listing 2.

Hopefully this looks far less invasive. Through the use of reflection in C# (or pre-processor macros in C++) we should be able to determine a suitable name for the operation using the method name.

By switching to flags for the instrument types, we've factored out some noise but made customising the instruments harder. Luckily customisation is pretty rare and so it's unlikely to be a problem in practice. The façade could also provide some common helper methods to simplify things further by wrapping up the common use cases, for example:

```
using (MeasureScope.WithWallClock())
```

or

```
using (Measure.CpuBoundTask())
```

The one other minor detail that I have so far glossed over is how the output sink is determined. For cases where it needs to be customised it can be passed in from above, but I've yet to ever need to do that in practice. Consequently I tend to configure it in `main()` and store it globally, then reach out to get it from behind the façade.

```
public Customers FetchCustomers()
{
  using (MeasureScope.With(Instruments.WallClock |
                           Instruments.CpuMonitor))
  {
    // Do the thing that fetches the customers.
  }
}
```

Listing 2

```
public static class MeasureScope
{
  public static Controller With
     (Instruments instruments)
  {
    string operation =
       new StackFrame(1).GetMethod().Name;
    return With(operation, instruments);
  }

  public static Controller With(string operation,
     Instruments instruments)
  {
    return new Controller(operation, instruments);
  }
}
```

Listing 3

```
public static class MeasureScope
{
  . . .
  public static Controller WithWallClock()
  {
    string operation =
       new StackFrame(1).GetMethod().Name;
    return With(operation, Instruments.WallClock);
  }
}
```

Listing 4

The facade

The façade used by the client code can be implemented like Listing 3.

Retrieving the calling method name in C# (prior to C# 4.0) relies on walking the stack back one frame. Of course if you write any wrapper methods you'll need to put the stack walking in there *as well* or you'll report the wrapper method name instead of the business logic method name (see Listing 4).

The same rule applies if you've used closures to pass your invocation along to some internal mechanism that hides grungy infrastructure code, such as your Big Outer Try Block [Errors]. See Listing 5.

Bitwise flags seem to be a bit 'old school' these days, but for a short simple set like this they seem about right, plus you have the ability to create aliases for common permutations (see Listing 6).

```
public Customers FetchCustomers()
{
  return Invoke(() =>
  {
    // Do the thing that fetches the customers.
  });
}
public T Invoke<T>(Func<T> operation)
{
  string operationName =
     new StackFrame(1).GetMethod().Name;
  using (MeasureScope.With(operationName,
                           Instruments.WallClock))
  {
    return operation();
  }
}
```

Listing 5

```
[Flags]
public enum Instruments
{
  WallClock = 0x01,
  CpuMonitor = 0x02,
  MemoryMonitor = 0x04,

  CpuBoundTaskMonitor = WallClock | CpuMonitor,
}
```

Listing 6

The instrument controller

The behind-the-scenes controller class that starts/stops the instruments and invokes the outputting is only a tiny bit more complicated than the façade (Listing 7).

The sink

The measurements that are returned from the instruments are just string key/value pairs. Given their different nature – datetime, timespan, scalar, etc. – it seems the most flexible storage format. I've implemented the sink in Listing 8 as a simple wrapper that writes to a log file through some unspecified logging framework.

Here I've assumed that the Log façade will not allow an exception to propagate outside itself. That just leaves an out-of-memory condition as

```
public class Controller : IDisposable
{
  public static ISink DefaultSink { get; set; }

  public Controller(string operation,
                    Instruments instrumentFlags)
  {
    var instruments = new List<IInstrument>();
    if ((instrumentFlags & Instruments.WallClock)
        != 0)
     instruments.Add(new WallClock());
    . . .
    _operation = operation;
    _instruments = instruments;
    _sink = DefaultSink;

    _instruments.ForEach((instrument) =>
    {
      instrument.Start();
    });
  }

  public void Dispose()
  {
    _instruments.ForEach((instrument) =>
    {
      instrument.Stop();
    });
    var measurements =
       new List<KeyValuePair<string, string>>();
    foreach (var instrument in _instruments)
       measurements.AddRange
       (instrument.Measurements);
    _sink.Write(_operation, measurements);
  }

  private readonly string _operation;
  private List<IInstrument> _instruments;
  private ISink _sink;
}
```

Listing 7

```
public class LogFileSink : ISink
{
  public void Write(string operation,
     IEnumerable<KeyValuePair<string,
                            string>> measurements)
  {
    var buffer = new StringBuilder();

    foreach (var measurement in measurements)
      buffer.AppendFormat("{0}={1};",
         measurement.Key, measurement.Value);
    Log.Performance("[{0}] {1}", operation,
                    buffer);
  }
}
```

Listing 8

the only other sort of failure that might be generated and I'd let that propagate normally as it means the process will likely be unstable.

The instruments are equally simple – a pair of methods to control it and a property to retrieve the resulting metric(s). Like all 'good' authors I've chosen to elide some error handling in the `Start()` and `Stop()` methods to keep the code simple (Listing 9), but suffice to say you probably want to be considering what happens should they be called out of sequence.

Earlier I mentioned that I would avoid writing measurements taken whilst an exception was being handled; that can be achieved in C# with the following check [Exception]. See Listing 10.

```
public interface IInstrument
{
  void Start();
  void Stop();
  Measurements Measurements { get; }
}

public class WallClock : IInstrument
{
  public Measurements Measurements {
     get { return _measurements; } }

  public void Start()
  {
    _measurements.Clear();
    _startTime = DateTime.Now;
  }

  public void Stop()
  {
    var stopTime = DateTime.Now;
    var difference = stopTime - _startTime;

    _measurements.Add
      (new KeyValuePair<string, string>
      ("WallClock:Duration",
       difference.ToString()));
  }

  private DateTime _startTime;
  private Measurements _measurements =
     new Measurements();
}
```

Listing 9

```
public static bool IsHandlingException()
{
  return
    (Marshal.GetExceptionPointers() !=
       IntPtr.Zero
     || Marshal.GetExceptionCode() != 0);
}

. . .

if (!IsHandlingException())
{
  var measurements =
     new List<KeyValuePair<string, string>>();
  foreach (var instrument in _instruments)
    measurements.AddRange
      (instrument.Measurements);
  _sink.Write(_operation, measurements);
}
```

Listing 10

Summary

This article has shown that it's relatively simple to add some instrumentation calls within your own code to allow you to track how it's performing on a continual basis. It explained what sort of measurements you might like to take to gain some useful insights into how your service calls and computations are behaving. It then followed up by showing how a simple framework can be written in C# that requires only out-of-the box techniques and yet still remains fairly unobtrusive.

Once in place (I recommend doing this as early as possible, as in right from the very start) you'll be much better prepared for when the inevitable happens and the dashboard lights up like a Christmas tree and the phones start buzzing. ■

Acknowledgements

Thanks to Tim Barrass for introducing me to the Round Robin Database as a technique for persisting a sliding window of measurements and generally helping me refine this idea. Additional thanks as always to the *Overload* advisors for sparing my blushes by casting their critical eye and spotting my mistakes.

References

[Causality] Chris Oldwood, 'Causality – relating distributed diagnostic contexts', *Overload* 114

[Errors] Andy Longshaw and Eoin Woods, 'The Generation, Management and Handling of Errors (Part 2)', *Overload* 93

[Exception] Ricci Gian Maria, 'detecting if finally block is executing for an manhandled exception', http://www.codewrecks.com/blog/index.php/2008/07/25/detecting-if-finally-block-is-executing-for-an-manhandled-exception

[Hoare] http://en.wikipedia.org/wiki/Program_optimization

[Lippert] Eric Lippert, 'Precision and accuracy of DateTime', http://blogs.msdn.com/b/ericlippert/archive/2010/04/08/precision-and-accuracy-of-datetime.aspx

[Memory] Chris Oldwood, 'Utilising more than 4GB of memory in 32-bit Windows process', *Overload* 113

[RRDtool] http://oss.oetiker.ch/rrdtool

Portable String Literals in C++

How hard can it be to make a file in C++
with international text literals in its name?
Alf Steinbach shows us.

C++ lacks a built-in or library-provided character encoding value type that reflects the main conventions for the encoding of international text literals, API arguments and, for *nix, external text, namely UTF-8 for *nix[1] and UTF-16 for Windows[2]. As a consequence, standard C++ code that works fine in *nix fails outright or produces erroneous results in Windows, as exemplified below. Portable code deals with this by converting strings at run time (efficiency/complexity cost), and by employing brittle conventions (programmer's time cost), and in teaching the problem is largely just ignored, letting students produce programs that, for example, are unable to deal with their Norwegian names (cost of negative perception of the language – a language so primitive that it can't even handle text).

The C++11 standard added the literal prefixes **`u8`**, **`u`** and **`U`** that specify known sizes and encodings, respectively UTF-8, UTF-16 and UTF-32. But no matter whether one chooses[3] **`u8`**, **`u`** or **`U`**, the code needs added runtime conversions on one or the other platform. Exacerbating the situation, the C++ standard library supports only char-based narrow strings in filenames and exception messages, which, for example, means that the current Boost filesystem library[4] can't access many Windows files – the main desktop platform's files – when it's used with the g++ compiler.

Happily the limited issue of suitable *original string data* for portable code, with UTF-8 for *nix and UTF-16 for Windows, can be dealt with 'simply' by using macros that adjust the form of literals. Proper core language support would be better still, but a suitable macro + supporting functionality addresses the problem at compile time, most efficiently, with a single common portable notation. And happily, when the macro always produces a Unicode literal then there is no problem with different character sets (only the encoding differs across systems), and when the macro produces a distinctly typed result[5] then there is no problem with inadvertent mixing of incompatible encodings such as Windows ANSI and UTF-8.

CP 1252 (Windows ANSI Western ext. of Latin 1)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
00	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	?
80	€	?	,	ƒ	"	…	†	‡	ˆ	‰	Š	‹	Œ	?	Ž	?
90	?	'	'	"	"	○	–	–	˜	™	š	›	œ	?	ž	Ÿ
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	"		–	®	¯
B0	°	±	²	³	´	µ		·	¸	¹	º	"	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figure 1

Relevant character encodings and terminology

In the middle 1960s, US government computers employed a large number of incompatible character encodings, which reduced interoperability and added needless costs and hassle. The *American National Standards Institute*, **ANSI**[6], therefore created a more general single-byte character encoding which became known as **ASCII**, the *American Standard Code for Information Interchange*. And on March 11 1960, President Lyndon B. Johnson approved ASCII as a US federal standard.

The ASCII code was English only. So, while ASCII largely solved the Tower of Babel problem within the English-speaking world, the same problem now resurfaced in the rest of the Western world. From this arose a single-byte ASCII extension intended to serve the needs of Western countries, called **ISO Latin 1**.

The first Windows versions were based on a Microsoft extension of Latin 1 called Windows ANSI. Today that term has taken on a more general meaning (discussed below), and the original Windows ANSI encoding is now known more precisely as **Windows ANSI Western**, or codepage 1252. A Windows **codepage** is a number that designates a character encoding in Windows; reportedly it originally referred to a tabular display of a single-byte encoding, literally a 'code page', like Figure 1.

[1] I haven't found any authoritative statements or data about *nix character encodings other than Markus Kuhn's Unix Unicode FAQ maintaining that "UTF-8 is the way in which Unicode is used under Unix, Linux, and similar systems". In Nov. 2011 I asked about it on Stack Exchange, but alas without a definitive answer. If you're interested in various opinions and details then check out that question at: http://unix.stackexchange.com/questions/24529/most-common-encoding-for-strings-in-c-in-linux-and-unix.

[2] The main Windows C++ compiler, Visual C++, supports only Windows ANSI as a narrow C++ execution character set, and UTF-16 for wide string literals. Windows ANSI cannot portably encode international text and incurs conversion costs. UTF-16, in Windows called 'Unicode', is therefore used by the vast majority of projects, and is the default in Visual Studio projects.

[3] At the time of writing, Visual C++ in version 11.0 does not yet support the C++11 **`u8`**, **`u`** and **`U`** prefixes.

[4] As of Boost version 1.54, released during the writing of this article.

[5] For standard C++ the **`u8`** prefix does produces a char based literal.

[6] At the time known as the United States of America Standards Institute, USASI; the name was changed to the American National Standards Institute, ANSI, in 1969.

Alf Steinbach learned Basic on a Tandberg EC-10 in 1980. He's worked as a senior consultant with Kantega and Accenture, as a lecturer (Norw. 'amanuensis') at Nordland University, and as a vocational school teacher at Bodin VGS. He's a moderator of Usenet group comp.lang.c++.moderated and was awarded Microsoft's Most Valued Professional in Visual C++ in 2012. He can be contacted at alf.p.steinbach@gmail.com

In Figure 1, table rows **`00H`** through **`70H`** constitute original ASCII. Rows **`80H`** through **`F0H`** were added in ISO Latin 1, except that in ISO Latin 1 rows **`80H`** and **`90H`** are undefined characters. The characters shown in rows **`80H`** and **`90H`** in Figure 1, including the Euro sign €, are the Windows ANSI Western extension of Latin 1 (in original Windows ANSI there was, of course, no Euro sign, since there was no Euro).

At some point[7] Windows started supporting local variants of Windows ANSI Western, e.g. with Cyrillic or Greek characters. Whatever narrow encoding used in the GUI, reported by **`GetACP()`**, is known as **Windows ANSI**, as opposed to the **OEM** character encoding which is the local chosen variant of the original IBM PC encoding, used in text consoles. The different variants of Windows ANSI ensures a global Tower of Babel problem, while the use of two incompatible narrow character encodings on the same machine, namely OEM and Windows ANSI, ensures that there's also a local Tower of Babel problem – at least for Windows users.

To address the general Tower of Babel problem a number of leading computer industry firms cooperated on developing a ‘universal’ character encoding, an extension of ISO Latin-1 which became known as **Unicode**. Original Unicode was a fixed size 16-bit per character encoding, and 32-bit Windows NT, introduced in 1992, was based on this 16-bit encoding. However, 16 bits didn’t suffice for e.g. Chinese ideograms, so Unicode was extended to 21 bits per character, and for the existing software the added characters were to be represented as *pairs* of 16-bit values, called **surrogate pairs**. Today this encoding is known as **UTF-16**, and the original 16-bit per character representation is known as **UCS-2** (two bytes per character). Windows’ console subsystem API supports copying of rectangular areas of console windows, but only with 16 bits per character, so console windows are effectively limited to UCS-2, while the rest of Windows is now generally UTF-16.

32-bit Windows includes many wrapper functions that automatically convert from legacy code’s Windows ANSI to the basic API's UTF-16, and back. Typically there is an UTF-16 based function called **`FooW`**, and a Windows ANSI wrapper called **`FooA`**. This legacy code support extends to the graphical user interface. However, with respect to *window messages* (small fixed format data packets used to control windows) Microsoft duplicated its file access API blunder, by using configurable encoding expectations. Pointers in window messages are untyped, and when a given message contains a pointer to a string, then that untyped string is encoded as Windows ANSI or UTF-16 depending on the particular window’s configuration… Thus the terms **ANSI window** and **Unicode window**. ‘Windows ANSI’ refers to the narrow character encoding used in the graphical user interface and reported by the **`GetACP`** API function, while ‘ANSI window’[8] refers to a window configured to expect and produce Windows ANSI encoded strings in its window messages.

UTF-8, very popular in *nix and for web pages, is an ASCII extension that encodes all of Unicode by using a variable number of bytes per character.

The inefficiency, complexity and current real world non-portability of standard C++ string literals

Let’s check how some basic, completely standard and therefore presumably automagically[9] portable C++ source code fares in Windows (see Listing 1).

Compiling with the MinGW g++ 4.7.2 compiler (a Windows build of the GNU toolchain’s C++ compiler), running the program and checking the result (see Figure 2).

```
// Source encoding: UTF 8 with BOM (necessary for
// Visual C++).
#include <assert.h>     // assert
#include <fstream>      // std::ofstream
auto main() -> int
{
  auto const filename = "π.recipe";
  // A pie recipe. :-)
  std::ofstream f( filename );
  assert( "File creation" && !!f );
}
```

Listing 1

```
> del a.exe *.recipe 2>nul &^
More? g++ cplusplus_stdlib_version.cpp &&^
More? a.exe && dir /b *.recipe
Ï€.recipe
```

Figure 2

This produced an *erroneous* result, a filename different from the specified one, namely `Ï€.recipe` instead of the specified `π.recipe`.

In some cases, but mostly with Microsoft’s Visual C++, this happens because an UTF-8-encoded source is misinterpreted as a Windows ANSI-encoded source (so it’s worth checking that the source encoding is correct!), but the reason above is that the MinGW g++ compiler and its standard library implementation have *different opinions* about what the C++ **execution character set** is or should be.

The g++ compiler defaults to UTF-8, which is the *de facto* standard narrow string encoding in *nix, while its standard library implementation, presumably delegating to Microsoft’s runtime library, defaults to **Windows ANSI**, which is the *de facto* standard narrow string encoding in Windows programming.

Adjusting the g++ compiler’s execution character set to match its standard library’s expectations will in general not help in obtaining a correct result, since most variants of Windows ANSI lack the lowercase Greek π character. But it does convert the silent erroneous result behaviour to a work-saving up-front *compilation error*. So, when using g++ in Windows, to avoid possible silent erroneous results do add the **`-fexec-charset=cp`***YourANSICodepageNumber* option, e.g. as shown in Figure 3.

So, how about using Windows’ own main compiler, Microsoft’s Visual C++, for this code? (See Figure 4.)

```
> del a.exe *.recipe 2>nul &^
More? g++ cplusplus_stdlib_version.cpp -fexec-
charset=cp1252 &&^
More? a.exe && dir /b *.recipe
cplusplus_stdlib_version.cpp: In function 'int
main()':
cplusplus_stdlib_version.cpp:7:27: error:
converting to execution character set: Illegal
byte sequence? Nice up-front compilation error.
cplusplus_stdlib_version.cpp:7:27: error: unable
to deduce 'const auto' from '<expression error>'
```

Figure 3

[7] According to Wikipedia’s codepage article, at http://en.wikipedia.org/wiki/Code_page, DOS gained codepage support in version 3.3, in 1987, while the first version of Windows was released in 1985.

[8] The term ‘ANSI Windows’ was used by one reviewer, who conflated it with ‘Windows ANSI’ (encoding) and ‘ANSI window’ (configuration). This term can appear to be used when ‘ANSI’ is used as a qualification. E.g. ‘ANSI Windows codepages’, meaning ‘ANSI (Windows codepages)’, the codepages that can be used as Windows ANSI, i.e., that can be returned by GetACP.

[9] … to compilers that support UTF-8 source code, which all the relevant compilers do. More in general *portable* for C++ means portable within the limits of the language implementation that one ports to. E.g., putting this to the point, the C++ standard does not specify the size of **`bool`** so that frivolous use of **`bool`** type local variables conceivably could exceed the available memory, yet such code is portable. One reviewer has however argued that C++ only supports source code with the characters formally guaranteed to be supported, i.e. only pure ASCII source code with no “$” signs, portably.

```
> del b.exe *.recipe 2>nul &^
More? cl cplusplus_stdlib_version.cpp /Fe"b.exe"
&&^
More? b.exe && dir /b *.recipe
cplusplus_stdlib_version.cpp
cplusplus_stdlib_version.cpp(7) : warning C4566:
character represented by universal character name
'\u03C0' cannot be represented in the current
code page (1252)
Assertion failed: "File creation" && !!f, file
cplusplus_stdlib_version.cpp, line 10
```

Figure 4

Here Visual C++ unfortunately accepted the source code, but happily the program then produced a *runtime error*. This is far better than g++'s default silent erroneous result, but it's rather ungood news for the portability of pure standard C++ source code as of 2013. Currently, the two main free C++ compilers for Windows are Visual C++ (Microsoft) and g++ (GNU), and as exemplified above neither of them support UTF-8 string constants for e.g. filenames.

It's not that Windows can't handle the `π.recipe` filename. Unicode filenames are supported by the Windows API, they're supported by Windows-specific library extensions such as **`_wfopen`**, and there's no problem creating or accessing such a file in e.g. Java or C# or Python 3. The problem is that such files can't be accessed using only portable, pure **standard C++** source code, and also that even if C++ had the wide string support that is *de facto* standard in Windows, using it directly for portable code would be needlessly inefficient in *nix; and the part of that problem that I address here is the support for string literals.

How Boost filesystem doesn't help

After the C++ standard library the next place to look for general functionality is usually the Boost library. For our example code the relevant sub-library is the Boost filesystem library. The Boost filesystem library, but apparently sans the **`boost::filesystem::ofstream`** class[10] that's used below, is slated for inclusion in *C++ Technical Report 2* (TR2)[11], which effectively means also in the next C++ standard.

However, the Boost filesystem library does not offer or visibly use[12] portable system dependent strings, and so for portable code, with the Boost filesystem library a filename such as `"π.recipe"` has to be specified as a wide string, like `L"π.recipe"`.

Since it's impractical to deal with two or more different string formats, one would then presumably standardize on using **`wchar_t`** based strings for all portable strings. This then incurs a *string conversion cost* in *nix, in the worst case for most every API call involving strings, which is counter to the general C++ principle of not paying for what you don't use. This cost (and others) is meant to buy a correct result, so let's check whether the Boost filesystem library actually does produce a correct result? (Listing 2)

```
// Source encoding: UTF 8 with BOM
// (necessary for Visual C++).
#include <assert.h>                  // assert
#include <boost/filesystem/fstream.hpp>
 // boost::filesystem::ofstream
namespace bfs = boost::filesystem;
auto main()
  -> int
{
  auto const filename = L"π.recipe";
  // A pie recipe. :-)
  bfs::ofstream f( filename );
  assert( "File creation" && !!f );
}
```

Listing 2

Compiling the program with the Visual C++ 11.0 compiler, using boost 1_54 filesystem and system libraries, and running gives the result shown in Figure 5.

```
> del b.exe *.recipe 2>nul &^
More? cl cplusplus_boost_version.cpp /MD /
Fe"b.exe" /I"%boost_pincludes%" /link
%msvc_link_bfs% &&^
More? b.exe && dir /b *.recipe
cplusplus_boost_version.cpp
π.recipe
```

Figure 5

Well, that worked nicely! At an encoding conversion cost for *nix, and at the general cost of using Boost. But how about building with MinGW g++ 4.7.2? (See Figure 6.)

```
> del a.exe *.recipe 2>nul &^
More? g++ cplusplus_boost_version.cpp -fexec-
charset=cp1252 %gnuc_using_bfs% &&^
More? a.exe && dir /b *.recipe
Assertion failed: "File creation" && !!f, file
cplusplus_boost_version.cpp, line 12

This application has requested the Runtime to
terminate it in an unusual way.
Please contact the application's support team for
more information.
```

Figure 6

The Boost filesystem library takes advantage of a Visual C++ extension to the standard library, namely a **`wchar_t`** based **`ofstream`** constructor, when the library is built with Visual C++. The g++ compiler's standard library implementation has a more clean extension, a **`std::streambuf`** subclass that can be initialized from a C **`FILE*`**. And a possibly more efficient workaround for the standard library's lack of Unicode filename support, which works with any compiler, is Windows' so called 'short' or 'DOS' or '8+3' filenames, which *were* used in Boost filesystem version 2.[13] But the current Boost filesystem library simply doesn't support Windows C++ compilers in general. For Windows it now only provides full functionality, the ability to portably access files with names such as `π.recipe`, when it's used with Visual C++ or a compiler with the same standard library extensions…

If the Boost filesystem library is just made part of the C++ standard we'll then have an absurdity: a part of the standard library making essential use of wide string based constructors, and thus effectively requiring them[14] of all Windows standard library implementations, without having them standardized and available to all.

[10] Judging by the N3693 draft Technical Specification at http://isocpp.org/files/papers/N3693.html

[11] Wikipedia lists the TR2 proposals at http://en.wikipedia.org/wiki/C++_Technical_Report_1#Technical_Report_2

[12] Internally the **`boost::filesystem::path`** class uses a representation of international text where the public definition **`value_type`** corresponds to the 'raw' encoding value type discussed in this article, with UTF-8 for *nix and UTF-16 for Windows. Presumably with C++14 (if that should be the next C++ standard), this article's **`Raw_syschar`** could be defined as **`std::filesystem::path::value_type`**.

[13] I filed a ticket about its disappearance in 2011, #6065 available at https://svn.boost.org/trac/boost/ticket/6065

[14] The N3693 draft Technical Specification contains this wording in its §8.4.6: "Implementations of the standard library for systems where **`string_type`** is **`wstring`**, such as Windows, are encouraged to provide an extension to existing standard library file stream constructors and open functions that adds overloads that accept **`wstring`**s for file names. Microsoft and Dinkumware already provide such an extension."

```
#ifdef _WIN32
   namespace cppx{ typedef wchar_t Raw_syschar; }
   // Implies UTF-16 encoding.
#  define CPPX_WITH_SYSCHAR_PREFIX( lit ) L##lit
#else
   namespace cppx{ typedef char Raw_syschar; }
   // Implies UTF-8 encoding.
#  define CPPX_WITH_SYSCHAR_PREFIX( lit ) lit
#endif
```

Listing 3

Summing up, since using Boost filesystem as a portability layer requires using wide strings it incurs an efficiency/complexity cost on *nix, a cost that in a great many cases buys you nothing. And worse, the current version doesn't even produce correct results with g++ in Windows, thus not providing the goods that the cost was meant to cover. Thus, as of this writing (July 2013) Boost filesystem is not a solution.

Strongly typed system dependent literals

In the same way that C++ integer types such as `int` are portable because their sizes depend on the system, one can define a character encoding value type[15] that's portable because its size and assumed encoding depends usefully on the system. I.e., a system dependent character encoding value type, which is portable precisely because it's system dependent – just as with the `int` type etc. It can look like Listing 3.

The `_WIN32` macro, a *de facto* standard in Windows C and C++ programming, is defined for both 32-bit and 64-bit Windows programming. There is one problem with the `Raw_syschar` type, though, namely that it's just a synonym for another type that it isn't **distinct**. For example, one cannot define a distinct `std::basic_string` specialization for it. It's practically possible[16] to define a distinct `Raw_syschar` type as a class, but in order to be able to put that inside a constructor-free `union` – as can happen with the short string optimization[17], where the `union` then occurs in the `std::basic_string` implementation – it would need to be without any user defined constructor. That means that it would need to expose a public data member, which is somewhat unclean, and different from use of basic types like `char` and `wchar_t`.

Happily with C++11, and with Visual C++ for a some time before that as a language extension, one can define an `enum` type with a specified underlying representation (this and all the following definitional code is in namespace `cppx`):

```
  enum Syschar : Raw_syschar {};
```

This produces a type with very much the desired properties[18] of a character encoding value type, namely, it's a distinct type that supports all the built-in comparison operators, and it provides an implicit conversion to integer.

And just by defining a `std::char_traits` specialization this type supports a distinct specialization of `std::basic_string`, if you should want that. Such a `std::char_traits` specialization is just a collection of static member functions that forward to the corresponding functions for the raw character type. However, such forwarding functions require general conversion between raw and typed characters and character strings - e.g. the following three **typed** functions for converting to strongly typed form, and corresponding **raw** functions the other way (Listing 4).[19]

This looks trivial, yes?

```
auto typed( Raw_syschar const c )
  CPPX_NOEXCEPT
  -> Syschar
{return static_cast< Syschar const >( c );}
auto typed( Raw_syschar* const s )
  CPPX_NOEXCEPT
  -> Syschar*
{return reinterpret_cast< Syschar*>( s );}
auto typed( Raw_syschar const* const s )
  CPPX_NOEXCEPT
  -> Syschar const*
{return reinterpret_cast< Syschar const* >( s );}
```

Listing 4

```
template< Size n >
auto typed( Raw_syschar const (&a)[n] )
  CPPX_NOEXCEPT
  -> Syschar const (&)[n]
{return reinterpret_cast< Syschar
   const (&)[n] >( a ); }
```

Listing 5

```
namespace detail {
  ...
  inline
  auto typed( Raw_syschar const* const& s,
              Pointer_kind = Pointer_kind() )
    CPPX_NOEXCEPT
    -> Syschar const* const&
  { return
    reinterpret_cast< Syschar const* const& >
    ( s ); }
  template< Size n >
  auto typed( Raw_syschar const (&a)[n],
              Array_kind = Array_kind() )
    CPPX_NOEXCEPT
    -> Syschar const (&)[n]
  { return reinterpret_cast
     < Syschar const (&)[n] >( a ); }
}  // namespace detail
```

Listing 6

Unfortunately, in order to later be able to construct a class type string very efficiently from a literal, it's very desirable to also have a function template like Listing 5, but this function template can then never be implicitly selected. The reason is that for an array type actual argument of any given size the corresponding specialization would not offer a better argument conversion than the pointer argument function. With the specialization the call would therefore be ambiguous. And then the C++11 standard decrees in its §13.3.3/1 fifth dash that F1 is a better function than F2 if "F1 is a non-template function and F2 is a function template specialization", which for the above functions means that the pointer argument function will always win.

My chosen fix is to route all calls to functions in a given set (e.g. `typed()` calls) via a single function template. The template just checks the actual argument type and dispatches the real work. To enable the dispatch call's function selection each of the typed overloads, and also each of the raw overloads, is outfitted with a defaulted nameless **dummy argument** that identifies the general kind of actual argument (see Listing 6).

[15] The `wchar_t` type can be argued to be such a type, but it's impractical for the purpose of portability.

[16] While it's not guaranteed by the C++ standard, as far as I know there's no compiler that by default will yield `sizeof(T)` > 1 when `T` is a POD class with just a single `char` data member.

[17] The last time I checked, two or three years ago, it did happen with Visual C++'s `std::string`.

[18] If names of e.g. control characters are desired then one can use an `enum class` in order to support easy name qualification, but for this article's exposition `enum class` would not have a purpose.

[19] Here `CPP_NOEXCEPT` is a macro that depending on the compiler is defined as C++11 `noexcept` (e.g. for g++ and clang) or C++03 `throw()` (for Visual C++ 11.0 and earlier).

```
template< class Arg >
auto typed( Arg const& arg )
  CPPX_NOEXCEPT
  -> decltype( detail::typed( arg,
    typename Type_kind_<Arg>::T() ) )
{ return detail::typed( arg,
    typename Type_kind_<Arg>::T() ); }
```

Listing 8

```
#pragma once
// Copyright (c) 2013 Alf P. Steinbach
// Mostly this is to enable a workaround for
// ordinary overload resolution.
#include <rfc/cppx/core/Size.h>  // cppx::Size
namespace cppx {
  enum Value_kind {};
  enum Pointer_kind {};
  enum Array_kind {};
  template< class Type >
  struct Type_kind_ { typedef Value_kind T; };
  template< class Type >
  struct Type_kind_<Type*> {
     typedef Pointer_kind T; };
  template< class Type >
  struct Type_kind_<Type* const> {
     typedef Pointer_kind T; };
  template< class Type, Size n >
  struct Type_kind_< Type[n] > {
     typedef Array_kind T; };
}  // namespace cppx
```

Listing 9

```
// Source encoding: UTF 8 with BOM (necessary
// for Visual C++).
#include "x/ofstream.h"     // x::ofstream
#include <assert.h>         // assert

auto main() -> int
{
  using cppx::typed;
  // A pie recipe. :-)
  auto const filename = typed
     ( CPPX_WITH_SYSCHAR_PREFIX( "π.recipe" ) );
  x::ofstream f( filename );
  assert( "File creation" && !!f );
}
```

Listing 10

The function template for this set of functions, through which all typed calls go (Listing 8), where **Type_kind_** is part of the small machinery that checks the argument type (see Listing 9).

Listing 10 is the file creation program again, but now using **Syschar** directly (only the machinery shown so far), producing a correct result. The just-for-this-example ad hoc header **x/ofstream.h** defines a subclass of **std::ofstream** called **x::ofstream** that provides a **Syschar**-based constructor by employing compiler-specific functionality. The necessity of compiler-specific or at least system-specific code for such basic functionality indicates to me that this area of functionality belongs in the standard.

But as the declaration of **filename** in Listing 10 shows, direct use of the conversion functionality defined so far yields *rather verbose* specifications of literal strings…

To support more concise usage expressions I therefore define two further macros, **CPPX_U** to express a typed literal and **CPPX_RAW_U** to express an untyped one (Listing 11).

```
#define CPPX_AS_SYSCHAR( lit ) \
 ::cppx::typed( CPPX_WITH_SYSCHAR_PREFIX( lit ) )

#define CPPX_U      CPPX_AS_SYSCHAR
#define CPPX_RAW_U  CPPX_WITH_SYSCHAR_PREFIX
```

Listing 11

```
// Source encoding: UTF 8 with BOM
// (necessary for Visual C++).
#include "x/ofstream.h"     // x::ofstream
#include <assert.h>         // assert
auto main() -> int
{
  auto const filename = CPPX_U( "π.recipe" );
  // A pie recipe. :-)
  x::ofstream f( filename );
  assert( "File creation" && !!f );
}
```

Listing 12

And with **CPPX_U** the file creation program looks, to my eyes, acceptable (Listing 12).

When it's compiled for Windows this program uses UTF-16 encoded **wchar_t** based strings, and when it's compiled for *nix it uses UTF-8 encoded **char** based strings. Unlike the C++ standard library and unlike Boost filesystem this ensures maximum efficiency for API calls, i.e. no runtime encoding conversion. And also unlike the C++ standard library and unlike Boost filesystem, with the necessary higher level functional support such as exemplified by **x::ofstream**, it provides access to all valid filenames on each system, lets students almost effortlessly write portable basic C++ programs that can handle Norwegian student names, etc.

Summary and final considerations

Standard C++11 does not provide the means to access Windows files in general, because the filenames can't be expressed as Windows ANSI encoded **char** based strings. The Boost filesystem library, slated for inclusion in TR2, imposes an efficiency cost for portable code used in *nix by requiring portable strings to be **wchar_t** based. And in Windows the Boost filesystem library only supports general Unicode filenames when it's used with the Visual C++ compiler.

The main idea for the library solution presented here is to use only the portable **CPPX_U** string notation in the portable code, and to have such strings reinterpreted as system specific **char** or **wchar_t** based strings for the system dependent implementation code, if any, and as necessary. By using a character encoding value type that's defined differently depending on the system, plus a macro that adds strong typing and an **L** literal prefix as required for each system, the exact same source code can specify strongly typed string literals with UTF-8 encoding for *nix, and with UTF-16 encoding for Windows. This is maximally efficient for each system's API function calls and favoured external text encoding, and makes it technically possible to access all valid filenames on each system, as shown.

To make this work most seamlessly the C++ source code should then be UTF-8 encoded with BOM, because that encoding is accepted and understood by default by both Visual C++[20] and g++, and because support for this source encoding is a reasonable requirement for any C++ compiler that one might consider using. ■

[20] As a practical matter, for UTF-8 encoded source code the Visual C++ compiler **requires** a Byte Order Mark (BOM) in order to correctly deduce the encoding. Some earlier versions of the g++ compiler didn't support a BOM for UTF-8, but now it does so that it's not even necessary to do that minimal source code encoding conversion. The same source can be used exactly as-is for both systems.

Dynamic C++ (Part 2)

Previously we saw how to use some simple dynamic features in C++. Alex Fabijanic and Richard Saunders explore more powerful dynamic tools.

> *[Y]ou can't build a system that is completely statically typed.*
> ~ Bjarne Stroustrup [Venners04]

In this installment of the 'Dynamic C++' series of articles, we continue to explore the dynamic solutions in C++ language. We start with **Boost** **`type_erasure`** [Boost.TypeErasure], a combination of **`Boost.Any`** [Boost.Any] and **`Boost.Function`** [Boost.Function], addressing the C++ runtime polymorphism shortcomings. Next, we look into **`Val`**, a class at the heart of the **PicklingTools** library [PicklingTools] aimed at interaction with Python environments. We conclude with **Facebook**'s **folly** library solution for interfacing the world of web and JSON from C++ – the **`dynamic`** class.

Boost.TypeErasure

According to the author, **`type_erasure`** is a generalization of **`Boost.Any`** and **`Boost.Function`** classes, allowing easy composition of arbitrary type erased operations; it addresses the shortcomings of C++ runtime polymorphism, in particular:

- intrusiveness
- dynamic memory management
- inability to apply multiple independent concepts to a single object.

Library uses some advanced constructs such as concepts and template metaprogramming constructs from **`boost::mpl`**. In a similar fashion to **`boost::variant`** specifying a set of types at construction time that can be contained at runtime, the **`type_erasure`** library specifies at construction time a set of operations that can be performed on it at runtime. This is achieved through a vector of *concepts* provided at object declaration site as shown in Listing 1 for an **`incrementable`** and **`ostreamable`** object.

In the example, **`copy_constructible`** allows *copying* and *destruction* of the object, while **`typeid_`** provides *run-time type information* so that **`any_cast`** can be used; these effectively make **`type_erasure any`** equivalent to **`any`** [Boost.Any]. Additionally, **`incrementable`** and **`ostreamable`** concepts are specified, allowing incrementing and streaming of the value **`x`**. Operations can have arguments, so replacing **`incrementable`** *concept* with **`addable`** allows adding of two **`any`**'s. The functionality is brittle in a subtle way, though – while adding two values of different types will compile, unfortunately it results in undefined behaviour at runtime. This problem can be alleviated by specifying **`relaxed_match`** concept (according to author, recently renamed to a more appropriate **`relaxed`** name), which causes exception to be thrown. The proper way of dealing with this problem is using *placeholders*, as shown in Listing 2.

```
any<
    mpl::vector<
        copy_constructible<>,
        typeid_<>,
        incrementable<>,
        ostreamable<>
    >
> x(10);
++x; // incrementable
std::cout << x << std::endl; // ostreamable
```

Listing 1

```
int array[5];

typedef mpl::vector<
    copy_constructible<_a>,
    copy_constructible<_b>,
    typeid_<_a>,
    addable<_a, _b, _a>
> requirements;

tuple<requirements, _a, _b> t(&array[0], 2);
any<requirements, _a> x(get< 0 > (t) +
  get< 1 >(t));
// x now holds array + 2
```

Listing 2

Placeholders are used extensively throughout the library. A placeholder is a substitute for a template parameter in a concept. The library automatically replaces all placeholders with the actual wrapped types.

Furthermore, **`type_erasure`** supports references (both **`const`** and non-**`const`**), as well as user-defined *concepts*. Listing 3 demonstrates adding **`stringable`** *concept* to **`type_erasure`**, allowing a **`to_string()`** member function call syntax directly on an *integer* value wrapped in **`any`**. Things can be simplified when implementation and interface are the same (i.e. a member of **`type_erasure::any`** called **`to_string`** calls a **`to_string`** member of the contained type), **`BOOST_TYPE_ERASURE_MEMBER`** 'shortcut' macro can be used.

Internally, a **`void*`** *pointer* points to the held heap-allocated value and a static equivalent of *virtual table* serves as a *binding* for attached operations as shown in Listing 4.

Alex Fabijanic has been a professional programmer since 1992, specializing in industrial automation and process control software using C and C++ since 1998. He leads the POCO (C++ POrtable COmponents, http://pocoproject.org) project and occasionally writes Javascript and Python code. He can be contacted at alex@pocoproject.org.

Richard T. Saunders has worked with C++ for 20+ years at Rincon Research Corporation doing soft real-time Digital Signal Processing. He has built and fielded many real systems using both C++ and Python. He also occasionally teaches Computer Organization, Software Engineering, Python and C at the University of Arizona in Tucson for the Computer Science and SISTA departments. Contact him at richismyname2001@yahoo.com

Dictionaries, supported in some form by most dynamic languages, became the currency of many systems

```
template<class F, class T>
struct to_string
{
  // conversion function
  static T apply(const F& from, T& to)
  {
    return to = NumberFormatter::format(from);
  }
};
namespace boost {
namespace type_erasure {
  template<class F, class T, class Base>
  // binding
  struct concept_interface
     <::to_string<F, T>, Base, F> : Base
  {
    typedef
       typename rebind_any<Base, T>::type IntType;
    T to_string(IntType arg = IntType())
    {
      return call(::to_string<C, T>(),
                  *this, arg);
    }
}; }
typedef any<to_string<_self, std::string>,
   _self&> stringable;
int i = 123;
stringable s(i);
std::string str = s.to_string(); // s == "123"
```

Listing 3

Boost.Type erasure is an interesting and valuable ‘merger’ of **any** and **function** features, providing dynamic-language like features within the confines of standard C++. Implementation is rather complex and use has some non-intuitive weak spots that can quickly get an inexperienced user in serious trouble. A more robust interface (even if only with **_typedef_**s provided for most frequently used types) would greatly improve the usability and safety for less experienced users.

```
// storage
struct storage
{
    storage() {}
    template<class T>
    storage(const T& arg) : data(new T(arg)) {}
    void* data;
};
// binding of concept to actual type
typedef ::boost::type_erasure::binding<Concept>
table_type;
// actual storage
::boost::type_erasure::detail::storage data;
                            table_type table;
```

Listing 4

The Val Name Rationale

Why *Val* and not something like dynamic or any? Three reasons:

- Everything is, in general, passed by value
- **PicklingTools** encourage use of valgrind to help ensure quality
- Val is only three letters, which is closer to a dynamic language with no letters for the type.

PicklingTools Val

The next reviewed solution to the dynamic typing problem is the **PicklingTools** [PicklingTools] Val [Saunders1], [Saunders2], [Saunders3]. The **PicklingTools** library is an open-source library made up of Python, C++ and Java code allowing cross language communication. The **PicklingTools** evolved from a need to allow Python and C++ to share Python dictionaries across language boundaries. Many modern applications are built using multiple languages: Python, C++, Java, JavaScript, Lua, Icon/Unicon. The front-end languages (JavaScript, Python, Lua) tend to be dynamic languages for handling scripting and basic data flow. The back-end languages (C++, C, Java, FORTRAN) handle the heavy-lifting of fast communications, data I/O and CPU intensive work. In these hybrid systems, front-end languages need to communicate with the back-end languages intensively. Dictionaries, supported in some form by most dynamic languages, became the *currency* of many systems. The **PicklingTools** library solution focuses on the Python dictionary and C++.

While making Python dictionaries easy to express in C++ was not a primary goal, given how much users enjoyed the ease of use, the C++ PicklingTools embraced them wholeheartedly. The goal, then, became to make Python dictionaries as easy to express in C++ as they are in Python.

Consider the ease of a dynamic dictionary manipulation in Python shown in Listing 5.

Because C++ is a statically-typed language, it requires a compile-time type for all variables. **PicklingTools** use Val to indicate a dynamic value.

```
# Create a Python literal
>>> d = { 'a': 1, 'b':2.2, 'c':
...{ 'X':1, 'Y':[1,2,3]  }}

>>> print d['a']   # lookup a single key: 'a' -> 1
>>> d['b'] = 3.3   # insert into dict

# Also easy to lookup/insert nested entities
>>> print d['c']['X']   # lookup nested key
>>> d['c']['Y'] = 0     # insert nested
}
```

Listing 5

The goal, then, became to make **Python dictionaries** as **easy to express** in C++ as they are in Python

```
# Python
>>> a = 1
>>> a = 2.2
>>> a = 'three'    # a takes three different types

// C++
Val a = 1;          // overload constructor
a = 2.2;            // overload op=
a = "three";
```

Listing 6

A side-by-side comparison of basic dynamic typing in Python and C++ using Val is shown in Listing 6.

The **PicklingTools** Val is implemented as a *union* and a *type-tag*, where the value is constructed using *placement new* inside the *union*. The destructor has to manually notice which type to destruct (for non-POD types) and explicitly call the correct constructor. The Val is really just a dynamic container, with storage as shown in Listing 7. C++11 standard introduces `alignof`/`alignas` to address alignment concerns; in pre-C++11, although standard does not explicitly guarantee it and some experts discourage it [GotW28], a union with a `double` member is practically close enough guarantee of the alignment to the largest member of the union.

```
struct Val
{
  // Flags: the ascii typetag, subtype for arrays,
  char tag;
  char subtype;
  char isproxy;
  char pad;

  Allocator *a;  // if using shared memory or
                 // special allocator

  union
  {
    int_1  s;      // type tag 's'
    int_u1 S;      // type tag 'S'
    int_2  i;      // type tag 'i'
    int_u2 I;      // type tag 'I'
    int_4  l;      // type tag 'l'
    int_u4 L;      // type tag 'L'
    int_8  x;      // type tag 'x'
    int_u8 X;      // type tag 'X'
    real_4 f;      // type tag 'f'
    real_8 d;      // type tag 'd'
    complex_8  F;  // type tag 'F'
    complex_16 D;  // type tag 'D'
    char t[sizeof(Tab)];
                   // type tag 't', usually 32
    char n[sizeof(Arr)];
                   // type tag 'n', usually 32
  } u;
};
```

Listing 7

```
// POD types
int_1, int_u1, int_2, int_u2,
int_4, int_u4, int_8, int_u8,
real_4, real_8,
complex_8, complex_16, size_t

// Non-POD types
string, None, Tab (like Python dictionary),
Arr (like Python list)
```

Listing 8

As can be concluded from Listing 7, by design `Val` can only contain certain types, shown in Listing 8.

There are no-user defined types by design. This allows library writers to concentrate on making the interface for C++ Python dictionaries as close to Python as possible without worrying about the problems generality brings.

Listing 9 shows how easy it is to construct a `Val` from basic types and get values out by means of user-defined conversion.

The user-defined conversions and overloading are syntactic sugar. Of course, overloaded cast operators direct calls are really just taking advantage of 'syntactic sugar' for function calls as shown below:

```
Val v = 1;
int_u4 i = v.operator int_u4();
```

The `Val` supports user-defined conversions for all basic types. If the conversion isn't direct, then the user-defined conversions follow the *Principle of Least Surprise*: convert as C++ would (example below).

```
Val v = 3.14159265;
int_u4 i4 = v; // Which conversion?  As C++ would:
   // int_u4 i4 = static_cast<int_u4>(3.14159265);
```

If the conversion doesn't make sense, behaviour is identical to that in Python – an exception is thrown, see Listing 10.

```
// overload Val constructor on all supported types
Val a = 1;
Val b = 2.2;
Val c = "three";
// Get a value out via user-defined conversions
int_u4 i = a;
```

Listing 9

```
# Python
>>> a = 3.14159265
>>> i4 = int(a)      # converts to 3
>>> d = dict(a)      # Exception!  doesn't make
                     # sense to convert to dict

// C++
Val a = 3.14159265;
int_u4 i4 = a;       // converts to 3
Tab d = i4;          // Compile-time error, can't
                     // construct Tab from float
```

Listing 10

The **Val** provides the basic container to support dynamic dictionaries. The C++ **Tab** is equivalent to the Python **dict** (think **Tab == Table**). The keys of the C++ **Tab**, as well as the values of the dictionary are **Val**'s, allowing us to construct dynamic dictionaries:

```
# Python
d = {'a':1, 'b':2.2, 'c':'three'}

// C++
Tab d = "{'a':1, 'b':2.2, 'c':'three'}";
```

Note that the C++ literal is inside a string: the library overloads the constructor of the **Tab** to take a string which contains the literal. While C++11 has some great new literal constructs, it can't quite mimic Python's syntax.

Literal construction of a Python dictionary is supported using exactly the Python syntax: you can cut-and-paste the Python dictionary literal and paste it into the C++ quotes. There is a little Python dictionary parser embedded in the **Tab** class so that it recognizes the same syntax. Rather than invent a new syntax for literal construction, library leverages the well-known Python syntax.

Facebook folly::dynamic

The Facebook **folly::dynamic** [Folly.Dynamic] class is another one in the spectrum of dynamic-typing classes. The class aims to relax the static typing constraints, especially in the *JSON* format data manipulation scenarios; it provides a runtime dynamically typed value for C++, similar to the way languages with runtime type systems work (e.g. Python). It can hold types from a predetermined set (ints, bools, arrays of other dynamics, etc), similar to **boost::variant** and **PicklingTools Val**, but the syntax is intended to be more akin to using the native type directly.

Facebook String Flavour

Facebook **folly::fbstring** is a drop-in replacement for **std::string**, providing the benefit of significantly increased performance on virtually all important primitives. This is achieved by using a three-tiered storage strategy and cooperating with the memory allocator; fbstring is designed to detect use of jemalloc [jemalloc] and cooperate with it to significantly improve speed and memory usage.

Storage strategies

- Small strings (<=23 chars) are stored in-situ without memory allocation.
- Medium strings (24-255 chars) are stored in **malloc**-allocated memory and copied eagerly.
- Large strings (>255 chars) are stored in **malloc**-ated memory and copied lazily.

Implementation highlights

- Compatible with **std::string**.
- Thread-safe, reference-counted copy-on-write for large (>255 chars) strings.
- Uses **malloc** instead of allocators.
- Jemalloc-friendly
- The **find()** is implemented using Boyer-Moore [Wikipedia].
- Offers conversions to and from **std::string**.

Supported architectures are x86 and x64; porting **fbstring** to big-endian architectures would require changes.

```
dynamic array = {
  "array ", "of ", 4, " elements" };
assert(array.size() == 4);
dynamic emptyArray = {};
assert(array.empty());
```

Listing 11

```
dynamic map = dynamic::object;
map["something"] = 12;
map["another_something"] = map["something"] * 2;
dynamic map2 = dynamic::object
  ("something", 12)("another_something", 24);
```

Listing 12

An example of creating a **dynamic** holding most common types is shown below. Strings are stored internally as **fbstring**, the Facebook drop-in replacement for **std::string**.

```
dynamic twelve = 12;
dynamic str = "string";   // fbstring
dynamic nul = nullptr;
dynamic boolean = false;
```

The library extensively uses C++11 features for both speed and syntactic advantages. For example, as shown in Listing 11, arrays can be initialized with *initializer lists*. This particular feature, however, also imposes a limitation – **dynamic** has no default constructor. The rationale for this design decision is due to the standard requirement for the expression **dynamic d = {}** to call default constructor. The conflict arises in the default construction either having to result in **d.isArray()** (a) being **false** for the expression **dynamic d = {}** or (b) being **true** for **dynamic d**. The solution the authors of **folly::dynamic** deemed most appropriate is to entirely disallow the default construction.

Maps from dynamics to dynamics are called objects. As shown in Listing 12, the **dynamic::object** constant is how an empty map from dynamics to dynamics is created. The same listing also shows how **dynamic** objects can be created by using **object::operator()**.

```
enum Type
{
  NULLT,
  ARRAY,
  BOOL,
  DOUBLE,
  INT64,
  OBJECT,
  STRING,
};
// ...
Type type_;
union Data
{
  explicit Data() : nul(nullptr) {}
  void* nul;    // void* used instead of
                // std::nullptr_t due to gcc bug
  Array array;
  bool boolean;
  double doubl;
  int64_t integer;
  fbstring string;
  typename std::aligned_storage<
    sizeof(std::unordered_map<int,int>),
    alignof(std::unordered_map<int,int>)
  >::type objectBuffer;
} u_;
```

Listing 13

```
struct dynamic::ObjectImpl :
std::unordered_map<dynamic, dynamic> {};

// ...

inline dynamic::dynamic(ObjectMaker (*)())
  : type_(OBJECT)
{
  new (getAddress<ObjectImpl>()) ObjectImpl();
}

inline dynamic::dynamic(char const* s)
  : type_(STRING)
{
  new (&u_.string) fbstring(s);
}
```

Listing 14

The internal **dynamic** storage is shown in Listing 13. Types that can be held are: **null**, **Array**, **bool**, **double**, **integer** (64-bit), **Object** and **String**.

Examples of object and string construction are shown in Listing 14. Most notably, **ObjectImpl** is not a mere typedef but inherits from hash map; the reason for this to avoid undefined behavior of parameterizing **std::unordered_map** with an incomplete type.

The gist of the **folly**'s conversion facilities is shown in the Listing 15. The listing shows the code involved in conversion of **dynamic** to **fbstring**. The actual code of converting 'anything to anything', as the documentation states, is in a separate header and too large for inclusion here. For binary/decimal and vice-versa conversion of IEEE doubles, the class uses V8 double-conversion [Double.Conversion].

As will be shown in one of the next installments, **dynamic** provides a very nice user interface, yet also provides a lot in terms of performance. It is a class designed with definite business goal in mind and it succeeds in that endeavor. The only downside for the whole **folly** library is a patchy build system which requires a significant effort to build the library. The library is also not portable, at least not in the out-of-the-box fashion.

Conclusion

In this installment, we reviewed three C++ dynamic typing solutions: **Boost type_erasure**, **PicklingTools Val** and **Facebook folly::dynamic**. While **dynamic** and **Val** provide dynamically-typed storage within the confines of the standard C++, **type_erasure** also ventures in a new direction by adding operations to C++ types. In the next installment, we'll look into more similar solutions, so stay tuned …■

Credits

Steven Watanabe provided valuable advice on **boost::type_erasure**. The list is, of course, not inclusive - many other people, discussions, libraries and code samples were an indispensable source of help in gathering and systematizing this writing.

```
template<> struct dynamic::GetAddrImpl<bool> {
  static bool* get(Data& d) {
    return &d.boolean; }
};
template<> struct dynamic::GetAddrImpl<int64_t> {
  static int64_t* get(Data& d) {
    return &d.integer; }
};
template<class T>
T* dynamic::getAddress() {
  return GetAddrImpl<T>::get(u_);
}
template<class T>
T* dynamic::get_nothrow() {
  if (type_ != TypeInfo<T>::type) {
    return nullptr;
  }
  return getAddress<T>();
}
template<class T>
T dynamic::asImpl() const
{
  switch (type())
  {
    case INT64:
      return to<T>(*get_nothrow<int64_t>());
    case DOUBLE:
      return to<T>(*get_nothrow<double>());
    case BOOL:
      return to<T>(*get_nothrow<bool>());
    case STRING:
      return to<T>(*get_nothrow<fbstring>());
    default:
      throw TypeError("int/double/bool/string",
                      type());
  }
}
inline fbstring dynamic::asString() const
{
  return asImpl<fbstring>();
}
```

Listing 15

References and further information

[Boost.Any] http://www.boost.org/doc/libs/1_53_0/doc/html/any.html

[Boost.Function] http://www.boost.org/doc/libs/1_53_0/doc/html/function.html

[Boost.TypeErasure] http://www.boost.org/doc/libs/1_54_0/doc/html/boost_typeerasure.html

[Double.Conversion] 'Double-conversion library' https://code.google.com/p/double-conversion/

[Folly.Dynamic] Facebook folly library, dynamic class – https://github.com/facebook/folly/blob/master/folly/docs/Dynamic.md

[GotW28] The Fast Pimpl Idiom http://www.gotw.ca/gotw/028.htm

[jemalloc] A general-purpose scalable concurrent malloc(3) implementation http://www.canonware.com/jemalloc/

[PicklingTools] The PicklingTools Library www.picklingtools.com

[Saunders1] 'Dynamic, Recursive, Heterogeneous Types in Statically-Typed Languages', Clinton Jeffery, Richard Saunders, *C++ Now* 2013 Presentation, http://cppnow.org/session/dynamic-recursive-heterogeneous-types-in-statically-typed-languages/

[Saunders2] 'Dynamic, Recursive, Heterogeneous Types in Statically-Typed Languages' Clinton Jeffery, Richard Saunders http://cppnow.org/files/2013/03/saunders-jeffery.pdf

[Saunders3] *C++ Now 2013* Presentation, Richard Saunders http://www.youtube.com/watch?v=W3TsQtnMtqg

[Venners04] 'Abstraction and Efficiency: A Conversation with Bjarne Stroustrup' by Bill Venners, February 16, 2004 http://www.artima.com/intv/abstreffi.html

[Wikipedia] Boyer–Moore string search algorithm http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string_search_algorithm

Further information

'Dynamic C++', ACCU 2013 Conference http://www.slideshare.net/aleks-f/dynamic-caccu2013

Facebook folly library, fbstring class – https://github.com/facebook/folly/blob/master/folly/docs/FBString.md

Auto – a necessary evil? (Part 2)

Should you declare (almost) everything auto?
Roger Orr considers when auto is appropriate.

To have a right to do a thing is not at all the same as to be right in doing it.
~ G.K.Chesterton

In the first article we covered the rules governing the `auto` keyword that was added to the language in C++11 (or added back, if your memory of C++ goes back far enough!)

It is important with a feature like `auto` not only to know the rules about what is permitted by the language – and the meaning of the consequent code – but also to be able to decide *when* the use of the feature is appropriate and what design forces need to be considered when taking such decisions.

In this article we look in more detail at some uses of `auto` with the intent of identifying some of these issues.

A 'complex type' example

One of the main motivations for `auto` was to simplify the declaration of variables with 'complicated' types. One such example is in the use of iterators over standard library containers in cases such as:

```
std::vector<std::set<int>> setcoll;
std::vector<std::set<int>>::const_iterator it =
   setcoll.cbegin();[1]
```

Many programmers were put off using the STL because of the verbosity of the variable declarations. With C++03 one recommendation was to use a `typedef` – and this approach remains valid in C++11:

```
typedef std::vector<std::set<int> > collType;
// C++03 code still works fine
collType setcoll;
collType::const_iterator it = setcoll.begin();
```

With the addition of `auto` to the language the code can be shortened considerably:

```
std::vector<std::set<int>> setcoll;
auto it = setcoll.cbegin();
```

But is it *better*?

To help answer that question let us consider the alternatives in more detail.

The original code is often seen as hard to read because the length of the variable declaration dwarfs the name itself. Many programmers dislike the way that the meaning of the code is masked by the scaffolding required to get the variable type correct.

Additionally, the code is fragile in the face of change. The type of the iterator is heavily dependent on the type of the underlying container so the two declarations (for `setcoll` and `it`) must remain in step if the type of one changes.

The second code, using a `typedef`, improves both the readability of the code and also the maintainability as, should the type of the container change, the nested type `const_iterator` governed by the `typedef` will change too. However, having to pick a type name adds to the cognitive overhead; additionally good names are notoriously hard to pin down.

In the final code the use of `auto` further helps readability by focussing the attention on the expression used to initialise `it` as this defines the type that `auto` will resolve to. Given this, code maintainability is improved as the type of `it` will track the type required by the initialising expression.

We retain the type safety of the language – the variable is still strongly typed – but implicitly not explicitly. The main downside of the final version of the code is that if you *do* need to know the precise type of the variable then you have to deduce it from the expression, to do which also means knowing the type of the container. On the other hand, it can be argued that to understand the semantics of the line of code you already have to know this information, so the new style has not in practice made understanding the code any more difficult.

In this case I am inclined to agree with this view and I can see little downside to the use of `auto` to declare variables for iterators and other such entities. So:

- the code is quicker and easier to write and, arguably, to read
- the purpose is not lost in the syntax
- code generated is identical to the explicit type
- the variable automatically changes type if the collection type changes

However, the last point can be reworded as the variable ~~automatically~~ **silently** changes type if the collection type changes. In particular this can be an issue with the difference between a `const` and non-`const` container. Note that the C++11 code uses `cbegin()`:

```
auto it = setcoll.cbegin();
```

If we'd retained the used of `begin()` we would have got a **modifiable** iterator from a non-`const` collection. The C++03 code makes it explicit by using the actual type name:

```
std::vector<std::set<int>>::const_iterator it;
```

The stress is slightly different and may mean making some small changes to some class interfaces, as with the addition of `cbegin()`.

DRY example

`auto` allows you to specify the type name once. Consider this code:

```
std::shared_ptr<std::string> str =
    std::make_shared<std::string>("Test");
```

1. We've repeated `the std::string`
2. `make_shared` exists solely to create `std::shared_ptr` objects

Roger Orr 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

[1] `cbegin` is another C++11 addition: it explicitly returns a `const` iterator even from a mutable container.

until use of C++11 is sufficiently widespread trying to use the style may simply result in a mix of the old and new

```
// in some header
struct X {
 int *mem_var;
 void aMethod();
};

// in a cpp file
void X::aMethod() {
  auto val = *mem_var; // what type is val?
  ...
```

Listing 1

We can write it more simply as:

```
auto str = std::make_shared<std::string>("Test");
```

The resulting code is just over half as long to write (and read) and I don't think we've lost any information. Additionally the code is easier to change.

Using **auto** rather than repeating the type is indicated most strongly when:

- the type names are long or complex
- the types are identical or closely related

auto is less useful when:

- the type name is simple – or important
- the cognitive overhead on the reader of the code is higher

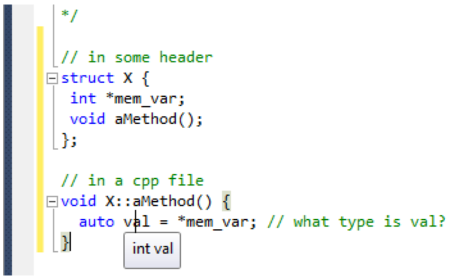
So I think **auto** may be less useful in an example like that in Listing 1.

YMMV (Your mileage may vary) – opinions differ here. The ease of answering the question about the type of **val** may also depend on whether you are using an IDE with type info.

For example, with Microsoft Visual Studio you get the type for the example in Listing 1 displayed in the mouse-over as shown in Figure 1.

Dependent return type example

auto can simplify member function definitions. Consider the class and member function definition in Listing 2.



```
*/

// in some header
struct X {
  int *mem_var;
  void aMethod();
};

// in a cpp file
void X::aMethod() {
  auto val = *mem_var; // what type is val?
}
```

Figure 1

```
class Example
{
public:
  typedef int Result;

  Result getResult();
};

Example::Result Example::getResult()
{ return ...; }
```

Listing 2

We have to use the prefix of **Example::** for the return type **Result** as at this point in the definition the scope does not include **Example**. **auto** allows the removal of the class name from the return type.

The syntax is to place the **auto** where the return type would otherwise go, then follow the function prototype with **->** and the actual return type:

```
auto Example::getResult() -> Result
{ return ...; }
```

Whether or not this makes the code clearer depends on factors including:

- familiarity
- consistent use of this style.

Personally, I still can't decide on this one. I think the new style is an improvement over the old one, but until use of C++11 is sufficiently widespread trying to use the style may simply result in a mix of the old and new styles being used. I do not think this would be a great step forward for existing code bases, but might be worth trying out for new ones.

Polymorphism?

One problem with **auto** is the temptation to code to the *implementation* rather than to the *interface*. If we imagine a class hierarchy with an abstract base class **Shape** and various concrete implementations such as **Circle** and **Ellipse**. We might write code like this:

```
auto shape = make_shared<Ellipse>(2, 5);
...
shape->minor_axis(3);
```

The use of **auto** has made the generic variable **shape** to be of the explicit type shared pointer to **Ellipse**. This makes it too easy to call methods – such as **minor_axis** above – that are not part of the interface but of the implementation.

When the type of shape is 'shared pointer to the abstract base class', you can't make this mistake. (Aside: I think this is a bigger problem with **var** in C# than with **auto** in C++ but your experience may be different.) The trouble is that **auto** is too 'plastic' – it fits the exact type that matches whereas *without* **auto** the author needs to make a decision about the most appropriate type to use. This doesn't only affect polymorphism: **const**,

```
auto main() -> int {
  auto i = '1';
  auto j = i * 'd';
  auto k = j * 100l;
  auto l = k * 100.;
  return l;
}
```

Listing 3

signed/unsigned integer types and sizes are other possible pinch points where the deduction of the type done by **auto** is not the best choice.

What type is it?

It is possible to go to the extreme of making everything in the program use **auto**, but I'm not convinced this is a good idea. For example, what does the program in Listing 3 do?

It is all too easy to assume the **auto** types are all the same – miss the promotion, the **'l'** or the **'.'**. Opinions also vary on whether writing **main** using **auto** aids readability – I am not at all sure it does, especially given the large amount of existing code predating this use of **auto**.

You can use the **auto** rules (on some compilers) to tell you the type. For example, if we want to find out the actual type of **j** we could write this code:

```
auto main() -> int {
  auto i = '1';
  auto j = i * 'd', x = "x";
  ...
```

When compiled this will error as the type deduction for **auto** for the variables **j** and **x** produces inconsistent types. A possible error message is:

```
  error: inconsistent deduction for 'auto':
'int' and then 'const char*'
```

You may also be able to get the compiler to tell you the type by using template argument deduction, for example:

```
template <typename T>
void test() { T::dummy(); }

auto val = '1';
test<decltype(val)>();
```

This generates an error and the error text (depending on the compiler) is likely to include text such as:

```
 see reference to function template instantiation 'void
 test<char>(void)' being compiled
```

What are the actual rules?

The meaning of an **auto** variable declaration follows the rules for template argument deduction.

We can consider the invented function template

```
template <typename T>
void f(T t) {}
```

and then in the expression **auto val = '1';** the type of **val** is the same as that deduced for **T** in the call **f('1')**.

This meaning was picked for good reason – type deduction can be rather hard to understand and it was felt that having a subtly different set of rules for **auto** from existing places where types are deduced would be a bad mistake. However, this does mean that the type deduced when using **auto** differs from a (naïve) use of **decltype**:

```
const int ci;
auto val1 = ci;
decltype(ci) val2 = ci;
```

val1 is of type **int** as the rules for template argument deduction will drop the top-level **const**; but the type of **val2** will be **const int** as that is the declared type of **ci**.

Adding modifiers to auto

Variables declared using **auto** can be declared with various combinations of **const** and various sorts of references. So what's the difference?

```
auto          i   = <expr>;
auto const    ci  = <expr>;
auto       &  ri  = <expr>;
auto const &  cri = <expr>;
auto       && rri = <expr>;
```

As above, **auto** uses the same rules as template argument deduction so we can ask the equivalent question about what type is deduced for the following uses of a function template:

```
template <typename T>;
  void f(T          i);
  void f(T const    ci);
  void f(T       &  ri);
  void f(T const &  cri);
  void f(T       && rri);
```

The answer to the question is, of course, 'it depends' ... especially for the **&&** case (which is an example of what Scott Meyers has named the 'Universal Reference').

const inference (values)

Let us start by looking at a few examples of using **auto** together with **const** for simple value declarations.

```
int i(0); int const ci(0);

auto       v0 = 0;
auto const v1 = 0;
auto       v2 = i;
auto const v3 = i;
auto       v4 = ci;
auto const v5 = ci;
```

This is the easiest case and, as in the earlier discussion of the difference between **auto** and **decltype**, **v0** is of type **int** and **v1** is of type **int const** (you may be more used to calling it **const int**). Similarly **v2** and **v4** are of type **int** and **v3** and **v5** are of type **int const**.

In general, with simple variable declarations, I prefer using **auto const** by default as the reader knows the value will remain fixed. This means if they see a use of the variable later in the block they do not have to scan the intervening code to check whether or not the value has been modified.

const inference (references)

Let's take the previous example but make each variable an l-value reference:

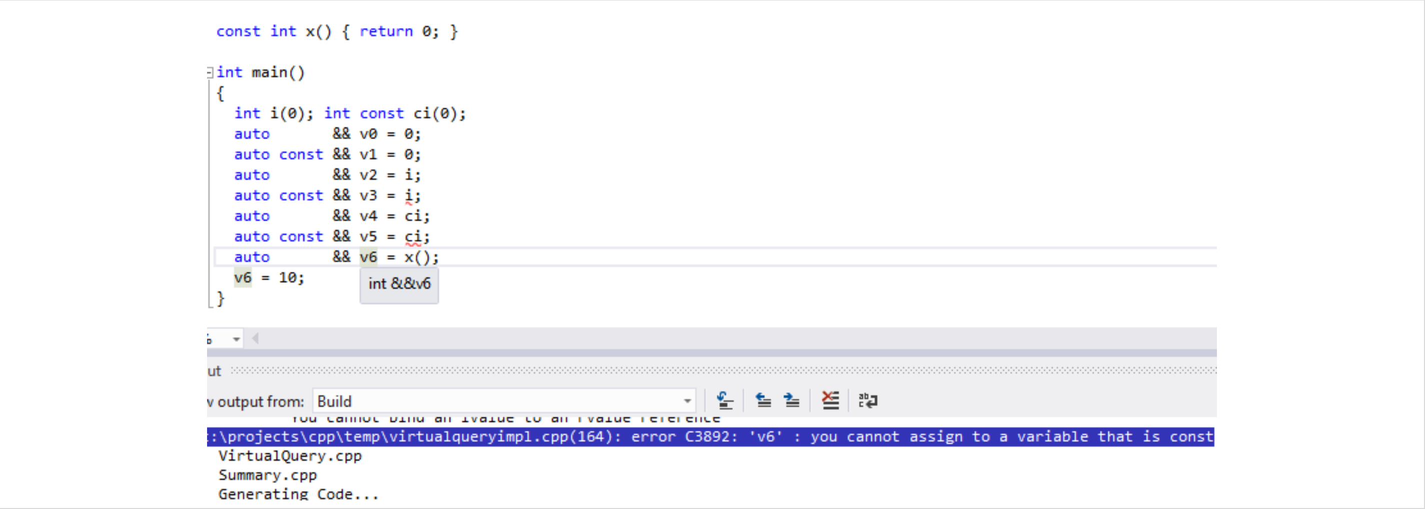
```
int i(0); int const ci(0);

auto       & v0 = 0;    // Error
auto const & v1 = 0;
auto       & v2 = i;
auto const & v3 = i;
auto       & v4 = ci;
auto const & v5 = ci;
```

The first one **fails** as you may not form an l-value reference to a temporary value. However, you **are** allowed to form a **const** reference to a temporary and so **v1** is valid (and of type **int const &**).

v2 is valid and is of type **int &** and the three remaining variables are all of type **int const &**. Notice that the **const** for **v4** is not removed, unlike in the previous example, as it is not a *top-level* use of **const**.

Reference collapsing and auto

Things get slightly more complicated again when we use the (new) r-value reference in conjunction with **auto**.



```
const int x() { return 0; }

int main()
{
  int i(0); int const ci(0);
  auto       && v0 = 0;
  auto const && v1 = 0;
  auto       && v2 = i;
  auto const && v3 = i;
  auto       && v4 = ci;
  auto const && v5 = ci;
  auto       && v6 = x();
  v6 = 10;
}
```

```
   You cannot bind an lvalue to an rvalue reference
:\projects\cpp\temp\virtualqueryimpl.cpp(164): error C3892: 'v6' : you cannot assign to a variable that is const
 VirtualQuery.cpp
 Summary.cpp
 Generating Code...
```

Figure 2

```
int i(0); int const ci(0);

auto       && v0 = 0;
auto const && v1 = 0;
auto       && v2 = i;
auto const && v3 = i;   // Error
auto       && v4 = ci;
auto const && v5 = ci;  // Error
```

The first variable, **v0**, becomes an r-value reference to the temporary 0 (type **int &&**) and the second, **v1**, is the **const** equivalent (**int const &&**). When it comes to **v2**, however, the reference type 'collapses' to an l-value reference and so the type of **v2** is simply **int &**. **v3** is invalid as the presence of the **const** suppresses the reference collapsing and you are not allowed to bind an r-value reference to an l-value. **v4** reference-collapses to **int const &** and the declaration of **v5** is an error for the same reason as for **v3**.

So this is the complicated one: **auto && var = <expr>;** as, depending on the expression, **var** could be

```
T &
T &&
T const &
T const &&
```

Deducing the last case is a little more obscure – you need to bind to a **const** temporary that is of class type. Here's an example of deducing **const &&**:

```
class T{};
const T x() { return T(); }
auto && var = x();   // var is of type T const &&
```

Note that non-class types, like **int**, decay to **&&**. This changed during the development of C++11 and at one point Microsoft's compiler and the Intellisense disagreed over the right answer (see Figure 2)!

(The compiler in the Visual Studio 2013 preview edition does now get this right.)

More dubious cases

auto does not work well with initializer lists as the somewhat complicated rules for parsing these results in behaviour, when used with **auto**, that may not be what you expect:

```
int main() {
  int var1{1};
  auto var2{1};
```

You might expect **var1** and **var2** to have the same type. Sadly the C++ rules have introduced a new 'vexing parse' into the language. The type of **var2** is **std::initializer_list<int>**. There is a proposal to make this invalid as almost everyone who stumbles over this behaviour finds it unexpected.

A mix of signed and unsigned integers – or integers of different sizes – can cause problems with **auto**. In many cases the compiler generates a warning, if you set the appropriate flag(s), and if you heed the warning you can resolve possible problems. But not in all cases ....

```
std::vector<int> v;
...
for (int i = v.size() - 1; i > 0; i -= 2)
{
  process(v[i], v[i-1]);
}
```

If you change **int** to **auto** then the code breaks. The trouble here is that **v.size()** returns **std::vector::size_type** which is an unsigned integer value. The rules for integer promotions means that **i** is also an unsigned integer value. If it starts out odd it will decrease by 2 round the loop as far as 1, then the next subtraction will wrap around – to a large **positive** value. Of course, care must be taken to ensure that an **int** will be large enough for all possible values of **size()** that the program might encounter.

I'm less convinced by the use of **auto** for variables defined by the results of arithmetic expressions as the correct choice of variable type may be necessary to ensure the desired behaviour of the program.

Conclusion

auto is a very useful tool in the programmer's armoury as it allows you to retain type safety without needing to write out the explicit types of the variables. I expect that use of **auto** will become fairly widespread once use of pre-C++11 compilers becomes less common.

However, I do have a concern that thoughtless use of **auto** may result in code that does not behave as expected, especially when the data type chosen implicitly is not the one the reader of the code anticipates.

Please don't use **auto** without thought simply to save typing, but make sure you use it by conscious choice and being aware of the potential issues and possible alternatives. ■

Acknowledgements

Many thanks to Rai Sarich and the *Overload* reviewers for their suggestions and corrections which have helped to improve this article.

This article is based on the presentation of that title at ACCU 2013.