overload 130

DECEMBER 2015 £3

Type Mosaicing with Consultables and Delegates

Plumbing classes together often requires lots of boilerplate code. Type mosaicing helps to avoid this.

CPU Clocks and Clock Interrupts, and Their Effects on Schedulers

Why "sleep" instructions are almost never accurate

Identify your Errors better with char[]

A technique for better error information

The Universality and Expressiveness of std::accumulate

Using C++14's polymorphic lambdas

The Two Sides of Boolean Parameters

How boolean parameters are tempting, but make life difficult

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 130

December 2015

ISSN 1354-3172

Editor

Frances Buontempo
overload@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 131 should be submitted by 1st January 2016 and those for Overload 132 by 1st March 2016.

The ACCU

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

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

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

Reduce, reuse, recycle

The introduction of a carrier bag tax has caused a little fuss. Frances Buontempo considers if it has lessons for programmers.

As some readers may be aware England recently introduced a tax on plastic carrier bags, barring exceptional circumstances such as buying an axe or a chicken. Other parts of the UK had done this a while ago, without so much fuss, as far as I could tell. I became mesmerised by various comedians on the television and radio talking about this. In particular, it seems that a writer in possibly the *Daily Mail* had given a Viz-style 'Top Tip' [citation needed] of avoiding the new tax by taking your own shopping bags to the supermarket. This led to much ridicule from topical comedy shows, since the more astute among us realise the point of the 5p 'tax' is to encourage the reduction in the use of the plastic bags which harm the environment. I suspect using the excuse of watching or listening to comedians and thinking about the environment might sound like a very poor excuse for not getting round to writing an editorial. I am told consistency is a virtue, and you must have expected this by now.

The 'green' or perhaps 'clean' campaigners have been pushing the three R's for a while; reduce, reuse, recycle. This campaign started off in the 1980s, and I suspect most people have heard of the three Rs, not to be confused with the British education system's three Rs – reading, writing, and 'rtihmetic. The internet [all-recycling] tells me "Recycling has a history that dates back to the historic times." That's the kind of history I like. In all seriousness, what struck me while reading about recycling is that if you spoke to Turkish people in about 400 BC about recycling glass they would possibly ask what a cycle was and then say "But we're just using it."

In between my hectic schedule of not writing an editorial and listening to comedy, I have started to read some of the books on our book cases that I've never read before. Prior to starting my Dad's algebra book [shameless_plug], I read *Ruminations* on C++ [Koenig]. I rediscovered people talking about the promise of object-oriented (OO) code allowing you to reuse code easily. One of our three Rs. Have you ever reused a class? Certainly, if your class were to be a singleton [GoF] then you would only be able to make one of them ever, during the course of one run of your program. If you had two cores and ran two instances of your program, you might end up with two, though as ever I digress. You still might use the singleton a few times during the course of the program. Just using it once would seem single-mindedly satirical. In the long run, you may find it very difficult to unit test or manage lifetimes and start considering never using any singleton ever again let alone reusing that one. Reducing the use of singletons is one nudge towards greener, cleaner code: our second R. Reduce. We will return to this thought later. Apart from singletons, the OO coders among us have probably written several classes. Have you ever put them in a library and referenced that library from a variety of projects? Or decided your `AbstractFactoryBuildManager` was very specific to your last project, and had a terrible name, so you just copy and paste some of the better parts into your new project, and rebuild it having learnt to be a better programmer. Rebuild is not one of the three Rs. Perhaps programmers need four. Though we haven't decided what recycle means yet. We will, dear reader.

Ruminations [op cit] talked about being able to reuse C++ classes more easily than C code. The authors tell us they had a C program for distributing other programs, which needed revision to handle larger loads. They decided to, "Try the revisions in C++ instead. The result was successful: My rewrite increased the speed of the old version dramatically, without compromising reliability. Although C++ programs are innately no faster than corresponding C programs, C++ made it intellectually manageable to use techniques that would have been too hard to ... implement reliably in C." The book then explores the possibilities C++ gave over C. Some of the main points include data abstractions, avoiding conventions by making it impossible to use these structures incorrectly – e.g. C++ strings versus C-style char arrays, templates, inheritance and many other features. We even find a section entitled 'Recycling software' in the first chapter which talks about easily extracting a string library from one project for reuse in theirs. Note that this pre-dates C++98 and the official STL. Perhaps reuse and recycling are slightly blurred in any context. Where reuse might mean just using something as it is, recycling might mean taking something and changing its form slightly to be better suited for a similar task. You can reuse or recycle C code too. I worked at a place that had a small library of useful things, such as a doubly-linked list, which was reused in various projects. C++ would have allowed us to use a generic type rather than the traditional `void *` yet we could reuse it as it stood. Library code might be the ultimate way to reuse code. A large part of *Ruminations* [op cit] talks about OO and specific foibles to be aware of in C++ such as virtual destructors, assignment operators and copy constructors, encouraging you to write easy to use library code. Though OO was hailed as a way to reuse code there are many articles bemoaning this as a failed promise. For example, Ambler [Dr Dobbs] tells us "Reusability is one of the great promises of object-oriented technology. Unfortunately, it's a promise that often goes unrealized." He digs into various ways in which code can be reused, beyond just 'inheritance reuse'. Worth a read. As an aside, I note that languages like Java and C# do constantly reuse at least one thing – Object, not something you find in standard C++.

Different languages have different paradigms. C++ is often described as a multi-paradigm language. Specifically, the STL contains classes but does not use OO as such, rather using generic programming. Never try to inherit from `std::vector`, for example. Stepanov, the so-called 'Father

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 works at Bloomberg, has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

of the STL', talks of starting with an algorithm, including the complexity requirements and allowing that to drive the implementation, always keeping the algorithm and the data it operates on separate. Actually, to be precise, he talks about a bacterial infection, algebra and the associativity of addition but people seem to be more interested in using the STL than understanding the mathematical foundations of his work or his health problem at the time. Now, I am aware of many other types of containers and various algorithms which are not in the STL. It is possible to code up your own, following the pattern of using iterators to access the data you wish to operate on generically rather than using OO. I started to wonder how the committee decided what to include. I see an R – reduce – in play again.

> Bjarne is personally responsible for reducing the number of components in STL by a factor of two. He was trying to make it as small as possible to placate the opposition. [Stepanov]

In order to arrive at a usable library, which is therefore reusable, the scope was reduced. Reduced scope or even reducing the amount of code is frequently a winning formula. Furthermore, it is often possible to fix bugs by deleting code. Reducing the number of Boolean flags is my personal favourite.

What have we learnt so far? It is possible to reuse code, and not just by copying and pasting it, however tempting that may be. It is frequently desirable to reduce the amount of code, either deleting chunks of unused legacy code, or making something more generic so you just have an algorithm once, rather than once for every type. You can take genericity too far, ending up with terse statements that are difficult to read. Sometimes a plain old `for` loop is simpler to understand that C#'s LINQ or similar. There's no point in reducing it down so far that the code is never used. Nonetheless, the 'Don't repeat yourself' (DRY) principle encourages us to refactor, yet another R, by abstracting repeated blocks. Given reduce, and reuse where does this leave recycle? Once upon a time, not that long ago in fact, I encountered a python file entitled NewMerge5.py. This very much suggested that there had been a 4, 3, 2 NewMerge.py and possible just a Merge.py at some point. We have all done something similar. Eventually, sense prevails and a programmer might start using version control. At this point you could be tempted to rename the file Merge.py, to cover up the chequered history prior to version control. Or not. Version control is an excellent tool to help us become cleaner coders. We can reuse something, just by invoking git clone, or the specific incantation for our flavour of version control. We can reduce the lines of code, and rollback easily if something goes wrong. We can recycle (or perhaps refactor) code easily – particular if we can find it in version control, rather than lying around in someone's home directory. Don't forget proper refactoring should be done with the safety of tests. How many times have you found copied and pasted code in a large code base that came from the internet, but the tests weren't pulled over too? Schoolboy error.

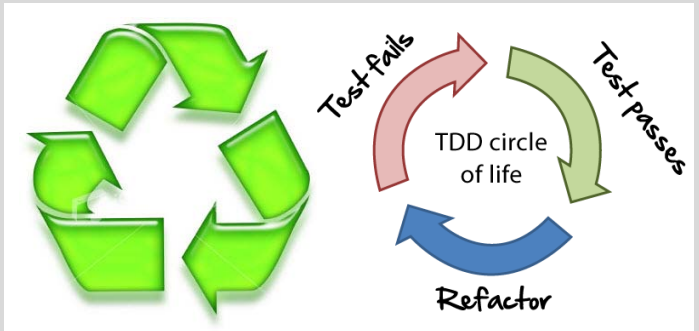

Figure 1

It seems to me that there's a strong parallel between the three Rs and the TDD cycle. Consider both parts in Figure 1. On the left we have the reduce, reuse, recycle in order to be greener, while on the right we have the test fails, test passes then refactor steps – also known as the red/green/refactor cycle. Many things come in threes.

I may have stretched the analogy a little far, but while the environmental activists have started a move towards calling themselves 'clean' rather than 'green', perhaps we coders should consider striving to be green, as well as clean. Reducing code is a delight. Reusing code is possible, if it is well written. And finally, recycling is possible, given version control, libraries that are platform independent, and a willingness to use someone else's code rather than feeling the need to rewrite your own version. If we fail to be green, we could always consider a 5p copy and paste tax. Every little helps.

References

[all-recycling] http://www.all-recycling-facts.com/history-of-recycling.html

[Dr Dobbs] 'A Realistic Look at Object-Oriented Reuse' Scott Ambler, January 01, 1998 from http://www.drdobbs.com/a-realistic-look-at-object-oriented-reus/184415594

[GoF] *Design patterns : elements of reusable object-oriented software* Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1994.

[Koenig] Ruminations on C++: Reflections on a Decade of C++ Programming, Koenig and Moo, Addison Wesley, 1996

[shameless_plug] *A Foundation Course In Modern Algebra*, David Buontempo, Macmillan, 1975.

[Stepanov] 'An Interview with A. Stepanov' by Graziano Lo Russo from http://www.stlport.org/resources/StepanovUSA.html

Once Again on TCP vs UDP

TCP and UDP have different properties. Sergey Ignatchenko weighs up their pros and cons.

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

Discussion on the advantages of TCP vs UDP (and vice versa) has a history which is almost as long as the eternal Linux-vs-Windows debate. As I have long been a supporter of the point of view that both UDP and TCP have their own niches (see, for example, [NoBugs15]), here are my two cents on this subject.

Note for those who already know the basics of IP and TCP: please skip to the 'Closing the Gap: Improving TCP Interactivity' section, as you still may be able to find a thing or two of interest.

IP: just packets, nothing more

As both TCP and UDP run over IP, let's see what the internet protocol (IP) really is. For our purposes, we can say that:

- we have two hosts which need to communicate with each other
- each of the hosts is assigned its own IP address
- the internet protocol (and IP stack) provides a way to deliver data packets from host A to host B, using an IP address as an identifier

In practice, of course, it is much more complicated than that (with all kinds of stuff involved in the operation of the IP, from ICMP and ARP to OSPF and BGP), but for now we can more or less safely ignore the complications as implementation details.

What we need to know, though, it is that IP packet looks as follows:

IP Header (20 to 24 bytes for IPv4)
IP Payload

One very important feature of IP is that it does not guarantee packet delivery. Not at all. Any single packet can be lost, period. It means that any number of packets can also be lost.

IP works only statistically; this behaviour is by design; actually, it is the reason why backbone Internet routers are able to cope with enormous amounts of traffic. If there is a problem (which can range from link overload to sudden reboot), routers are allowed to drop packets.

Within the IP stack, it is the job of the hosts to provide delivery guarantees. Nothing is done in this regard en route.

'No Bugs' Hare 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 currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

UDP: datagrams ~= packets

Next, let's discuss the simpler one of our two protocols: UDP. UDP is a very basic protocol which runs on top of IP. Actually, it is that basic, that when UDP datagrams run on top of IP packets, there is always 1-to-1 correspondence between the two, and all UDP does is add a very simple header (in addition to IP headers), the header consisting of 4 fields: source port, destination port, length, and checksum, making it 8 bytes in total.

So, a typical UDP packet will look as follows:

IP Header (20 to 24 bytes for IPv4)
UDP Header (8 bytes)
UDP Payload

The UDP 'Datagram' is pretty much the same as an IP 'packet', with the only difference between the two being the 8 bytes of UDP header; for the rest of the article we'll use these two terms interchangeably.

As UDP datagrams simply run on top of IP packets, and IP packets can be lost, UDP datagrams can be lost too.

TCP: stream != packets

In contrast with UDP, TCP is a very sophisticated protocol, which does guarantee reliable delivery.

The only relatively simple thing about TCP is its packet:

IP Header (20 to 24 bytes for IPv4)
TCP Header (20 to 60 bytes)
TCP Payload

Usually, the size of a TCP header is around 20 bytes, but in relatively rare cases it may reach up to 60 bytes.

As soon as we're past the TCP packet, things become complicated. Here is an extremely brief and sketchy description of TCP working[1]:

- TCP interprets all the data to be communicated between two hosts as two streams (one stream going from host A to host B, and another going in the opposite direction)
- whenever the host calls the TCP function `send()`, the data is pushed into the stream
- the TCP stack keeps a buffer (usually 2K–16K in size) on the sending side; all the data pushed to the stream goes to this buffer. If the buffer is full, `send()` won't return until there is enough space in the buffer[2]
- Data from the buffer is sent over the IP as TCP packets; each TCP packet consists of an IP packet, a TCP header, and TCP data. TCP data within a TCP packet is data from the sending TCP buffer; data is *not* removed from the TCP buffer on the sending side at the

1. For simplicity, the discussion of flow control and TCP windows is omitted; so are the optimizations such as `SACK` and fast retransmit
2. Alternatively, if the socket is non-blocking, in this situation `send()` can return `EWOULDBLOCK`

moment when TCP packet is sent (!)

- After the receiving side gets the TCP packet, it sends a TCP packet in the opposite direction, with a TCP header with the **ACK** bit set – an indication that a certain portion of the data has been received. On receiving this **ACK** packet, the sender may remove the corresponding piece from its TCP sending buffer.[3]
- Data received goes to another buffer on the receiving side; again, its size is usually of the order of 2K–16K. This receiving buffer is where the data for the **recv()** function comes from.
- If the sending side doesn't receive an **ACK** in a predefined time – it will re-send the TCP packet. This is the primary mechanism by which TCP guarantees delivery in case of the packet being lost.[4]

So far so good. However, there are some further caveats. First of all, when re-sending because of no-**ACK** timeouts, these timeouts are doubled for each subsequent re-send. The first re-send is usually sent after time T1, which is double the so-called RTT (round-trip-time, which is measured by the host as a time interval between the moment when the packet was sent and another moment when the **ACK** for the packet was received); the second re-send is sent after time T2=2*T1, the third one after T3=2*T2, and so on. This feature (known as 'exponential back-off') is intended to avoid the Internet being congested due to too many retransmitted packets flying around, though the importance of exponential back-off in avoiding Internet congestion is currently being challenged [Mondal]. Whatever the reasoning, exponential back-off is present in every TCP stack out there (at least, I've never heard of any TCP stacks which don't implement it), so we need to cope with it (we'll see below when and why it is important).

Another caveat related to interactivity is the so-called Nagle algorithm. Originally designed to avoid telnet sending 41-byte packets for each character pressed (which constitutes a 4000% overhead), it also allows the hiding of more packet-related details from a 'TCP as stream' perspective (and eventually became a way to allow developers to be careless and push data to the stream in as small chunks as they like, in the hope that the 'smart TCP stack will do all the packet assembly for us'). The Nagle algorithm avoids sending a new packet as long as there is (a) an unacknowledged outstanding packet and (b) there isn't enough data in the buffer to fill the whole packet. As we will see below, it has significant implications on interactivity (but, fortunately for interactivity, usually Nagle can be disabled).

TCP: Just the ticket? No so fast :-(

Some people may ask: if TCP is so much more sophisticated and more importantly, provides reliable data delivery, why not just use TCP for every network transfer under the sun?

Unfortunately, it is not that simple. Reliable delivery in TCP does have a price tag attached, and this price is all about loss of interactivity :-(.

Let's imagine a first-person shooter game that sends updates, with each update containing only the position of the player. Let's consider two implementations: Implementation U which sends the player position over UDP (a single UDP packet every 10ms, as the game is fast and the position is likely to change during this time anyway), and Implementation T which sends the player position over TCP.

First of all, with Implementation T, if your application is calling **send()** every 10 ms, but RTT is, say, 50ms, your data updates will be delayed (according to the Nagle algorithm, see above). Fortunately, the Nagle algorithm can be disabled using the **TCP_NODELAY** option (see the section 'Closing the gap: improving TCP interactivity' for details).

If the Nagle algorithm is disabled, *and* there are no packets lost on the way (and both hosts are fast enough to process the data) – there won't be any difference between these implementations. But what will happen if some packets *are* lost?

With Implementation U, even if a packet is lost, the next packet will heal the situation, and the correct player position will be restored very quickly (at most in 10 ms). With Implementation T, however, we cannot control timeouts, so the packet won't be re-sent until around 2*RTT; as RTT can easily reach 50ms even for a first-person shooter (and is at least 100–

3. In practice, **ACK** is not necessarily a separate packet; efforts are taken by the TCP stack to 'piggy-back' an **ACK** on any packet going in the needed direction
4. There are other mechanisms of re-sending, which include re-sending when an **ACK** was received, but was out-of-order, but they are beyond the scope of present article

when implementing reliable UDP, the more TCP features you implement, the more chances there are that you end up with an inferior implementation of TCP

150ms across the Atlantic), the retransmit won't happen until about 100ms, which represents a Big Degradation compared to Implementation U.

In addition, with Implementation T, if one packet is lost but the second one is delivered, this second packet (while present on the receiving host) won't be delivered to the application until the second instance of the first packet (i.e. the first packet retransmitted on timeout) is received; this is an inevitable consequence of treating all the data as a stream (you cannot deliver the latter portion of the stream until the former one is delivered).

To make things even worse for Implementation T, if there is more than one packet lost in a row, then the second retransmit with Implementation T won't come until about 200ms (assuming 50ms RTT), and so on. This, in turn, often leads to existing TCP connections being 'stuck' when new TCP connections will succeed and will work correctly. This can be addressed, but requires some effort (see 'Closing the gap: improving TCP interactivity' section below).

So, should we always go with UDP?

In Implementation U described above, UDP worked pretty well, but this was closely related to the specifics of the messages exchanged. In particular, we assumed that *every packet has all the information necessary*, so loss of any packet will be 'healed' by the next packet. If such an assumption doesn't hold, using UDP becomes non-trivial.

Also, the whole schema relies on us sending packets every 10ms; this may easily result in sending too much traffic even if there is little activity; on the other hand, increasing this interval with Implementation U will lead to loss of interactivity.

What should we do then?

Basically, the rules of thumb are about the following:

- If characteristic times for your application are of the order of many hours (for example, you're dealing with lengthy file transfers) – TCP will do just fine, though it is still advisable to use TCP built-in keep-alives (see below).
- If characteristic times for your application are below 'many hours' but are over 5 seconds – it is more or less safe to go with TCP. However, to ensure interactivity consider implementing your 'Own Keep-Alives' as described below.
- If characteristic times for your application are (very roughly) between 100ms and 5 seconds – this is pretty much a grey area. The answer to 'which protocol to use' question will depend on many factors, from "How well you can deal with lost packets on application level" to "Do you need security?". See both 'Closing the gap: reliable UDP' and 'Closing the gap: improving TCP Interactivity' sections below.
- If characteristic times for your application are below 100ms – it is very likely that you need UDP. See the 'Closing the gap: reliable UDP' section below on the ways of adding reliability to UDP.

Closing the gap: reliable UDP

In cases when you need to use UDP but also need to make it reliable, you can use one of the 'reliable UDP' libraries [Enet] [UDT] [RakNet]. However, these libraries cannot do any magic, so they're essentially restricted to retransmits at some timeouts. Therefore, before using such a library, you will still need to understand very well *how exactly* it achieves reliable delivery, and how much interactivity it sacrifices in the process (and for what kind of messages).

It should be noted that when implementing reliable UDP, the more TCP features you implement, the more chances there are that you end up with an inferior implementation of TCP. TCP is a very complex protocol (and most of its complexity is there for a good reason), so attempting to implement 'better TCP' is extremely difficult. On the other hand, implementing 'reliable UDP' at the cost of dropping most of TCP functionality, is possible.

Closing the gap: improving TCP interactivity

There are several things which can be done to improve interactivity of TCP.

Keep-alives and 'stuck' connections

One of the most annoying problems when using TCP for interactive communication is 'stuck' TCP connections. When you see a browser page which 'stuck' in the middle, then press 'Refresh' – and bingo! - here is your page, then chances are you have run into such a 'stuck' TCP connection.

One way to deal with 'stuck' TCP connections (and without your customer working as a freebie error handler) is to have some kind of 'keep alive' messages which the parties exchange every *N* seconds; if there are no messages on one of the sides for, say, 2**N* time – you can assume that TCP connection is 'stuck', and try to re-establish it.

TCP itself includes a Keep-Alive mechanism (look for `SO_KEEPALIVE` option for `setsockopt()`), but it is usually of the order of 2 hours (and worse, at least under Windows it is not configurable other than via a global setting in the Registry, ouch).

So, if you need to detect your 'stuck' TCP connection earlier than in two hours, and your operating systems *on both sides of your TCP connection* don't support per-socket keep alive timeouts, you need to create your own keep-alive over TCP, with the timeouts you need. It is usually not rocket science, but is quite a bit of work.

The basic way of implementing your own keep-alive usually goes as follows:

- You're splitting your TCP stream into messages (which is usually a good idea anyway); each message contains its type, size, and payload
- One message type is `MY_DATA`, with a real payload. On receiving it, it is passed to the upper layer. Optionally, you may also reset a 'connection is dead' timer.

- Another message type is `MY_KEEPALIVE`, without any payload. On receiving it, it is not passed to the upper layer, but a 'connection is dead' timer is reset.
- `MY_KEEPALIVE` is sent whenever there are no other messages going over the TCP connection for *N* seconds
- When a 'connection is dead' timer expires, the connection is declared dead and is re-established. While this is a fallacy from traditional TCP point of view, it has been observed to help interactivity in a significant manner.

  As an optimization, you may want to keep the original connection alive while you're establishing a new one; if the old connection receives something while you're establishing the new one, you can resume communication over the old one, dropping new one.

TCP_NODELAY

One of the most popular ways to improve TCP interactivity is enabling `TCP_NODELAY` over your TCP socket (again, as a parameter of `setsockopt()` function).

If `TCP_NODELAY` is enabled, then the Nagle algorithm is disabled (usually `TCP_NODELAY` also has some other effects such as adding a PSH flag, which causes the TCP stack on the receiving side to deliver the data to the application right away without waiting for 'enough data to be gathered', which is also a Good Thing interactivity-wise. Still, it cannot force packet data to be delivered until previous-within-the-stream packet is delivered, as stream coherency needs to be preserved).

However, `TCP_NODELAY` is not without its own caveats. Most importantly, with `TCP_NODELAY` you should always assemble the whole update *before* calling `send()`. Otherwise, each of your calls to `send()` will cause the TCP stack to send a packet (with the associated 40–84 bytes overhead, ouch).

Out-of-Band Data

TCP OOB (Out-of-Band Data) is a mechanism which is intended to break the stream and deliver some data with a higher priority. As OOB adds priority (in a sense that it bypasses both the TCP sending buffer and TCP receiving buffer), it may help to deal with interactivity. However, with TCP OOB being able to send only one byte (while you can call `send(...,MSG_OOB)` with more than one byte, only the last byte of the block will be interpreted as OOB), its usefulness is usually quite limited.

One scenario when `MSG_OOB` works pretty well (and which is used in protocols such as FTP), is to send an 'abort' command during a long file transfer; on receiving OOB 'abort', the receiving side simply reads all the data from the stream, discarding it without processing, until the OOB marker (the place in the TCP stream where `send(...,MSG_OOB)` has been called on sending side) is reached. This way, all the TCP buffers are effectively discarded, and the communication can be resumed without dropping the TCP connection and re-establishing a new one. For more details on `MSG_OOB` see [Stevens] (with a relevant chapter available on [Masterraghu]).

Residual issues

Even with all the tricks above, TCP is still lacking interactivity-wise. In particular, out-of-order data delivery of over-1-byte-size is still not an option, stale-and-not-necessary-anymore data will still be retransmitted even if they're not necessary, and dealing with 'stuck' connections is just a way to mitigate the problem rather than to address it. On the other hand, if your application is relatively tolerant to delays, 'other considerations' described below may easily be a deciding factor in your choice of protocol.

Other considerations

If you're lucky enough and the requirements of your application can be satisfied by both TCP and UDP, other considerations may come into play. These considerations include (but are not limited to):

- TCP guarantees correct ordering of packets, UDP as such doesn't (though 'Reliable UDP' might).
- TCP has flow control, UDP as such doesn't.
- TCP is generally more firewall- and NAT-friendly. Which can be roughly translated as 'if you want your user to be able to connect from a hotel room, or from work – TCP usually tends to work better, especially if going over port 80, or over port 443'.
- TCP is significantly simpler to program for. While TCP is not without caveats, not discussed here (see also [Nobugs15a]), dealing with UDP so it works without major problems generally takes significantly longer.
- TCP generally has more overhead, especially during connection setup and connection termination. The overall difference in traffic will usually be small, but this might still be a valid concern.

Conclusions

The choice of TCP over UDP (or vice versa) might not always be obvious. In a sense, replacing TCP with UDP is trading off reliability for interactivity.

The most critical factor in selection of one over another one is usually related to acceptable delays; TCP is usually optimal for over-several-seconds times, and UDP for under-0.1-second times, with anything in between being a 'grey area'. On the other hand, other considerations (partially described above) may play their own role, especially within the 'grey area'.

Also, there are ways to improve TCP interactivity as well as UDP reliability (both briefly described above); this often allows to close the gap between the two. ■

References

[Enet] http://enet.bespin.org/

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

[Masterraghu] http://www.masterraghu.com/subjects/np/introduction/unix_network_programming_v1.3/ch24lev1sec2.html

[Mondal] Amit Mondal, Aleksandar Kuzmanovic, 'Removing Exponential Backoff from TCP', *ACM SIGCOMM Computer Communication Review*

[NoBugs15] '64 Network DO's and DON'Ts for Game Engines. Part IV. Great TCP-vs-UDP Debate' http://ithare.com/64-network-dos-and-donts-for-game-engines-part-iv-great-tcp-vs-udp-debate/

[NoBugs15a] '64 Network DO's and DON'Ts for Game Engines. Part VI. TCP' http://ithare.com/64-network-dos-and-donts-for-multi-player-game-developers-part-vi-tcp/

[RakNet] https://github.com/OculusVR/RakNet

[Stevens] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, *UNIX® Network Programming Volume 1, Third Edition: The Sockets Networking API*

[UDT] http://udt.sourceforge.net/

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

Type Mosaicing with Consultables and Delegates

If several classes need to work together lots of boilerplate code is often needed. Nicolas Bouillot introduces type mosaicing to avoid this.

When assembling a class with complex behaviour out of several other feature classes, the most common approaches are multiple inheritance[1] and object composition.[2] This article introduces an other alternative called *type mosaicing* which enables the assembly of multiple object interfaces, all being visible independently to the class user without the need for writing wrappers. This is achieved using an update of previously described making and forwarding of consultables and delegates [Bou15], which use C++ template meta-programming for automating the wrapping of delegate's public methods, possibly enabling const only methods (*consultable*) or all public methods (*delegate*).

Consider for instance a base class called `Element` being composed of members offering complex features, such as a documentation tree with serialization and deserialization, an event system, a state system and others. All these features and their interfaces already exist independently (or are wanted to be kept independent). Feature being class members are not enough: how to use them remains to be specified. It is wanted for instance that the user has a read-only (const methods) access to the documentation tree interface, while the base class and subclasses has read/write access. How can this kind of behaviour be implemented? Several approaches can be used: writing wrappers gives full control, but while the event system interface is powerful, it also very detailed and requires many wrappers. A less verbose approach could be writing simple feature accessors for feature members, but this introduces ownership related issues. One could instead use multiple inheritance, but both the documentation three member and the state system member have a `get` method. Would it be easier if the `Element` class could 'import' read-only or read/write member interfaces into its public and/or protected declarations, with a compile time generated wrapper that avoid possible conflicts? This would allow expressive design of feature assembled classes without ownership related issue, no wrapper to maintain, no conflicts among feature interfaces and no constraint imposed on feature classes.

So the motivation here is the assemblage of feature classes interfaces.[3] In this context, there are some issues with common approaches. With C++ *multiple inheritance*, base classes interfaces are visible to the subclass user as if they were inside in the subclass, requiring explicit specification of a method scope name only when method name is ambiguous. Accordingly, including a new feature with multiple inheritance may add ambiguity and therefore may break already present code. Another issue occurs when two features of the same type are wanted in the same assemblage: inherit directly twice from the same base class does not import two isolated interfaces, but rather conflicts and fail to compile.

With *object composition*, the assemblage is achieved declaring features as members, and then delegate them manually writing boilerplate wrappers, resulting (according to my experience) in a task prone to errors and muscle pains, with a very likely incomplete wrapping of delegated methods. However, unlike inheritance, object composition is avoiding ambiguity among feature's methods and allows for manual hooking of wanted calls for various purpose, such and thread safety, log, etc.

Type mosaicing with *consultable* and *delegate* aims at avoiding the previously described limitations, while keeping flexibility of object composition. Ambiguity avoiding is achieved using automated wrapper generation that makes delegate methods accessible through an *accessor method* (see Listing 1) that isolate delegated features from each other. Additionally, the possibility of specifying particular behaviour around delegate invocation is enabled through two new additions to consultable. They are described here: **selective hooking** allows for replacing the invocation of a delegated method wrapper by a delegator method sharing the same signature. This enables either bypassing the invocation, or adding a particular behaviour when the hook is subcalling the delegated method (log and call, count and call, etc). The second addition is the **global wrapping** feature that allows to user-defined code to be executed before and after delegate method invocation.

Compared to a simple accessor, the *consultable* and *delegate* method provides a much safer approach with a more powerful control over feature use. Suppose the use of a simple accessor for a feature member: once the user has the object, invocation on it cannot be handled by the assembled class. The second reason is ownership: if the simple accessor returns as reference or a pointer to the feature object, the obtained reference or pointer can be saved by the user. Moreover, with a reference or a pointer, a read-only object can be made writable with `const_cast`. With wrapper generation in *consultable* and *delegate*, the class user does not have access to the object (ownership remains with the assembled class) and each call to its methods can be controlled from the assembled class if wanted.

Nicolas Bouillot is a research associate at the Society for Arts and Technology ([SAT], Montreal, Canada). He likes C++ programming, team-based working, writing research papers, networks and distributed systems, data streaming, distributed music performances, teaching, audio signal processing and many other unrelated things.

1. From [GHJV95], "class inheritance lets you define the implementation of one class in terms of another's. Reuse by subclassing is often referred to as white-box reuse. The term 'white-box' refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses."
2. From [GHJV95], "object composition is an alternative to class inheritance. Here new functionality is obtained by assembling or composing objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as 'black boxes'."
3. While the decorator pattern intent is to 'attach additional responsibilities to an object dynamically' and seems close to the motivation of assembling feature interfaces, the decorator pattern [GHJV95] does not apply here. Indeed, it requires a decorator object's interface to conform to the interface of the component it decorates. Accordingly, a decorator's interface is not added to the decorated component, but used in order to add a specific behaviour to the already existing interface.

the consultable and delegate method provides a much safer approach with a more powerful control over feature use

```
// 'Name' is the class delegated twice by the
// 'NameOwner' class, as declared lines 13 and
// 14. The two delegated member are used lines 22
// and 23, invoking their 'print' method.

 1 class Name {
 2  public:
 3   Name(const string &name): name_(name){}
 4   string get() const { return name_; }
 5   void print() const { cout << name_; }
 6   // ...
 7  private:
 8   string name_{};
 9 };
10
11 class NameOwner {
12  public:
13   Make_consultable(NameOwner, Name, &first_,
      first);
14   Make_consultable(NameOwner, Name, &second_,
      second);
15  private:
16   Name first_{"Augusta"};
17   Name second_{"Ada"};
18 };
19
20 int main() {
21   NameOwner nown;
22   nown.first<MPtr(&Name::print)>();  // Augusta
23   nown.second<MPtr(&Name::print)>(); // Ada
24 }
```

Listing 1

The following will provide a more in-depth view of type mosaicing use and implementation, providing first a description/reminder of the updated initial design of consultables and delegates. Then *selective hooking* and *global wrapper* are introduced with example code. After, a more complete example of type mosaicing is given, where a class **`Element`** will have several templated properties **`Prop`** installed. Finally, brief explanations about implementation will be given.

Update about consultables & delegates

Consultables and delegates basically provide automated class member delegation. This is implemented using C++ templates, generating wrapper in user code. The generated wrapper is actually a templated method (called the *accessor method*) taking the original delegated method pointer as parameter. Accordingly, declaring a member as consultable or delegate (see Listing 1) results in the declaration of the *accessor method* templates. **`Make_delegate`** enables the invocation of **all public methods**, and **`Make_consultable`** enables the invocation of **public const methods only**.

```
// 'first' and 'second' are accessor methods
// declared in the 'NameOwner' class and
// therefore available as methods of 'nown_'
// member. Forward_consultable (lines 3 and 4)
// allows them to be forwarded as Box methods,
// respectively 'fwd_first' and 'fwd_second'.

1 class Box {
2  public:
3   Forward_consultable(Box, NameOwner, &nown_,
     first, fwd_first);
4   Forward_consultable(Box, NameOwner, &nown_,
     second, fwd_second);
5  private:
6   NameOwner nown_{};
7 };
8
9 int main() {
10   Box b;
11   b.fwd_first<MPtr(&Name::print)>(); // Augusta
12   b.fwd_second<MPtr(&Name::print)>(); // Ada
13 }
```

Listing 2

The previous implementation of consultables and delegates [Bou15] used a delegated method pointer as the accessor method's first argument. However, the implementation of *selective hooking* needs to detect which delegated method is invoked at compile time. The updated implementation now takes delegate method pointer as template parameter, with invocation arguments only aimed at being forwarded to the delegated method.

The new syntax for invocation is shown with Listing 1, lines 22 & 23. Notice the use of the **`MPtr(X)`** (for method pointer) macro that shortcuts **`decltype(X), X`**. Indeed, with pointer to method as a template parameter, the method type must be specified before the method pointer. This however does not handle ambiguity among possibly overloaded methods in the delegator. Handling this case, additional shortcut macros are provided: **`OPtr`** (for overloaded method pointer) and **`COPtr`** (for const overloads). They however require the specification of return and arguments types, along with the method pointer. For instance, if the **`print`** method (line 5) would have been overloaded, a user could have specified it using **`COPtr(&Name::print, void)`**. For this reason, it is preferable to avoid overloaded methods when writing a class that will be delegated with a consultable or a delegate.

Forwarding a consultable or a delegate has the same meaning as in the previous implementation: an accessor method of a class member can be forwarded to a local class accessor method. Listing 2 shows an example of **`Forward_consultable`**: the **`NameOwner`** member (line 6) accessor methods **`first`** and **`second`** are forwarded as Box accessor methods, respectively **`fwd_first`** (line 3) and **`fwd_second`** (line 4).

```
// 'Box2' is having a 'NameOwner' member, from
// which it is forwarding accessor methods.
// However, for a particular invocation, i.e.
// '&Name::get' with 'fwd_second' (declared
// lines 5-8), the call is hooked and replaced by
// the invocation of 'hooking_get' (declared
// lines 11-15).

 1 class Box2 {
 2  public:
 3   Forward_consultable(Box2, NameOwner, &nown_,
      first, fwd_first);
 4   Forward_consultable(Box2, NameOwner, &nown_,
      second, fwd_second);
 5   Selective_hook(fwd_second,
 6                  decltype(&Name::get),
 7                  &Name::get,
 8                  &Box2::hooking_get);
 9  private:
10   NameOwner nown_{};
11   string hooking_get() const {
12     return "hooked! (was "
13         + nown_.second<MPtr(&Name::get)>()
14         + ")";
15   }
16 };
17
18 int main() {
19   Box2 b;
20   cout << b.fwd_first<MPtr(&Name::get)>()
             // Augusta
21        << b.fwd_second<MPtr(&Name::get)>()
             // hooked! (was Ada)
22        << endl;
23 }
```

Listing 3

As with previous implementation, a delegate (read/write access) can forwarded as a consultable (read-only), blocking then write access to the feature.

Accessor method hooks

Making and forwarding consultable reduces code size and allows for automating delegation without giving reference to the delegated member. However, when *making* or *forwarding*, a class might want to implement some specific behaviour for all (or for a given) delegated method. This is achieved with *selective hooking* and *global wrapping* described here.

Selective hooking

Sometimes one wants to forward all methods of a delegate, but also wants to change or augment the behaviour of one particular method. Motivations can be testing performance of a new algorithm, emulating a behaviour during debug, counting the calls, measuring the call duration of the delegate method, etc. As presented in Listing 3, this is achieved declaring a *selective hook* (lines 5-8). The selective hook applies to a particular delegated method, here **`Name::get`**, and is attached to a particular accessor method, here **`fwd_second`**.

The last argument of **`Selective_hook`** is a method that will be invoked instead of the original delegated method, here the private **`Box2::hooking_get`** method. It must therefore have an identical signature (not merely compatible) as the original delegated method. In this hook, the original method is invoked (line 13) and the returned string ("Ada") is placed into an other string, resulting in the printing of a new string "hooked! (was Ada)" when invoked on line 21.

Notice the type of the delegated method is given as second argument (line 6). This allows consultable internals to support overloaded delegate methods, whose type is the only way to distinguish them from other overloaded methods. This argument might have been made not required, but at the cost of not supporting overloads.

```
// the 'Box3' class is forwarding (line 3)
// accessor method of the 'nown_' member (line 5).
// A global wrap is declared for this particular
// forwarding (line 11), that will cause the
// wrapper to call 'make_TorPrinter' and store
// the result (a 'TorPrinter') before invoking the
// delegated method. The result will be destructed
// when the wrapper will be unstacked.

 1 class Box3 {
 2  public:
 3   Forward_consultable(Box3, NameOwner, &nown_,
      first, fwd_first);
 4  private:
 5   NameOwner nown_{};
 6   struct TorPrinter{
 7     TorPrinter(){ cout << " ctor "; }
 8     ~TorPrinter(){ cout << " dtor "; }
 9   };
10   TorPrinter make_TorPrinter() const {
      return TorPrinter();}
11   Global_wrap(fwd_first, TorPrinter,
      make_TorPrinter);
12 };
13
14 int main() {
15   Box3 b;
16   b.fwd_first<MPtr(&Name::print)>();
       // ctor Augusta dtor
17   cout << b.fwd_first<MPtr(&Name::get)>()
       // ctor dtor Augusta
18        << endl;
19 }
```

Listing 4

Global wrapping

A *Global wrap* is specified for a particular accessor method, but is applied regarding the delegated method invoked. It basically allows for invoking code before and after any delegate invocation, with the exception of delegate methods having a selective hook attached.

Listing 4 shows a use of a *global wrap*. As seen line 11, the **`fwd_first`** accessor method has a global wrap that will invoke the **`make_TorPrinter`** method, which returns a TorPrinter object. As seen in lines 6–9, a **`TorPrinter`** object prints "ctor" when constructing, and "dtor" when destructing. As a result, the invocation of **`Name::print`** (line 16) first constructs a **`TorPrinter`** (printing "ctor"), then invokes print (printing "Augusta"), and then destructs **`TorPrinter`** (printing "dtor").

Notice that *global wrapping* is here used with the custom type (**`TorPrinter`**) but can be used with other types, as for instance **`std::unique_lock`**. The use of a delegate can then be made thread safe with the declaration of a global wrapper that returns a lock applied to a mutex member of the assembled class.

Type mosaicing with consultables and delegates

Listing 5 is an example of *type mosaicing*. The public interface of the class **`Element`** is built as the assembly of interfaces from its private members, declaring class members as delegates and/or consultables (lines 15–19). An equivalent assembly would be impossible with inheritance since two members share the same type (**`info_`** and **`last_msg_`**). With composition, such an assemblage would require the writing of many wrappers, increasing significantly class definition size.

Before entering the listing details, suppose that a new feature is eventually wanted, such as log **`set`** invocation on **`last_msg`** (as invoked line 35). Then you would declare a selective hook for the set method applied to

`last_msg` accessor method that would invoke set and log. This feature addition would consist only in adding a selective hook without modifying existing methods.

With a closer look at the listing, it can be seen that the use of the `Element` class is achieved through creation of a `Countess` subclass instance line 34. The `last_msg_` can be updated (line 35) through invocation of the `last_msg` accessor method. `num_` can be used at line 36 since `num_` has been declared as a consultable line 16.

Things are a little different for the member `info_`. It has two accessor methods declared: one as consultable for public use (line 15) and one as delegate for subclasses (line 19). As it is it, the class user has a read only access to `info_`. In other words, `Prop<string>::get` is allowed for `info_` and return "programmer" when invoked line 38. Then, invocation of the `mutate` method (declared in the `Countess` subclass) is updating `info_` to "mathematician". This `mutate` method is allowed to write `info_` since the `Countess` class have access to the `prot_info` access method. A new call to `info_` getter (line 40) is accordingly returning "mathematician".

The `info_` related part of the code demonstrates how declaration of *consultables* and *delegates* makes explicit which part of the interface can be invoked by a subclass or by a user, without the noise introduced by wrapper declaration when wrapping multiple features. Achieving the same with manually written wrappers would have resulted into wrapping non-const methods for subclasses and wrapping `info_` const methods once or twice (if protected and public wrappers need to have different implementations). Moreover, suppose a const method is added latter into the `Prop` class, then additional wrapper(s) need to be written in the `Element` class. At the inverse, with *type mosaicing*, the `Element` class stays unchanged, but the addition is visible to the user and the subclass(es).

Implementation

My previous paper [Bou15] gave several details about consultable and delegate implementation. The focus here is about description of *selective hooking* and *global wrapping* implementation and a discussion about the use of macros.

About the use of macros

All previously presented features are implemented using macro declaring types and templates in the class definition. It is argued here that making, forwarding and hooking of consultable and delegates follows a declarative programming paradigm. Indeed, when declaring consultables, delegates and hooks, the programmer is describing *what* the program should accomplish (which wrappers are generated), rather than describing *how* to go about accomplishing (writing wrappers). Unfortunately, C++11 offers very few (or nothing excepted macros?) that enable such a paradigm.

While I have read on the Internet that the use of macro is almost always bad, there might be no other option for approaching declarative programming. Here, making consultable if very close to the declaration of a member qualifier (such as const for instance). For instance, (and naively), if one could add custom qualifier to class members, a consultable could have been declared as follow:

```
Prop<string> info_{} : public consultable info,
  protected delegate prot_info;
```

Which would have the same meaning for `info_` than related declarations in Listing 5, generating two accessor methods, `info` and `prot_info`.

This is not possible with C++ (and maybe not desirable), but as demonstrated with consultable implementation, this can be approached using macros.

Notice also that the use of macro for building declarative statement in C++ has already be promoted, for instance by Andrei Alexandrescu for implementation of `scope_exit`, `scope_fail` and `scope_success`.[4]

4. See Andrei Alexandrescu's 'Declarative flow control' talk at the NDC 2014 conference: https://vimeo.com/97329153

```
// the 'Element' class interface is an assemblage
// of several private members interfaces, using
// 'Make_consultable' and 'Make_delegate'.

 1 template<typename T> class Prop {
 2  public:
 3   Prop() = default;
 4   Prop(const T &val) : val_(val){}
 5   T get() const { return val_; }
 6   void set(const T &val) { val_ = val; }
 7   // ...
 8  private:
 9   T val_{};
10 };
11
12 class Element {
13  public:
14   Element(string info, int num) : info_(info),
      num_(num){}
15   Make_consultable(Element, Prop<string>,
      &info_, info);
16   Make_consultable(Element, Prop<int>, &num_,
      num);
17   Make_delegate(Element, Prop<string>,
      &last_msg_, last_msg);
18  protected:
19   Make_delegate(Element, Prop<string>, &info_,
      prot_info);
20  private:
21   Prop<string> info_{};
22   Prop<int> num_{0};
23   Prop<string> last_msg_{};
24 };
25
26 struct Countess : public Element {
27   Countess() : Element("programmer", 1){}
28   void mutate(){
29     prot_info<MPtr(&Prop<string>::set)>
        ("mathematician");
30   }
31 };
32
33 int main() {
34   Countess a;
35   a.last_msg<MPtr(&Prop<string>::set)>
      ("Analytical Engine");
36   a.num<MPtr(&Prop<int>::get)>();      // "1"
37   // a.info<MPtr(&Prop<string>::set)>("...");
     // does not compile
38   a.info<MPtr(&Prop<string>::get)>();
     // "programmer"
39   a.mutate();
40   a.info<MPtr(&Prop<string>::get)>();
     // "mathematician"
41 }
```

Listing 5

Selective hook

Selective hooking is implemented using template specialization. When declaring a consultable or delegate, a primary template for the `get_alternative` method is declared. The parameter of this template is the delegated method pointer, allowing specialization per delegated method. This primary template returns `nullptr`. However, when a `selective_hook` is declared, an explicit specialization for `get_alternative` is declared, returning the pointer to the *hook* method.

In the generated wrapper (see Listing 6), the pointer to the *hook* pointer is stored into the `alt` variable (line 1). Then, if a method for hooking has

```
// 'fun' is the delegate method pointer, 'args' the
// arguments passed for invocation and 'BTs' their
// types. '_member_raw_ptr' is the pointer to the
// delegated member.

1  auto alt = get_alternative<decltype(fun),
    fun>();
2  if(nullptr != alt)
3    return (this->*alt)
       (std::forward<BTs>(args)...);
4  auto encap = internal_encaps();
5  return ((_member_rawptr)
    ->*fun)(std::forward<BTs>(args)...);
```

Listing 6

been declared, **alt** is holding a non null pointer. In this case, the hook is invoked instead of delegated method (line 3).

Global wrapper

Global wrapping is implemented with the **internal_encaps** method, used by the accessor method (line 4 of Listing 6). When declaring a **Global_wrap**, the **internal_encaps** method is declared, wrapping the user method and returning its result. When the consultable or delegate is declared, a dummy **internal_encaps** returning **nullptr** is declared if none has been declared. This is achieved with the use of **enable_if** and a template testing existence of a specific class method.

As a result, when the accessor method is invoked, i**nternal_ encaps** returns either **nullptr** or the object produced by the global wrapper, which is stored in the stack with the **encap** variable. **encap** will then be freed when the accessor method will be unstacked, after invocation of the delegated method (line 5).

Summary

The notion of *type mosaicing* has been introduced as an alternative to *multiple inheritance* and *object composition* when a class interface is designed as an assembly of several other feature interfaces. *Type mosaicing* is made possible thanks to *consultables* and *delegates* which allow for automated wrapper generation with C++ templates. New additions to *consultables* and *delegates* are introduced: *selective hooking* for attaching a specific behaviour to a given feature's method wrapper, and *global wrapping* for attaching code to execute before and after any wrapper invocation of a given feature. This results in an automated interface assembly with full control over feature wrappers.

With *type mosaicing*, being feature classes (consultable or delegate) does not require for compliance with particular constrains like inheriting from a specific base class or implementing one or several specific method(s). It is therefore easier to reuse an existing class as a feature class, and reuse a feature class in a different context. In addition, features are used through wrappers that control read/write accesses with constness, encouraging const-correct feature classes. Moreover, there is no way for a user to access the feature object by copy, reference or address, which helps avoid ownership related issues with features.

As with previous implementation, samples and source code are available on github[5] and have been compiled and tested successfully using with gcc 4.8.4-2ubuntu1, clang 3.4-1ubuntu3 & Apple LLVM version 6.0.

Design and development of *type mosaicing* with *consultable* and *delegate* has been initially made into production code in which major refactoring has been engaged several times: updating wrappers became a major concern. In this production code, the addition of new features is also required periodically and is sometimes necessitating modifications/additions in the core. In this context, feature assembly with minimal boilerplate code became critical.

After searching for similar approaches, comparable techniques have been published. Alexandrescu's *policies* [Ale01], based on template-controlled inheritance, "fosters assembling a class with complex behaviour out of many little classes". MorphJ [HS11] is a Java-derived languages for "specifying general classes whose members are produced by iterating over members of other classes" (Morphing), with possible inlining of delegate into the delegator and optional selecting all members or a subset. DelphJ [GBS13] is extending Morphing with possible transformation of delegated methods (including exposed signature). Delegation proxy in Smalltalk [WNTD14] is comparable to consultable and delegate forwarding, allowing to implement "variations that propagate to all objects accessed in the dynamic extent of a message send. [...] With such variations, it is for instance possible to execute code in a read-only manner". ■

Acknowledgement

Many thanks to Frances Buontempo and the Overload review team for providing constructive comments. This work has been done at the Société des Arts Technologiques and funded by the Ministère de l'Économie, de l'Innovation et des Exportations (Québec, Canada).

References

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Bou15] Nicolas Bouillot. Make and forward consultables and delegates. *Overload* (127):20–24, 2015.

[GBS13] Prodromos Gerakios, Aggelos Biboudis, and Yannis Smaragdakis. Forsaking inheritance: Supercharged delegation in DelphJ. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 233–252, New York, NY, USA, 2013. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[HS11] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, 33(2):6:1–6:44, February 2011.

[WNTD14] Erwann Wernli, Oscar Nierstrasz, Camille Teruel, and Stéphane Ducasse. Delegation proxies: The power of propagation. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 1–12, New York, NY, USA, 2014. ACM.

5. Source code and examples are available at https://github.com/nicobou/cpp_make_consultable

The Universality and Expressiveness of std::accumulate

Folding is a highly generic operation available through std::accumulate. Paul Keir goes beyond reduction, with the help of C++14's polymorphic lambdas.

Graham Hutton's 1999 monograph: *A tutorial on the universality and expressiveness of fold* [Hutton99] provides a fascinating and eminently readable analysis of an elementary reduction operator from functional programming: *fold*. Hutton's tutorial is punctuated by increasingly compelling examples, written in Haskell, of the use of *fold* to implement common list operations, including summation; concatenation; reversal; function composition; *left* folds; and concluding with the Ackermann function. The *fold* combinator is directly comparable to the C++ standard library function: **`std::accumulate`**. In this article, I focus on the *fold* examples from Hutton, and present equivalent generic C++ incarnations of each; making thorough use of appropriate C++14 library and language features, including polymorphic lambda functions[1].

The four-parameter **`accumulate`** function constructs a result from the repeated pairwise application of a given binary function, to elements supplied from both an iterator range; and a single "zero" value, having the type of the result. A canonical example is summation; the value returned from the call to **`accumulate`** in listing 1 is 10.

```
int xs[]{1,2,3,4};
accumulate(cbegin(xs), cend(xs), 0, plus<>{});
```

Listing 1

Hutton's first four examples use *fold* to implement arithmetic and logic operations using built-in functions to specify the relevant binary operations. Similar function objects are provided by the C++ standard library within the **`<functional>`** header. The example above demonstrates the use of **`plus<>`** for **`operator+`**; while **`operator*`**, **`operator&&`**, and **`operator||`** correspond to **`multiplies<>`**, **`logical_and<>`**, and **`logical_or<>`** respectively. Note that C++14 conveniently defines a default template argument for such function objects; **`std::plus<>`** invokes a specialisation which infers the relevant types from its arguments.

Accommodating std::accumulate

A binary operator is said to be *associative* when an expression involving a sequence of its applications produces the same result, regardless of the order of evaluation. The four C++ function objects from the previous section all denote associative operations. Consider addition: both $(1+2)+3$ and $1+(2+3)$ produce the same result; 6. Operator precedence is irrelevant when faced with $a \oplus b \oplus c$; the question is whether to evaluate from the *left*; or from the *right*. In particular, which order does **`accumulate`** use? It uses a left ordering.

An elementary *non-associative* binary operation is *subtraction*. The call to **`accumulate`** in listing 2 would then produce a result equal to $((0-1)-2)-3$, i.e. -6; evaluating from the left. In contrast, an evaluation

```
vector<double> xs{1,2,3};
accumulate(cbegin(xs), cend(xs), 0, minus<>{});
```

Listing 2

ordered from the *right*, say $1-(2-(3-0))$, produces a result of 2. Alas, the remaining examples from Hutton [Hutton99] firmly assume the fold operation evaluates from the *right*.

Producing a result from **`accumulate`** equal to a *right fold* requires two interventions: we need the order of traversal across the container elements reversed; *and* for the order of the arguments given to the binary operation to be switched[2]. Listing 3 defines such a wrapper for **`accumulate`**, called **`foldr`**. The C++14 functions **`crbegin`** and **`crend`** return **`const`** iterators to the *reverse beginning* and *reverse end* of the container argument **`c`**. Meanwhile, the **`flip`** function, uses **`std::bind`** to reverse the argument order for the binary function object; e.g. **`flip(minus<>{})(0,1)`** evaluates to 1; not -1.

```
template <typename F>
auto flip(const F &f) { return bind(f,_2,_1); }

template <typename F, typename T, typename C>
T foldr(const F &f, const T &z, const C &c) {
  return accumulate(crbegin(c), crend(c), z,
    flip(f));
}
```

Listing 3

The definition of **`foldr`** in listing 3 removes the need to call **`crbegin`**, **`crend`** and **`flip`**. It also allows a single container argument to drive the C++ *fold*; much as with C++ *ranges* [Niebler15]. This allows listings here to remain concise; while also facilitating a comparison of the syntactical differences between Haskell and C++. We can now invoke a *right fold*. Assuming `` `C` `` creates an arbitrary standard sequence container, with inferred value type[3], the call to **`foldr`** in listing 4 returns the integer 2.

```
foldr(minus<>{}, 0, C(1,2,3))
```

Listing 4

Non-reducing folds

Using a *fold* to concatenate two containers first requires a small helper function, which should return a new container, by adding a single element to the front of an existing one. Haskell provides the **`(:)`** operator for this job. Listing 5 defines this using its traditional Lisp name: "cons".

1. Note that most of Hutton's Haskell fold examples can also be found in [Bird88] or [Gill93].
2. This is described as the third duality theorem in [Bird88, p68].
3. Assume a **`vector`** default; i.e. **`C(1,2)`** becomes: **`C<vector,int>(1,2)`**.

Paul Keir is a lecturer in the School of Engineering and Computing atars of industry development experience, along with an M.Sc. in HPC; an M.Sc. in Computer Graphics; and a Ph.D. in compilation for heterogeneous multicore. Drop him a mail at paul.keir@uws.ac.uk

Ackermann's function is commonly cited as a recursive function which is not primitive recursive in a first-order programming language

```
auto cons = [=](auto x, auto xs) {
  decltype(xs) ys{x};
  copy(begin(xs), end(xs), back_inserter(ys));
  return ys;
};
```

Listing 5

Like *subtraction*, the **cons** function is *non-associative*; and *non-commutative*. **cons** though, expects two different argument types. Listing 6 provides **foldr** with **cons** as the binary function, and an empty container as the "zero" or starting value; to define an identity *fold*. That is, **id(C(1,2,3))** will return a container object of the same type; with the same 3 elements. Meanwhile, the genericity of C++ allows a similar invocation which only changes the container type: **foldr(cons, list<int>{}, C(1,2,3))**.

```
auto id = [=](auto xs) {
  return foldr(cons, decltype(xs){}, xs);
};
```

Listing 6

To append one container to another, listing 7 again uses **cons** for **foldr**'s first argument; while providing the second, a container, as its "zero" value. Note that while the elements of, say, **append(C('a'),C('b'))** and **C('a','b')** are equal, so too are they equal to **append(C<list>('a'),C<vector>('b'))**; as the definition is sufficiently generic.

```
auto append = [=](auto xs, auto ys) {
  return foldr(cons, ys, xs);
};
```

Listing 7

Folding with lambda arguments

The three functions[4] of listing 8 provide each corresponding **foldr** invocation with a binary lambda function, as, like Haskell, no equivalents exist within the standard library. The **length** function returns the size of its container argument, using a lambda function with unnamed first argument. Both **reverse** and **map** return a container[5]; with **map** utilising the closure aspect of lambda expressions to capture **f**.

Tuples allow a single *fold* to perform more than one calculation. For example, listing 9 defines a function which returns both the size of a container, and the sum of its elements[6].

```
auto length = [=](auto xs){
  return foldr(
    [=](auto, auto n){ return 1+n; },
    0,
    xs);
};
auto reverse = [=](auto xs){
  return foldr(
    [=](auto y, auto ys)
      { return append(ys,decltype(xs){y}); },
    decltype(xs){},
    xs);
};
auto map = [=](auto f, auto xs){
  return foldr(
    [=](auto y, auto ys){ return cons(f(y),ys); },
    vector<decltype(f(*xs.begin()))>{},
    xs);
};
```

Listing 8

```
auto sumlength = [=](auto xs){
  return foldr(
    [=](auto n, auto p){
      return make_pair(n + p.first, 1 + p.second);
    },
    make_pair(0,0),
    xs);
};
```

Listing 9

Functions from folds

The result of applying the *composition* of two functions **f** and **g** to an argument **x** can be achieved in C++ with the expression: **f(g(x))**. In Haskell an elementary binary operator, **(.)**, can also be used; accepting two functions as arguments, and returning a *new* function representing their composition. In listing 10, the *fold*'s binary function argument is a comparable lambda expression for composition. The result of invoking **compose** with a container of callable elements is a function object representing the chained composition of each function in sequence. The "zero" argument of **foldr** uses a simple lambda identity function; though notice it is wrapped by the type of the container element: an instantiation of **std::function**. Why? While the type of each lambda expression is unique, the type of each container element is the same. **std::function** provides exactly the required homogeneity; each lambda accepting and returning say an **int**, becomes simply **std::function<int(int)>**. The "zero", meanwhile, needs the same wrapper, as it provides the type of the *fold*'s result.

One of the most intriguing functions capable of definition by a *right fold*, such as our **foldr**, is a *left fold*. Listing 11 provides such a definition. As

4. A definition of **filter** from [Hutton99] appears in the appendix.
5. Note that **map** returns a **vector** object here solely for brevity.
6. A similar tuple-based definition of **dropWhile** appears in the appendix.

```
auto compose = [=](auto fs){
  using fn_t = typename decltype(fs)::value_type;
  return foldr(
    [=](auto f, auto g)
      { return [=](auto x){ return f(g(x)); }; },
    fn_t([=](auto x){ return x; }),
    fs);
};
```

Listing 10

before, an identity function is required for the *fold*'s starting value, and again this wild lambda needs the guiding hand of `std::function`; though the type in question is calculated in a different manner. Unlike `compose`, the function object returned by `foldr` is invoked within `foldl`; upon `z`. Our journey has brought us full circle to a *left fold*, akin to `std::accumulate`; an invocation such as `foldl(minus<>{}, 0, C(1,2,3))` will produce -6; much as listing 2.

```
auto foldl = [=](auto f, auto z, auto xs){
  using z_t  = decltype(z);
  using fn_t = std::function<z_t(z_t)>;
  return foldr(
    [=](auto x, auto g){
      return [=](auto a){ return g(f(a,x)); };
    },
    fn_t([=](auto x){ return x; }),xs)(z);
};
```

Listing 11

One last comment regarding *left* and *right* folds: should you ever be in the embarrassing situation of being uncertain of the handedness of your *fold* definition, the expression in listing 12 could be useful. Simply replace `fold` with either `foldr` or `foldl`; for a `true` or `false` evaluation respectively.

```
fold([](auto x, auto){ return x; },
  false, C(true))
```

Listing 12

Our final fold example, and so too in [Hutton99], is Ackermann's function [Ackermann28]. Ackermann's function is commonly cited as a recursive function which is not *primitive recursive* in a first-order programming language. John Reynolds, however, demonstrated [Reynolds85] that the function *is* primitive recursive in a higher-order programming language. The C++ implementation in listing 13 includes similar idioms to previous examples, but is given additional treatment to avoid the use of *currying* seen in Hutton's definition. While the binary lambda function provided to the innermost *fold* in the curried original appears unary, $\lambda\, y \rightarrow g$, the C++ version must be uncurried: $\lambda\, y\, as \rightarrow g(as)$. Note too, that these Ackermann *folds* encode the traditional natural number arguments within the size of the input and output container values.

Summary

C++ lambda functions, including the polymorphic variety now available in C++14, allow the generic *fold* operation supported by `std::accumulate` to extend well beyond simple reductions. While a complex *fold* can be less readable or idiomatic than the traditional form, the approach can be refined to construct and transform programs, as well as prove specific program properties; while building on established software engineering principles of reuse and abstraction.

The article places less emphasis on performance considerations, instead focusing on pedagogy and algorithmic aspects; while maintaining parity with the equivalent Haskell, with consistent use of `auto` for type inference.

C++17 will introduce *fold expressions* [Sutton14]. Here, a *finite* set of operators, will share the brevity of the *comma* in pack expansion; consider `*` versus `std::multiplies<>{}`. One restriction is the variadic template pack length must be known at compile-time. ■

```
auto ack = [=](auto xs, auto ys){
  using ys_t = decltype(ys);
  using fn_t = std::function<ys_t(ys_t)>;
  return [=](auto zs){
    return foldr(
      [=](auto, auto g){
        return [=](auto ws){
          return foldr(
            [=](auto, auto as){ return g(as); },
            g(ys_t{1}),
            ws
          );
        };
      },
      fn_t([=](auto bs){ return cons(1,bs); }),
      zs
    );
  }(xs)(ys);
};
```

Listing 13

Appendix

All examples, along with code for `dropWhile`, `filter` and other folds are available at https://bitbucket.org/pgk/accumulate.

References

[Ackermann28] W. Ackermann. *Zum Hilbertschen Aufbau der reellen Zahlen* Mathematische Annalen, 1928.

[Bird88] R. Bird and P. Wadler. *Introduction to Functional Programming* Prentice Hall, 1988.

[Gill93] A. Gill, J. Launchbury and S.P. Jones. *A Short Cut to Deforestation*. ACM Press, 1993.

[Hutton99] G. Hutton. *A tutorial on the universality and expressiveness of fold* Cambridge University Press, 1999.

[Niebler15] E. Niebler. *Working Draft, C++ extensions for Range*s. The C++ Standards Committee, 2015.

[Reynolds85] J.C. Reynolds. *Three approaches to type structure* Springer-Verlag, 1985.

[Sutton14] A. Sutton and R. Smith. *Folding expressions*. The C++ Standards Committee, 2014.

QM Bites – The two sides of Boolean Parameters

Boolean parameters are tempting but make life difficult. Matthew Wilson advises us to avoid them (almost) all the time.

TL;DR

Writing fns with bool params quick&easy; costs maintainers of cli-code far more effort than U save

Bite

As I've discussed previously on a number of occasions ([XSTLv1, QM#1, QM#2]) and as is evident from deductive reasoning alone, combinations of characteristics of software quality are often in conflict: *expressiveness* vs *efficiency*; *efficiency* vs *portability*; and so forth. This is not always the case, however, and usually the sub-divisions of the composite characteristic ***discoverability & transparency*** (see sidebar) collaborate well and, further, tend to also to work well with others, especially *modularity*, but also (depending on how good is the design of the given component) with *expressiveness*, *flexibility*, *portability*, *correctness*/*robustness*/*reliability*, and, even, *efficiency*.

However, sometimes these very collegial subdivisions work against each other. In my experience, the most common example of this is when it comes to using Boolean parameters.

Consider a programmer tasked with writing an API for the manipulation of fonts, such that one is able to create a font from a base family along with selecting emboldenment, italicisation, underlining, and superscripting. The API might look like the following:

```
Font CreateFont(string family, bool bold, bool
italicised, bool underlined, bool superscripted);
```

From the perspective of **discoverability** of the API, this is great: As a user I specify the family, then each of my (Boolean) choices as to whether it is emboldened, italicised, underlined, and superscripted. What could be clearer?

Similarly, the **transparency** of the *implementation* of this is likely to be high, since the name of each of the parameters clearly indicates its intent.

However, the problem comes in the **transparency** *of the client code*. What does any of us know of the majority of the intended semantics of the following statement:

```
Font f = CreateFont("Courier", true, false,
false, false);
```

This violates one of the most important of the Principles of UNIX Programming (PoUP) [AoUP, XSTLv1]:

> **Principle of Transparency**: Design for visibility to make inspection and debugging easier.

If I want to understand the statement in the client code, I have to look at the API. The use of booleans has made me do work I should not need to do in an ideal world. Thus, there's another PoUP violation:

> **Principle of Economy**: Programmer time is expensive; conserve it in preference to machine time.

QM Bites – Introduction

Welcome to QM Bites!

Those gentle readers of patience and good memory will remember that I write the Quality Matters column in this journal, only I haven't done so in quite some time. The problem is a simple one: I'm stuck halfway in my exceptions series, I'm a perfectionist (and not in a good way, in truth), and I haven't been able to get past my commitment to finish the series and move on to the many other topics I'd planned (and still intend) to cover.

Our patient editor, Frances Buontempo, has offered gentle encouragement from time to time, and very recently had the liberating (for me) insight that I should write about other things until my exception-mojo is fully charged. Further, I've just taken on a new long-term role as Application Architect for a company that does extremely high-performance software (in many technologies, but mainly C++), with a remit to transform and modernise. This means I'm going to have a tremendous amount of new material to draw on and huge motivation to rekindle my previous intentions to wander the software quality landscape dropping my magic pixie dust.

Thus: QM Bites. This new vehicle will let me guarantee my contributions to every (or most) *Overload* issue, usually two or three Bites and the occasional full-size Quality Matters instalment when I'm able. It's win-win, baby! ;-)

Recap

I'll be relying on a host of previously discussed topics in the Bites. As I examine each one, I'll provide a short definition, reference to previously discussed points in QM, and relevant references to established works.

TL;DR

Even prior to the advent of the Twitter age I was informed with disheartening frequency that my written prolixity was enervating to an unsettling and, often, self-defeating degree. As a consequence, not only will I be putting very real effort into being succinct, I will also pander to the 140-character generation by including a pithiest-possible executive summary at the head of each Bite. (Sigh!)

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

Definition of Discoverability & Transparency

Discoverability is how easy it is to understand a component in order to be able to use it.

Transparency is how easy it is to understand a component in order to be able to change it.

(NOTE: I'm not even going to touch on the nightmare of what happens if the API designer decides to reorder some of the parameters …)

So what's the humble programmer to do? The answer is simple to state and, at least for those languages that support enumeration(-like) types, straightforward to implement: make each parameter be of a different type.

API:

```
Font CreateFont(string family, Emboldenment bold,
Italicisation italicised, Underlining underlined,
Superscripting superscripted);
```

Client-code:

```
Font f = CreateFont("Courier", Emboldenment.Bold,
Italicisation.None, Underlining.None,
Superscripting.None);
```

For certain, this is somewhat more effort to write (although if you have an Intellisense-like editor it may well be less effort and faster). But without a doubt your maintenance efforts will be *massively* less.

Note that this phenomenon doesn't even have to involve multiple parameters. One of my longest standing mistakes in this respect was in the constructors of reference-counting smart pointers, as seen in listing 1.

To be sure, when looking at uses of this in client code you don't have the ambiguity of multiple parameters, but you still have to know what the second parameter of `stlsoft::ref_ptr`'s constructor does. And it's harder to grep your code for reference copying vs owning in a codebase.

So, the advice is: **Avoid Boolean parameters (almost) all the time.** ■

```
// namespace stlsoft
  template <typename T>
  class ref_ptr
  {
    . . .
  public: // Construction
    ref_ptr(
      T* pt
    , bool addRef
    );
    . . .
```

Listing 1

References

[AoUP] *The Art of UNIX Programming*, Eric S. Raymond, Addison-Wesley, 2003

[QM#1] Quality Matters: Introductions and Nomenclature, Matthew Wilson, *Overload* 92, August 2009

[QM#2] Quality Matters: Correctness, Robustness, and Reliability, Matthew Wilson, *Overload* 93, October 2009

[XSTLv1] *Extended STL, volume 1: Collections and Iterators*, Matthew Wilson, Addison-Wesley, 2007

Identify your Errors better with char[]

Error codes still get used instead of exceptions. Patrick Martin and Dietmar Kühl consider how to use char arrays for better information.

The use of exceptions isn't a viable error handling approach in all cases and returning codes for error handling is sometimes preferable. However, using integral types for error identification is problematic as there is no good mechanism to guarantee that each value uniquely identifies a specific error. This article proposes the use of `char` arrays instead as these are unique straight away. In addition `char` arrays also yield a simple way to get an indication of the cause of an error.

Problem statement

High quality software requires a well defined and straightforward error handling strategy to allow a system to protect or verify its invariants in the face of invalid input or runtime state. There are many positions taken on how to achieve this (see [Google15], [Bloomberg15] [Mozilla15] [Wikipedia15]). It seems clear that there is not yet a consensus on the issue.

Nevertheless, error handing is everyone's responsibility and particularly so for applications coded in C++ and C. In this article we will make a proposal, which we'll call `error_id`, that can be used as an *identity* concept (concept with a little 'c') to ensure when a specific course of action is desired, the error state reported by an API can be unambiguously recognised at arbitrarily remote call sites.

Review of C++ and C approaches

A very common style for reporting from functions in C and C++ is using `enum` or `int` values to enumerate all possible reported statuses and returning these values from library calls. An extension of this approach is to encode a value and some additional category info into an integral value, forming a system of return statuses like `HRESULT` [Wikipedia 2015]. However these different sets of independent `enum` and `int` return values cause significant issues from the mapping of these independent sets when interfaces must be composed into new interfaces that themselves must define new return statuses. `HRESULT`-style status values do not have this issue, but a given system must have all possible error return statuses registered, so that they can be reported and handled consistently. This scales poorly to larger software system. Note that in COM/DCOM `HRESULT`s can be the outcome of IPC calls, thus extending into other universes of `HRESULT` values.

It is possible to define even more complex error handling schemes, such as registering callbacks or having an explicit error stack. And finally, global or thread-local system library variables for an `errno` style approach are of course available, with the usual basket of caveats.

Patrick Martin Patrick's github repo was classified using a machine learning gadget as belonging to a 'noble corporate toiler'. He can't top that. Patrick can be contacted at patrickmmartin@gmail.com.

Dietmar Kühl Dietmar is a senior software developer at Bloomberg L.P. working on the data distribution environment used both internally and by clients. In the past, he has done mainly consulting for software projects in the finance area. He is a regular attendee of the ANSI/ISO C++ standards committee and a moderator of the newsgroup comp.lang.c++.moderated.

Fundamentally, the problem when library boundaries are crossed is that without access to a shared *identity* type whose value describes the status the solutions all tends towards the familiar 'a non zero return indicates an error', which is sensibly enough indeed the position of many error handling schemes employing return codes exclusively. Schemes have been constructed to allow additional information relevant to that status to be extracted, but composing them can be difficult or verbose.

The concern is that a large amount of code becomes devoted to merely mapping values between the return code sets of various libraries; this has a number of critiques on how this will scale:

- claiming to handle the return codes from dependency systems (and transitively via their dependencies) is a forward commitment, which may or may not remain valid over time
- the amount of code written / code paths will result in issues
- the most prudent approach is to have a consistent 'non-zero return code indicates an error' policy, which has the deficiency of requiring all library clients 'opt into' the steps required to obtain more information on a failed operation

C++11's std::error_code

C++11 introduced an interesting approach to deal with errors through the classes `std::error_code` and `std::error_condition`: the idea of these classes is that error values are represented as enumerators with the enum type identifying an error category. The difference between `std::error_code` and `std::error_condition` is that the former is for implementation specific handling of errors while the latter is portable handing of error messages. Although the two classes are different we only refer to `std::error_code` below: the same concepts apply to `std::error_condition`.

Each enum type used with an `std::error_code` is mapped to a different error category. An error category is a class derived from `std::error_category`. An `std::error_code` object holds an enumerator value stored as int and a reference to the corresponding `std::error_category`. When comparing `std::error_code` objects both the stored int and the `std::error_category` can be taken into account, allowing for a mechanism to create unique `std::error_code` values.

The standard C++ library defines error enums and corresponding `std::error_category`s defined for typical system domain. Users can create new error enums and corresponding `std::error_category`s to cover non-standard errors. Unfortunately, creating new error categories is relatively involved: an enum needs to be created and registered as an error enum by specializing `std::is_error_code_enum<E>`, an error category needs to be created by inheriting from `std::error_category` and implementing its pure virtual functions, and `std::make_error_code()` needs to be overloaded for the error `enum`.

Although `std::error_code` can address uniqueness of errors propagated in a system, its use is unfortunately fairly complicated.

a constant of this type can be used as an identity concept, whose values can be passed through opaquely from any callee to any caller

Especially when people are not easily convinced that meaningful errors need to be returned a much simpler approach is needed.

error_id type proposal

The proposed type for `error_id` is `typedef char const error_id[]` which is the *error constant* whereas the *variable* is of course `typedef char const* error_value`. The idea is that each `error_id` is made unique by the linker in a process without any need of registration. Note that functions returning an `error_id` need to be declared to return an `error_value` because functions can't return arrays.

We strongly recommend this value should be printable and make sense in the context of inspecting system state from logs, messages, cores etc.

Interestingly, a brief search for prior art in this area reveals no prior proposals, though we'd love to hear of any we have overlooked. An honourable mention should be made of the second example in the *rogues' gallery* of exception anti-patterns we present later as that author was part of the way towards our proposal.

As such, we contend that a constant of this type can be used as an *identity* concept, whose values can be passed through opaquely from any callee to any caller. Caller and callee can be separated by any number of layers of calls and yet the return values can be passed transparently back to a caller without any need of translating error codes, resulting in less effort and less opportunity for inappropriate handling.

To compare with prior art in this area: note that Ruby's symbols [Ruby15] and Clojure's keywords [Clojure15] supply similar concepts supported at the language level.

error_id desirable properties

An `error_id` has a number of good properties, in addition to being a familiar type to all C and C++ programmers.

- it is a built in type *in C and C++* – and has the expected semantics: the comparison `if (error)` will test for presence of an error condition.
- default use is efficient
- safe for concurrent access
- usage is exception free
- globally registered, the linker handles it [If patching your binary is your thing, then enjoy, but please beware what implications that process has]
- if the content is a printable string constant, (which we strongly recommend) then it inherently supports printing for logging purposes and can be read for debugging. Sadly, printing the 'missing `error_id`' (`NULL`) results in undefined behaviour for streams and `printf`, i.e., upon success the 'error result' can't be printed directly.

For the last point it is helpful to use a simply utility function which arranges to turn the result into always printable values:

```
// declaration of an error_id:
extern error_id MY_KNOWN_ERROR;
extern error_value get_an_error();
...
// definition of an error_id:
error_id MY_KNOWN_ERROR = "My Foo Status";
...
// you can't do this (no existential forgery)

error_value ret = get_an_error();
/*
if (ret == "My Foo Status")
  // does not compile with -Wall -Werror
  // "comparison with string literal results in
  // unspecified behaviour"
{
    ...
}
*/

if (ret)
{
   if (ret == MY_KNOWN_ERROR)
     // this is how to test
   {
     // for this interesting case, here we might
     // need to do additional work
     // for logging, notification and the like
   }
   mylogger <<  "api_call returned " << ret
     << "\n";
}
return ret;  // we can always do this with no
             // loss of information
```

Listing 1

```
error_value error_string(error_value ret) {
  return ret? ret: "<no error>";
}
```

error_id usage examples – 'C style'

As a consequence of these good properties, we can see the following styles are available

- Listing 2 – manual error handling (Mozilla style)
- Listing 3 – error conditions can be composed dynamically

Use of error_id with exception based error handling

So far, all the previous code examples would have worked almost as well if `error_id` were an integral type, however the identity of an `error_id` is exploitable to provide good answers to a number of the issues that come

```
error_value ret;

ret = in();
if (ret)
    return ret;

ret = a_galaxy();
if (ret)
    return ret;

ret = far_far_away();
if (ret)
    return ret;

ret = awaken_force();

if (ret)
{
  // list known continuable errors here
  if ((ret == e_FORD_BROKEN_ANKLE) ||
      (ret == e_FORD_TRANSPORTATION_INCIDENT) &&
      (Ford::scenes::in_the_can()))
  {
    print_local_failure(ret); // whoops!
  }
  else
  {
    panic(ret); // THIS FUNCTION DOES NOT RETURN
  }
}
order_popcorn();
```

Listing 2

```
error_value ret;
for (test_step : test_steps)
{
    ret = test_step(args);
    if (ret)
    {
        log << "raised error [" << ret << "] "
               "in test step " <<
               test_step << '\n';
        return ret;
    }
    // alternatively we might run all,
    // or more and produce a nicely formatted
    // table for debugging / monitoring
}
```

Listing 3

with using exceptions while retaining the known established good practices and features of exception handling.

Not everyone uses exceptions, and not everyone has used exceptions well; to be fair there has been a history of dubious prior art in this area. All the following are real world examples of code that went to production, or can be found in patent submissions, etc. The names of the guilty parties have been removed while we await the expiry of any relevant statutes.

- `throw 99`
- `catch (const char * err)`
- reliance upon `catch (...)`
- reliance upon checking `what()`
- every exception is `std::runtime_error`

However, making use of `error_id` while simultaneously inheriting from a standard exception class in the `std::exception` hierarchy is useful for the same reasons as for using the raw value. As an example: exception class templates specialised on `error_id` are very apt:

```
// we can define a simple template parameterised
// upon the error_id value
template <error_id errtype>
class typed_error_lite : public std::exception {};

// or we can go a little further and allow for
// some additional information this one has a base
// type and additional info
template <error_id errtype>
class typed_error : public std::runtime_error {
public:
  typed_error(const char* what = errtype):
              std::runtime_error(what) {}
  const char *type() const { return errtype; }
  operator const char *() { return errtype; }
};
```

Listing 4

- Listing 4 – exception template allowing exceptions with *identity*
- Listing 5 – define and handle exceptions concisely based upon an `error_id`

This approach has some rather neat properties: we can avoid 'false matches' caused by code handling exception types too greedily. The parameter has to be an `error_id`, not a string literal.

Having a unified set of identities allows callees to throw an exception, relying upon callers higher in the stack to make the decision on how to

```
// somewhere
struct FooErrors {
  static constexpr error_id eFOO =
                   "FOOlib: Foo error";
  static           error_id eBAR;
  //...
};
// elsewhere
constexpr error_id FooErrors::eFOO;
          // a definition is still required
error_id FooErrors::eBAR = "FOOlib: Bar error";
...
// we can define new unique exception instances
typedef typed_error<FooErrors::eFOO> foo_err;
typedef typed_error<FooErrors::eBAR> bar_err;
void f() {
    try
    {
      // something that throws a typed_error
    }
    catch (typed_error<FooErrors::eFOO> const &e)
    {
      // use the template
    }
    /* you can't even write this
    catch (typed_error<"FOOlib: Foo error"> &e)
    {
      // use the template
    }
    */
    catch (bar_err &e)
    {
      // or a typedef
    }
    catch (...)
    {
      // we don't get here
    }
}
```

Listing 5

```
try
{
  // something that throws a
  // typed_error<LibA::ePOR>
  // if LibA::ePOR is not a publicly visible
  // value, it is not possible to write a handler
  // for that specific case nor throw one, except
  // for the code owning that identity
}
catch (typed_error<LibA::eBAR> &e)
{
  // not caught
}
catch (std::runtime_error &e)
{
  // typed_error<LibA::ePOR> is caught here,
  // conveniently
}
```

Listing 6

handle that status, and avoiding the need to re-throw the exception. Even if re-thrown – if the same **error_id** is used, the *identity* is of course preserved even if the stack at the point of re-throw is different from the originating thrower. Listing 6 shows exception handling with fall-through.

There is one responsibility that is granted along with the benefit of universality: since everyone could receive an error code, there is a need to approach declaring **error_id** instances to some standard of consistency. This may well require defining a scheme to generate standard formats and help ensure unique values, perhaps based upon library, component, etc. – see Listing 7, which is a simple example for generating a 'standard' **error_id** value.

In summary, the primary risk from identical strings in two logically distinct **error_id** declarations is when these **error_id** symbols need to be distinguishable by some calling code when an **error_value** may receive a value of either identity. **error_id** does not have an issue and 'does the right thing' from the viewpoint of reading the code. However it should be remembered the identity of an **error_id** is intended to derive entirely from its content, and in the prior case, the printed values will be the same, further reinforcing the utility of a rule requiring **error_id** content which is printable and distinct for each unique identity.

No existential forgery of error_id

So, what is meant by 'existential forgery'? There are two types:

- the first is caused innocently enough by interfacing C++ client code with a C style API which define an enum for the return status type from an interface. This *forces* us to make a mapping some status to another – clearly this is a good place for incorrect logic to creep in, on the basis on the nature of the code.
- the second is caused by the problems caused by a policy of not documenting return values; integral values cannot be made an implementation detail and in large systems it is all too common for code to handle the return **-99** to appear when clients perceive a need to perform handling for that situation.

This problem is addressed by **error_id** in multiple ways:

- possibly most valuably, we can break out of the cycle because the moderate level of self-description in the string of the raw value should facilitate implementing a better approach as trivially the component, file and issue can be delivered
- additionally, **error_id** values can be made internal or for public consumption, enforcing a consistent discipline using the language, again the string contents can back this up, but the clear contract supplied by the library can be 'please feel free to interpret these error states, but any others must be handled using the "Internal Error" strategy'
- note also that exposing an **error_id** is no longer a forward commitment to support that value for all future time, in contrast to integral value return codes, as prior values can be *removed* in new revisions of the interface, in addition to new ones being introduced and clients addressing removed values will simply fail to compile.

Listing 8 shows the generation of identities and unique identities.

```
#define SCOPE_ERROR(grp, pkg, error_str)      \
                    grp "-" pkg ": " error_str
// this can be used thus
const char LibA::ePOR[] =
  SCOPE_ERROR("GRP", "FOO", "Foo not reparable");
// which give us the string:
// "GRP-FOO: Foo not reparable"

// Organisations can exploit other preprocessor
// features to ensure uniqueness of output
#define TOSTRING(x) #x

#define SCOPE_ERROR_LOCATION(grp, pkg,        \
                             error_str)       \
  __FILE__ ":" TOSTRING(__LINE__) " " grp "-" \
  pkg ": " error_str " "
// which give us a string like
// ../test_error_id.cpp:39 GRP-FOO: Foo not Bar
```

Listing 7

What error_id cannot do

No solution is perfect, and this approach is no exception. In the spirit of allowing people to choose for themselves, let us attempt to list some of the common concerns one would come up with and address them:

- **error_id**s cannot be cases in **switch** statements

  we do not see this as much of an issue as this restriction only applies to code using raw **error_id**, and not exceptions and there are two main use cases:

  - hand crafting the mapping between incompatible return statuses. This use should not be necessary as **error_id** values would ideally only need to be thrown/returned and consumed
  - finally reaching code responsible for handling a set of specific codes differently. In this case, chained **if**/**else if** blocks for integral types should suffice.
- return additional info

```
const char N::new_bar[] =
    SCOPE_ERROR("GRP", "FOO", "Foo not Bar");
assert(strcmp(N::new_bar, FooErrors::eBAR) == 0);
assert((N::new_bar != FooErrors::eBAR));
try
{
  throw typed_error<N::new_bar>
    ("bazong not convertible to bar");
}
catch (typed_error<FooErrors::eBAR> &e)
{
  assert(0 == "in typed_error<FooErrors::eBAR>
    handler");
}
catch (typed_error<N::new_bar> &e)
{
  // ok!
}
catch (...)
{
  assert(0 == "Fell through to catch all
    handler");
}
```

Listing 8

- this restriction is of course a natural consequence of using a pure `error_id` as an identity
- exception classes similar to `typed_error` of course allow as much context as one is prepared to pay for in each object instance
- if status need more context – conditions like '[file|table|foo] not found' being the most infuriating – then we have to leave it to the user to code up a solution to pass back more context. The same restriction applies when using integral results.

- defend against abuse

  in a C / C++ application, there is no way to completely prevent abuse such as `error_id` values being appropriated and used inappropriately; the intent of the proposal is to illustrate the benefits arising from the simplicity and effectiveness of using `error_id`. We hope that the solution would be adopted widely upon its own merits.
- yield stable values between processes / builds
  - firstly, it should be remembered the value is not be inspected – only what it points to
  - secondly, this is unavoidable and the solution of course stops working at the boundary of the process – marshalling status codes between different processes, binaries and even different aged builds of the same code cannot rely upon the address. This is a job for some marshalling scheme layered on top of an existing status code system.
- be used as translatable strings
  - the most important point to make here is that these strings should never be treated as trusted output safe to be displayed to system end users. An `error_id` can travel as an opaque value, and hence there is no rigorous mechanism that could prevent information leakage
  - finally `error_id` is a pointer to a const array of char: dynamic translation into user readable strings can only be done by mapping values to known translations. For even a modest size system it becomes more effective to have a facility for text translation which would offer more features relevant to that task than just an `error_id`.

Comparison of error_id and std::error_code

The proposed `error_id` and `std::error_code` have some common features. In particular, both address error propagation with and without exceptions, both provide uniquely identified errors, and both can be globally consumed without requiring declarations of all errors.

There are also important differences. For `std::error_code` the category yields one level or hierarchical grouping while there is no grouping of `error_id`s at all. On the other hand, creating new errors with `std::error_code`s requires definition of multiple entities while creating an `error_id` is just one definition. If the respective specific error should be detectable by users suitable declarations in headers are needed in both cases.

A major difference is that `error_id` can be used both with C and C++ code while `std::error_code` only works with C++ code. In code bases where different languages are used in the same executable it is helpful to use an error reporting scheme available to all of these languages.

Wrap up

In summary, once the perhaps slightly odd feeling of using `error_id` fades, we hope it is a technique that people will adopt in composing larger systems when the error handling strategy is being designed. The process wide identity concept allows for composition of large-scale applications comprising many components, while affording the opportunity of an exception-like error handling with or *without* employing actual exceptions, and maintaining a high level of usability for debugging. This approach will allow both C and C++ libraries to become first class citizens in a design where error handling need never be left to chance or assumption.

Please note that it is implementation defined whether identical string literals use identical or different addresses [c++ std lex.string para 13]. In fact, constant folding where one string literal becomes the data for many identical declarations of string literals in code occurs in many current compilers. Hence it is key to use char arrays where this does not happen.

Recommendations

- Define your identities for system states
- Define how you wish to expose these identities and distribute them. For example, component level, subsystem level, application wide?
- Use them rigorously!

Curate's eggs

There are yet some potentially interesting ramifications that fall out from `error_id` that have not been demonstrated in the interest of brevity, but which we'll touch upon here to pique your interest.

- *missing switch*: it is possible to write template metaprograms that will
  - allow statically typed handlers to be registered for a switch statement to ensure values are always handled, with various outcomes for a fall-through, (Fail, Pass, etc.)
  - even prevent compilation if handlers are not located for specific `error_id` instances
- *private values*: it is possible to define an `error_id` with an value not visible to clients
  - for *typed error* this would allows a standard 'abort via exception' for reporting those error conditions not understood explicitly by callers
  - for raw *error id* this can allow a crude hierarchy of error conditions to be defined. ■

Footnote

Michael Maguire discovered that due to an apparent compiler bug in IBM's xlC V11.1 the arrays of unspecified size `char[]` need to be explicitly decayed. The fix is to use `+eFOO` instead of `eFOO` when `eFOO` is to be passed to a function template.

References

Code illustrating the concept can be found at https://github.com/patrickmmartin/errorcodeNX/tree/article_nov_2015

[Bloomberg15] https://github.com/bloomberg/bde/wiki/CodingStandards.pdf

[Clojure15] http://clojure.org/data_structures#toc8

[Google15] http://google.github.io/styleguide/cppguide.html

[Mozilla15] https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style#Error_handling

[Ruby15] http://ruby-doc.org/core-1.9.3/Symbol.html

[wikipedia15] https://en.wikipedia.org/wiki/HRESULT

CPU Clocks and Clock Interrupts, and Their Effects on Schedulers

Instructions to sleep for a second almost never result in precisely one second's sleep. Bob Schmidt walks us through the mechanics of why.

Suppose you are walking down the hallway of your office, and a Summer Intern (SI) intercepts you and asks, "If I put a line of code in my program that simply reads `sleep(10)`, how long will my program sleep?"[1]

You look at the harried SI and reply, "It depends," and you continue on your way.

The SI rushes to catch up with you, and asks, "It depends on what?"

And you answer, "That, too, depends," as you continue walking.

At this point our young SI is frantic (and in immediate danger of going bald). "Stop talking in riddles, grey hair! I'm in real need of help here."

Your stroll has taken you to the entrance of the break room, so you grab your interlocutor, duck inside, grab two cups of your favourite caffeinated beverage, and sit down.

"It depends," you say, "on many things, so let's start with first things first."

First things first

To understand what's going on 'under the hood' when a `sleep()` is executed, it helps to know a little about how CPUs work, and that means knowing something about CPU clocks, interrupts, and schedulers. The former two are hardware concepts; the latter is a software concept.

But before we get tucked into those details, we should be clear on what `sleep()` we are talking about. There is quite a variety:

- The Posix function `sleep()` (with a lower-case ess) takes a parameter that specifies the number of seconds to sleep;
- The Posix function `usleep()` takes a parameter that specifies the number of microseconds to sleep;
- The Posix function `nanosleep()` takes a parameter that specifies the number of nanoseconds to sleep;
- Boost has the `boost::this_thread::sleep()` function, which accepts an object specialized from the `date_time::subsecond_duration` template;
- C++ 11 has the `std::this_thread::sleep_for()` function that accepts an object specialized from the `duration` template;
- The Windows Platform SDK function `Sleep()` (with an upper-case ess) takes a parameter that specifies the number of milliseconds to sleep;
- Other, operating system variations too numerous to list.

For the purposes of this discussion I'm going to assume a generic version of `sleep()` that accepts a parameter representing time in milliseconds. The concepts scale up or down with the scale of the parameter.

Bob Schmidt Bob Schmidt is president of Sandia Control Systems, Inc. in Albuquerque, New Mexico, a company he founded 20 years ago when he had nothing else to do. When he's not writing software he often can be found brandishing a hot soldering iron threateningly at some recalcitrant piece of prototype hardware. He can be contacted at bob@sandiacontrolsystems.com.

Schedulers

In order to discuss schedulers, I will first define two terms: *process* and *thread*. A process is a unit of execution that, within the context of its operating system, contains all of the resources to execute as a stand-alone entity. A thread is usually a subset of a process, and is the smallest unit of executable code that can be scheduled on its own. (A process has at least one thread, but a thread is not necessarily a process.) Processes are sometimes considered 'heavy-weight' while threads are considered 'light-weight', referring to the amount of resources allocated to each type. Processes have unique address spaces; threads within a process share the address space of the process.

There are several common types of operating environments in the computer world. A single process, non-threaded (SPNT) OS runs one process at a time; Microsoft's DOS is a good example of this type. A single process, multi-threaded (SPMT) OS, such as General Software's Embedded DOS, runs only one process at a time, but supplies an interface that allows for multiple threads to execute in that process. A multi-process, non-threaded (MPNT) OS, such as those used in ModComp mini-computers, may have many processes with a single thread of execution. Linux and Windows are examples of multi-process, multi-threaded (MPMT) environments.

A SPNT OS has no real need of a scheduler. The process is started, it runs until completion, and while it is running no other processes can be executed. The other three types have schedulers of varying complexity. What they have in common is a requirement to determine what should be executing at any given time.

Again, depending on the OS, a scheduler may run at a fixed interval, when needed, or both. We'll come back to this after discussing clocks and interrupts.

CPU clocks

Most CPUs have some sort of circuitry that generates a periodic waveform of alternating ones and zeros. (Clock-less CPUs exist – this discussion is not about them.) The easiest way to represent this signal is shown in Figure 1a. Horizontal lines represent logic levels, with the lower horizontal line typically representing a zero, and the higher horizontal line representing a one. (In reality these horizontal lines represent voltage levels, with the lower line typically at approximately 0 volts DC, and the higher horizontal line typically representing 3.3 VCD or 5.0 VDC, depending on the operating voltage of the logic.) Vertical lines represent the rise and fall of the clock.

1. I shamelessly have stolen two different rhetorical devices here. My first semester calculus professor (whose name I have long since forgotten) used to ask questions by starting off "Suppose you are walking down the street and a stranger asks you…" The idea of using the Summer Intern as a convenient target for all manner of mayhem comes from Stephen Dewhurst, one of my favourite speakers at past Software Development and Embedded Systems conferences

Whatever your CPU was doing is, well, interrupted, and the CPU branches off to do some other, ostensibly important, function

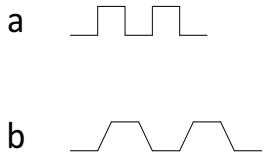

Figure 1

In logic datasheets this type of waveform is often represented by Figure 1b. The waveform represented by Figure 1b takes into account the fact that the rise and fall of the clock is not instantaneous; it takes some very small but none-the-less non-zero amount of time. The slope of the rise and fall of the clock is usually exaggerated in these diagrams to make it easy to see; there often are other lines superimposed on the clock signal to indicate logic level hysteresis and timing. (An example can be found at [TI393], Figure 1, Page 6.)

Both of these diagrams are idealized versions of the waveform. In reality, clock signals are much messier. For a good example of an actual waveform, see [McArdell15]. The rising and falling edges overshoot their levels, then dampen out toward the nominal voltage level.

I'm going to use the idealized clock drawing in Figure 1a for the remainder of this discussion.

Interrupts

A CPU interrupt is a signal that causes some out-of-band processing to occur. Your CPU is chugging along merrily, and then BAM – an interrupt occurs. Whatever your CPU was doing is, well, interrupted, and the CPU branches off to do some other, ostensibly important, function. When that important function is complete, the interrupt is released (or cleared), and your CPU goes back to what it was doing, right where it left off.

(There's a lot that goes on under the hood when processing interrupts, and different CPU architectures implement those details in differing ways. Those details are not important to this discussion.)

Interrupts originate from two places: internal and external to the CPU. Internal interrupts are generated by the CPU hardware itself, or commanded by software. External interrupts are those caused by peripheral devices. These types of interrupts are non-deterministic – they can occur at any time relative to anything else occurring in the CPU. (As an example, consider the input from a keyboard. The CPU cannot predict when a mere human might press a key. The act of pressing a key causes an interrupt in the CPU, so the encoded data can be read from the keyboard and stored for later retrieval by a process.)

Schedulers, part deux

The raw clock rate of today's CPUs is blisteringly fast, even for low-power embedded cores.[2] There are some things that a CPU needs to do that occur at a fraction of that speed, and one of those things is scheduling. (I told you we'd get back to this.)

If we consider a CPU with only one core, and no hyper-threading, it is only possible for one process, and one thread in that process, to be executing at any one time. In a multi-process and/or multi-threaded environment the scheduler is responsible for allocating this limited CPU resource, temporarily suspending the execution of one process and/or thread, and temporarily resuming the execution of a different process and/or thread. (This is called a *context switch*.)

A scheduler contains a list of all of the processes (and their threads) currently executing. The list contains the context in which each process is running. This context is saved when a process or thread is suspended, and restored just before the process or thread is resumed.

A process or thread can be in one of several states. If a process or thread is executing, it is the current context. A process or thread that is suspended is waiting for something to occur. A process or thread that is ready to execute (but is not executing) is one that can become the current context at any time.

Schedulers determine which process or thread to execute next based on their nature. Two common scheduler types are real-time and time-sharing. In a real-time scheduler, processes are assigned priorities, and the process with the highest priority that is ready to execute will become the current context (and execute) the next time the scheduler runs. A time-sharing scheduler allocates CPU time on a round-robin basis, where each process gets a certain fraction of the available CPU time. A real-time scheduler might allocate time to processes of equal priority on a round-robin basis, or a run-until-completion basis.

Clock interrupts

So how and when, you might ask, does the scheduler run? Mostly, the scheduler runs when an interrupt fires, but it may run at other times, too.

An operating system typically will have some sort of clock interrupt, which causes code to execute that takes care of tasks that need to be processed periodically. One of those tasks is the scheduler.

In the absence of any other (external) interrupt, the scheduler will run whenever the clock interrupt fires. At that time the scheduler will look at its list of processes, figure out which one should be executing, and (possibly) execute a context switch to that process.

Clock interrupts fire at a rate much slower than the underlying speed of the CPU. The rate at which the clock executes is a balancing act: the more often a scheduler runs, the easier it is to balance the load between processes (in a time-sharing scheduler), or the faster a higher priority process will actually execute when it is ready (in a real-time scheduler). On the other hand, the more the clock interrupt fires, the less time there is for actual

2. I use RabbitCore modules for some of my embedded designs, and the slowest clock speed they offered was around 20 MHz. Compare that to the clock on the ModComp mini-computers I maintained for 30 years, which ran at only 5 MHz.

When you call sleep(), its operating system-dependent implementation most likely calls an operating system function that suspends the process

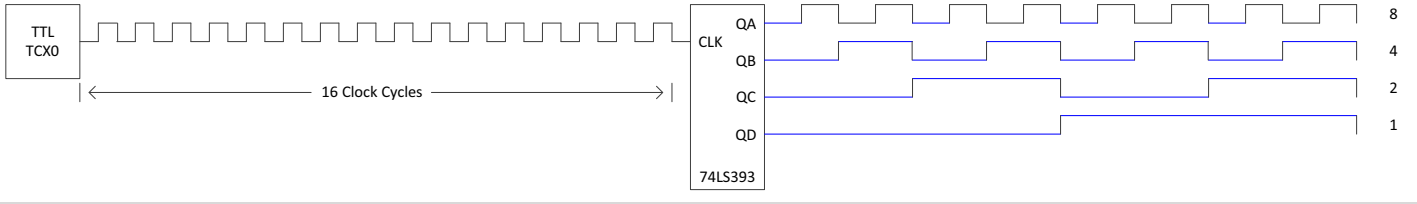

Figure 2

Figure 3

work to get done, until at its most absurd the only code that is executing is the clock interrupt code.

Looking at Figure 2, the box labelled TCXO at the left is a clock oscillator. It generates a clock at the oscillator's specified frequency. The box labelled 74LS393 is a 4-bit decade and binary counter [TI393]. The clock signal drives the CLK input of the 74LS393; the resulting four outputs show how the clock frequency is divided: QA is the clock divided by two; QB is the clock divided by four; QC is the clock divided by eight; and QD is the clock divided by 16.[3]

A modern clock oscillator divided by 16 may not be much good, but QD could be directed to be the input to another binary counter, whose QD output would be the clock divided by 256. Run the signal through two more binary counters, and you have divided the clock by 65 536. If your CPU's clock is running at 6 553 600 Hz (slow by today's standards), you get a clock interrupt 100 times a second, which was a fairly common frequency for a scheduler (modern x86 architecture clock interrupts execute at a faster rate).[4] [Linux2] (I am going to assume a 100 Hz clock interrupt for the remainder of this discussion. I am also going to assume a multi-process, non-threaded environment to simplify the examples.)

Earlier I said that a scheduler may run independent of the interrupt. Consider the case where a high-priority process is suspended waiting for I/O to complete. While it is suspended lower priority processes are getting a chance to execute; however, when the I/O completes we want the high-priority process to resume immediately. This typically is accomplished by having the I/O interrupt handler change the state of the process to ready-to-execute and invoke the scheduler.

3. There are inputs to the 74LS393 that are not shown.
4. This is meant only as an example. There are other, more efficient ways to derive a 'slow' clock interrupt, such as using a second, slower clock oscillator as part of the clock interrupt circuitry.

Delays

"We've got all of the pieces of the puzzle now," you say to the SI, who is getting restless. "It's just a matter of putting them all together to get the answer to your question."

When you call `sleep()`, its operating system-dependent implementation most likely calls an operating system function that suspends the process and causes a context switch – independent of the clock interrupt. In implementations I've seen, the parameter to `sleep()` – the number of milliseconds to sleep – is converted to an integer equal to the parameter divided by the clock interrupt period. (A 1000 millisecond sleep is equal to 100 clock interrupts.) It is easy for the scheduler to decrement a count every time it is executed as the result of the interrupt firing, and change the state of the process to ready-to-execute when the value reaches zero.

You should be starting to see that the accuracy of the `sleep()` call is only as good as the underlying clock interrupt frequency. If the scheduler only runs every 10 milliseconds, you cannot get sub-10 millisecond accuracy from your call to `sleep()`.

Let us consider a system where the clock interrupt fires at our assumed rate – 100 times a second, for a period of 10 milliseconds. If your process is such that it is possible to call `sleep()` at any time between clock interrupts, a 10 millisecond `sleep()` call will, on average, cause your process to sleep for 5 milliseconds (in the absence of any other process activity). A 20 millisecond `sleep()` call will, on average, cause your process to sleep for 15 milliseconds. Take a look at Figure 3 to see why.

In Figure 3 the horizontal line represents time, increasing from left to right, and the vertical lines represent the firing of the clock interrupt. If you can

a single-process, non-threaded operating system may result in the most accurate delays

call **sleep()** at any time, then it is just as likely that **sleep()** will be called immediately after a clock interrupt (point A) as immediately before a clock interrupt (point B). When **sleep()** is called at point A the result is an almost 10 millisecond delay; at point B the result is almost no delay at all. Over time this averages out.

If you want to guarantee a minimum amount of delay, you must call **sleep()** with a value that is at least twice the clock interrupt period. For a minimum 10 millisecond delay that means calling **sleep()** with a parameter equal to 20, which results in a delay from 10 to 20 milliseconds, and an average delay of 15 milliseconds. (I'm assuming that the intermediate values 11 through 19 round down to 10 milliseconds, since that has been my experience.)

```
        load            register1,number_of_milliseconds_to_delay
        load            register2,iterations_per_millisecond
outer:  decrement       register2
        jump_not_zero   register2,outer
inner:  decrement       register1
        jump_zero       register1,end
        load            register2,iterations_per_millisecond
        jump            outer
  end:  no_op
```

Listing 1

Exceptions to this rounding rule-of-thumb include the Posix routines mentioned above (there may be others). The functions **sleep()**, **usleep()** and **nanosleep()** all guarantee a delay that is at least as long as the value of their respective parameters, unless a **SIGALARM** signal is generated for the calling process. [POSIX] The delays are still subject to the limitations of the underlying clock interrupt period, so for these functions the delay will be, on average, one half-period longer than the clock interrupt period (15 milliseconds for a 10 millisecond delay, 25 milliseconds for a 20 millisecond delay, etc.)

Ironically, a single-process, non-threaded operating system may result in the most accurate delays. The reason is that the **sleep()** function may just have to waste CPU cycles by spinning in a loop (a *busy wait*), since there are no other processes or threads to which to switch.[5] In a pseudo-assembly language that might look like Listing 1.

It is possible to guarantee a minimum delay by specifying the value (or next higher value) needed, but in a multi-process (multi-threaded) environment it is not possible to guarantee a maximum delay. The reason for this should be clear – there is no guarantee that your process will be executed on the next clock interrupt context switch. If your process is not the highest priority in a real-time system, or your process is in a round-robin list, one or more other processes may get to execute before the scheduler gets back to your process. Even if your process is the highest priority process, and should be expected to be scheduled on the next clock interrupt, another interrupt higher in priority than the scheduler's may occur, delaying the execution of the scheduler and by extension further delaying your process. This is the 'other activity' alluded to several paragraphs ago.

```
sleep( 0 )
```

A **sleep()** call with a parameter equal to zero often devolves into a relinquish. Calling **sleep(0)** indicates to the scheduler that your process wants to cause a context switch to allow a lower priority process (or other process in the round-robin list) to execute. Your process remains ready-to-execute, so that it can be executed on the next context switch, assuming no other higher priority process also is ready-to-execute.

Synching to the clock

There is a common pattern in process control software, represented by the following code fragment:

```
while ( true )
{
  // do something, over and over again
  sleep ( some_value );
}
```

The first pass through this loop can occur at any time relative to the clock interrupt; however, the second and subsequent passes through the loop are somewhat synched to the clock interrupt. The higher the priority of the process, the more synched to the clock interrupt it will be.

How is this useful? Suppose you have a system that needs to generate regular output pulses to some bit of external hardware in response to some asynchronous (to the clock interrupt) input. (Assume the system's clock interrupt frequency is adequate to space the pulses.) As in Figure 3, the input can occur any time relative to the clock interrupt. Looking at Figure 4, if the interrupt occurs at point A, and the first pulse is generated immediately, you end up with a short duration between the first and second pulses. If the interrupt occurs at point B, you may get a long duration between the first and second pulses; alternatively, you can get a long pulse. Which one you get is based on the priority of the process. You get the regular-width pulse followed by the long gap if your process is high enough priority to remain scheduled after the clock interrupt. You get the long-width pulse if your process is context switched away from executing.

If you synch to the clock, you have a much better chance of getting the result that you want.[6]

5. The Linux documentation for **nanosleep()**, under Old Behavior, indicates that delays of less than 2 milliseconds were handled with a busy wait in kernels prior to 2.5.39. [Linux1]
   This documentation also refers to the different clocks available in modern x86 hardware and their effects on timing. [Linux2] A modern x86 CPU running at 3.0 GHz (3.0x109) has a 0.3333… nanosecond (3.333x10-10) period. That means that a call to nanosleep with a parameter equal to one nanosecond will take more time to make the call and return from it than the delay calls for.

If we have a 10 Hz clock oscillator, we expect there to be 10 cycles per second, and we expect that each cycle will have a period of exactly one tenth of a second

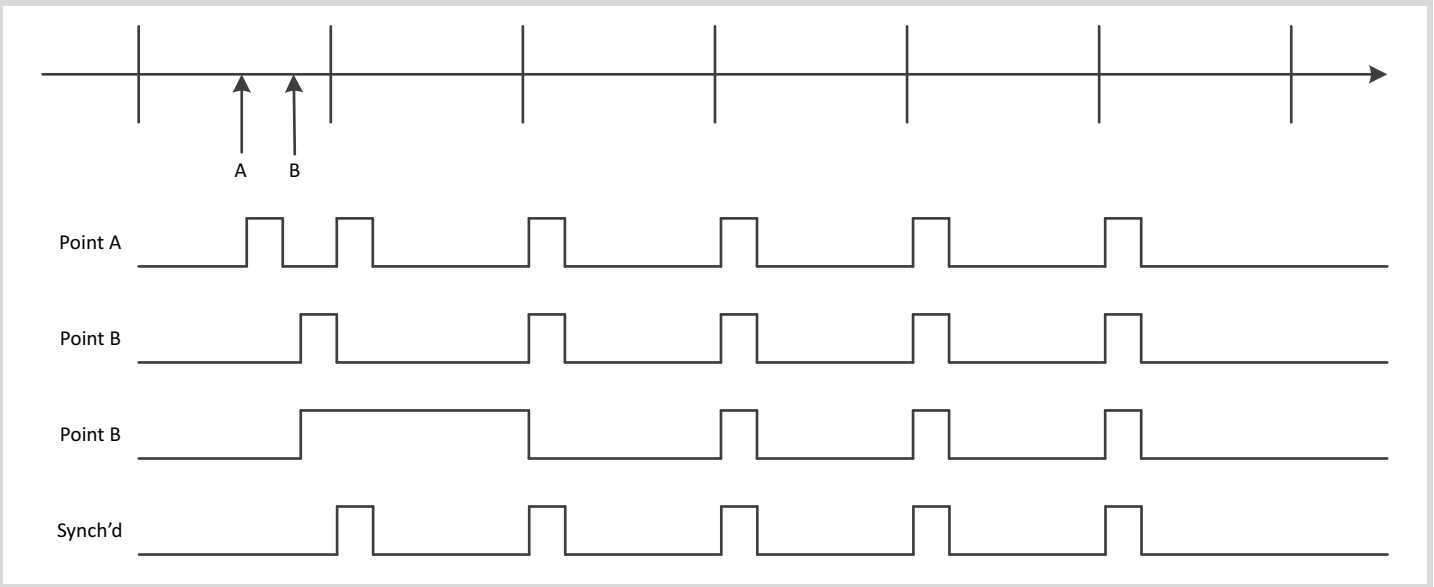

Figure 4

```
sleep ( 10 ) // This synchs to the clock
             // interrupt for each pulse
{
  Set signal high
  waste clock cycles for the width of the pulse
    (busy wait)
  Set signal low
  sleep ( 10 )
}
```

6. In this example the pulse width is generated by busy waiting – wasting clock cycles by executing a tight loop. This is generally not a good idea in any multi-process or multi-threaded system – while your process is spinning nothing else is executing, either, but it is sometimes necessary.

But again, getting these regularly spaced pulses is only possible for high priority processes, or processes running on SPNT systems.

Jitter

If we have a 10 Hz clock oscillator, we expect there to be 10 cycles per second, and we expect that each cycle will have a period of exactly one tenth of a second. Jitter is the deviation from the exact periodicity of that periodic signal. Temperature compensated clock oscillators do a good job of minimizing jitter; unfortunately, the same can't be said about schedulers.

Figure 5 illustrates this point. Here we have the same waveform, synched to the clock, as Figure 4, just expanded in size a bit to make it easier to visualize. The black lines represent the ideal waveform; this is what our process hopes to achieve. The grey boxes represent one possible example of the areas in which the rising or falling edges might actually occur.

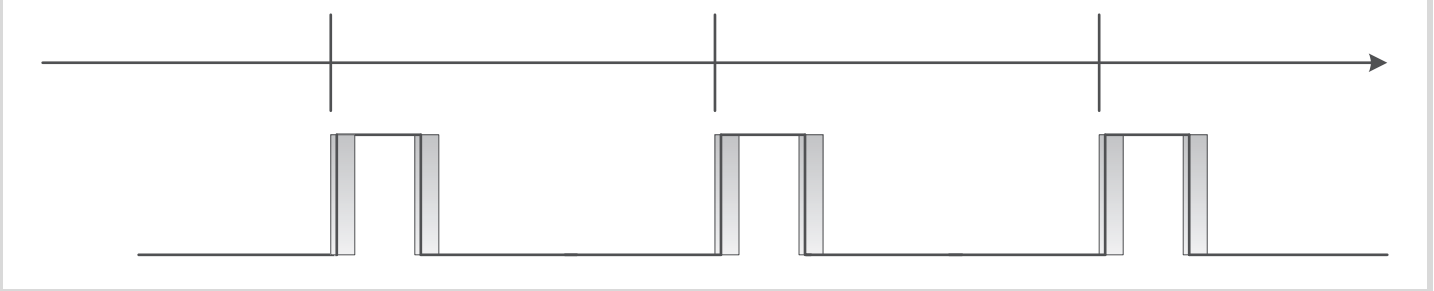

Figure 5

Dealing with Jitter the Hard Way

Several years ago I developed a new interface to an old hardware subsystem. The old hardware had been designed to interface to a DEC PDP-11 through a DEC DRV11 parallel input/output board, using a pair of ribbon cables. For the new implementation I used an Advantech PCI-1739U 48-bit input/output board installed in a PC running Windows.

The DRV11 had input and output signals on the same ribbon cable. The bits on the PCI-1739U were configurable as input or output in groups of eight bits, so I designed a little circuit board that connected the signals at each end. The PCI-1739U had one group of outputs on one ribbon cable and a group of inputs on the other ribbon cable. The circuit board placed the signals on the appropriate conductors of the ribbon cables that connected to the old hardware.

Data was sent to the hardware by loading the data into bits configured as output bits on the I/O board, then toggling a dedicated signal to tell the hardware to load the bits. Most of the timing wasn't critical, and a lot of the jitter didn't matter, but the width of the signal to load the data had to be within certain limits. No matter what I tried, I couldn't toggle that signal in software and reliably hit the time window.

I ended up putting a 74LS123 one-shot timer on the custom board [TI123]. The rising edge of my software-generated signal triggered the 74LS123 to generate an output pulse of exactly the right width, using an R-C circuit. The falling edge of my software-generated signal had no effect on the output of the 74LS123. My software generated pulse could be wildly varying in width, while the output pulse of the 74LS123 was a constant width.

In general, for a real-time scheduler the lower the priority of the process the higher the jitter is likely to be.[7] In a round-robin scheduler jitter is the norm rather than the exception. (See Dealing with Jitter the Hard Way.)

Afterword

"So you see, young intern, the amount of time you sleep does depend on many things," you conclude. The intern gets up and wanders off, muttering something about asking the time and being told how to build a watch. "That's a clock, not a watch… oh, never mind," you say to the retreating SI. "And my hair is not grey, it's blond!"[8] ■

Acknowledgments

As always, thanks to Fran and the reviewers. They always make my stuff so much better. I am particularly indebted to the reviewer who pointed me to the Posix sleep routines. I was not familiar with these functions and their minimum delay guarantees. Special thanks to Szymon Gatner for his feedback (see Update on the Monostate NVI Proto-Pattern).

7. The exception to this is a hard real-time system tuned such that every process always meets its deadline. There should be very little jitter in such a system. I personally have worked on only one hard real-time system – an airplane flight control system; all of the other systems have been soft real-time systems where jitter was an issue.
8. For a given value of blond.

Update on the Monostate NVI Proto-Pattern

My last article in *Overload* [Schmidt15] generated a response from Szymon Gatner. Szymon suggested that all of the functions duplicated from the abstract base class to the Mono NVI class could be replaced with an overloaded pointer operator.

The set of functions represented by **`foo()`** from listing 3 (page 11), implemented like this:

```
inline void foo ()
{
  mp->foo (); // call to the virtual function
              // in the concrete class
}
```

could be replaced with one pointer operator overload function

```
T* operator-> ()
{
  return  mp.get ();
}
```

and used something like this instead of the method used in Listings 5 and 11 (pages 11 and 13):

```
typedef mono_nvi_template< abstract_base >
  mono_nvi;
void nested_function ( void )
{
  mono_nvi  mnvi;  // assumes its non-default
                   // constructor has mnvi->foo ();
                   // been called
}
```

My first impression focused on the unusual way that the pointer operator was being used with a non-pointer variable. Szymon pointed out that "for classes of pointer semantics I think it is something C++ are used to by now. We already have **`unique_ptr`** and **`shared_ptr`** (and **`auto_ptr`** for a long time), also iterators have pointer semantics."

I encouraged Szymon to write up his idea for *Overload*. (Fran is always looking for good material; sometimes she has to settle for mine.) He said he would be "happy if you would like to side-note it in your next article". Here it is, Szymon. Thanks, Bob.

References

[Linux1] **`nanosleep()`** – http://linux.die.net/man/2/nanosleep

[Linux2] **`time(7)`** – http://linux.die.net/man/7/time

[McArdell15] 'Raspberry Pi Linux User Mode GPIO in C++ [Part 2]' in *CVu*, Volume 27, Issue 4, Pg. 17

[POSIX] **`sleep ()`** – http://pubs.opengroup.org/onlinepubs/009695399/functions/sleep.html

`usleep()` – http://pubs.opengroup.org/onlinepubs/009695399/functions/usleep.html

`Nanosleep()` – http://pubs.opengroup.org/onlinepubs/009695399/functions/nanosleep.html

[Schmidt15] 'Alternatives to Singletons and Global Variables' in *Overload* 126, April 2015, Pgs. 9–13

[TI123] SN54122, SN54123, … SN74LS123: Retriggerable Monostable Multivibrators, http://www.ti.com/lit/ds/symlink/sn54123.pdf

[TI393] SN54930, SN54LS390, … SN74LS393: Decade, Divide-By-Twelve and Binary Counters, http://www.ti.com/lit/ds/symlink/sn54ls393.pdf