OVERLOAD 152

August 2019

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 153 should be submitted by 1st September 2019 and those for Overload 154 by 1st November 2019.

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

Reactive or Proactive

Reactive systems are all the rage. Frances Buontempo compares them with a proactive approach.

Having just moved house, I am writing this with a pen on a bit of paper. My PC is not yet set up, and I broke my desk, so have solid excuse for not writing an editorial this time. Having ordered a new desk, we noticed we have no door bell, so spent a day hovering by the front door, keeping our eyes open for a delivery van. In the end, it turned out that next day delivery meant the day after the company were ready, not the day after the customer placed the order. We ordered a door bell, which did turn up and I do now have a desk. Some internet may follow shortly.

We discovered the temporary lack of doorbell could be worked round with a note along with our mobile numbers explaining the predicament. A genius idea that took a couple of days to think of. In order to communicate, whether between people or computers, something like a doorbell or understood protocol is required. If the doorbell rings, you know someone is at the door. The cat thinks the world is about to end and he must run somewhere, anywhere, away from the noise. The same message will be handled differently, depending on the audience. The communication is the bing-bong and the response is up to the listener.

Even a single machine, running a single multithreaded process or several processes may need to communicate. For threaded code with shared mutable state, synchronisation points are required. More like a door, or gate, than a doorbell, to be honest; however, in order to deal with writers writing when readers may be trying to read, action must be taken. Barriers must be in place. Locks must be acquired. Deadlocks must be avoided. In contrast to this seemingly old-school approach, various Reactive frameworks are now taking hold. Our regular writer Sergey Ignatchenko has also written about Actors on various occasions [Ignatchenko18] [Ignatchenko16]. At a high level, this approach involves registering observers for specific messages, for example a doorbell ringing. A producer will announce its news, and any interested parties will act accordingly; for example, the cat will leg it. You could argue the low-level approach with locks and so on, is proactive, putting things in place in advance to cover various eventualities, while Actors/Reactive is just that: reactive. Code reacts when things happen, rather than setting things up to stop things happening. Is one approach better than the other? It depends.

A surprising number of setups I have used make it hard to unsubscribe, including an events/delegates methodology, emails, phone calls from my network service provider insisting I need to buy a new tablet with SIM card, and similar. Attempting to poll my emails from my phone, while awaiting proper internet, makes me realise how much nonsense I get. I am sure I had previously attempted to unsubscribe from many of them. A few are periodically interesting, when I have time to look, but I don't want to be cornered into an immediate response, or get further emails reminding me I have an email, and the offer ends at midnight. Which time zone? Or Google assist reminding me about deadlines for conference submissions I don't have time to submit to. In fact, on many occasions an unsubscribe event generates further emails, asking, "Are you sure?", "We miss you already!" I am sure you have similar experiences.

To be honest, unsubscribing from an event in .Net has proved difficult, at least for me [Skeet15]. I've only recently tried using the .Net Reactive framework [Github], so haven't fully explored how it works yet. I may find unsubscribing easier here once I've investigated more. Most of the time, I tend to have a listener listening for the lifetime of the program, so it doesn't really matter. However, it is something that I expect to be possible. I expect symmetry. What starts can stop. What subscribes can unsubscribe. Each action has an equal and opposite reaction. However, defining opposite and equal can be difficult [Love11]. Furthermore, expectations tend to be based on experience.

Most of my experience does come from the more old-school 'proactive' approach. I wonder if people's lives tend to fall into one of these two camps. Some 'fly by the seat of their pants', as it were, not planning much and managing to respond or react as things happen. Others might try to plan an itinerary or todo list, and include backup plans for several scenarios. When I behave like this, I can dream up worst case scenarios better than most. It isn't always helpful to spend so long plotting for every 'What if' you can think of. In code, as in life, it can come to the point where simply reporting what happened and stopping is more sensible than trying to dig your way out of a hole. You cannot handle every exceptional circumstance [Sutter19].

Now, if you are trying to write robust software, you do want to throw every scenario you can think of at the system. This can be in a logical, ordered fashion, or via a formal proofs' methodology, such as Z, for specification, development and verification [Wikipedia-1]. It could also be powered by a bit of randomness, such as fuzzers, property-based testing, mutation testing or something akin to Netflix's Chaos Monkey [Netflix]. Either approach can flush out problems and increase confidence that the code works as intended. Furthermore, seeing what happens if things go wrong gives you practice operating the system you built. You can see if the logs and error messages make any sense. You can try out turning everything off and on again, and see if the order matters or not. You will discover what happens if people disobey instructions, which is usually good to know. This is exactly why larger offices have fire drills. Proactively training participants, just in case, so people don't respond in a panic if something bad does happen. It is worth planning for some outcomes. It's worth discussing an escape plan if there's a fire in your own home, but

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

you might not need to practise fire drills every Wednesday morning. You don't need to plan every single thing in meticulous detail. Some things can be figured out on the spot, and therefore spending time panicking about unlikely outcomes is a waste of time, and can lead to some very unhealthy states of mind. Some kind of sense is required. Be proactive, and organised, to a point.

Perhaps something similar goes for project management. Certainly, a waterfall approach compared to an agile approach might at first glance seem to be proactive, detailed forward planning as opposed to a just-in-time, reactive method. This is not entirely accurate. An agile method will have some kind of backlog of work. Some forethought about things that need to be done will happen. They may get fleshed out in more detail later on, rather than up front, but it's not all purely reactive. A just-in-time Kanban [Wikipedia-2] scheme in a manufacturing setting will be more reactive: when parts are needed the system reacts and orders them, instead of planning at the start of a production run, and ordering everything up front, thereby needing storage space for all the components ordered in advance. Furthermore, a waterfall project will end up moving around blocks of work on the Gantt chart or project plan, and everyone expects this to happen. We used a Kanban board for our house move. It was useful to have the TODO list in one place, and be able to find updates without having to search through mountains of emails. We did end up with four columns: to do, doing, done, can't be bothered any more. Much more sensible that some Kanban boards I've seen, that sprawl into something more like a tree structure.

The trouble with planning things out in detail in advance is the unexpected things that happen. Or don't happen. If the builders need some scaffolding, but the scaffolders forget to put it up on both sides of the house, delays happen. The order in which work happens needs to be re-ordered. If the completion date for the house buying ends up beyond the original estimate, you might need to find a new removal firm who are free at the time. In many ways, moving home is like project management. A mixture of forward planning and being agile enough to react as things changed got us through. Having packed important things in an overnight bag meant we didn't have to rake through boxes for coffee, wine or other important elements. Reacting to the contents of the boxes as we opened them was another matter. And that was ok. A note on each box in marker pen indicating which room items had come from helped. Who knew we had so many coats? Yes, we have lots of books and cables, but coats?!

This is all a matter of pipelining and logistics. The same is true for communicating processes or threads. In fact, Communicating Sequential Processes (CSP) offer an alternative, or at least different slant to Actors and Reactive approaches. CSP is a formal language for describing patterns of interaction in concurrent systems, according to Wikipedia [Wikipedia-3]. CSP's design suited the transputer, with its pipeline processor. CSP allows modelling and analysis nearer formal proofs that can find errors testing may miss. It is built from primitives, defined as events and processes, and an algebra, think adding up – making new primitives, choosing, interleaving and hiding (according to Wikipedia). Now, some say [Vernon15] sequential processes will be a bottleneck, so Actors are much better. Specifically, it says "Sequential processes can't create other sequential processes" (page 11), going on to suggest that Actors are powerful than sequential approaches. However, the Wikipedia article claims

> the 'Sequential' part of the CSP name is now something of a misnomer, since modern CSP allows component processes to be defined both as sequential processes, and as the parallel composition of more primitive processes

Now, CSP are similar to Actors but made "fundamentally different choices with regard to the primitives they provide":

- CSP processes are anonymous, while actors have identities.
- CSP uses explicit channels for message passing, whereas actor systems transmit messages to named destination actors.
- CSP message-passing fundamentally involves a rendezvous between the processes involved in sending and receiving the message, while message-passing in actor systems is fundamentally asynchronous.

Each approach has different pros and cons, but all deal with things happening in an unknown order. Different languages may use different words, for example 'yield' for coroutines in Python [Ramalho15], or rendezvous in Ada [Miranda04]. In .Net, adding 'async' to circumspect places in code moves from proactive to reactive. The original CSP paper [Hoare78] talks about solving various problems including how to communicate, how to synchronise, and how to choose between synchronisation methods. How ever you approach code, or life, you will need to communicate, meetup, and decide a suitable method for each. You need to make a decision, and whatever you choose may work well some of the time, and be more difficult other times. There is no One True Way. Sometimes, proactive planning helps, but you need to be able to react on the spot too. Sometimes a deadlock can be solved by a sleep. If anyone wants to write up different ways of communicating or synchronising, do get in touch. I suspect most of you have tried at least one of the many approaches. Tell us about it.

References

[Github] Rx.Net: https://github.com/dotnet/reactive

[Hoare78] C.A.R. Hoare (1978) 'Communicating Sequential Processes' in *Communications of the ACM* 21:8 on pages 666– 677, available at https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf

[Ignatchenko16] Sergey Ignatchenko 'On Zero-Side-Effect Interactive Programming, Actors, and FSMs' *Overload* 131 pages 9–12, February 2016, available at https://accu.org/index.php/journals/2199

[Ignatchenko18] Sergey Ignatchenko, Dmytro Ivanchykhin and Marcos Bracco '(Re)Actor Allocation at 15 CPU Cycles' *Overload* 146 pages 14–19, August 2018, available at https://accu.org/index.php/journals/2533

[Love11] Steve Love and Roger Orr 'Some Objects Are More Equal Than Others' *Overload* 103 pages 4–9, June 2011, available at https://accu.org/index.php/journals/1971

[Miranda04] Javier Miranda and Edmond Schonberg (2004) 'The Rendezvous' in *GNAT: The GNU Ada Compiler*, online at https://www2.adacore.com/gap-static/GNAT_Book/html/node22.htm

[Netflix] Chaos Monkey: https://netflix.github.io/chaosmonkey/

[Ramalho15] Luciano Ramalho (2015) *Fluent Python: Clear, Conise and Effective Programming* published by O'Reilly, Aug 2015

[Skeet15] Jon Skeet 'Clean Event Handler winvocation with C# 6' https://codeblog.jonskeet.uk/2015/01/30/clean-event-handlers-invocation-with-c-6/

[Sutter19] Herb Sutter 'De-fragmenting C++: Making exceptions more affordable and usable' at ACCU19: https://www.youtube.com/watch?v=os7cqJ5qlzo

[Vernon15] Vaughn Vernon (2015) *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka* published by Addison-Wesley Professional, August 2015

[Wikipedia-1] 'Formal methods': https://en.wikipedia.org/wiki/Formal_methods

[Wikipedia-2] 'Kanban': https://en.wikipedia.org/wiki/Kanban

[Wikipedia-3] 'Communicating sequential processes': https://en.wikipedia.org/wiki/Communicating_sequential_processes

A Low-Latency Logging Framework

Logging can be a bottleneck in systems. Wesley Maness and Richard Reich demonstrate a low-latency logging framework that avoids common problems.

If anybody wants to build highly scalable systems, I recommend you study logging systems and then do completely the opposite. You got some hope of making a scalable system or high performance system at that stage.
Martin Thompson in *Designing for Performance*. [Thompson16]

With that in mind we hope the idea proposed in this article is an exception to Martin's observation.

We wish to utilize some of our findings related to cache-line awareness in a previous publication [Maness18] to solve a more practical real world problem often encountered by many developers: a low latency logging frame work (LLLF). What is an LLLF? One could ask five software engineers and probably get ten different answers. The same question about the concept of low latency can have numerous definitions and ranges as well as acceptable deterministic behaviors. Before we get into absolute measurements, we can instead focus on the basic concepts of an LLLF. In general, an LLLF could be thought of as a framework that allows you to capture a minimally complete set of information at run time in a path of execution. This path of execution is often referred to as a hot path or critical path. Ideally this path of execution has the properties of being as fast as possible and deterministic. The goal of the critical path is to execute business logic while at the same time capturing data about the state of business logic at important points in the execution.

One of the more common building blocks for achieving concurrent execution is a ring buffer. In our case, we are separating critical and non-critical paths of execution. The ring-buffer is a general purpose tool, which can be customized for many different use-cases. In this article, the use-case will be the LLLF. We will pay special attention to the entry into the ring-buffer, or the produce method; this is where the focus of our analysis will take place as this could potentially be the major bottleneck in the performance of the critical path. Delivering information from one thread to another is one of the fundamental operations of concurrent execution. Doing so with low latency, high determinism is paramount in an LLLF. Later, in the code section of the paper, we provide a complete source code listing of the ring buffer utilized in all analyses.

Wesley Maness has been programming C++ for over 15 years, beginning with missile defense in Washington, D.C. and most recently for various hedge funds in New York City. He has been a member of the C++ Standards Committee and SG14 since 2015. He enjoys golf, table tennis, and writing in his spare time and can be reached at wesley.maness@gmail.com

Richard Reich has 25 years of experience in software engineering ranging from digital image processing/image recognition in the 90s to low latency protocol development over CAN bus in early 2000s. Beginning in 2006, he entered the financial industry and since has developed seven low latency trading platforms and related systems. He can be reached at richard@rdrtech.com

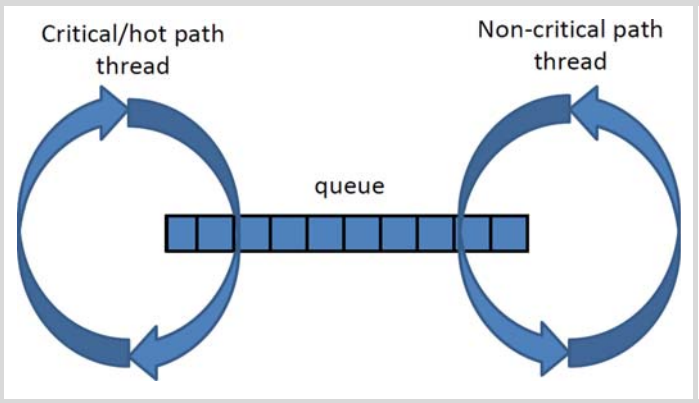

Figure 1

Once the state has been captured in the critical path of your process, it must be efficiently stored in memory for another thread and/or process to serialize the information to be archived. This could be in a human-readable format or in binary (and another process could perform the binary to human-readable conversion later). There are numerous ways in which information is passed from the thread that is executing the fast path to a secondary process and/or thread performing the serialized output as shown in Figure 1. The critical path LLLF could write into shared memory, memory, messaging infrastructure, or some shared queue. Once written to the transport medium the LLLF in the critical path would then need to notify or inform the secondary process to process the written information and write to disk. The notification is often handled by the form of an atomic increment, but there are other techniques to relay this information. The process writing to disk could have a busy loop checking the atomic variable and simply notices the value has increased and will process that information from the transport medium and serialize to disk. These are the basic building blocks of a LLLF.

Intel intrinsics [Intel] are used in this article as a way of executing Intel specific instructions. The first `_mm_prefetch` is used to load the first part of logging memory upon initialization and helps with performance at the 95th percentile and above. The second, `_mm_clflush`, is used at initialization to push all the logging memory out of the cache hierarchy to leave as much of the cache free as possible; this is a serializing instruction and is well suited for initialization purposes. Lastly `_mm_clflushopt` is used to clear populated cache lines when the consumer thread has completed its usage of the cache lines.

Definitions

In this article, the parts in an LLLF that we want to address are cache-line awareness, cache pollution and memory ordering. We will focus on optimizing the critical path's insertion of minimally complete information into some shared queue, for generating a log of operations. We need to first define some core concepts in our design and their potential impact on our LLLF. Cache-line awareness was addressed in our first article [Maness18].

Both the critical path and the non-critical path are generally spinning threads. Each thread is pinned to a different core, and the cores do not necessarily need to be on the same NUMA node

1. Cache pollution – Occupying space in the cache when not necessary.
   - When accessing or creating data that will not be used in an amount of time that it will reasonably exist in the cache. The data is evicted due to other activity before it is accessed again.
   - When accessing data that will not be accessed again, such as data that has been sent over the network.
   - When accessing large amounts of data that will only be used once.
2. Memory ordering – Memory ordering is vital for the creation of critical sections to ensure state is maintained between concurrent threads. More information can be found in the reference here [CPP].
3. Explicit atomics – when using non-default memory ordering in C++ atomic operations, careful attention must be applied to their use. However, in some cases, significant performance gains can be realized.
4. CPU pipeline – The process of executing many instructions independently of each other and discarding results that have dependencies.
5. Structure.

In Figure 1, we illustrate a very common approach to logging in low latency environments and it is the same approach we have taken for our work in this article. Both the critical path and the non-critical path are generally spinning threads. Each thread is pinned to a different core, and the cores do not necessarily need to be on the same NUMA node. There was no performance difference in the critical path if the logging thread was on another NUMA node. The queue can exist in memory, shared memory or perhaps NVDIMM.

The critical path is the path of execution that must carry out a series of well-defined operations under very specific performance criteria. These metrics are often measured in terms of latency or CPU cycles under various percentiles. For example, you would want to know how many microseconds it would take to execute a complete cycle in the critical path, or some segment of the critical path, at the 99th percentile. This measured time also includes the time it takes to place a work item onto the queue for later consumption by the non-critical thread. The work item should be a minimally complete set of information necessary to capture state at that spot in the critical path.

The non-critical thread is spinning and once it can determine there is a work item in the queue for consumption, it pops the work item off the queue, does any mappings or lookups in needs to perform, translates the work item into some human readable format and serializes to a destination, often to disk based storage.

Code

Listing 1 takes the ring buffer as an argument and casts the data to the payload type defined using the parameter pack type. This is not intended for production use, but simply demonstrate functionality.

```
using TimeStamp_t = uint64_t;

template <template<typename> typename A,
  typename... Args>
uint64_t writeLog (RingBuff& srb)
{
  using Payload_t = Payload<Args...>;
  A<Args...> arch;
  Payload_t *a =
    reinterpret_cast<Payload_t*>(
        srb.pickConsume(sizeof(Payload_t)));
  if (a == nullptr )
    return 0;
  // detect empty parameter pack
  if constexpr(sizeof...(Args) != 0)
  {
    arch.seralize(a->data);
    // properly deconstruct, may have
    // complex objects
    a->~Payload_t();
    memset((char*)a, 0, sizeof(Payload_t));
    srb.consume(sizeof(Payload_t));
  }
  return sizeof(Payload_t);
}
```

Listing 1

The snippet below shows a type that extracts underlying types from the r-value. The `NR` in `TupleNR` means no reference.

```
template <typename... T>
using TupleNR_t = std::tuple
  <typename std::decay<T>::type...>;
```

Listing 2 is the structure that is created in the ring buffer. It contains the function pointer to the method containing the parameter pack type. It is aligned to the pointer size.

Listing 3 simply constructs the payload using placement new within the ring buffer. It is worth pointing out that the code here will drop payloads that are newest in the queue, not the oldest ones. We chose this approach as it is often a requirement in financial systems to prioritize retaining older log messages over newer ones. This is because of certain regional regulatory requirements (although all should be captured and saved off). We could construct a drop policy where we can specify which to drop, older or newer payloads, and measure each policy's impact on performance (not shown here).

Listing 4 (on page 8) shows the ring buffer in its entirety.

Results

The graph shown in Figure 2 (on page 7) captures the number of cycles it takes to push the number of arguments (each argument is an 8-byte integer)

The critical path is the path of execution that must carry out a series of well-defined operations under very specific performance criteria

```
template <typename... Args>
struct alignas(sizeof(void*)) Payload
{
  using Func_t = uint64_t (*)(RingBuff&);
  Payload(Func_t f, Args&&... args)
    : func_(f)
      , data(args...)
  {
    // all of this washes away at compile time.
    auto triv_obj = [](auto a)
    {
      static_assert(
        std::is_trivially_default_constructible
        <decltype(a)>::value,
        "Trivial Default Ctor required");
      static_assert(
        std::is_trivially_constructible
        <decltype(a)>::value,
        "Trivial Ctor required");
      static_assert(std::is_trivially_destructible
        <decltype(a)>::value,
        "Trivial Dtor required");
    };
    std::apply([triv_obj](auto&&... a)
      {((triv_obj(a), ...));}, data);
  }
  Func_t func_;
  TupleNR_t<Args...> data;
};
```

Listing 2

into the ring buffer for the percentiles shown for the G10 machine. Specifications for the G10 are shown in the references section.

The graph in Figure 3 captures the number of cycles it takes to push the number of arguments into the ring buffer for the percentiles shown for the Linux laptop machine in the references section.

Clearly the benefits of a more modern architecture are shown. For example, comparing G10 to the personal laptop at 99.99th percentile, the number of cycles for 16 arguments was more than cut in half from 1132 to 576. Both systems are locked at 3GHz with CPU and IRQ isolation.

Conclusions/Summary

If you not have access to the CLFLUSHOPT [Intel19a] [Intel19b] calls, please contact us so that we can provide an auxiliary path implementation with compiler options, which we have not shown here.

Another point to make, that isn't shown here, is that if we didn't utilize the CLFLUSHOPT calls to minimize the cache pollution, we observed (in production code) much higher latencies, at 90th percentile and above. We observed no noticeable improvements in latency due to CLFLUSHOPT in micro benchmarking. It's important to note also that due to the test

```
template <typename... Args>
uint64_t userLog (Args&&... args)
{
  auto timeStamp = __rdtsc();
  using Payload_t = Payload<TimeStamp_t, Args...>;
  char* mem = data.pickProduce(sizeof(Payload_t));
  if (mem == nullptr)
  {
    ++logMiss_;
    return 0;
  }
  // The beauty of placement new!
  // A simple structure is created and memory is
  // reused as ring buffer progresses
  [[maybe_unused]]Payload_t* a = new(mem)
    Payload_t(
      writeLog<TimeStamp_t, Args...>
      , std::forward<TimeStamp_t>(timeStamp)
      , std::forward<Args>(args)...);
  data.produce(sizeof(Payload_t));
  // Consider RIAA
  data.cleanUpProduce();
  return timeStamp;
}
```

Listing 3

performances themselves, we noticed some numbers jumping around, most attributed to pipelining and branch prediction.

There are several logging frameworks [GitHub-1] [GitHub-2] that are open sourced and target the low latency crowd. We have decided not to compare them in this paper, but instead reference the loggers here and leave it as an exercise for the reader to do their own analysis and come to their own conclusions. ■

Acknowledgments

Special thanks to Frances and the review board of ACCU *Overload*.

References

[CPP] 'Memory model' on cppreference.com: https://en.cppreference.com/w/cpp/language/memory_model

[GitHub-1] 'Super fast C++ logging library', available at: https://github.com/KjellKod/spdlog

[Github-2] 'G3log', available at: https://github.com/KjellKod/g3log

[Intel] Intel Intrinsics Guide at https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[Intel19a] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, published April 2019 by the Intel Corporation, available at: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

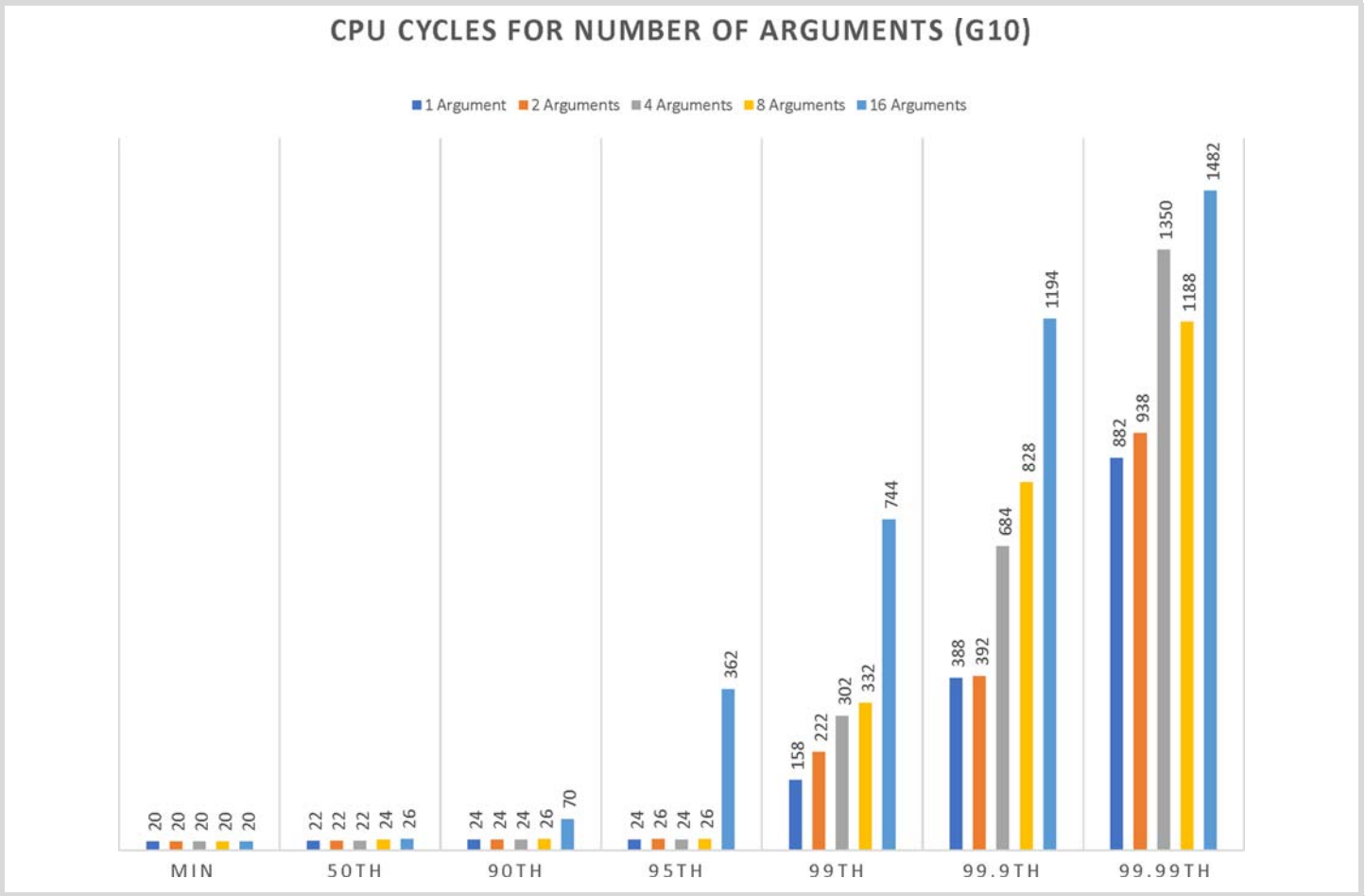

Figure 2

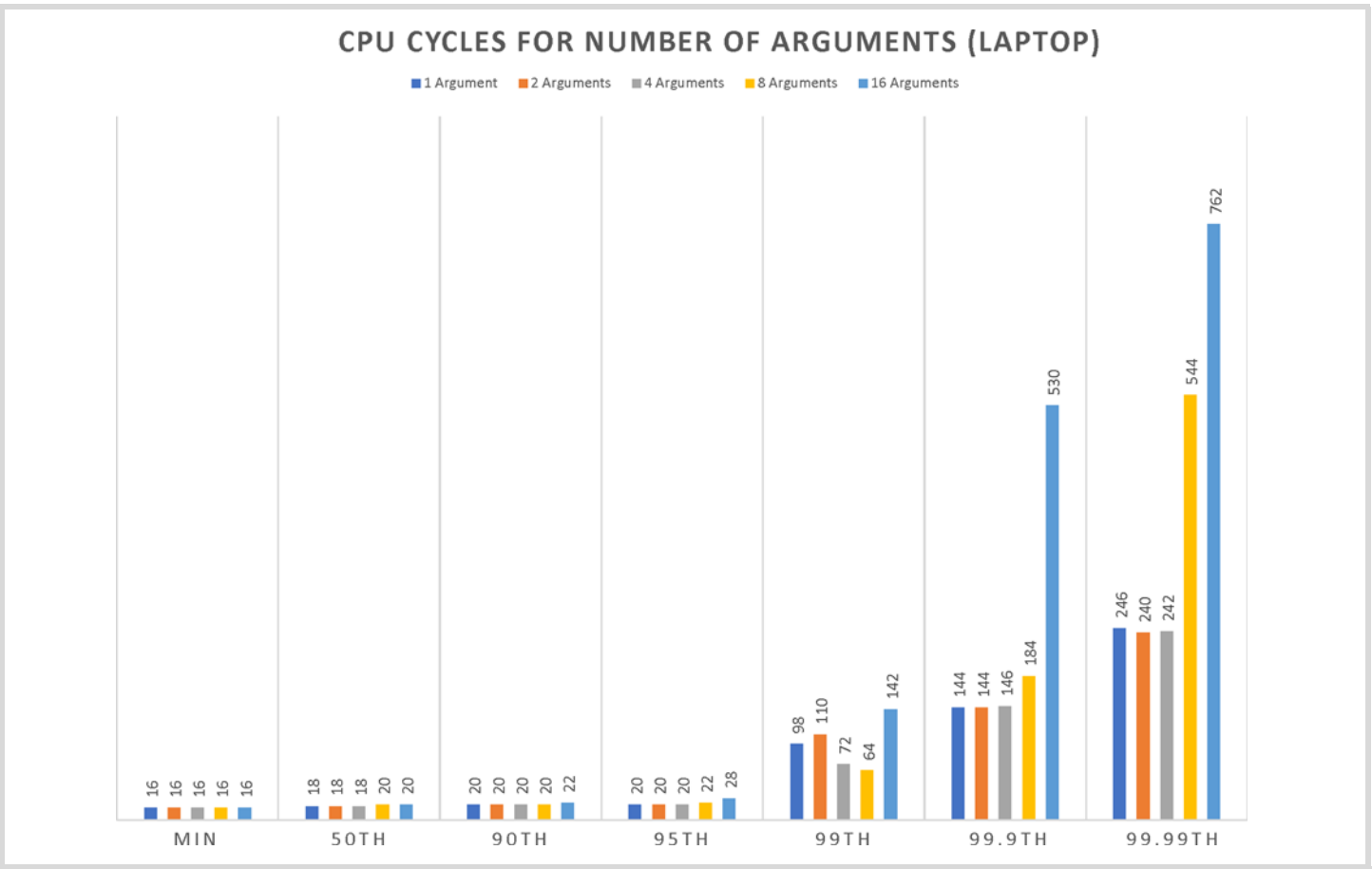

Figure 3

[Intel19b] *Intel 64 and IA-32 Architectures Software Developer's Manual*, published May 2019 by Intel Corporation, available at: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[Maness18] Wesley Maness and Richard Reich (2018) 'Cache-Line Aware Data Structures' in *Overload* 146, published August 2018, available at: https://accu.org/index.php/journals/2535

[Thompson16] Martin Thompson (2016) 'Designing for Performance' from the *Devoxx* conference, published to YouTube on 10 November 2016: https://youtu.be/03GsLxVdVzU

G10 specifications

https://h20195.www2.hpe.com/v2/getpdf.aspx/a00008180ENUS.pdf

GCC 7.1. was used on the G10 with the flags std+c++17 -Wall -O3. Dual socket 18 core (36 total) Intel ® Gold 6154 CPU @ 3GH. Hyper-threading was not enabled. CPU isolation is in place.

Laptop specifications

Gentoo with GCC 8.2

Linux localhost 4.19.27-gentoo-r1 #1 SMP Tue Mar 19 10:23:15 -00 2019 x86_64 Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz GenuineIntel GNU/Linux

```
#pragma once
#include <iostream>
#include <atomic>
#include <emmintrin.h>
#include <immintrin.h>
#include <x86intrin.h>
constexpr uint64_t cacheLine = 64;
constexpr uint64_t cacheLineMask = 63;

class RingBuff
{
  public:
    using RingBuff_t = std::unique_ptr<char[]>;
  private:
    const int32_t ringBuffSize_{0};
    const int32_t ringBuffMask_{0};
    const int32_t ringBuffOverflow_{1024};
    RingBuff_t ringBuff0_;
    char* const ringBuff_;
    std::atomic<int32_t> atomicHead_{0};
    int32_t head_{0};
    int32_t lastFlushedHead_{0};
    std::atomic<int32_t> atomicTail_{0};
    int32_t tail_{0};
    int32_t lastFlushedTail_{0};
  public:
    RingBuff() : RingBuff(1024) {}
    RingBuff(uint32_t sz)
      : ringBuffSize_(sz)
        , ringBuffMask_(ringBuffSize_-1)
        , ringBuff0_(new
          char[ringBuffSize_+ringBuffOverflow_])
        , ringBuff_{(char*)(((intptr_t)
            (ringBuff0_.get()) + cacheLineMask) &
            ~(cacheLineMask))}
    {
      for ( int i = 0;
        i < ringBuffSize_+ringBuffOverflow_;
        ++i)
      memset( ringBuff_, 0,
              ringBuffSize_+ringBuffOverflow_);
```

Listing 4

```
      // eject log memory from cache
      for ( int i = 0;
            i <ringBuffSize_+ringBuffOverflow_;
            i+= cacheLine)
        _mm_clflush(ringBuff_+i);
      // load first 100 cache lines into memory
      for (int i = 0; i < 100; ++i)
        _mm_prefetch( ringBuff_ + (i*cacheLine),
            _MM_HINT_T0);
    }
    ~RingBuff()
    {
    }
    int32_t getHead( int32_t diff = 0 )
    { return (head_+diff) & ringBuffMask_; }
    int32_t getTail( int32_t diff = 0 )
    { return (tail_+diff) & ringBuffMask_; }
    char* pickProduce (int32_t sz = 0)
    {
      auto ft = atomicTail_.load(
        std::memory_order_acquire);
      return (head_ - ft > ringBuffSize_ -
        (128+sz)) ? nullptr :
          ringBuff_ + getHead();
    }
    char* pickConsume (int32_t sz = 0)
    {
      auto fh = atomicHead_.load(
        std::memory_order_acquire);
      return fh - (tail_+sz) < 1 ? nullptr :
        ringBuff_ + getTail();
    }
    void produce ( uint32_t sz ) { head_ += sz; }
    void consume ( uint32_t sz ) { tail_ += sz; }

    uint32_t clfuCount{0};
    void cleanUp(int32_t& last, int32_t offset)
    {
      auto lDiff = last - (last & cacheLineMask);
      auto cDiff = offset -
        (offset & cacheLineMask);
      while (cDiff > lDiff)
      {
        _mm_clflushopt(ringBuff_ +
          (lDiff & ringBuffMask_));
        lDiff += cacheLine;
        last = lDiff;
        ++clfuCount;
      }
    }
    void cleanUpConsume()
    {
      cleanUp(lastFlushedTail_, tail_);
      atomicTail_.store(tail_,
          std::memory_order_release);
    }
    void cleanUpProduce()
    {
      cleanUp(lastFlushedHead_, head_);
      // signifigant improvement to fat tails
      _mm_prefetch(ringBuff_ +
        getHead(cacheLine*12), _MM_HINT_T0);

      atomicHead_.store(head_,
        std::memory_order_release);
    }
    char* get() { return ringBuff_; }
};
```

Listing 4 (cont'd)

Empty Scoped Enums as Strong Aliases for Integral Types

Scoped enums have many advantages. Lukas Böger demonstrates their use as strong types of numbers.

Scoped enumerations were one of the easy-to-grasp C++11 features that quickly spread and became the intended, superior alternative to their unscoped siblings. Local enumerator scope and forward declarations reduce namespace pollution and compilation dependencies, while prohibited implicit conversions to integral types promote type safety.

```
enum class Season {winter, spring, summer,
  autumn};

Season s1 = Season::spring; // Ok

// Error, no conversion to int:
int s2 = Season::summer;
// Error, must be Season::summer:
Season s3 = summer;
```

As with unscoped enums, it is possible to create an enumerator object from objects of its underlying type. In C++14, this is the way to go:

```
auto s4 = Season(1);
auto s5 = static_cast<Season>(2);
```

Both versions are equivalent; an explicit type conversion on the right hand side (functional notation and cast expression) is used for copy initializing s4 and s5. The necessity to detour via copy initialization seems clumsy though, the **`static_cast`** version even looks like someone forced the compiler to perform a dubious conversion without warnings. Thanks to P0138 [Reis16], C++17 mitigates this scenario by allowing for direct list initialization of scoped enums (braces mandatory, no narrowing conversions).

```
Season s6{1};
```

But why even try to construct an enumeration from a literal? Does such an initialization not defeat the whole purpose of an **`enum`**, i.e., accessing a set of constants via comprehensible names instead of magic numbers? This concern is justified, and **`auto s = Season::spring`** should indeed be the preferable way to initialize the above enumerator. But recall that enumerations without any explicit enumerator are valid, too, and then, there is no comprehensible name at hand. Empty scoped enumerations can be extraordinarily useful as strong type aliases for integral types, and the new list initialization adds the missing piece for their mainstream usage as such. But let's first cover some ground.

What is a strong type alias and what problem does it solve?

Type aliases in C++ (via the **`typedef`** or **`using`** keyword) introduce new type names, but not new types. They are *transparent*: various type aliases referring to the same underlying type can be interchanged without errors or even warnings.

```
using InventoryId = int;
using RoomNumber = int;

void store(InventoryId what, RoomNumber where);

// Ok, nothing but ints (bad!):
store(RoomNumber{2}, InventoryId{10});
```

Strengthening the restrictions on a type alias with respect to substitutability and computational base (its associated functionality) renders it a *strong type alias* or a *strong typedef*. This requires a distinct type and is used to enforce semantics at compile time and to improve the expressiveness of function parameters. Assuming a **`StrongTypeDef`** template at hand, the above example could be written as

```
// Use a tag type as 2nd template parameter to
// create unique types:
using InventoryId = StrongTypeDef<int,
  struct InventoryIdTag>;
using RoomNumber = StrongTypeDef<int,
  struct RoomNumberTag>;

void store(InventoryId what, RoomNumber where);

// Error, types don't match (good!):
store(RoomNumber{2}, InventoryId{10});
```

References to such techniques are numerous, see e.g. Matthew Wilson's early outline and example implementation [Wilson03], Scott Meyer's 'Make interfaces easy to use correctly and hard to use incorrectly' in *Effective C++* [Meyers05] or Ben Deane's talk 'Using Types Effectively' [Deane16]. Exemplary implementations for internal purposes can be found in the Boost Serialization library [Ramey], Llvm [Lattner04] or Chromium [Chromium], while distinct libraries with strong type templates are e.g. type_safe [Müller] and Named Type [Boccara].

What is the design space of strong type aliases?

The smallest and most restrictive set of operations is explicit construction from the underlying type and explicit conversion to it. The other extreme is a mirror of the complete computational base of the wrapped type (in case of an **`int`**, this includes bitshifting, modulo operators and so on). Most approaches are somewhere in the middle. Their design requires answers to the following questions.

- Type safety and constructability: allow implicit conversions from or to the underlying type (both doesn't make any sense)? Provide a default constructor?
- Uniqueness upon reuse: create a new type by an additional tag type template parameter or wrap the strong typedef definition into a macro?
- Comparison and arithmetic operators: when wrapping types that support those, mirror a subset? When construction is explicit, should binary functions be duplicated for one parameter of the underlying type?

Lukas Böger is a civil engineer who stuck with Fortran77 during his PhD program and started a C++ side project to alleviate his frustration. This worked out, and he now develops power electronics simulation software for Plexim, Zürich. He likes reading, brass music, Newton mechanics and his family. Reach him via mail@lboeger.de.

The smallest and most restrictive set of operations is explicit construction from the underlying type and explicit conversion to it

- Hashing, serialization, parsing: support insertion into `std::unordered_set`/`map`? Offer `operator<<` and/or `operator>>` for standard library streams? Support the upcoming fmt library?

Finding agreeable answers for these questions is hard. An attempt to standardize 'Opaque Typedefs' as first-class C++ citizens could not succeed, see N3741 [Brown18], and hence, when a strong typedef is needed, we must choose a library solution or ship our own – except when the wrapped type is an integral one.

How do empty scoped enumerators fit in?

Let's clearly state that once again: scoped enumerators are restricted with respect to the wrapped type: it must be an integral type (`bool`, `int`, `unsigned short`, etc.). When a `double` or a `std::string` are involved, you are out of luck. But it turns out that integral types are the most commonly used ones; a quick-and-dirty regex scan of the Chromium sources showed that around half of all strong types wrap integral values. This is what the above example looks like with an empty enumeration:

```
enum class InventoryId {};
enum class RoomNumber {};

void store(InventoryId what, RoomNumber where);

// Error, types don't match (good!):
store(RoomNumber{2}, InventoryId{10});
```

Scoped enumerators must be explicitly constructed. Narrowing conversions during construction are invalid (which cannot even be enforced with a generic library type), default construction is allowed and yields zero or `false`. Retrieval of the underlying type requires a cast (functional, C-style or `static_cast`), so no laziness here. By default, the underlying type of a scoped enumeration is `int`, but this can be adjusted. Every definition creates a completely new type, but their definition is trivial – this is a clean, built-in solution, requiring neither a macro nor an additional tag type. Objects of one type are totally ordered through the usual comparison operators, and std::hash works out of the box. Standard arithmetic or IO operations are not supported. Manually adding them as needed is straightforward, though admittedly, with many such enumerators, you will either end up polluting some namespace with greedy operator templates or go with a macro to not repeatedly implement the same functions for different types[1].

1. Strong type alias templates scale better here, as they provide operators through the Barton-Nackmann trick. This can get out of hand, though; ambitious solutions risk duplicating the Boost Operator Library.

So when am I supposed to use empty scoped enumerations?

Every time a strong integral type seems handy for a function signature, an API, a vocabulary type in your project – empty scoped enums should be your first consideration. These types are dead simple, they are as efficient as expressive and require no external dependency. C++17 makes them easy to instantiate, there is no burden left that keeps you from leveraging their strengths. Just keep that in mind for the next time you write an interface that uses integral parameters! ■

References

[Boccara] Jonathan Boccara *et al.*, Named Type, a header-only library for strong types. On GitHub at https://github.com/joboccara/NamedType/blob/master/named_type_impl.hpp

[Brown18] Walter Brown (2018) 'Toward Opaque Typedefs for C++1Y', v2 ISO/IEC JTC1/SC22/WG21 document N3741, 2018-08-30. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3741.pdf

[Chromium] Chromium Project, sources as retrieved in June 2019 from https://chromium.googlesource.com/chromium/src.git/+/refs/heads/master/base/util/type_safety/strong_alias.h

[Deane16] Ben Deane (2016) 'Using Types Effectively' from *CppCon 2016*, available at https://www.youtube.com/watch?v=ojZbFIQSdl8

[Lattner04] Chris Lattner and Vikram Adve (2004) 'Llvm: A Compilation Framework for Lifelong Program Analysis and Transformation' in *Proc. of the 2004 International Symposium on Code Generation and Optimization*, Palo Alto, California, 2004. The implementation on GitHub is available: https://github.com/llvm/llvm-project/blob/master/llvm/include/llvm/Support/YAMLTraits.h

[Meyers05] Scott Meyers (2005) *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd. edition, Addison-Wesley.

[Müller] Jonathan Müller *et al.*, type_safe: Zero overhead utilities for preventing bugs at compile time. The implementation on GitHub is available: https://github.com/foonathan/type_safe/blob/master/include/type_safe/strong_typedef.hpp

[Ramey] Robert Ramey *et al.*, Boost Serialization Library, Version 1.70. BOOST_STRONG_TYPEDEF (documentation) available at https://www.boost.org/doc/libs/1_70_0/libs/serialization/doc/strong_typedef.html

[Reis16] Gabriel Dos Reis, Construction Rules for enum class Values', ISO/IEC JTC1/SC22/WG21 document P0138, 2016-03-04. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0138r2.pdf

[Wilson03] Matthew Wilson (2003) 'True typedefs' in *Dr. Dobb's Journal*, dated 01 March 2003. Available at: http://www.drdobbs.com/true-typedefs/184401633

C++ Reflection for Python Binding

There are various approaches to generating Python bindings in C++. Russell Standish shows how Classdesc can be used to achieve this.

Since 2000, Classdesc has provided reflection capability for C++ objects, with no dependencies other than the standard C++ library. Typical applications include serialisation of objects to a variety of different stream formats, and language bindings to non-C++ languages. With the increased popularity of Python, this work looks at automatically providing Python bindings to C++ objects using the Classdesc system.

Introduction

Classdesc is a mature C++ reflection system with nearly two decades of continuous development [Madina01], [Standish16]. It has often been used to implement automatic serialisation of objects, but also for the automatic creation of language bindings, in particular for Objective C [Leow03] (largely obsoleted by the Objective C++ language), TCL [Standish03] and Java, leveraging the JNI API [Liang99].

At its core, Classdesc is a C++ preprocessor that reads C++ header files and generates overloaded functions that recursively call themselves on members of the class. The collection of overloaded functions is called a *descriptor*. More details can be found in [Standish16].

With the recent popularity of Python as a scripting language, and also this author's adoption of Python as a general purpose scripting language, this work seeks to apply Classdesc to the problem of automatically generating Python language bindings for C++ objects.

The CPython implementation supplies a C language API [Rossum02] which is quite low level. The Boost library provides a much higher level API over the top of the C API [Abrahams03], more closely suited to reflecting C++ objects into Python objects, providing a much appreciated leg up for creating a Python binding descriptor.

Background

Boost Python

The ever popular Boost library contains a C++ abstraction layer over the top of Python's C API [Abrahams03]. This provides a mapping between C++ objects and equivalent Python objects. Since C++ does not provide native reflection capability, a programmer using Boost.Python must explicitly code bindings for methods and attributes to be exposed to Python. This is made as simple as possible, using C++ templates, but still involves quite a bit of boilerplate code on behalf of the user, with the concommitent maintenance overhead as class definitions evolve during software development. This work seeks to extend the Classdesc library to automatically provide these bindings from the Classdesc processor. In this section I summarise the features of Boost.Python used for the Classdesc Python descriptor.

- **Exposing C++ classes to Python**

  To start defining a Python class, the programmer needs to instantiate an object of the type `class_<T>`, where `T` is the C++ class type being exposed, and the single string argument is used to give the Python type name. By default, `T` is assumed to be default constructible and copyable. If either one of these assumptions is not true, then the `boost::python::no_init` object or the `boost::noncopyable` type must be passed to the `class_` constructor as appropriate. No particular reason is given why it is an object in one case, and a type in the other. In this Classdesc library, we use standard C++ type traits to pass these additional arguments to Boost.Python, according to the type `T` being processed. This will be discussed in more detail in 'Default and Copy constructibility' on page 15.

- **Exposing methods**

  With the previously defined class object, you can expose methods with the `def` method, for example:

```
class<Foo>("Foo").
  def("bar", &Foo::bar).
  def("foobar", &Foo::foobar);
```

  The `def` method (as well as the equivalent attribute methods) returns a reference to `this`, so that the calls can be chained as shown above.

  It should be noted that by default, the exposed methods return a copy of the returned object. This is not always what you want – particularly if the method returns a reference to an internally contained object, such as the indexing method of a vector or map. In such a case, you would want to be able to call mutating method or set attributes of the returned object, not a copy.

  Boost.Python does provide a method for manually specifying return by reference, but this requires additional manual specification by means of passing an additional argument of type `boost::python::return_internal_reference<>` to `def`. In this Classdesc library, we use C++-11 metaprogramming techniques to distinguish methods returning a reference, and those returning a value, and supply the extra Boost.Python specification automatically. This will be described in more detail in 'Reference returning methods' on page 14.

- **Overloaded methods**

  Boost.Python does support overloaded methods. In order for this to work, you need to specify the functional signature for the method pointer, eg:

```
void (Foo::*bar0)()=&Foo::bar;
void (Foo::*bar1)(int)=&Foo::bar;
class_<Foo>("Foo").
  def("bar",bar0).
  def("bar",bar1);
```

Russell Standish gained a PhD in Theoretical Physics, and has had a long career in computational science and high performance computing. Currently, he operates a consultancy specialising in computational science and HPC, with a range of clients from academia and the private sector. Russell can be contacted at hpcoder@hpcoders.com.au

Surprisingly, Boost.Python does not provide any way of exposing global variables to the module namespace

will expose the overloaded bar method to Python. In this work, the Classdesc processor, which hitherto ignored overloaded methods, is extended to provide these functional signatures in order to support overloaded methods. This is described in more detail in 'Handling overloaded methods' on page 13.

- **Exposing attributes**

  Similar to exposing methods, attributes are exposed using the specific `boost::python::class_<T>` methods `def_readwrite` and `def_readonly`. Similar to the case with return reference values, we use metaprogramming techniques to automatically distinguish between const attributes and mutable ones. Details can be found in 'Reference returning methods' on page 14.

- **Global functions**

  Global or namespace scope functions can be exposed to the python module namespace via a global `def` function.

- **Global variables**

  Surprisingly, Boost.Python does not provide any way of exposing global variables to the module namespace. In the Classdesc Python descriptor, a method `python_t::addObject` is provided that explicitly adds a reference to the global object to the `__dict__` of the namespace C++ objects are being added to. More details to come in 'Global objects' on page 15.

- **Python object wrappers**

  Boost.Python provides the `boost::python::object` class, which wraps the `Py_Object` type from the C API. There is an `boost::python::extract<T>()` template which attempts to downcast the object to a C++ object of type `T`, rather analogous to C++'s `dynamic_cast` operator. An exception is thrown (which is automatically converted to a Python exception if propagated into python) if the object doesn't refer to the named type `T`. Various C++ wrapper types are provided to represent native Python types such as `tuple`, `list` and `dict`, each of which support the usual Python operations such as a `[]` operation or `len` function, using C++ operator overloading where necessary.

Classdesc

Classdesc is a C++ processor that parses user defined types (classes, structs, unions and enums), and emits definitions of descriptors, which are overloaded functional templates that recursively call the descriptors on the members of the structured type. For enums, a table of symbolic name strings to enum label values is constructed. A hand written descriptor needs to be provided for dependent library types for which you don't want to run Classdesc on. Classdesc provides implementations of these for most of the base language and standard library types.

In earlier versions of Classdesc, descriptors were quite literally overloaded global functions. However because functions do not support partial specialisation, and the difficulty in generating templated friend statements for functions, in more recent versions of Classdesc, descriptors are templated functor objects. A global template function is provided to instantiate and call into the functor objects. In the explanation that follows, the function form of the descriptor will be used, as conceptually that is easier to understand.

Classdesc has been in active development since 2000 [Madina01], [Standish16].

Extending the Classdesc processor

Modification of Classdesc to emit descriptors for unbound member pointers

Traditionally, Classdesc's descriptors work by being passed an instance of an object, and the automatically generated descriptors recursively call that descriptor on references to the members of that instance object. Boost.Python, however is oriented around defining class objects, registering C++ member pointers for the attributes and methods of the class. This allows Python to potentially control the lifetime of C++ objects, for example to create temporary copies for value returning methods.

So fairly early on, it became clear that Classdesc needed to be extended to define descriptors for classes, rather than instance objects. So the traditional Classdesc descriptor

```
template <class T> void python(python_t&,
  const string&,T& obj);
```

needs to be augmented with an additional form:

```
template <class T> void python(python_t&,
  const string&);
```

The former descriptor is used like:

```
Foo f;
python_t p;
python(p,"f",f);
```

The latter form requires the explicit specification of the class:

```
python_t p;
python<Foo>(p,"");
```

In the traditional mode, the Classdesc processor emits definitions of the form:

```
void python(python_t& p, const string& d, Foo& a)
{
  python(p,d+".bar",a.bar);
  python(p,d+".method1",a,&Foo::method1);
}
```

we need definitions that do not pass an instance object:

```
template <class C=Foo>
void python<C,Foo>(python_t& p, const string& d)
{
  python_type<C,Foo>(p,d+".bar",&Foo::bar);
  python_type<C,Foo>(p,d+".method1",
    &Foo::method1);
  }
```

it was decided for consistency to change the form of instance descriptors for object attributes to pass both object and member pointer

where **python_type** is a type descriptor, as opposed to the traditional instance object descriptor. The reason why two type arguments are required in the template arguments is to handle inheritance. If Bar is derived from Foo, then we would see

```
template <> void python<Bar,Bar>(python_t& p,
  const string& d)
{
  python<Bar,Foo>(p,d); // process base class
  ...
}
```

As part of the process of adding support for type descriptors, it was decided for consistency to change the form of instance descriptors for object attributes to pass both object and member pointer, just like how method pointers are handled. This mode is enabled by the Classdesc processor switch **-use_mbr_ptrs**, and this will become the default way things are done in the next major version (4.x) of Classdesc. Support for the old way of doing things is enabled with a macro **CLASSDESC_USE_OLDSTYLE_MEMBER_OBJECTS** defined in the `use_mbr_pointers.h` header file. So

```
CLASSDESC_USE_OLDSTYLE_MEMBER_OBJECTS(pack)
```

creates a descriptor overload that binds the member pointer to the object, and calls the traditional overload:

```
template <class C, class M>
void pack(pack_t& p, const string& d, C& o, M y)
{pack(p,d,o.*y);}
```

Handling overloaded methods

Given:

```
void (Foo::*bar0)()=&Foo::bar;
void (Foo::*bar1)(int x)=&Foo::bar;
python(py,"bar",bar0);
python(py,"bar",bar1);
```

the first descriptor is called on argumentless method and the second on the single integer argument one.

Method overloading support was added to Classdesc by modifying the processor to emit these function signature qualified method pointers, instead of the inline 'address of' traditionally used. This required parsing the method declaration lines to extract the return type, the method name, the method arguments and the method type. For example:

```
virtual const Bar& foo(const FooBar& x={}) const;
```

In this example, the return type is **const Bar&**, the method name **foo**, the argument list **const FooBar& x** and the type **const**. Some support for extracting this information had been added to Classdesc for the Objective C work [Leow03]; however, that was deficient in a number of ways. In particular keywords like **virtual**, **inline**, **static** and **override** need to be filtered out. So the member name is immediately recognisable as the token prior to the first parenthesis, however we must take all tokens preceding the member name (minus those keywords) as the return type. Similarly for the argument list, we must strip out all initialisers present. We can leave the argument name in place (if present), which is fortunate, as we cannot assume the last token of an argument is a name, and not part of the type. Finally, the type is important, as the emitted declaration must vary accordingly:

type	declaration
none	`const Bar& (Foo::*m)(const FooBar& x)=&Foo::foo;`
const	`const Bar& (Foo::*m)(const FooBar& x)const=&Foo::foo;`
static	`const Bar& (*m)(const FooBar& x)=&Foo::foo;`
constructor	`void (*m)(const FooBar& x)=0;`

The constructor case will be discussed in 'Constructors', below.

Overloading support is enabled by the **-overload** command line switch. It is not enabled by default (nor will it be in Classdesc 4.x) because Classdesc is not aware of the full context of the class it is processing. Type **Bar** may be declared in the current namespace, but in a different header file to what Classdesc is processing, it may come into scope by inheritance. In which case, the type name will not be in scope at the site of the descriptor declaration. Classdesc tries to import symbols it knows about – so publicly defined types of the current class are explicitly imported, as are types defined in the enclosing namespace the class is defined in. However, there will always be situations it cannot work out where a type is defined, and so a compilation error will ensue. The answer is that you need to explicitly qualify these types such that they can be found in the **classdesc_access** namespace – for example you may need to change the above declaration to

```
virtual const FooBase::Bar&
  foo(const FooBar& x={}) const;
```

in the case where **Bar** is defined in the base class **FooBase**.

Of course, if the descriptor ignores methods (eg any of the serialisation descriptors), then it is not necessary to enable overloading, eliminating this problem.

At the time of writing no effort is made to parse default arguments. If you wish to emulate default arguments in the Python interface, then you will need to provide explicit method overloads. This may change in the future.

Constructors

Traditionally, all constructors were ignored by Classdesc, as it is impossible to obtain a method pointer to a constructor. But it is such a powerful feature to be able to construct a C++ object by whatever constructors it provides (and to construct objects that have no default constructor), that this work added the ability to expose constructors. We use the code for parsing method signatures, and declare a temporary function pointer with the same arguments as the constructor, initialised to NULL.

we use the Classdesc-provided logical metaprogramming operations for combining different type traits

```
template <class M,
  int N=functional::Arity<M>::value> struct Init;
template <class... A> struct InitArgs;
template <class A, class... B>
struct InitArgs<InitArgs<B...>, A>:
  public InitArgs<B...,A> {};

template <class M, int N, class... A>
struct InitArgs<Init<M,N>, A...>:
  public InitArgs<Init<M,N-1>,
     typename functional::Arg<M,N>::T,A...>
{};

template <class M, class... A>
struct InitArgs<Init<M,0>, A...>
{typedef boost::python::init<A...> T;};

template <class M, int N>
struct Init: public InitArgs<Init<M, N>>
  {};
```

Listing 1

When passed to the python descriptor, we need to extract the types of each argument and construct a `boost::python::init<>` specialisation with those argument types. Instantiating an object of this type and passing it to `def()` is all that is required to expose the constructor to Python.

Metaprogramming is used for this purpose, leveraging the `classdesc::functional` metaprogramming library and modern C++ variadic templates. The code implementing this is in Listing 1.

The idea is that `M` is the type of the function pointer passed to the descriptor, and the second argument being the number of arguments to process, initialised by default template argument set to the function's arity. This value is decremented as the arguments are unpacked into the variadic type argument pack `A...`, and when finally reaching 0, defines the output type to `boost::python::init<A...>`.

This technique does require a helper template class to carry the argument pack – in this instance called `InitArgs`. An initial attempt at repurposing `boost::python::init` failed because that class carried too much baggage.

The python descriptor

Reference returning methods

As mentioned in 'Boost Python' on page 11, Boost.Python wrapped methods return a copy of the returned object, even if the method returns a reference to an object intended for mutating the object state. Boost.Python provides an alternate version of def that handles this case using the following syntax:

```
class_<Foo>.def("bar",&Foo::bar,
  return_internal_reference<>());
```

```
// value returns
template <class C, class M>
typename enable_if<
  And<
    Not<is_reference<typename
      functional::Return<M>::T>>,
    Not<is_pointer<typename
      functional::Return<M>::T>> >,
      void>::T
addMemberFunction(const string& d, M m)
{
  auto& c=getClass<C>();
  if (!c.completed)
    c.def(tail(d).c_str(),m);
  DefineArgClasses<M, functional::Arity<M>
    ::value>::define(*this);
}
// reference returns
template <class C, class M>
typename enable_if<is_reference
  <typename functional::Return<M>::T>,void>::T
addMemberFunction(const string& d, M m)
{
  auto& c=getClass<C>();
  if (!c.completed)
    c.def(tail(d).c_str(),m,
    boost::python::return_internal_reference<>());
  DefineArgClasses<M,functional::Arity<M>::
    value>::define(*this);
}
// ignore pointer returns
template <class C, class M>
typename enable_if<is_pointer
  <typename functional::Return<M>::T>,void>::T
addMemberFunction(const string&, M) {}
```

Listing 2

In order to emit this alternative syntax, we use the std type_traits library, the `enable_if` metaprogramming trick (Classdesc provides its own implementation of this, modelled on the version supplied as part of the Boost metaprogramming library) and the Classdesc functional library, which provides metaprogramming templates returning a function object's arity, its return value and the types of all its arguments. So we see code like that in Listing 2.

Here we use the Classdesc-provided logical metaprogramming operations for combining different type traits, as you can see in the default `def` case, where we want to exclude both reference and pointer returning methods from being exposed via a copied return object.

The third case corresponds to pointer returning methods. Because ownership of pointees may or may not be passed with the pointer being returned, calling these functions from python potentially creates a memory

The ClassBase base class is a non-templated virtual base class, allowing the use of this type in containers

leak, or worse. So if you want to expose a pointer returning function to python, do it the old-fashioned way, ie explicitly, not automatically. Better is to recode the method to return a reference, for which it is clear by language semantics that ownership remains with the method's C++ object, if that is your intention.

Another point of interest is that the python descriptor is called recursively on the return type, and on the types of each argument which ensures that a python definition for those types exists. To do this, we again leverage the Classdesc functional library, and use template recursion to call the python descriptor on each argument. Care must be taken to avoid an infinite loop caused when the method takes an argument of the same type as the class being processed. For this purpose, we use a single shot singleton pattern to return false the first time it is called and true thereafter for a given type:

```
template <class T>
inline bool classDefStarted()
{
  static bool value=false;
  if (value) return true;
  value=true;
  return false;
}
```

This is adequate for when each class needs to exposed just once per execution (such as on loading a dynamic library).

The code for recursively exposing the types of each argument is in Listing 3 and is called from within a `python_t` method by

```
DefineArgClasses<F,functional::Arity<F>::value>
  ::define(*this);
```

Const and mutable attributes

We can similarly deal with the different syntactic requirements of const or noncopyable versus non const attributes, using standard metaprogramming techniques (see Listing 4).

In this case, not only do we check whether an attribute is declared const, but we also need to check whether the attribute is even copy assignable.

Default and Copy constructibility

By default, Boost.Python assumes that an exposed C++ object is default constructible and copy constructible. As already mentioned in 'Boost Python' on page 11, non-default constructible classes can be handled by passing an object of type `boost::python::no_init` to its constructor. Noncopyable objects can be exposed if the `class_` template takes an extra template parameter of `boost::noncopyable`. To make the code more symmetric, and shareable in these cases, in Classdesc the `class_` template is subclassed as shown in Listing 5.

The ClassBase base class is a non-templated virtual base class, allowing the use of this type in containers, and the use of a boolean template parameter allows us to instantiate the Class object via a factory function:

```
// recursively define classes of arguments
template <class F, int N>
struct DefineArgClasses {
  static void define(python_t& p) {
    typedef typename remove_const<
      typename remove_reference<
        typename functional::Arg<F,N>::T>
      ::type>
    ::type T;
    if (!pythonDetail::classDefStarted<T>())
      p.defineClass<T>();
    DefineArgClasses<F,N-1>::define(p);
  }
};
template <class F>
struct DefineArgClasses<F,0> {
  static void define(python_t& p) {
    typedef typename remove_const<
      typename remove_reference<
        typename functional::Return<F>::T>
      ::type>
    ::type T;
    if (!pythonDetail::classDefStarted<T>())
      // define return type
      p.defineClass<T>();
  }
};
```

Listing 3

```
template <class T>
Class<T,is_copy_constructible<T>::value>&
  getClass();
```

Use of the default constructor is deliberately suppressed by the `no_init` argument to the `class_` constructor, but then added back in by the call to `addDefaultConstructor` if `T` is default constructible.

At the same time, an equality operator is defined that at minimum returns true if the same C++ object is referenced by two different python objects, but also calls into the C++ equality operation if that is defined.

Global objects

Surprisingly, Boost.Python doesn't provide any means of exposing a global object. Unlike the global `def()` function which exposes functions to python, there is no equivalent global `def_readwrite()` or `def_readonly()`, nor is the module available as a class object that we could run those as methods.

Instead, we can use more primitive objects – the module is available as Boost.Python object via the default constructor of `scope`. We can extract the `__dict__` attribute of this object, and insert the global object into the module dictionary via a pointer proxy:

```
template <class X>
typename enable_if<
  And<
  std::is_copy_assignable <typename
     pythonDetail::MemberType<X>::T>,
    Not<is_const<typename
     pythonDetail::MemberType<X>::T>>
    >,void>::T
addProperty(const string& d, X x)
  {this->def_readwrite(d.c_str(),x);}

template <class X>
typename enable_if<
  Or<
    Not<std::is_copy_assignable<typename
       pythonDetail::MemberType<X>::T>>,
      is_const<typename
       pythonDetail::MemberType<X>::T>
    >,void>::T
addProperty(const string& d, X x)
{this->def_readonly(d.c_str(),x);}
```

Listing 4

```
template <class T, bool copiable> struct PyClass;
template <class T> struct PyClass<T,true>:
  public boost::python::class_<T>
{
  PyClass(const char* n):
    boost::python::class_<T>(n,
    boost::python::no_init()){}
};
template <class T> struct PyClass<T,false>:
  public
    boost::python::class_<T,boost::noncopyable>
{
  PyClass(const char* n):
    boost::python::class_<T,boost::noncopyable>(n,
    boost::python::no_init()){}
};
template <class T, bool copiable> struct Class:
  public ClassBase,
  public ClassBase::PyClass<T,copiable>
  {
    addDefaultConstructor(*this);
    def("__eq__",
      pythonDetail::defaultEquality<T>);
  }
```

Listing 5

```
  extract<dict>(scope().attr("__dict__"))
  ()[tail(d).c_str()]=ptr(&o);
```

Thus exposing a global object to Python is a matter of calling the `python_t::addObject(string name, T& obj)` method.

Containers

Boost.Python does not explictly support standard containers, such as `std::vector` or `std::list`. In Classdesc, the philosophy is to support standard containers, or better still the concepts behind standard containers. In Classdesc, two concepts are defined: *sequence* and *associative_container*. Sequences include `std::vector`, `std::list` and `std::deque`. Associative containers include `std::set`, `std::map`, `std::multimap`, and the unordered versions of these, such as `std::unordered_map`.

Users can exploit these concepts for their own containers by defining the appropriate type trait: `is_sequence` or `is_associative_array` (see Listing 6).

```
namespace classdesc
{
  template<>
  struct is_sequence<MySequence>
  {
    static const bool value=true;
  };
}
```

Listing 6

In the case of the python descriptor, we want containers to support the Python sequence protocol. It suffices to define the methods `__len__`, `__getitem__` and `__setitem__`. This is sufficient to support Python operations such as `len()`, `[]` and to iterate over a container like:

```
for i in someVector:
  print(i)
```

Additionally, it is useful to be able to assign lists of objects to sequence containers. For this, we create an `assign` method, which takes `boost::python::object` as an argument, and attempt to assign each component of the boost object (if it supports the sequence protocol).

Finally, it is desirable to construct a new C++ container from a Python list or tuple. Doing this is not well documented in Boost.Python, but involves def'ing the `__init__` method with a very special function:

```
template <class T>
boost::shared_ptr<T>
  constructFromList(const boost::python::list& y)
{
  boost::shared_ptr<T> x(new T);
  assignList(*x,y);
  return x;
}

getClass<std::vector<T> >().
  def("__init__",boost::python::make_constructor
    (constructFromList<std::vector<T>>));
```

The crucial key is that the actual constructor implementation must return a `boost::shared_ptr<T>`.

Finally, standard container are archetypical template types, and after Classdesc processing, the class names are not syntactically valid Python types. For example, a `std::vector<int>` cannot be directly instantiated, however it is possible to reference that type from within Python. Assuming the C++ classes are exposed within namespace `example`, then you can rename the constructor functions within Python like:

```
IntVector=example.std.__dict__['vector<int>']
x=IntVector([1,2,3,4])
```

One final nice to have feature not currently implemented is to directly pass a list or tuple to a C++ sequence parameter of a method. For now, you have to explicitly instantiate a C++ object of the appropriate type to pass to the argument, as above, or alternatively code an overloaded method in C++ that takes a `boost::python::object` in place of the sequence parameter, and then use the python sequence protocol (`len()`, `operator[]`) to construct an appropriate C++ container. The difficulty in arranging this to happen is that it is an area poorly documented in Boost.Python, so is still a subject of future research.

Smart Pointers

The standard library smart pointers implement a concept *smart_ptr*, which has the following methods and attributes:

- **target** is the object being referenced by the smart pointer. You can access or assign that smart pointer's target via this attribute. So

  ```
  x.target.foo()
  ```

  is equivalent to the C++ code

  ```
  x->foo();
  ```

and

```
x.target=y
```

is equivalent to the C++ code

```
*x=y;
```

If the object is null for either of these operations, a null dereference exception is thrown.

- **reset()** sets the smart pointer to null, deleting the target object if the reference count goes to zero.
- **new(args)** creates a new target object by its default constructor, or with *args* if an init method exists for that object.
- = assigning a smart pointer will cause the reference to be shared to the new variable (in the **shared_ptr** case) or transferred (in the **unique_ptr** case).
- **refCnt()** returns the reference count pointing to the current target. For **unique_ptr** this will be 1. This can be of use for debugging why a destructor is not being called when expected.

Conversion of an existing codebase

In order to test these ideas out and to harden the implementation, it is necessary to use them in a real world application. The SciDAVis plotting application [Benkert14] was chosen for this purpose, as the author is the project manager, and it already sports a python interface via the SIP reflection system [Riverbank] by Riverbank Computing for exposing C++ objects to Python. SIP was exploited to expose Qt [Blanchette06] and Qwt [Rathmann] classes, in the form of the library PyQt library.

This work also hopefully addresses a problem with using SIP in that the MXE (http://mxe.cc) cross-compiler build environment does not readily build PyQt (the build process requires a working python interpreter for the target system, which is not so useful for cross compilers). This has led to the lack of python scriptability on the Windows build of SciDAVis.

A second problem, hopefully addressed with this work, is that the API change from Qwt5 to Qwt6 does not interact well with SIP. It could be argued that the API change is a backward step – the new API is harder to use and more error prone, but suffice it to say it becomes important to wrap the new Qwt6 classes with ones that have a more C++ style, supporting RAII semantics for example. This wrapping will insulate the Python layer from the Qwt layer.

Neither of these last two advantages have been realised yet – that is the scope of future work. However, the full Python interface, as documented in the SciDAVis manual has been implemented, in a feedback process that led to many additional features in the Classdesc python descriptor, such as supporting method overloading and constructor support.

qmake project file changes

SciDAVis uses the qmake [Blanchette06] build system. Unlike GNU make, which can exploit the C++ compiler to automatically generate dependencies of object files on the included header files, qmake requires all header files to be explicitly listed. Whilst qmake does understand the dependency relations between object files and headers, it doesn't appear to have any way of specifying a dependency between a Classdesc descriptor implementation file (.cd) and its header (.h). Instead, a separate list of header files to be processed by Classdesc is maintained, and if any of those header files change, then all classdesc'd headers are processed again by Classdesc. This does cause more compilation work than is necessary, but in practice was not a major problem for SciDAVis, which has fairly modest build times. It should be noted that a direct make solution, such as used by the Minsky project does not have this problem.

Qt meta object compiler

SciDAVis is a Qt project, which has certain implications. The first is that Qt has a form of reflection called *moc*, short for meta object compiler. Qt header files are written in a superset of C++, the most significant change being keywords supporting Qt's *signals and slots* mechanism. The keywords **signals** and **slots** appear in class definitions in the same place that the class access specifiers **public**, **protected** and **private** are used. Signals are always protected, but slots may be declared public, private or protected. Additional code was added to the Classdesc processor to parse these declarations, and set the **is_private** flag appropriately.

The other aspect of Qt code is specific macros used to indicate things to the moc preprocessor. These are **Q_OBJECT**, **Q_PROPERTY()** and **Q_ENUMS()**, which are filtered out by the Classdesc processor.

This additional processing is enabled with the **-qt** flag on the Classdesc processor.

Organising the use of the python descriptor in SciDAVis

Most of the python support code is handled in the one file `PythonScripting.cpp`. So exposing class definitions involved adding the SciDAVis header file and the Classdesc descriptor definition file for each exposed class to the beginning of that file. We started with exposing the ApplicationWindow class, which is the main omnibus class implementing the SciDAVis application. The code to expose this class becomes:

```
BOOST_PYTHON_MODULE(scidavis)
{
  classdesc::python_t p;
  p.defineClass<ApplicationWindow>();
}
```

As mentioned in 'Reference returning methods' on page 14, this will automatically expose classes referenced by each exposed method, provided the appropriate header files have been included. The compiler will let you know if the header file is not present.

As the full API support was developed, additional classes needed to be added, mainly for things like the various fitting algorithm, and filter algorithms such as integration and interpolation. These options can be instantiated from Python, and then passed to methods taking a fit or filter base class reference. In all, 21 classes needed to be added to the **BOOST_PYTHON_MODULE** block.

I decided not to process the Qt library headers, as these tended to use a lot of conditional macros that Classdesc doesn't have the context to deal with. The alternative strategy of preprocessing the Qt headers to remove macros was rejected, as this typically leads to an uncontrollable explosion of classes that Classdesc must process. Instead, for each Qt class exposed on the SciDAVis python interface, a wrapper class was created, with delegated methods. Whilst a bit of work, by starting from a copy of the class taken from the relevant Qt header file, it is a fairly mechanical process creating the delegated methods inlined in the class.

Static objects in the Qt namespace, such as Qt's global colours, could be reimplemented in local code. I grouped these into a single class (called **QtNamespace**, as the identifier **Qt** clashes with the global Qt namespace). A single line was added to the **BOOST_PYTHON_MODULE** block creating an alias of this object to **Qt** in python's global namespace:

```
modDict("__main__")["Qt"]=modDict("scidavis")
["QtNamespace"];
```

This pretty much implements the needed functionality from the PyQt library, eliminating the latter from SciDAVis's software dependency list.

The final pieces were supporting the **typeName** functionality for Qt types. For any type derived from QObject, this was easily implemented as a call to the moc generated **staticMetaObject::className()** method, however there were numerous Qt classes not derived from QObject, such as QString. These were implemented individually for each one, although the common cases were easily handled with a macro to reduce the amount of boilerplate code.

Code changes to SciDAVis

The biggest code changes involved methods that return pointers to objects. For the reasons outlined in 'Reference returning methods' on page 14, pointer returning methods are never exposed by Classdesc, so instead they must be converted to methods return a reference. However, these methods typically return null when an error condition is encountered. So these

methods were refactored to throw an exception (a handy `NoSuchObject` exception type was created for this purpose). The Boost.Python library converts all C++ exceptions propagated through the C++/Python interface into Python exceptions, so this was clearly the right thing to do. One could take the lazy way out, and simply provide a wrapper method that converts a pointer returning method into one returning a reference, or throwing on null pointer return, but I took the opportunity to refactor caller code to use the reference interface too, in line with conventional C++ practices.

The second set of changes revolved around making the Python API consistent with C++ API, as Classdesc will faithfully expose the C++ interface to the equivalent Python one. In the original SciDAVis code, the API is specified in 2 places. It is documented in the manual, and specified in the scidavis.sip specification file. As might be expected, these two definitions were sometimes contradictory, and also were not consistent with C++. When resolving these inconsistencies, I chose to follow what was documented in the manual, even though it potentially introduces breaking changes for scripts that rely on how the API was actually defined. The most significant change were in methods that took arguments that satisfy Python's sequence semantics, such as lists or tuples. So such a method call should look like:

```
foo.bar((1,2,3))
```

or

```
foo.bar([1,2,3])
```

but instead the SIP implementation did it in a variadic way:

```
foo.bar(1,2,3)
```

Whilst it is possible to supply a variadic definition from within Boost.Python, it needs to be coded explicitly, as Classdesc ignores variadic methods.

Ideally, in C++, one should be able to initialise a C++ sequence with one of these python sequence objects, but currently that is not possible. So for now, supporting this call from Python involves adding an addition overloaded method taking a `pyobject` reference. The `pyobject` type is defined `boost::python::object`, which implements `operator[]` and `len()`, which suffices for constructing a C++ sequence object. In the future, I hope to be able to automatically generate this code. In the case where python support is disabled, `pyobject` is declared as a class, but otherwise not defined. In the C++ implementation file, the body of the method is simply `#ifdef`'d out.

Most of the code changes were then to make the C++ API consistent with the published Python API.

Originally, in order to get a runnable executable as quickly as possible, all unrecognised types were given a null python descriptor. However, that proved to be a mistake – it was better just to define dependent library classes (Qt, Qwt) as having null descriptors, and ensure Classdesc was run on all necessary SciDAVis defined classes.

Results

The core SciDAVis code (libscidavis directory) consisted of 100,466 lines of code, and after the conversion to Classdesc weighed in at 100,838 lines of code. The saving from eliminating the 2K loc scidavis.sip file was mostly eaten up by having to implement shim classes to expose Qt and Qwt classes. There is room for improvement by eliminating dead code that has been made redundant in the classdesc-boost.python way of doing things.

The result on compile times though is rather disappointing. On a quad core Intel i5-8265U CPU, the original SciDAVis code takes 1'25" to compile and link the application. The refactored classdesc-enabled code takes 6'3" to do the same thing, much of which is spent compiling the one module PythonScripting.cpp. This could be improved by splitting the classdesc descriptor calls for the different classes into different compilation units. In further work, the ApplicationWindow class python support was compiled into a separate object file from other classes, and the build time was reduced to 5'7". Further build time optimisations will be needed too.

The resulting binary is larger too, at 15.6 MiB versus the original 6.0 MiB, probably because classdesc exposes a fatter interface than the manually crafted SIP interface. Indeed, the approach is to expose a maximally fat interface – all SciDAVis public classes are exposed to Python, as well as select Qt and Qwt classes historically exposed to Python, via SciDAVis implemented wrapper classes. The compiler and python regression test together defined what these needed to be. Also, the equivalent PyQt functionality is inlined into the executable, rather than in a dynamically loaded library % get the size of PyQt dynamically loaded % lib...

Executable times, running the tests scripts appears to be much of a muchness between classdesc and SIP, the runtime differences between the two versions within experimental noise. ■

References

[Abrahams03] David Abrahams and Ralf W Grosse-Kunstleve (2003) 'Building hybrid systems with Boost.Python' in *C/C++ Users Journal*, 21(7).

[Benkert14] T Benkert, K Franke, D Pozitron, and R Standish (2014) Scidavis 1. *D005* (Free Software Foundation, Inc: 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA), 2014.

[Blanchette06] Jasmin Blanchette and Mark Summerfield (2006) *C++ GUI programming with Qt 4* Prentice Hall Professional.

[Leow03] Richard Leow and Russell K. Standish (2003) 'Running C++ models under the Swarm environment' in *Proceedings SwarmFest 2003*. arXiv:cs.MA/0401025.

[Liang99] Sheng Liang (1999) *The Java Native Interface: Programmer's Guide and Specification* Addison-Wesley Professional.

[Madina01] Duraid Madina and Russell K. Standish (2001) 'A system for reflection in C++' in *Proceedings of AUUG2001: Always on and Everywhere*, page 207. Australian Unix Users Group.

[Rathmann] Josef Wilgen Uwe Rathmann 'Qwt - Qt widgets for technical applications' on https://qwt.sourceforge.io/ (retrieved 13 June 2019).

[Riverbank] Riverbank Computing 'What is SIP?' https://www.riverbankcomputing.com/software/sip/intro (retrieved 13 June 2019).

[Rossum02] Guido Van Rossum and Fred L Drake Jr. (2002) *Python/C API reference manual* Python Software Foundation.

[Standish03] Russell K. Standish and Richard Leow (2003) 'EcoLab: Agent based modeling for C++ programmers' in *Proceedings SwarmFest 2003* arXiv:cs.MA/0401026.

[Standish16] Russell K. Standish (2016) 'Classdesc: A reflection system for C++11' in *Overload* 131 pages 18–23, published February 2016 http://accu.org/index.php/journals/c358/

Trip Report: Italian C++ 2019

Milan held Italy's largest C++ conference. Hans Vredeveld reports back.

On June 15th I was in Milan, Italy, to attend the Italian C++ conference [ItalianC++Conf19]. It was not the first time the Italian C++ community had held a conference, but it was the first time that I attended.

The day started with registration and the welcome message from Marco Arena, followed by the keynote. This time, the keynote speaker was Andrei Alexandrescu. With the usual Alexandrescu humour, he talked about sorting algorithms. He started by comparing a couple of standard algorithms, looking for the one with best performance. After he found a good one, he started tweaking it, performing all kinds of theoretical optimizations that, when measured, only resulted in worse performance. Finally he made some changes that should result in worse performance (at least theoretically), but that actually improved performance.

After the coffee break, the two tracks with sessions started. The first talk I attended was Rainer Grimm's talk about concepts in C++20. He started with the motivation for concepts and how they were inspired by Haskell. Next he explained what concepts are and how they can be used. Having run into the problem that I needed something more generic than functions, but that templates left too much open and were too generic, I cannot wait and look forward to using concepts in my code.

The next talk I went to was Vector++17 by Davide Bianchi. The talk was not about `std::vector`, but about the vector that we all know from mathematics. Although it was not directly relevant for my day-to-day programming, I was intrigued and wanted to know more about it. Davide's goal was a class `Vector` that behaves like the mathematical vector and that can be used with the operators on vectors that we all know from mathematics. He made good use of fold expressions and constexpr `if` to get clean code that supports SIMD like operations and also resulted in better error messages. In the end, I found the 30 minutes for this talk too short and would have preferred a 60 minute talk on the subject.

Next it was time for the lunch break and some more networking. During the lunch break there was a bonus talk that I missed, because I was caught up in a conversation.

After the lunch break, I went to Dmitry Kozhevnikov's talk about the future of C++ with modules. If you have already delved into the subject of modules, most of what Dmitry talked about is already known to you. I knew that they were coming, but hadn't spent much time on them yet. For me, this was a nice overview of what they will give us, solving a lot of the problems we have with include files and the current compilation model that we inherited from C and its preprocessor.

Then it was time for 'Custom Clang Tooling' with James Turner. He introduced us to Clazy [GitHub], a static analyser based on Clang [Clang-1]. In many ways it is similar to clang-tidy [Clang-2], but Clazy specializes in Qt and enforcing Qt best practices. It can automatically fix issues found, it has support for your own coding conventions, it automates code refactoring and it integrates into CI. It was a good presentation that gave me a decent overview of the tool. As always with this kind of tool, I have to play with it to fully understand its usefulness. I don't know when that will happen.

Finally we had another break and the last talk of the day. Arne Mertz missed his connecting flight the day before and could not make it. That made the choice simple. Do I or don't I go to Michele Caini's talk 'ECS back and forth'? I did. ECS stands for Entity Component System [Wikipedia] and it is an architectural pattern. It favours composition over inheritance, sacrificing encapsulation. Unfortunately, my attention started wavering, in part because we already had a full day of talks and in part because I had no idea how I could use it in my daily programming practices. But, if I want to play with it, there is an open source library that implements the pattern: EnTT. [Caini]

Marco Arena and Alessandro Vergani concluded the day with the closing message. During the day attendees were encouraged to tweet about the conference, and in the closing message they handed out the prizes for the best tweets. They also invited the other member of the organizing committee and the speakers to come forward. The room thanked both groups with a heart-warming applause.

All in all, it was an excellent day and I look forward to go to Milan again next year. ■

References

[Caini] Michele Caini (skypjack) EnTT library on GitHub: https://github.com/skypjack/entt

[Clang-1] Clang: http://clang.llvm.org/

[Clang-2] Clang-tidy: https://clang.llvm.org/extra/clang-tidy/

[GitHub] Clazy: https://github.com/KDE/clazy (A mirror of the KDE project.)

[ItalianC++Conf19] *Italian C++ Conference 2019*: https://www.italiancpp.org/event/itcppcon19/

[Wikipedia] Entity Component System: https://en.wikipedia.org/wiki/Entity_component_system

Hans Vredeveld started working in the software industry 20+ years ago as a system administrator. Via application administration he soon moved into software development, where he was bitten by the C++ virus. Not wanting to be cured, he is always searching for the next cool C++ thing. He can be contacted at accu@closingbrace.nl

Afterwood

Many people are risk-averse. Chris Oldwood considers this position – in verse.

Don't change the specification!
We made a plan, we should follow it to the letter,
so what if customer feedback can make the product better.
The schedule's secured
and the team's been chosen,
from here on in the deadline's frozen.

Don't refactor the code!
The code isn't broken, there's nothing to fix,
so what if the author used incomprehensible tricks.
If it was hard to write
it will be hard to read,
a 10x developer doesn't come cheap.

Don't upgrade the libraries!
We already use all the features we require
switching versions is tantamount to playing with fire.
We can save on testing
when things stay the same,
minimizing QA is the name of the game.

Don't switch compilers!
We know how the tool works, we've got the build stable,
so what if compliance makes the code more maintainable
We're here to add features
not become a language fanatic,
idealism needs tempering; look to favour pragmatic.

Don't alter the configuration!
The support team have a play-book, it covers all cases,
tinkering with settings from now only leads to red faces.
The set-up has hardened
it took weeks to achieve,
now is the time to demand a system-wide freeze.

Don't change the platform!
The architecture's formalized and infrastructure's mature,
we have an enterprise contract which is there to ensure
that there's someone on call
when things go awry
while we're paying good money it's not going to die.

But change is inevitable!
Heraclitus said: everything changes and nothing stands still,
for some it's an assertion to swallow, a bitter-tasting pill.
You can't mitigate risk
by instilling fear,
instead embrace it, and make change something the team will revere. ■

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