overload 146

AUGUST 2018 £3

Cache-Line Aware Data Structures

Structuring your program to consider memory can improve performance. We demonstrate this using a producer–consumer queue.

Compile-Time Data Structures in C++17

Using compile-time data structures to speed up runtime

How to Write a Programming Language

We continue writing a simple programming language, this time looking at the parser

(Re)Actor Allocation At 15 CPU Cycles

A lightweight allocator for (Re)Actors

Afterwood

Much ado about nothing

A magazine of ACCU

ISSN: 1354-3172

CARE about code?

passionate about programming?

Join ACCU www.accu.org

OVERLOAD 146

August 2018

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Balog Pal
pasa@lib.h

Ben Curry
b.d.curry@gmail.com

Paul Johnson
paulf.johnson@gmail.com

Klitos Kyriacou
klitos.kyriacou@gmail.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Philipp Schwaha
<philipp@schwaha.net>

Anthony Williams
anthony@justsoftwaresolutions.co.uk

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 147 should be submitted by 1st September 2018 and those for Overload 148 by 1st November 2018.

Overload is a publication of the ACCU

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

4 Cache-Line Aware Data Structures
Wesley Maness and Richard Reich demonstrate with a producer–consumer queue.

8 miso: Micro Signal/Slot Implementation
Deák Ferenc presents a new implementation of the Observer pattern.

14 (Re)Actor Allocation at 15 CPU Cycles
Sergey Ignatchenko, Dmytro Ivanchykhin and Marcos Bracco pare malloc/free to a minimum.

20 How to Write a Programming Language: Part 2, The Parser
Andy Balaam continues his series on writing a programming language.

23 Compile-time Data Structures in C++17: Part 1, Set of Types
Bronek Kozicki details an implementation of a compile time data structure.

28 Afterwood
Chris Oldwood considers what it means to have nothing.

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.

Copyrights and Trade Marks

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

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

Should I Lead by Example?

Stuck on a problem? Frances Buontempo considers where to turn to for inspiration.

I started to clear my desk and found a cue card with the words “Lead by example” written on it, which distracted me from writing an editorial. This card is from Seb Rose’s closing keynote ‘Learning to Walk again’ at this year’s ACCU conference [Rose18]. The idea was to write down something you had learnt from the conference on a card and give the card, along with contact details, to someone so they could later remind you what you wrote. I failed to comply with the instructions precisely, since I still have my own card, but it did jog my memory, though not enough to recall what had taught me this. It could have been Arne Mertz’s ‘Code review’ session [Mertz18]. You cannot expect to get away with telling others how they can improve their code if you don’t follow your own suggestions. As a mentor, I encouraged new programmers to add unit tests to their code and put it all in version control, but frequently noticed I had strategic scripts with no tests at all and some not even in version control. For shame! Lead by example. Of course, it’s not just unit tests. I complain when others don’t keep a diary, but don’t always write appointments in my own diary. Far too much ‘Do as I say, not as I do.’

OK, what does lead by example mean? In one sense, leading by example contrasts with trying to bully people into doing things your way. How do you persuade people to adopt your approach? I was recently asked this at an interview. If faced with a task that might take a day, and two possible implementations, it’s surely worth knocking together both alternatives rather than spending three days arguing over which is best? You do not always need to make everyone go along with one idea. Sometimes it matters, sometimes it doesn’t. Tempers can get frayed if people don’t see things your way, and I value working software over proving my idea is the best. A recent blog post [DestroyAllSoftware] broke down a response from Linus Tovarlds on union aliasing. Tovarlds’ language was inflammatory and unkind. The blog post showed an alternative way of making the same points without being so hateful. You can disagree with people without resorting to bullying by telling them they are brain-dead or worse. I can recall many time when I’ve categorically told someone they were stupid and didn’t know what they were doing. I believe I have stopped doing this now. Pull me up if you notice me being a bully. Taking the lead by bullying others into submission is not a good idea. What are the alternatives?

To encourage the adoption of a new approach to a problem, instead of arguing or laying down the law, you may be able to knock together a prototype showing the alternative works. Sometimes the better tech wins, so giving people alternatives allowing them to try out your new ideas can be more persuasive than banning older tool-chains or similar. If you want to change an API, try adding new functions and gradually deprecating the older ones as people stop using them. The strangle vine pattern or strangle applicator [Fowler04] describes ways to ensure a new approach strangles or kills off the old way, drawing an analogy with strangle vines, which grow over other plants. It contrasts with a complete re-write allowing new approaches to live side by side with old approaches, at least for a while. This probably doesn’t count as leadership per se, but does give alternative paths. More suggest by alternatives than lead by example, though it incorporates the nub of the idea: have a demonstration or example to make your point.

Any prototype is certainly an example, though whether this counts as leading is another matter. In the sense of pioneering, or going out in front, it surely does? Sometimes attempting to trail-blaze leaves you more a lone lunatic in no-mans’ land than a leader. You need a level of self-confidence to be able to demonstrate an idea or working example without wanting to hide under a blanket or behind a sofa. I hope we manage to encourage *Overload* writers, even if we don’t always agree with everything that gets written. That’s ok. Thank you for sharing your ideas with us. An article is often an example, sometimes a novel idea even, which can lead readers to try new things, learn or even write in to disagree. That’s ok too. However, an article as a way of leading by example is probably not what my cue card meant.

What did I mean? I am not sure. It sounds like sensible advice, but I am unclear what it really means, as is often the case with slightly trite phrases. Furthermore, it begs the question: should I lead by example? Perhaps this is straying near Betteridge’s law of headlines: ‘Any headline that ends in a question mark can be answered by the word no’. [Wikipedia-1]. When used in a headline, a question is often sensationalist, in order to drum up an audience, something I suspect my title is unlikely to achieve. One website [BetteridgeLaw] has examples filling over two thousand pages. I can’t vouch for the veracity of any of those, though many are hilarious. ‘Does Bill Gates still know what computer users want?’ and so on. I’ll leave you to explore.

You may not aspire to be a leader, but can find yourself out in front from time to time. I sometimes walk quicker than others, finding myself ahead when walking with friends or family to an unfamiliar location, having just said, ‘I’ll follow you.’ It is difficult to follow if you are in front. Perhaps you find yourself reluctantly in front, being the first person to try to make a new technology work, or resurrect some old code no one knows how to build. Bad luck. In that situation, you are unlikely to have examples to follow. You can make sure you put the code in version control, add some kind of tests or at least ways of spotting regressions. You can add a make file, or other build script, once you have figured out how to build the code. A short readme file is a good idea too. Even if it says little more than ‘type

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

make then **run_tests**'. In a sense, this is leading by example because you have improved the situation, just quietly in the background, without needing long meetings to decide what approaches to take. You might need these too, but at least the fundamental parts are in place and you have recorded what you spent time discovering in a simple and clear format. Perhaps that's all I meant. Instead of moaning about the state of the world, or the project, or codebase, step up and make the changes needed. This might need to be in a non-invasive way, so people can still email themselves files, write word documents and have meetings if they want to. Meanwhile the code builds and runs. And crashes. But that's another story.

This nuance is leading in the sense of forging ahead and getting stuff done. It's not leadership in the sense of an authoritarian head of state or someone guiding or conducting a process, team or project. A leader can also be a front page news splash or similar. Something at the front, in your face, trying to stir up discussion. An editorial of sorts. The etymology of the ending -ship might trace to the Dutch for cut or hack [Etymology], I presume along the lines of essence of something rather than thrown together or a newspaper hack. Such a pen for hire is not to be confused with relatively recent phone hacking scandals by News International [Wikipedia-2]. In a sense, the hack makes a path through something or in a direction. Any example gives a hint of how to do something or the direction to take. An entrepreneur or pioneer may take a lead, one dealing with the enterprisey requirements to form a business, the other possible being more like a lone ranger going off in front, perhaps a alone. Do such people lead by example? They lead. The best team leads I have ever worked with lead from behind. They were happy to take a back seat and enable the developers. The *Harvard Business Review* attributes this to Nelson Mandela [Hill04], equating a leader with a shepherd who "stays behind the flock, letting the most nimble go out ahead, whereupon the others follow, not realizing that all along they are being directed from behind."

Are there leaders, whether team leads at work or from other realms, you admire? I suspect each of us can think of at least one person who seems to have a knack of getting things done in an effective manner. I have a few people I bring to mind when I get stuck on various problems. For mathematics, I often wonder what my Dad would have done. For some coding problems, I wonder what specific coders I know would do. I won't name and embarrass anyone, but I've met many such people through the ACCU. The meme, 'What would [insert name here] do?' has run for a long time. As with many memes or inspirational sayings, the question is a cliché. Commonplace sayings become clichés because they capture a heart of a common idea or experience, which rings true for many people. They can give an accurate encapsulation of an idea, or an example to put out in front. I have caught myself a few times thinking a project is going badly wrong and rather than asking 'What would XXX do?' I start asking myself what do I want to do. What would I do, if I were leading this project? What would I do if this were a personal project? That's why I have managed to sneak in a make file and a way to run some regression tests on my current project. I'm not suggesting you use me as a fine example of how to solve any problems. I am asking if you have some self-belief. If you're suffering from a confidence nose-dive, be kind to yourself. Remind yourself what you are good at, or at least enjoy. Spending time listening to people you admire, reading what they've written; articles, code or blogs. Or stories. At least get to a point where you can reflect on the bigger picture and get what you think clear. That might be no more than deciding you are stuck and haven't got a clue what to do.

If you're not sure how to tackle a problem, do consider asking yourself what XXX would do. Not necessarily Vin Deisel.[1] Find an example to lead you through your problem. I think I have now devolved into giving myself advice or suggestions using a few suspiciously platitudinous phrases. Thanks for listening. I may have intended to lead by example, but now wonder if perhaps I should re-word my cue card to say "Hack by cliché." Whichever you think is most appropriate, take a moment to ask yourself, what would Linus do? What would Bjarne do? What should I do? Be the person you want to be. Don't be mean. Lead by example, or hack by cliché.

References

[BetteridgeLaw] http://betteridgeslaw.com/

[DestroyAllSoftware] 'A case study in not being a jerk in open source' (2018) https://www.destroyallsoftware.com/blog/2018/a-case-study-in-not-being-a-jerk-in-open-source

[Etymology] -ship in the *Online Etymology Dictionary* at https://www.etymonline.com/word/-ship?ref=etymonline_crossreference

[Fowler04] Martin Fowler (2004) 'Strangler Application', at: https://www.martinfowler.com/bliki/StranglerApplication.html

[Hill04] Linda Hill (2010) 'Leading from Behind' in the Harvard Business Review, https://hbr.org/2010/05/leading-from-behind

[Mertz18] Arne Mertz (2018) 'Code Reviews – Why, what and how' https://isocpp.org/blog/2018/01/code-reviews-why-what-and-how-arne-mertz (not recorded at the ACCU conference)

[Rose18] Seb Rose (2018) 'Software development – learning to walk again' from the ACCU18 conference, available at https://www.youtube.com/watch?v=iFwm-_04rLg

[Wikipedia-1] 'Betteridge's law of headlines', https://en.wikipedia.org/wiki/Betteridge%27s_law_of_headlines

[Wikipedia-2] 'News International phone hacking scandal', at https://en.wikipedia.org/wiki/News_International_phone_hacking_scandal

1. The film *xXx* staring Vin Diesel was released in 2002 (https://www.imdb.com/title/tt0295701/)

Cache-Line Aware Data Structures

Structuring your program to consider memory can improve performance. Wesley Maness and Richard Reich demonstrate this with a producer–consumer queue.

In this paper, we explore cache-line friendly data structures, in particular queues built with atomics that will be used in multi-threaded environments. We will illustrate our topic with a real-world use case that is not cache-line aware, measure, incorporate our suggestions, measure again, and finally review our results. Before we get into the nuts and bolts of our data structures, we need to define a few terms followed with some examples. For those readers who are not familiar with the topics of NUMA and CPU cache, we highly recommend reviewing them (at [Wikipedia-1] and [Wikipedia-2]).

Jitter

Jitter can be defined as the variance of time around operations that can arguably have constant time expectations.

A few examples to better illustrate are:

- The wall clock on a system in which each 'tick' of the highest-precision unit has some variance. Back in 2012, we measured the wall clock on the best server we had at the time. It was accurate down to 1 microsecond. However, we had jitter of ±1.5 microseconds. Consider that the clock's accuracy increases over longer time periods, but each tick jittered.
- When we are linearly accessing memory in a system, each access may take a constant time until you hit a page fault. That page fault can introduce delay. In this case, we have a constant time operation that ends up having predictable jitter.
- Sparsely accessing an array can have many cache misses as well as cache hits. Depending on the location and layout of the memory, what should be constant will be different depending on:
  - Is the memory in the same NUMA node? [Tsang17]
  - Is the memory located in the cache?
  - Which cache is the memory located in? (L3, L2, L1)
  - Is the memory in the current cache line?
  - Is the memory contended with another CPU socket/core?

Many of the above are just examples which are not investigated in the scope of the paper and many can also be mitigated by using prefetching.

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@aya.yale.edu.

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

Cache line

The cache line is the smallest unit of RAM the CPU can load to perform operations. On the Intel CPU, this is 64 bytes, or 8 pointers in a 64-bit operating system. If at least one of the cores is writing, then cache coherency causes the cache line to be synchronized between the cores as each write forces a synchronization between the cores. Many cores reading from the same cache line causes no performance issues if there is no writing to that same cache line.

Cache awareness

Cache awareness really comes down to structuring your memory layout (memory model) of your program and its data structures. Careful consideration of what memory goes where can significantly improve the performance of the resulting machine code that must be verified by measuring.

For example, if we look at the Intel core i7 [Levinthal09] we can see its specifications are: L1 is 4 CPU cycles, L2 is 11 CPU cycles and L3 is 30-40 CPU cycles. Main memory can range from 200-350 CPU cycles on a 3GHz system. Crossing NUMA nodes incurs even more penalties. Code that sparsely accesses memory incurring many cache misses can spend 95% of the time doing nothing!

Motivation

In previous sections, we hinted as to why we would want to be cache aware, but here we will explain in a bit more detail, and then consider the benefits of potential future hardware progression and its impact.

From a multi-threading perspective, we need to be sure that data that are independently accessed by different threads are not shared over a cache line. Doing so will cause all reading threads to stall while the dirty cache line is synchronized across all cores. This is compounded if one or more of the threads exists on a different NUMA node.

The most direct benefit of fully independent data between threads not sharing cache lines is linear scalability. Some of the most obvious costs are:

- Increased code complexity due to increased complexity of the memory model.
- If each thread is accessing small amounts of data, some of the space in each cache line may not be utilized.
- Because of the above, we may need more memory to force alignment. If memory is limited, this could require special consideration.
- Perhaps there are multiple copies of the same data in various locations resulting in less efficient usage of memory.

Each socket is being packed with more and more cores, and FPGA integration into Intel CPUs is on the horizon (as this is written) [Intel] [Patrizio17]. This will effectively model memory to separate memory access at the cache-line level (to avoid false sharing [Bolosky93]). Furthermore, it will have a greater impact as cores and specialized hardware compete for resources.

we need to be sure that data that are independently accessed by different threads are not shared over a cache line

```
struct Benchmark
{
  uint64_t cycles{0};
  uint32_t serial{0};
};

template <typename Bench, int X>
struct Alignment
{
  alignas(X) Bench cb;
  Bench& get() { return cb; }
};
```

Listing 1

Next, we will demonstrate the necessity of cache-line awareness with a modern-day use case that would benefit nicely from such consideration.

Common use case

There exist situations in technology where we find ourselves having to set up and use a shared environment with many agents performing various coordinating tasks. In this use case, we will be given a machine in which we have a single multi-producer multi-consumer queue (MPMC) instance shared amongst a set of producers and consumers. The producers, we can assume, are clients which are running some computations and generating work, or some partial work. Hence each client would be running a different set of computations and each of those publishing their resulting work items to the MPMC for later consumption. Each consumer would be responsible for taking the work off the queue, perhaps performing some post-checks, or validation of the work on an item in the queue then dispatching that work out to some client who will need to process or make some determination. Each consumer and producer would run in their own thread and potentially could be pinned to any CPU. Depending on what is required, it's quite reasonable that the number of producers and consumers would need to scale up or down as clients and or workloads are running (some can go on and off line driven by various external events). The most critical measurement we would want to consider here would be the time it takes to produce or push a work item on the queue and for the work item to be removed from the queue. We consider other types of measurements as well later.

Running the benchmark

To enforce that we are not aligning the data, we use the `alignas` specifier for our data. The `struct` that we will be using is shown in Listing 1. We will store the number of cycles in the `struct` along with some other data fields. How we store these is shown in Listing 2 (not shown is the `GetTravelStore` method which is the same as `GetStore`). We used the `boost::lockfree::queue` for our testing the aligned section and we modify their implementation to disable alignment (not shown) for our baseline numbers.

```
std::unique_ptr<uint64_t>
  GetStore ( uint64_t iter )
{
  thread_local uint64_t* store{nullptr};

  if (store == nullptr)
    store = new uint64_t[iter];
  return std::unique_ptr<uint64_t>(store);
}
```

Listing 2

In establishing our baseline number, we will consider various scenarios or ratios of numbers of producers to consumers. Currently we only display results for the ratio 2:2. Each consumer and each producer will spin in a busy loop to minimize overall cycles required to publish and or consume data. The code for producer and consumer is shown in Listing 3. The total

```
template <typename T, typename Q>
void producer(Q* q, uint32_t iterations)
{
  auto store = Thread::GetStore(iterations);
  while (Thread::g_pstart.load() == false) {}
  T d;
  d.get().serial = 0;
  int result = 0;
  for ( uint32_t j = 0; j < 2; ++j) // warm up
  for ( uint32_t i = 0; i < iterations; ++i)
  {
    ++d.get().serial;
    do { d.get().cycles = getcc_b();  }
      while (!q->push(d));
    store.get()[i] =
      getcc_e() - d.get().cycles;
    // busy work to throttle production
    // to eliminiate "stuffed" queue
    //* No noticable effect
    for (uint32_t k = 0; k<1000; ++k)
    {
      result += k+i;
    }
    // */
  }
  ++result;
  std::stringstream push;
  genStats(iterations, store, "1 Push", push);
  std::lock_guard<std::mutex>
    lock(Thread::g_cout_lock);
  std::cout << result << std::endl;
  Thread::g_output.emplace(push.str());
}
```

Listing 3

```
template <typename T, typename Q>
void consumer(Q* q, uint32_t iterations)
{
  auto store = Thread::GetStore(iterations);
  auto travel_store =
    Thread::GetTravelStore(iterations);

  while (Thread::g_cstart.load() == false) {}

  T d;
  uint64_t start;
  uint64_t end;
  for ( uint32_t j = 0; j < 2; ++j) // warm up
  for ( uint32_t i = 0; i < iterations; ++i)
  {
    do { start = getcc_b(); }
      while (!q->pop(d));
    end = getcc_e();
    travel_store.get()[i] =
        end - d.get().cycles;
    store.get()[i] = end - start;
  }

  std::stringstream trvl, pop;
  genStats(iterations,
        travel_store,
        "3 Travel",
        trvl);
  genStats(iterations,
        store,
        "2 Pop",
        pop);
  std::lock_guard<std::mutex>
    lock(Thread::g_cout_lock);
  Thread::g_output.emplace(pop.str());
  Thread::g_output.emplace(trvl.str());
}
```

Listing 3 (cont'd)

number of items produced in each scenario will be scaled to the number of consumers. In these examples, we will use the standard atomics (as part of the MPMC queue) in C++ and spin on their values to identify when data is available. The producers and consumers are constructed in the run method shown in Listing 4.

```
template<typename T,template<class...>typename Q>
void run ( int producers, int consumers )
{
  std::cout << "Alignment of T "
        << alignof(T)
        << std::endl;
  std::vector<std::unique_ptr<std::thread>>
    threads;
  threads.reserve(producers+consumers);
  Q<T> q(128);
  // need to make this a command line option
  // and do proper balancing between
  // consumers and producers
  uint32_t iterations = 10000000;
  for (int i = 0; i < producers; ++i)
  {
    threads.push_back(
        std::make_unique<std::thread>
          (producer<T,Q<T>>
          , &q
          , iterations));
    // adjust for physical cpu/core layout
    setAffinity(*threads.rbegin(), i*2);
  }
  for (int i = 0; i < consumers; ++i)
  {
    threads.push_back(
      std::make_unique<std::thread>
        (consumer<T,Q<T>>
        , &q
        , iterations));
    // adjust for physical cpu/core layout
    setAffinity(*threads.rbegin(), 4+(i*2));
  }
  Thread::g_cstart.store(true);
  usleep(500000);
  Thread::g_pstart.store(true);
  for (auto& i : threads)
  {
    i->join();
  }
  for (auto& i : Thread::g_output)
  {
    std::cout << i << std::endl;
  }
}
```

Listing 4

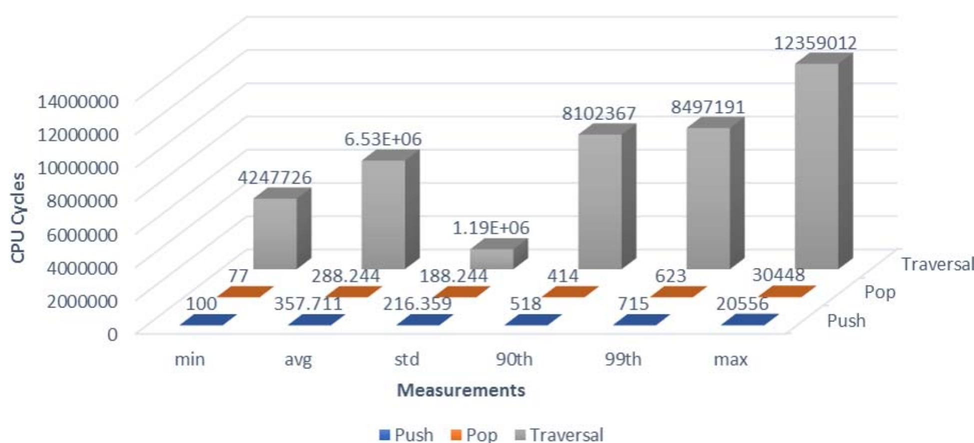


Figure 1

Finally, for our benchmark without cache-line awareness, we put all of this together:

```
run<Alignment<
  Benchmark
  , alignof(Benchmark)>
  , boost::lockfree::bad_queue>
  (producers, consumers);
```

We will pin each thread created to its very own CPU to reduce overall variance on measurements. This is done in the method **setAffinity** mentioned in the code examples but not shown. You can find more information about thread affinity on the Linux man pages [Kerrisk]. We will measure push, pop, and total queue traversal time (pop end time – push start time) along with various other metrics. These results are shown in Figure 1. All measurements are in cycles (using RDTSCP and CPUID instructions) using the recommended guidelines from Intel

[Paoloni10]. These measurements are computed inside the functions `getcc_b` and `getcc_e` (not shown).

Applying cache-line awareness to our example

Now that we have identified how we are going to take advantage of cache-line awareness in our queue, we will re-run our previous tests with the changes we just applied. We will now enable cache-line awareness by using the alignas specifier for our data. The code used here is:

```
run<Alignment<
    Benchmark, 64>
  , boost::lockfree::queue>
  (producers, consumers);
```

We will run through the same scenarios as we saw in Figure 1. These results are shown in Figure 2.

Note: If the measurements are not clear in Figure 2, the max measurements for Push, Pop, and Traversal are observed as 22977, 20221, and 18102 respectively.

Analysis

Comparing where we started and where we ended up, there are several items of considerable mention. The first is that the distribution of the percentiles for Figure 1 clearly show fat tail properties mostly noticeably in the traversal measurements. Another quite fascinating point is that the 90th and 99th percentiles of all measurements dropped considerably in all operations. Push went from 518 cycles to 431 and 715 cycles to 575 for 90th and 99th percentiles. Pop, not necessarily as impressive as Push, went from 414 cycles to 370 and 623 cycles to 500 cycles respectfully. Traversal was even more impressive going from 810236 cycles to 12307 and 8497191 cycles to 15571 for 90th and 99th percentiles respectively.

Conclusion and future direction

There are many items we didn't address in this paper. For example, we did not discuss the data throughput through the queue as a function of cache-line awareness changes. We could have also considered measuring different scenarios and the ratios of producers to consumers. It is important to note that the problem addressed in this paper is just one instance of a group of problems collectively known as flow control problems. These problems exist in many domains, and are quite common in some aspects of financial technology, but have relevancy in many others. We mention the points above to illustrate different ways in which this problem can be expanded upon and perhaps further analysis may be continued.

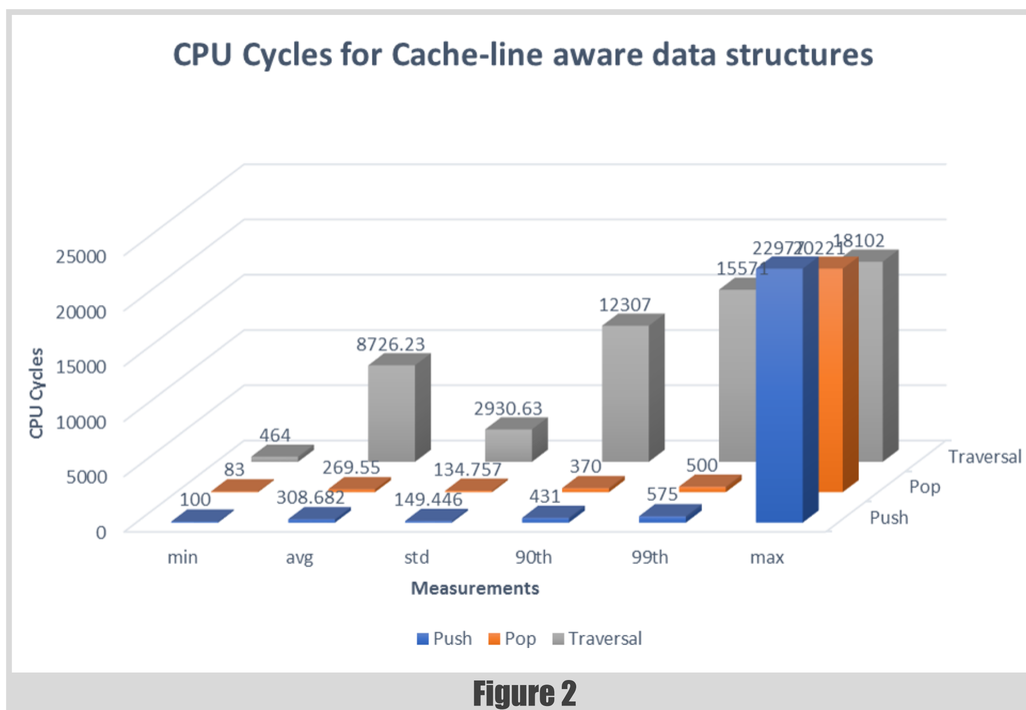


Figure 2

Finally, as we were writing this paper, running measurements of various scenarios, the results were in many ways quite interesting and often initially appeared to be counter-intuitive, but after carefully examining the assembly and measuring the performance more closely the results made more sense. We can't stress enough how important it is to measure your applications and the tools used to measure those applications. ■

Notes

Boost 1.63.0 and GCC 5.4.0 were used. For a complete package of the code used in this article and that which is referenced and not shown, please contact the authors directly. All measurements were the average of 1000 runs of 1e6 iterations for 2 consumers and 2 producer threads. Intel i7-3610QM CPU 2.30GHz 4 cores per socket for 8 cores total was used to produce all measurements discussed in the paper. Operating system used was Linux ll 4.12.5-gentoo.

References

[Bolosky93] William Bolosky and Michael Scott (1993) 'False Sharing and its Effect on Shared Memory Performance', originally published in *Proceedings of the USENIX SEDMS IV Conference Sept 22–23 1993*, available at http://static.usenix.org/publications/library/proceedings/sedms4/full_papers/bolosky.txt

[Intel] 'The Power of FPGAS' available at https://www.intel.com/content/www/us/en/fpga/solutions.html

[Kerrisk] Michael Kerrisk (maintainer), *Linux man pages online*, http://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html

[Levinthal09] David Levinthal PhD (2009) 'Performance Analyis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors' at https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[Paoloni10] Gabriele Paolone (2010) *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*, published by Intel Corporation, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[Patrizio17] Andy Patrizio (2017) 'Intel plans hybrid CPU-FPGA chips', posted 5 October 2017 at https://www.networkworld.com/article/3230929/data-center/intel-unveils-hybrid-cpu-fpga-plans.html

[Tsang17] Stanley Tsang (2017) 'Real-Time NUMA Node Performance Analysis Using Intel Performance Monitor' at http://www.acceleware.com/blog/real-time-NUMA-node-performance-analysis-using-intel-performance-counter-monitor

[Wikipedia-1] 'CPU cache' at https://en.wikipedia.org/wiki/CPU_cache

[Wikipedia-2] 'Non-uniform memory access' at https://en.wikipedia.org/wiki/Non-uniform_memory_access

miso: Micro Signal/Slot Implementation

The Observer pattern has many existing implementations. Deák Ferenc presents a new implementation using modern C++ techniques.

miso is short for **mi**cro **s**ignals and sl**o**ts and, as the name suggests, it is an implementation of the wel-known language construct largely popularized by Qt: The signals and slots mechanism [Wikipedia]. As the Wikipedia article suggests, the signal-slot construct is a short, concise and pain-free implementation of the Observer pattern, ie. it provides the possibility for objects (called observers) to be recipients of automatic notifications from objects (called subjects) upon a change of state, or any other event worthy of notification.

Reasoning

So, you may ask, why another signal/slot implementation? Since we already have the granddaddy of them all, the Qt signal/slot implementation which, as presented in [Qt4SigSlot], is a very powerful mechanism invented just for this purpose and which was even further enhanced with Qt5's new syntax for signals and slots [Qt5SigSlot].

Or we have the boost signal libraries [BoostSigSlot], which are another excellent implementation of the same mechanism for the users of the boost library.

And we also have other less well-known signal/slot implementations, such as Sarah Thompsons' signal and slot library [sigslot-1] or the VDK signals and slots written in C++ 11 [VDK], GNOME's own libsigc++ [libsigc++], the nano signal slot [nanosigslot], Patrick Hogans' Signals [Hogan] or several fresher ones from github ([nod] [sigcxx] [sigslot-2] [cpp-signal]) or the more hidden ones, which I was not able to discover even with Google's powerful search algorithm.

All these excellent pieces of software were written specifically for this purpose, and they all serve the needs of software developers wanting to use the signals and slots mechanism without too much hassle.

And on the other side, the Observer pattern is a very widely adopted and successful pattern which has also been widely studied in various articles, including *Overload*'s own ones, such as Phil Bass's articles in *Overload* 52 and 53: 'Implementing the Observer Pattern' [Bass02] or Pete Goodliffe's articles in *Overload* 37, 38 and 41 ('Experiences of Implementing the Observer Design Pattern') [Goodliffe00] – both excellent articles which were not backed up by the power of C++11's syntax and standard library at the time … due to the fact they were written in the first years of this century – but also Alan Griffiths' article from 2014 ('Designing Observers in C++11') [Griffiths14], which lifted this pattern into the modern age using C++11 constructs.

So with a good reason, you may ask why…

But please bear with me … the implementation of this mechanism seemed to be such an interesting and highly challenging research project that I could not resist it. I wanted to use the elegance of the constructs introduced with C++11 to avoid as much as possible the syntactical annoyances that I found in various signal/slot projects, which were bound to old-style C++ syntax, and I also wanted to keep this implementation short and concise. Hence, this header-only micro library appeared, and in the spirit of keeping it simple, it is under 150 lines, but still tries to offer the full functionality one would expect from a usable signal/slot library.

Deák Ferenc Ferenc has wanted to be a better programmer for the last 15 years. Right now he tries to accomplish this goal by working at FARA (Trondheim, Norway) as a system programmer, and in his free time, by exploring the hidden corners of the C++ language in search for new quests. fritzone@gmail.com

This article not only provides a good overview of the usage of and operations permitted by this tiny library, but also presents a few interesting C++11 techniques I have stumbled upon while implementing the library that I considered to be of sufficient calibre to be worth mentioning here.

The library itself

miso, being a single header library, is very easy to use. You just have to include the header file into your project and you're good to go: `#include <miso.h>` and from this point on you have access to the `namespace miso`, which contains all the relevant declarations that you need to use it. Later in this article, we present all the important details of this namespace.

The library was written with portability and standard conformance in mind, and it is compilable for both Linux and Windows; it just needs a C++11 capable compiler.

Signals, slots, here and there

The notion of a slot is sort of uniform between all signal-slot libraries: It must be something that can be called. Regardless whether it's a function, a functor, a lambda or some anomalous monstrosity returned by `std::bind` and placed into a `std::function`… at the end: It must be a callable. With or without parameters. Since this is what happens when you emit a signal: a 'slot' is called.

However, there is no real consensus regarding the very nature of signals. Qt adopted the most familiar, clear and easy to understand syntax of all the signatures:

```
signals:
  void signalToBeEmitted(float floatParameter,
    int intParameter);
```

Simple, and clean, just like a the definition of a member function, with a unique signature, representing the parameters this signal can pass to the slots when it is `emit`ed. And the Qt meta object compiler takes care of it, by implementing the required supporting background operations (ie: the connection from the signal to actually calling the slot function), thus removing the burden from the programmer who can concentrate on implementing the actual functionality of the program.

The other big player in platform independent C++ library solutions, boost, on the other end has chosen a somewhat more complex approach to defining the same signal:

```
boost::signals2::signal<void (float, int)> sig;
```

This way of defining a signal feels very similar to the declaration of a function packed in a templated signal declaration and, because what it means is widely understood, it was adopted not only by [VDK],

I have found including a function signature in the declaration of my signal not to work, so I went for the simplest syntax that was able to express the desired type of my signal

Qt signals and slots

For those who haven't had the chance to work with Qt's signals and slots, a small note: Qt has a handy tool, called `moc` (Meta-Object Compiler) which handles the C++ extensions of the Qt framework, such as signals and slots among other more handy helping features. The `moc` tool parses a header file containing Qt extensions and generates a C++ source file, which must be included in the compilation in order to get the desired Qt functionality working [QtMoc].

[neosigslot] and [nanosigslot] but also by nod, sigcxx, sigslot (the one from github) and cpp-signal. This syntax has the advantage of not requiring an extra step in the compilation phase (like `moc` of Qt) since it is already syntactically correct C++ which the compiler can handle without too much hassle. This declaration also has the side effect that unless like Qt's signal declaration, we have a tangible C++ variable which possibly is a class with methods and properties we can act upon.

Signals in miso

The signal definition of **miso** uses the following syntax in order to declare the same signal:

```
signal<float, int> float_int_sig;
```

Achieving the simplicity of Qt's signal syntax seemed to not to be possible without using an extra step in the compilation phase (I am thinking of the convenience offered by `moc`) and personally I have found including a function signature in the declaration of my signal not to work, so I went for the simplest syntax that was able to express the desired type of my signal (such as a `signal`, having a `float` and an `int` parameter) and with the supporting help of the variadic templates introduced in C++11 this seemed to be the ideal combination. This syntax is also used by the library presented in [Hogan], with the difference being the name of the class and the fact that, in [Hogan], you need to specify a different class name based on the number of parameters.

So, from the above we see that a signal in the **miso** framework will be an object, constructed from a templated class which handles a various number of types. A signal which carries no extra information in the form of parameters must be declared as:

```
signal<> void_signal;
```

The design decision to not have to explicitly specify the void signal as a template specialization (ie: `signal<void>`) has its advantages, both from the users' point of view, and also the library's internal design gained a bit of ruggedness from it.

A tiny application

The easiest way to introduce a new library is to present a small and simple example which showcases the basic usage of the library, so Listing 1 is the "Hello world" equivalent of **miso**.

After skipping the mandatory inclusions, let's analyze the important pieces:

```
#include "miso.h"
#include <iostream>

struct a_class
{
  miso::signal<const char*> m_s;
  void say_hello()
  {
    emit m_s("Hello from a class");
  }
};
void a_function(const char* msg)
{
  std::cout << msg << std::endl;
}
int main()
{
  a_class a;
  miso::connect(a.m_s, a_function);
  a.say_hello();
}
```

Listing 1

Firstly, we declare a class (for now with the `struct` keyword to keep the code short and uncluttered): `struct a_class`. In the **miso** framework the signals belong to classes: it is not possible to have a signal living outside of an enclosing entity. This sort of resonates on the same frequency as Qt's signal and slot mechanism; however, the boost signals are more independent and are not required to be bound to a class.

As mentioned above, the **miso** signals are to be bound to a class so now is the perfect time to declare the `signal` object itself: `miso::signal<const char*> m_s;`. All the **miso** types live in the `miso` namespace in order to avoid global namespace pollution; however, this does not stop you from using the namespace as per your needs. The signal we have declared is expected to come with a parameter, which is of type `const char*`.

The next line in the class is a plain method, which has just one role: to emit the signal. This is done with the intriguing line: `emit m_s("Hello from a class");`. After spending several years with Qt, it just feel so natural to `emit` a signal and since I wanted to keep the essence of the library close to already existing constructs to ease the transition, the `emit` was born. `emit` will be dissected later in the article to understand how it works.

The global method `void a_function(const char* msg)` is the slot which is connected to this signal. It does not do very much; it only prints the message it receives from the signal to stdout, but for demonstration purposes this is acceptable.

And now we have reached to the main method of the application, which creates an object of type `a_class` and connects its signal: `a.m_s` to the global function `a_function`. And, last but not least, the `say_hello`

the compiler will take care that slots with matching signatures to the ones the signal requires are actually available

method of the class is called, which in its turn will emit the signal. Upon emitting, the mechanism hidden in the library will kick in and the `a_function` will be called. There is support in the library to obtain the object which emitted the signal the current slot is handling by calling the `miso::sender` method; however, this is not presented in this short example.

This was a short example, now it is time to break down the application into tiny pieces, and start examining it.

The miso namespace

There are the following interesting elements in the `miso` namespace

1. The `signal` class
2. The `connect` and the `sender` methods
3. The macro definition for `emit`. Although this is not namespace dependent, it just felt right to place it there.

There is also another namespace, called `internal` with the intention that this is not to be used by the end-users.

Due to these being interconnected, I will present them one by one; however, be prepared for several jumps between various components, and since the namespace level entities use the internals very heavily it will be necessary to dig into them too.

The signal class

The class responsible for creating signals has the following declaration:

```
template <class... Args> class signal final
```

My intention was to keep the signal objects final, in order to have a clean interface and easy implementation; however, this does not stop you from removing the `final` and providing good implementation for use cases for signal derived classes.

Since the class is a template class, nothing stops you from creating signals for your own data types and using them properly in the emit and the receiving slot declaration.

A short overview of the public members is as follows: The default constructor and destructors are marked `default`, we just let the compiler do its default work.

The following two methods are `connect` and `disconnect`, and as their name suggest these will connect (or disconnect) the signal to (from) a slot.

Right now the following entities can be used as slots:

A function

The function must be declared with parameters corresponding to the parameters of the signal, and these parameters are not restricted only to basic C++ types. Using `std::function` values also works, and so do the `static` methods of various classes (see Listing 2).

The example in Listing 2 will call `b_function` twice, which will print 'Hello from the Other class method' twice because it is connected twice to the same signal.

```
struct other_class {
  void method() const {
    std::cout
      << "Hello from the Other class method";
  }
};
struct a_class {
  miso::signal<other_class> m_s;
  void say_hello() {
    emit m_s(other_class());
  }
};
void b_function(other_class oc) {
  oc.method();
}
int main() {
  a_class a;
  std::function<void (other_class)> f
    = b_function;
  miso::connect(a.m_s, f);
  miso::connect(a.m_s, b_function);
  a.say_hello();
}
```

Listing 2

A lambda expression

The lambda expression can either be coming from an `auto l = []() {...}` expression, or simply be written as a parameter to the `connect` method. Again, correct matching of lambda parameters is required. So, an example for the above source code would be:

```
miso::connect(a.m_s, [](other_class b) {
  b.method(); });
```

A functor

A function object allows the instantiation of a functor class to be invoked in a manner similar to functions by providing an overload to `operator ()`. So, in order to use a functor as a slot we can have the code in Listing 3.

As a side note, if there is more than one overload of `operator()` it is possible to connect more than one signals to the same functor, each being handled by its own `operator()`. And since this is an over-templated solution, the compiler will take care that slots with matching signatures to the ones the signal requires are actually available, otherwise it will spectacularly fail with a long list of cryptic messages.

Connect internals

In the `signal` class,`connect` and `disconnect` are implemented both using `internal::connect_i`, by calling it as shown in Listing 4, where the `T&& f` is just the slot where we want this signal to reach upon

I consider this piece of code to be one of the small wonders of the powers of modern C++

```
struct functor {
  void operator()(int aa) {
    std::cout << "functor class's int slot:"
              << aa << std::endl;
  }
};
struct a_class {
  miso::signal<int> m_s;
  void say_hello() {
    emit m_s(42);
  }
};
int main() {
  a_class a;
  functor f;
  miso::connect(a.m_s, f);
  a.say_hello();
}
```

Listing 3

```
template<class T>
void connect(T&& f, bool active = true) {
  internal::connect_i<T,
    typename slot_holder<T>::FT, slot_holder<T>>
    (std::forward<T>(f), slot_holders, active);
}
```

Listing 4

emitting, and **active** decides whether this signal is active or not (**disconnect** calls the same function with **active = false**).

The parameters to the internal function follow by using forward on the **f** parameter to the current function, then **slot_holders** which is a local variable of type:

```
std::vector<internal::common_slot_base*>
  slot_holders;
```

And finally, **active** to tell the framework whether this signal is active or not (ie: should be called upon **emit** or not).

Since **common_slot_base** has appeared now, here is a definition for it:

```
struct common_slot_base {
  virtual ~common_slot_base() = default;
};
```

so, basically it is just an interface to be used by all the different kinds of signals as a means of calling their corresponding slots. An immediate usage of it is in the **signal** class:

```
struct slot_holder_base :
  public internal::common_slot_base {
    virtual void run_slots(Args... args) = 0;
};
```

with further specialization following in Listing 5.

Reading the last method, it is obvious that the main action happens here, i.e. the actual call of a slot as per the corresponding signal takes place in these lines.

A bit more investigation of this structure gives us the declaration of **FT** being an **std::function** which at compile time identifies its return type from the template parameter of the **slot_holder** class (**T** which is supposed to be a 'Callable') which is fed into the **std::result_of** of the **<type_traits>** header having the parameters **Args...** of the signal class that this **slot_holder** resides in, combined again with the **Args...** of the signal to obtain a fully understandable expression. Just a clarification, **FT** stands for **F**unction **T**ype. And last but not least about this construct: Personally, I consider this piece of code to be one of the small wonders of the powers of modern C++... (read: even after writing it, and knowing that it's syntactically correct and valid code, in my weaker moments I still wonder that it compiles...)

Since in this structure we have introduced a new structure (**func_and_bool**), here is its definition:

```
template<typename FT>
struct func_and_bool final {
  std::shared_ptr<FT> ft;
  bool active;
  void *addr;
};
```

which roughly holds the lowest level of a slot, i.e.: a function object, whether it is active or not, and its address, thus revealing that at the lowest level all slots are decaying into an **std::function** (the one which was declared in the type name **FT** of the **struct slot_holder**).

Now, that we have covered the necessary structures and functions of a signal, it is time to look at the actual function from the internals, which performs the real connect (see Listing 6).

So, dissecting it into bits we can observe the following:

```
template<class T>
struct slot_holder : public slot_holder_base {
  using FT = std::function<typename
    std::result_of<T(Args...)>::type(Args...)>;
  using slot_vec_type =
    std::vector<internal::func_and_bool<FT>>;
  slot_vec_type slots;
  void run_slots(Args... args) override
  {
    std::for_each(slots.begin(), slots.end(),
      [&](internal::func_and_bool<FT>& s)
    { if (s.active) (*(s.ft.get()))(args...); }
        );
  }
};
```

Listing 5

we check whether the newly created object is in the slot holder already (by comparing its physical address to those already in the container)

- The type of the **`static SHT sh;`** local variable came in via the template parameters, and for our case it will have the structure **`slot_holder`** declared in the **`signal`** class. Now, this **`sh`** (slot holder, for the uninitiated) variable will be common for all the **`connect_i`** functions sharing a common prototype (hence, the static). For the perverse among you, **`SHT`** stands for **S**lot **H**older **T**ype; don't you dare start thinking of anything else. There is just one small drawback to using this static variable: **miso** in its current incarnation is not thread safe (so if someone is feeling tempted to fix this issue … feel free to make a pull request on github or depending on time and resources, the author might fix it).
- The next step is to create a **`func_and_bool`** object with the type of the **`FT`** we discussed in the **`slot_holder`** class, and we check whether the newly created object is in the slot holder already (by comparing its physical address to those already in the container). If yes, we set its activeness state to the one required in the parameter, but since we don't want to add it again, we also flip a boolean flag for later usage.
- The next step is updating the incoming parameter **`sholders`** in order to append the local **`sh`** object. This is where the magic happens since this parameter is the same that is declared in the **`signal`** class, and since **`slot_holder<T>`** is a **`common_slot_base`** specialization we successfully managed to gather all the slots regardless of their parameters, this type of signal class is connected to into one common entity we can operate on.

```
template<class T, class FT, class SHT>
void connect_i(T &&f,
  std::vector<common_slot_base *> &sholders,
  bool active = true)
{
  static SHT sh;
  func_and_bool<FT> fb{
    std::make_shared<FT>(std::forward<T>(f)),
    active, reinterpret_cast<void *>(&f)
  };
  bool already_in = false;
  std::for_each(sh.slots.begin(), sh.slots.end(),
    [&](func_and_bool<FT> &s)
    {
      if (s.addr == fb.addr)
      {
        s.active = active;
        already_in = true;
      }
    }
  );
  if (!already_in)
  {
    sh.slots.emplace_back(fb);
  }
  if (std::find(sholders.begin(),
    sholders.end(),
    static_cast<common_slot_base *>(&sh)) ==
      sholders.end())
  {
    sholders.push_back(&sh);
  }
}
```

Listing 6

With these covered we have successfully surveyed the mechanisms behind the connection of a slot to a specific signal, so we can jump to the next stage of our library, namely, emitting a signal.

Emitting a signal

The syntax, as seen from the tiny example application provided, is:

```
emit signalname(param1, param2, ...);
```

By digging further in the header file, we find that **`emit`** basically is:

```
#define emit \
miso::internal::emitter\
<std::remove_pointer<decltype(this)>\
::type>(*this) <<
```

(Yes, that is a **`<<`** operator at the end of the line)

So, a simple **`emit`** will create in its turn a temporary **`miso::internal::emitter`** object, which is a helper class like Listing 7, whose role is to keep track of the current object that emitted the signal. I'm confident that the logic for this is covered in the nice self-

```
template<class T>
struct emitter final {
  explicit emitter(const T &emtr) {
    sender_objs.push(&emtr);
    minstance = this;
  }
  ~emitter() {
    sender_objs.pop();
    minstance = nullptr;
  }
  static T *sender() {
    return const_cast<T *>(sender_objs.top());
  }
  static emitter<T> *instance() {
    return minstance;
  }
private:
  static std::stack<const T *> sender_objs;
  static emitter<T> *minstance;
};
```

Listing 7

```
struct a_class {
  miso::signal<int> m_s;

  void say_hello() {
    emit m_s(42);
  }
  int x = 45;
};
struct functor {
  void operator()(int aa) {
    std::cout << "functor class's int slot:"
      << aa << std::endl;
    a_class* ap = miso::sender<a_class>();
    std::cout << "x in emitter class:" << ap->x
      << std::endl;
  }
};
int main() {
  a_class a;
  functor f;
  miso::connect(a.m_s, f);
  a.say_hello();
}
```

Listing 8

explanatory code above, so let's just give an example of how can we retrieve the sender of the current signal (see Listing 8).

So, in the slot, we just simply call:

```
a_class* ap = miso::sender<a_class>();
```

and this gives us the type of the class that has emitted the signal resulting in us being in the current **slot**. Be aware, that if we are not handling the slot class due to an emit from a signal, and we call the **miso::sender** we will get a **std::runtime_error** exception.

The internals of calling the signal handler

If you wonder about the **<<** operator in the macro definition of **emit**, please note the **signal** class has a very complex friend declaration, in the form of:

```
template<class T, class... Brgs> friend
internal::emitter<T> && internal::operator
  << (internal::emitter<T> &&e,
  signal<Brgs...> &s);
```

which looks like:

```
template<class T, class... Args>
emitter<T> &&operator
  <<(internal::emitter<T> &&e,
    signal<Args...> &s) {
  s.delayed_dispatch();
  return std::forward<internal::emitter<T>>(e);
}
```

To make the syntax possible, please also note the following in the **signal** class:

```
std::tuple<Args...> call_args;
signal<Args...>& operator()(Args... args) {
  call_args = std::tuple<Args...>(args...);
  return *this;
}
```

(so, the signal class in its turn is also a functor :)) otherwise the required syntax for **emit** wouldn't have been possible. The **call_args** member is nothing more than the calling arguments for emitting the signal populated by this **operator()**.

Now we can see that the temporary **emitter** object created above will call the overloaded **<<** operator with the signal (which in its turn has already consumed the input parameters via the **operator()** call) , and in there the **delayed_dispatch** method of the signal is called.

When it comes to delayed dispatch, [Stackoverflow] shows us how to unpack a tuple holding various values of various types to a function with matching parameter types. This is necessary in order to have a perfect match between the values the signal's call arguments were populated with and the slots that are supposed to get the same values.

When the delayed dispatch method runs, it in turn calls **run_slots** from the slot holders vector (which, if you remember, were populated in the **connect** step).

The future

With this lengthy overview, I am confident, that everyone needing to use a lightweight signal-slot library has at least one more choice to select from, making the decision even harder. At the same time, I'm hoping that this article has shed some light into how to use this library. Whether it is a good choice for your team or not that depends entirely on you.

The library

You can find the implementation of the library in `miso.h` (released under MIT license) at https://github.com/fritzone/miso

Also, please note: For now, this is far from a fully fledged signal-slot library, offering the power and functionality you would expect from Qt or Boost. Depending on time and resources, I would be happy to add features you request (or even approve your pull request in case you consider it worth fixing a few bugs here and there, or adding a new nice to have item to it) but till then kindly treat it lightly.

References

[Bass02] Phil Bass (2002) 'Implementing the Observer Pattern in C++' in *Overload* 52, https://accu.org/var/uploads/journals/overload52-FINAL.pdf, and *Overload* 53, https://accu.org/var/uploads/journals/overload53-FINAL.pdf

[BoostSigSlot] http://www.boost.org/doc/libs/1_61_0/doc/html/signals2.html

[cpp-signal] https://github.com/Montellese/cpp-signal

[Goodliffe00] Pete Goodliffe (2000) 'Experiences of Implementing the Observer Design Pattern' in *Overload* 37, https://accu.org/index.php/journals/488, *Overload* 38, https://accu.org/index.php/journals/481, and *Overload* 41, https://accu.org/index.php/journals/464

[Griffiths14] Alan Griffiths (2014) 'Designing Observers in C++11' in *Overload* 124, https://accu.org/var/uploads/journals/Overload124.pdf#page=5

[Hogan] https://github.com/pbhogan/Signals

[libsigc++] http://libsigc.sourceforge.net/

[nanosigslot] https://github.com/NoAvailableAlias/nano-signal-slot

[neosigslot] http://i42.co.uk/stuff/neosigslot.htm

[nod] https://github.com/fr00b0/nod

[Qt4SigSlot] *C++ GUI Programming with Qt 4* by Jasmin Blanchette & Mark Summerfield, ISBN-13: 978-0131872493

[Qt5SigSlot] http://doc.qt.io/qt-5/signalsandslots.html

[QtMoc] http://doc.qt.io/qt-5/moc.html

[sigcxx] https://github.com/zhanggyb/sigcxx

[sigslot-1] sigslot, a signals and slots library written by Sarah Thompson, http://sigslot.sourceforge.net

[sigslot-2] https://github.com/supergrover/sigslot

[Stackoverflow] http://stackoverflow.com/questions/7858817/unpacking-a-tuple-to-call-a-matching-function-pointer

[VDK] vdk-signals at http://vdksoft.github.io/signals/index.html

[Wikipedia] https://en.wikipedia.org/wiki/Signals_and_slots

(Re)Actor Allocation at 15 CPU Cycles

(Re)Actor serialisation requires an allocator. Sergey Ignatchenko, Dmytro Ivanchykhin and Marcos Bracco pare malloc/free down to 15 CPU cycles.

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

Task definition

Some time ago, in our (Re)Actor-based project, we found ourselves with a need to serialize the state of our (Re)Actor. We eventually found that app-level serialization (such as described in [Ignatchenko16]) is cumbersome to implement, so we decided to explore the possibility of serializing a (Re)Actor state at allocator level. In other words, we would like to have all the data of our (Re)Actor residing within a well-known set of CPU/OS pages, and then we'd be able to serialize it page by page (it doesn't require app-level support, and is Damn Fast™; dealing with ASLR when deserializing at page level is a different story, which we hope to discuss at some point later).

However, to serialize the state of our (Re)Actor at allocator level, we basically had to write our own allocator. The main requirements for such an allocator were that:

- We need a separate allocator for each of (Re)Actors running in the same process
- At the same time, we want our app-level (Re)Actors to be able to use simple *new* and *delete* interfaces, without specifying allocators explicitly
- We need each of our allocators to reside in a well-defined set of CPU pages (those pages obtained from the OS via `mmap()`/ `VirtualAllocEx()`)

  This will facilitate serialization (including stuff such as Copy on Write in the future).
- We do NOT need our allocator to be multi-threaded. In the (Re)Actor model, ***all*** accesses from within (Re)Actor belong to one single thread (or at least 'as if' they are one single thread), which means that we don't need to spend time on thread sync, not even on atomic accesses.

  The only exception is when we need to send a message to another (Re)Actor. In this case, we MAY need thread-safe memory but, in comparison with intra-(Re)Actor allocations, this is a very rare occurrence – so we can either use standard `malloc()` for this purpose or write our own message-oriented allocator. The latter will have very different requirements and therefore very different optimizations from the intra-(Re)Actor allocator discussed in this article.

Actually, when we realized that we only needed to consider single-threaded code was when we thought, 'Hey! This can be a good way to improve performance compared to industry-leading generic allocators'. Admittedly, it took more effort than we expected, but finally we have achieved results which we think are interesting enough to share.

What our allocator is NOT

By the very definition of our task, our allocator does ***not*** aim to be a drop-in replacement for existing allocators (at least, not for ***all*** programs). Use of our allocator is restricted to those environments where all accesses to a certain allocator are guaranteed to be single-threaded; two prominent examples of such scenarios are message-passing architectures (such as Erlang) and (Re)Actors a.k.a. Actors a.k.a. Reactors a.k.a. ad hoc FSMs a.k.a. Event-Driven Programs.

In other words, we did ***not*** really manage to outperform the mallocs we refer to below; what we managed to do was to find a (very practical and very important) subset of use cases (specifically message passing and (Re)Actors), and write a highly optimized allocator specifically for them. That being said, while writing it, we did use a few interesting tricks (discussed below), so some of our *ideas* ***might*** be usable for regular allocators too.

On the other hand, as soon as you're within (Re)Actor, our allocator ***does not*** require additional programming effort from the app-level; this gives it an advantage over manually managed allocation strategies such as used by `Boost::Pool` (not forgetting that, if necessary, you can still use `Boost::Pool` over our allocator).

Major design decisions

When we started development of our allocator (which we named **iibmalloc**, available at [Github]), we needed to make a few significant decisions.

First, we needed to decide how to achieve multiple allocators per process, preferably without specifying an allocator at app-level explicitly. We decided to handle it via TLS (`thread_local` in modern C++). Very briefly:

- By task definition, our allocator is single-threaded.

  This allows us to speak in terms of the 'function which is *currently* executed within the current thread'.

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

Sergey Ignatchenko has 15+ years of industry experience, including being a co-architect of a stock exchange, and the sole architect of a game with 400K simultaneous players. He currently holds the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

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

Marcos Bracco has a degree in electronics engineering from UNLP in Argentina and 15 years of software development experience. Marcos can be contacted at marcosbracco@gmail.com

- Moreover, at any point in time, we can say which allocator is currently used by the app level (it is the allocator which belongs to the currently running (Re)Actor).

  Even if there are multiple (Re)Actors per thread, this still stands.
- Hence, a `thread_local` allocator will do the job:
  - For a *single (Re)Actor per thread*, we can have a per-thread allocator (this is the model we were testing for the purposes of this article).
  - For *multiple (Re)Actors per thread*, our Infrastructure Code (which runs threads, instantiates (Re)Actors, and calls `Reactor::react()`) can easily put a pointer to the allocator of the current (Re)Actor right before calling the respective `Reactor::react()`.
- In addition, we found that the performance penalties of accessing TLS (usually one indirection from a specially designated CPU register into a highly-likely cached piece of memory) are not too high even for our very time-critical code.

Second, we needed to decide whether we want to spend time keeping track of whole pages becoming empty so we can release them. Based on the logic discussed in [NoBugs18], we decided in favor of ***not*** spending any effort on keeping track of allocation items as long as there is more than one such item per CPU page.

Third, in particular based on [NoBugs16], we aimed to use *as few memory accesses as humanly possible*. Indeed, on modern CPUs, register–register operations (which take ~1 CPU cycle) are pretty much free compared to memory accesses (which can go up to 100+ CPU cycles).

Implementation

We decided to split all our allocations into four groups depending on their size:

- ‘small’ allocations – up to about one single CPU page.

  We decided to handle them as ‘bucket allocators’ (a.k.a. ‘memory pools’). Each page contains buckets of the same size; available bucket sizes are some kind of exponent so that we can keep overheads in check.

  Whenever a new page for a specific bucket size is allocated, we ‘format’ it, creating a linked list of available items in the page, and adding these items to the ‘bucket’, which is essentially a single-linked list with all the free items of this size.
- ‘medium’ allocations – those taking just a few pages (currently – up to 4 pages IIRC).

  These are also handled as ‘bucket allocators’, but they may span several CPU pages (we named these ‘multi-pages’). Note that for ‘medium’ allocations, all the allocation items are page-aligned.
- ‘large’ allocations – those which are already too large for buckets, but which are still too small to request from the OS directly as a single range (doing so would create too many virtual memory areas a.k.a. VMAs, and may result in running out of available VMA space – which is manifested by `ENOMEM` returned by `mmap()` even if there is still ***lots*** of both of address space and physical RAM). Currently, ‘large’ allocations go up to about a few hundred kilobytes in size.

  ‘large’ allocations are currently handled as good ol’ Knuth-like first-fit allocators working at page level (i.e. granularity of allocations is one page), and with some further relatively minor optimizations.

  ‘Large’ allocations are not aligned at page boundaries (though, of course, they’re still aligned at 8-byte boundaries).
- ‘very large’ allocations – those allocations which are large enough to feed them to the OS directly. Currently, they start at about a few hundred kilobytes.

  Like ‘large’ allocations, ‘very large’ allocations are not aligned on page boundaries.

It was when we faced the problem of how to do deallocation efficiently that we got into the really interesting stuff

'very large' allocations are the only kind of allocations which can be returned back to the OS.

Optimizing allocation – calculating logarithms

Up to now, everything has been fairly obvious. Now, we can get to the interesting part: specifically, what did we do to optimize our allocator? First, let's note that we spent *most* of our time optimizing 'small' and 'medium' allocations (on the basis that they're by far the most popular allocs in most apps, especially in (Re)Actor apps).

The first problem we faced when trying to optimize small/medium allocations was that – given the allocation size, which comes in a call to our **`malloc()`** – we need to calculate the bucket number. As our bucket sizes are exponents, this means that effectively we had to calculate an (integer) logarithm of the allocation size.

If we have bucket sizes of 8, 16, 32, 64, … – then calculating the integer logarithm (more strictly, finding the greatest integer so that two raised to that integer is less or equal to the allocation size) becomes a cinch. For example, on x64 we can/should use a BSR instruction, which is extremely fast. (How to ensure that our code generates a BSR is a different compiler-dependent story but it can be done for all major compilers.) Once we have our BSR, skipping some minor details, we can calculate **`bucket_number = BSR(size-1)-2`**, or, in terms of bitwise arithmetic, the ordinal number of the greatest bit set of (size-1) minus two.

However, having bucket sizes *double* at each step leads to significant overheads, so we decided to go for a 'half-exponent' sequence of 8, [12 omitted due to alignment requirements], 16, 24, 32, 48, 64, … In this case, the required logarithm to find our bucket size can still be calculated very quickly along quite similar lines: it is a doubled ordinal number of a greatest bit set of (size-1) plus second greatest bit of (size-1) minus five.

These are still register-only operations, are still branch-free, and are still extremely fast. In fact, when we switched to 'half-exponent' buckets, we found that – due to improved locality – the measured speed *improved* in spite of the extra calculations added.

Optimizing deallocation – placing information in a dereferenceable pointer?!

The key, the whole key, and nothing but the key, so help me Codd

~ unknown

Up to now, we have described nothing particularly interesting. It was when we faced the problem of how to do deallocation efficiently that we got into the *really* interesting stuff.

Whenever we get a **`free()`** call, all we have is a pointer, and nothing but a pointer (for C++ **`delete`**). And from this single pointer we need to find: (a) whether it is to a 'small', 'medium', 'large', or 'very large' allocated block, and for small/medium blocks, we have to find (b) which of the buckets it belongs to.

Take 1 – Header for each allocation item

The most obvious (and time-tested) way of handling it is to have an allocated-item header preceding each allocation item, which contains all the necessary information. This works, but requires 2 memory reads (cached ones, but still taking 3 cycles or so each) and, even more importantly, the item header cannot be less than 8 bytes (due to alignment requirements), which means up to twice the overhead for smaller allocation sizes (which also happen to be the most popular ones).

We tried this one, it did work – but we were sure it was possible to do it better.

Take 2 – Dereferenceable pointers and bucket page headers

For our next step, we had two thoughts:

- All large/very-large allocated items have values of **`CPU_page_start+16`** (this happens naturally as these items in our implementation always start at page beginning, after a 16-byte header). BTW, '16' is not really a magic number, it is just the size of a large/very-large item header.

  We can also ensure that all small/medium pointers *never* start at **`CPU_page_start+16`**. This is assured by the 'bucket page formatting' routine, which, if it runs into such a size, simply skips this one single item (note that it won't happen for larger item sizes, so the memory overhead due to such skipping is negligible).

  This means that (assuming a 64-bit app and a 4K-page, but for other page sizes the logic is very similar) an expression **`((pointer_to_be_freed&0xFFF)==16)`** will give us an answer to the question of whether we're freeing a small/medium alloc or a large/very-large alloc.

  BTW, this means that we already achieved the supposedly-impossible feat of effectively placing a tiny bit of information into a dereferenceable pointer. In other words, having *nothing but the pointer itself* (not even accessing the memory the pointer refers to), we can reach conclusions about certain properties of the memory it points to.

- And for small/medium allocs, we can exploit the fact that all of the buckets within the same page are of the same size. This means that if we place a header into each page (instead of placing it into each allocated item), we'll be able to reach it using **`((page_header*)(pointer_to_be_freed&0xFFFFFFFFFFFFF000))`** – and get information about the bucket number out of our page_header.

This approach worked, but while it reduced memory overhead, the cost of the indirection to the **`page_header`** (which was less likely to be cached than the allocation item header) was significant, so we observed minor performance degradation.☹

Take 3 - Storing the bucket number within a dereferencable pointer

However, (fortunately) we didn't give up - and came up with the following schema, which effectively allows us to extract the bucket number from each small/medium allocated pointer. It requires a bit of explanation.

Whenever we're allocating a bunch of pages from the OS (via `mmap()`/`VirtualAllocEx()`) – we can do it in the following manner:

- Let's assume we have 16 buckets (this can be generalized for a different number of buckets, even for non-power-of-2 ones, but let's be specific here).
- We're *reserving* 16 pages, without *committing* them (yet). Sure, it does waste a bit of address space – but at least for 64-bit programs it is not really significant; and as we're *not* committing, we do *not* waste any RAM (well, except for an additional VMA, but a number of VMAs have to be addressed separately anyway).

  As for the wasting of *address space*, in the worst possible case such a waste is 16x (it won't happen in the real-world, but let's assume for the moment it did). And while 16x might look a lot, we can observe that modern OSs running under x64 CPUs have 47-bit address spaces; even with the 16x worst-case overhead, we still can physically allocate 2^{43} bytes of RAM, or 8 Terabytes of RAM – which is still well beyond practical capabilities of any x64 box I've ever heard of (as of this writing, even the largest TPC-E boxes which cost $2 million, use 'only' 4T of RAM). If you ever have such a beast at your disposal, we'll still have to see whether it will need all this memory within a single process. In any case, it is clear that this waste won't matter for the vast majority of currently existing systems.
- Very basic maths *guarantees* us that among our reserved 16 pages, there is *always exactly one* page for which the expression `page_start&0xF000` has the value 0x0000, and *exactly one* page for which the expression `page_start&0xF000` has the value 0x1000, and so on all the way up to 0xF000. In other words, while we do *not* align our reserved page range, we still can rely on having one page with each of 16 possible values of a certain pre-defined expression over a `page_start` pointer(!).
- Now, we're saying, that we need to allocate buckets for `bucket_number` 7, so let's pick the page which has the expression `page_start&0xF000 == 0x7000` (as noted above, such a page *always* exists in our allocated range). Then *commit* and 'format' this page to have buckets corresponding to bucket index == 7.
- Of course, whenever we need a page for a different bucket size, we can (and should) still re-use those reserved-but-not-yet-committed pages, committing memory for them and formatting them for the sizes which follow from their `page_start&0xF000`.

After we're done with this, we can say that:

> *For each and every 'small'/'medium' pointer to be freed, the expression (*`(pointer_to_be_freed>>12)&0xF`*) gives us the bucket number.*

This information can be extracted purely from the pointer, *without* any indirections(!). In other words, by doing some magic we *did* manage to put information about the bucket number into the pointer itself(!!).

In practice, it was a bit more complicated than that (to avoid creating too many VMAs, we needed to reserve/commit pages in larger chunks – such as 8M), but the principles stated above still stand in our implementation.

This approach happened to be the best one both performance-wise *and* memory-overhead-wise.

How our deallocation works

To put all the pieces of our deallocation together, let's see how our deallocation routine works:

- We take the pointer to be freed (which is fed to us as a parameter of a `free()` call), and use something along the lines of (`(pointer_to_be_freed&0xFFF)==16`) to find out if it was small/medium alloc, or large/very-large one. NB: there is a branch here, but large/very-large blocks happen rarely, so mispredictions are rare.
- If it is a large/very-large item, we're using a traditional header-before-allocation-item. As this happens rarely, performance in this branch is not too important (it is fast, but it doesn't need to be Damn Fast™).
- If it is a small/medium item, we calculate the bucket size using (`(pointer_to_be_freed>>12)&0xF`) and then simply add the current pointer to be freed to the single-linked list of the free items in this bucket.

  This is the most time-critical path – and we got it in a *very* few operations (maybe even close to 'the-least-possible'). Bingo!

Test results

Of course, all the theorizing about 'we have very few memory accesses' is fine and dandy, but to map them into real world, we have to run some benchmarks. So, after all the optimizations (those above and others, such as forcing the most critical path – and only the most critical path – to be inlined), we ran our own 'simulating real-world loads' test [Ignatchenko18] and compared our **iibmalloc** with general-purpose (multithreaded) allocators. We feel that the results we observed for our **iibmalloc** were well-worth the trouble we took while developing it.

The testing is described in detail in [Ignatchenko18], with just a few pointers here:

- We tried to simulate real-world loads, in particular:
  - The distribution of allocation sizes is based on *p*~(1/*sz*) (where *p* is the probability of getting allocations of size *sz*).
  - The distribution of life times of allocated items is based on a Pareto distribution.
  - Each allocated item is written once and read once.
- All 3rd-party allocators are taken from the current Debian 'stable' distribution.
- Unless specified otherwise, we ran our tests with total allocation size of 1.3G.

  When running multiple threads, the total allocation size was split among threads, so the total allocation size for the whole process remained more or less the same.

As we can see (Figure 1), CPU-wise, we were able to outperform all the allocators at least by 1.5x.

And from the point of view of memory overhead (Figure 2), our **iibmalloc** has also performed well: its overhead was pretty much on par with the best alloc we have seen overhead-wise (jemalloc) – while significantly outperforming it CPU-wise.

Note that comparison of **iibmalloc** with other allocs is not a 100% 'fair' comparison: to get these performance gains, we had to give up on support for multi-threading. However, *whenever you can afford to keep the Shared-Nothing model* (='sharing by communicating instead of communicating by sharing memory'), this allocator *is* likely to improve the performance of malloc-heavy apps.

Another interesting observation can be seen in the graph in Figure 3, which shows results of a different bunch of tests, changing the size of allocated memory.

NB: Figure 3 is for a single thread, which as we seen above is the very best case for tcmalloc; for larger number of threads, tcmalloc will start to lose ground.

On the graph, we can see that when we're restricting our allocated data set to single-digit-megabytes (so everything is L3-cached and significant parts are L2-cached), then the combined costs of a `malloc()`/`free()` pair for our **iibmalloc** can be as little as 15 CPU clock cycles(!). For a `malloc()`/`free()` pair, 15 CPU cycles is a pretty good result, which we expect to be quite challenging to beat (though obviously we'll be happy if somebody

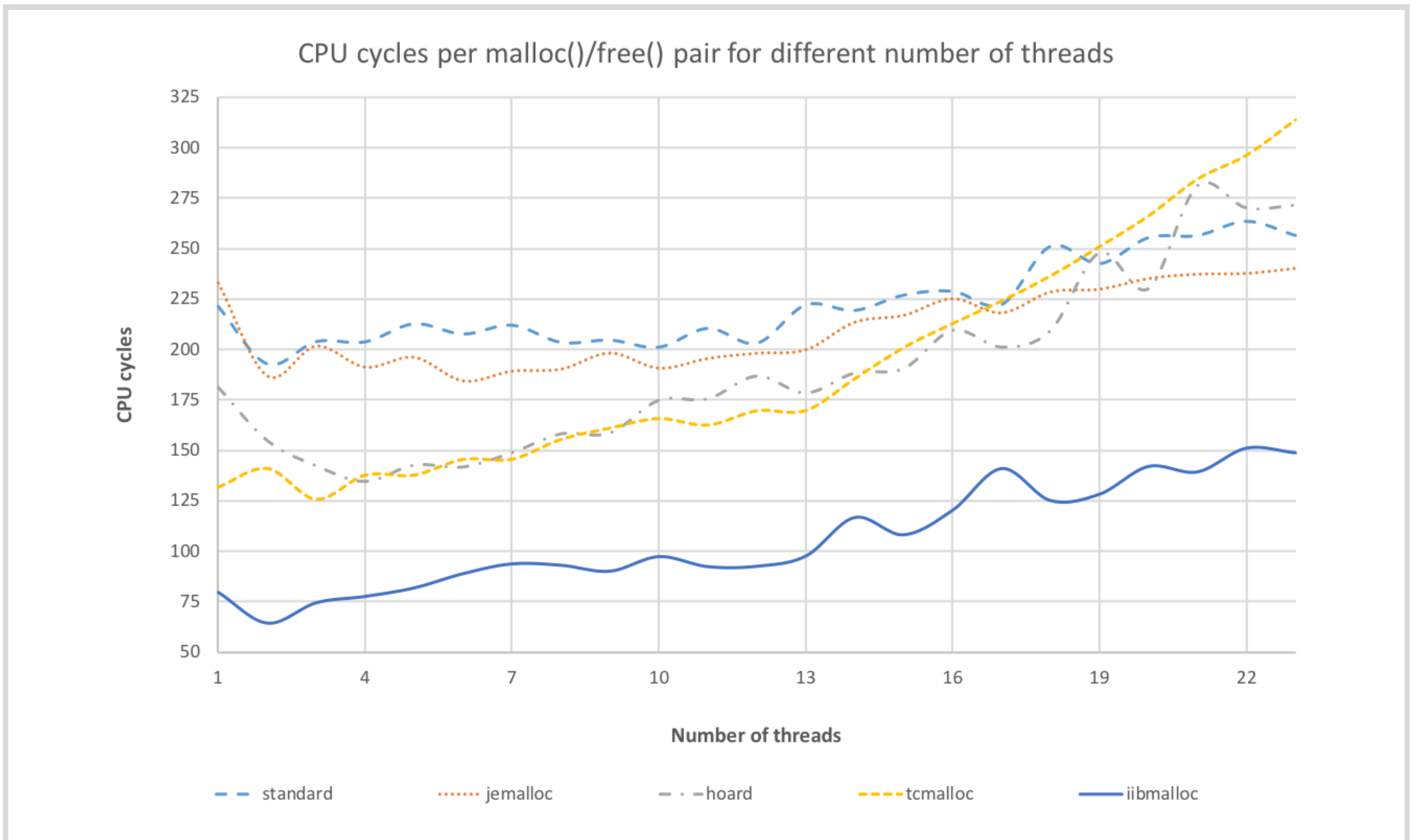


Figure 1

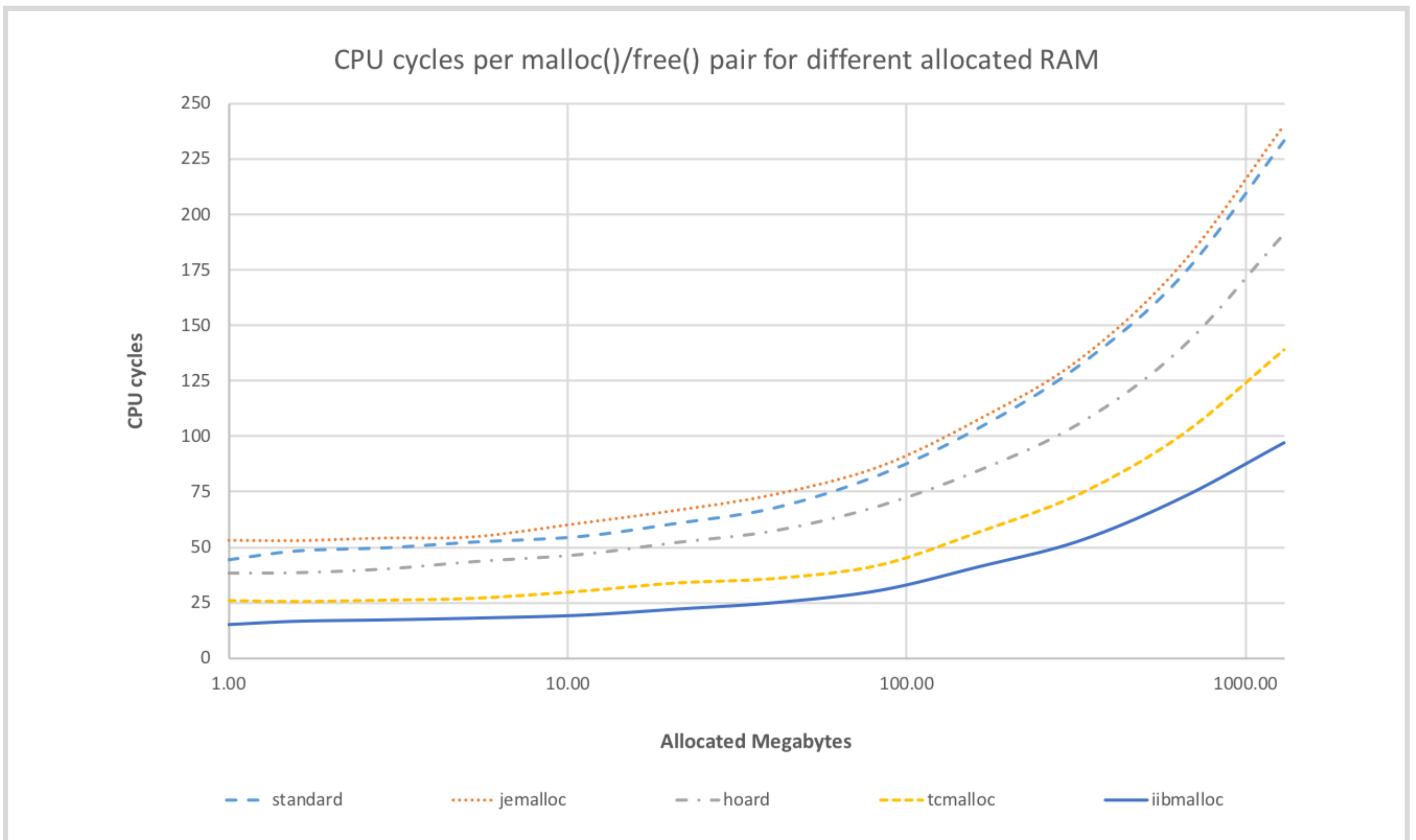


Figure 2

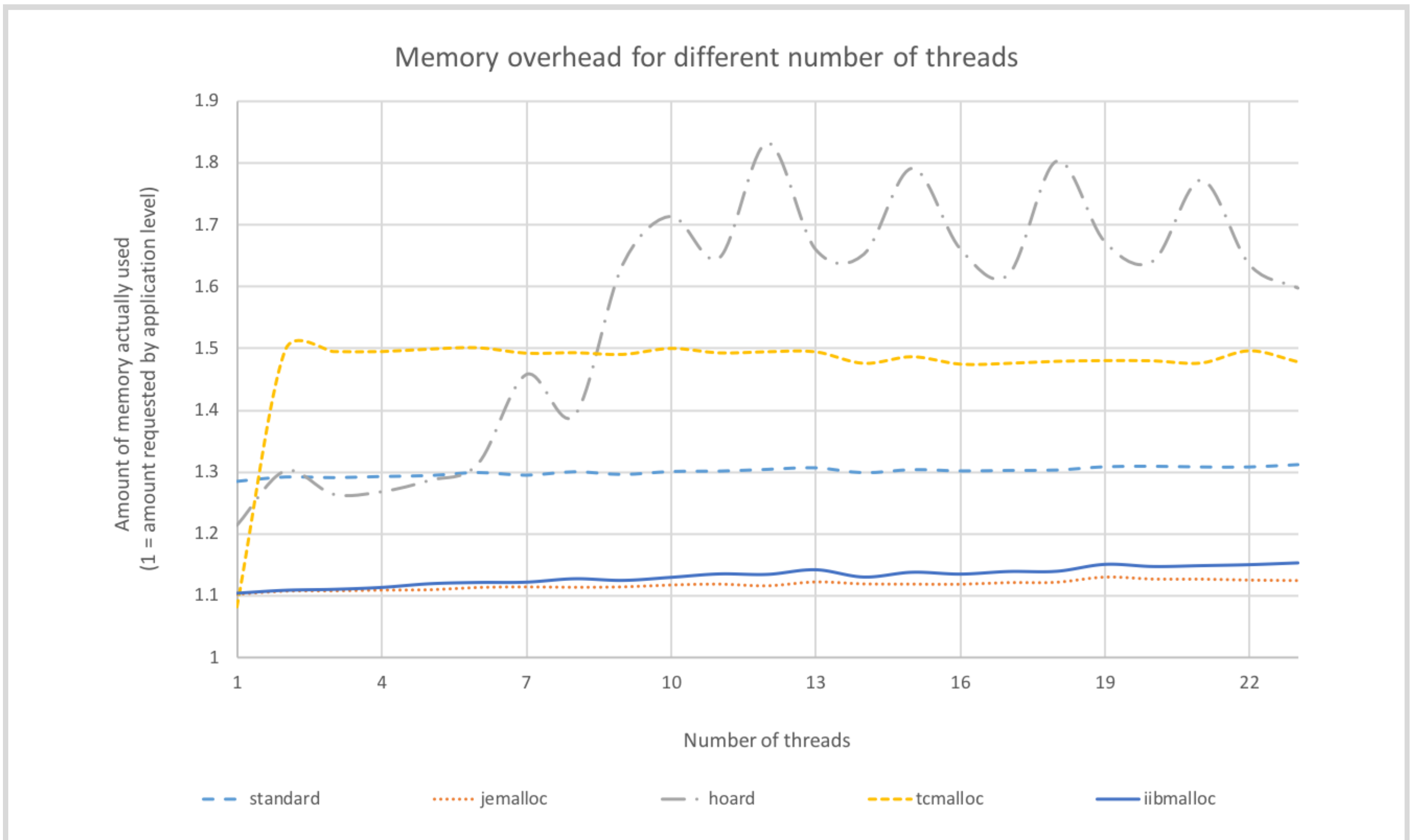


Figure 3

does). On the other hand, as we have spent only a few man-months on our allocator, there is likely quite a bit of room for further improvements.

Conclusions

We presented an allocator which exhibits significant performance gains by giving up multi-threading. We did not really try to compete with other allocators (we're solving a different task, so it is like comparing apples and oranges); however, we feel that we can confidently say that

> *For (Re)Actors and message-passing programs in general, it* ***is*** *possible to have a significantly better-performing allocator than a generic multi-threaded one.*

As a potentially nice side-effect, we also demonstrated a few (hopefully novel – at least we haven't run into them before) techniques, such as storing information in dereferenceable pointers, and these techniques *might* (or *might not*) happened to be useful for writers of generic allocators too. ■

References

[Github] https://github.com/node-dot-cpp/iibmalloc

[Ignatchenko16] Sergey Ignatchenko and Dmytro Ivanchykhin (2016) 'Ultra-fast Serialization of C++ Objects', *Overload* #136

[Ignatchenko18] Sergey Ignatchenko, Dmytro Ivanchykhin, and Maxim Blashchuk (2018) 'Testing Memory Allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc While Trying to Simulate Real-World Loads', http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/

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

[NoBugs16] 'Operation Costs in CPU Clock Cycles', 'No Bugs' Hare, http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

[NoBugs18] 'The Curse of External Fragmentation: Relocate or Bust!', 'No Bugs' Hare, http://ithare.com/the-curse-of-external-fragmentation-relocate-or-bust/

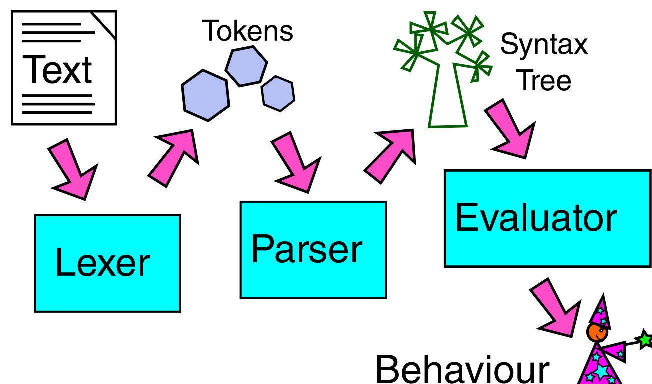
How to Write a Programming Language: Part 2, The Parser

We've got our tokens: now we need to knit them together into trees. Andy Balaam continues writing a programming language with the parser.

In this series we are writing a programming language, which may sound advanced, but is actually much easier than you might expect. Last time, we wrote a lexer, which takes in a text characters (source code) and spits out tokens, which are the raw chunks a program is made up of such as numbers, strings and symbols.

This time, we will write the parser, which takes the tokens coming out of the lexer and understands how they fit together, building structured objects corresponding to meaningful parts of our program, such as creating a variable or calling a function. These structured objects are called the syntax tree.

Next time, we'll work on the evaluator, which takes in the syntax tree and does the things it says, executing the program. By the end of this series, you will have seen all the most fundamental parts of an interpreter, and be ready to build your own!



A bit more about Cell

Last time we saw that the language we are writing, Cell, is designed to be simple to write, rather than being particularly easy to use. It also lacks a lot of the error handling and other important features of a real language, but it does allow us to do the normal things we do when programming: make variables, define functions and perform logic and mathematical operations.

One of the ways Cell is simpler than other languages is that things like **if** and **for** that are normally special keywords in other languages are just normal functions in Cell. This program demonstrates this idea for **if**:

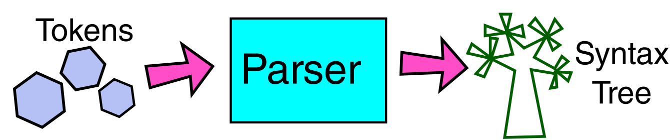
```
if(
  is_even( 2 ),
  { print "Even!"; },
  { print "Odd."; }
);
```

Andy Balaam Andy is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

In Cell, **if** is a function that takes three arguments: a condition, a function to call if the condition is true, and another to call otherwise (the **else** part). By passing functions as arguments, we avoid the need for a special keyword to define logical structures like **if** and **for**. This makes our parser simple, and it also means Cell programmers can write their own functions similar to the **if** function, and have them be first-class citizens, on a par to built-ins like **if** and **for**.

Because of the simplicity this allows, Cell's parser only needs to recognise a few simple structures.

Cell's parser



Cell has four expression types:

- Assignment: **x = 3**
- Operations: **4 + y**
- Function calls: **sqrt(-1)**
- Function definitions: **{:(x, y) x + y;}**

The parser's job is to recognise from the tokens it sees which of these expression types it is seeing, and build up a tree structure of the expressions. For example, this code snippet:

```
x = 3 + 4;
```

should be parsed to a tree structure something like this:

```
Assignment:
  Symbol: x
  Value:
    Operation:
      Type: +
      Arguments:
        3
        4
```

In Cell, you can tell what kind of expression you are looking at from the first two tokens. So in the example above, if we look at the first token ("**x**" we can't tell whether this is going to be an operation like **x + 2**, or an assignment. Once we have the second token (**=**) we know we are dealing with an assignment.

Once the parser has recognised we are dealing with an assignment, it can treat everything on the right-hand side of the **=** as a new expression. This new expression will be parsed and nested inside the tree structure of the first one. That is how **Operation** ends up inside the **Assignment** section above.

Cell is written in Python, and the tree structures built up by the parser are Python tuples like (**"operation"**, **"+"**, **3**, **4**) or (**"assignment"**, **"x"**, **18**).

the parser … takes the tokens coming out of the lexer and understands how they fit together, building structured objects corresponding to meaningful parts of our program

```
def parse(tokens_iterator):
  parser = Parser(PeekableStream(tokens_iterator),
    ";")
  while parser.tokens.next is not None:
    p = parser.next_expression(None)
    if p is not None:
      yield p
    parser.tokens.move_next()
```

Listing 1

```
class Parser:
  def __init__(self, tokens, stop_at):
    self.tokens = tokens
    self.stop_at = stop_at
```

Listing 2

They can be nested inside each other like this:

```
("assignment",
  "x",
  ("operation", "+", 3, 4)
)
```

which is the syntax tree representing the code `x = 3 + 4`.

Note: above we wrote `"x"`, `3` and `4` but in the actual syntax tree these will be full lexer tokens like (`"symbol"`, `"x"`) and (`"number"`, `"3"`).

Enough introduction – let's get into the code.

The parse() function

Listing 1 shows the `parse()` function. Its job is to create a `Parser` object and call its `next_expression` method repeatedly until we have processed all the tokens coming from the lexer. It uses the `PeekableStream` stream class that we saw in the previous article to create a stream of tokens that we can 'peek' ahead into to see the next token that is coming.

When we create the Parser object, we pass two objects in to its constructor: the stream of tokens, and `";"`, which tells the parser when to stop. Here we end when we hit a semi-colon because we are parsing whole statements, and all statements in Cell end with a semi-colon. Later we will make other `Parser` objects that stop parsing when they hit other types of token like `","` and `")"`.

The Parser class

Listing 2 shows the constructor of `Parser`, which just remembers the stream of tokens we are operating on and `stop_at`, the token type that tells us we have finished.

Listing 3 shows the real heart of the parser – the `next_expression` method of the `Parser` object. Similar to the `lex()` function we saw in the previous article, the `next_expression` method is built around a big `if`/`elif` block.

`next_expression` takes one argument, `prev`, that represents the progress we have made parsing so far. Earlier we found that we only need to see the first two tokens of an expression to know what type it is. By passing the previous expression in to `next_expression`, we can use it, along with the current token, to understand what kind of expression we have. If we're just starting to parse an expression, we pass in `None` as the value for `prev`.

Several of the branches of `next_expression` call `next_expression` from inside itself – this is because we are building up a nested tree of expressions within expressions. Every time we look for a sub-expression within an expression (for example the `"3 + 4"` part of `"x = 3 + 4"`) we call `next_expression` again, and use the return value as part of the original expression we are constructing.

Before we enter the big `if`/`elif` block, `next_expression` has an introductory section in which we get hold of the type and value of the next token we are dealing with, and stop parsing if we have hit one of the `stop_at` types. Since we were passed the expression so far in the `prev` argument, when we hit a `stop_at` token, we can immediately return it.

```
def next_expression(self, prev):
  self.fail_if_at_end(";")
  typ, value = self.tokens.next
  if typ in self.stop_at:
    return prev
  self.tokens.move_next()
  if typ in ("number", "string", "symbol") and prev is None:
    return self.next_expression((typ, value))
  elif typ == "operation":
    nxt = self.next_expression(None)
    return self.next_expression(("operation", value, prev, nxt))
  elif typ == "(":
    args = self.multiple_expressions(",", ")")
    return self.next_expression(("call", prev, args))
  elif typ == "{":
    params = self.parameters_list()
    body = self.multiple_expressions(";", "}")
    return self.next_expression(("function", params, body))
  elif typ == "=":
    if prev[0] != "symbol":
      raise Exception("You can only assign to a symbol.")
    nxt = self.next_expression(None)
    return self.next_expression(("assignment", prev, nxt))
  else:
    raise Exception("Unexpected token: " + str((typ, value)))
```

Listing 3

```
def multiple_expressions(self, sep, end):
  ret = []
  self.fail_if_at_end(end)
  typ = self.tokens.next[0]
  if typ == end:
    self.tokens.move_next()
  else:
    arg_parser = Parser(self.tokens, (sep, end))
    while typ != end:
      p = arg_parser.next_expression(None)
      if p is not None:
        ret.append(p)
      typ = self.tokens.next[0]
      self.tokens.move_next()
      self.fail_if_at_end(end)
  return ret
```

Listing 4

If we haven't finished, we enter the **if**/**elif** block that checks the type of the token we are processing and returns an expression of the right type.

First, we check what to do if we see a normal type (string, number or symbol) and we have no previous expression (because **prev** is **None**). This means we have only seen one token so far, so we can't decide what type of expression we are dealing with. To avoid making the decision yet, we call ourselves recursively, using **(typ, value)** – the token we were given – as the value for **prev**. This time we will have a non-**None** value for **prev** (because we just passed it in), and so will be able to make a decision about the expression we are parsing.

Next we check whether **typ** is **"operation"**. If it is, we are nearly ready to return an **"operation"** syntax tree. We have been given the left-hand side of the operation as **prev**, we've just found the operation, so all we need is the right-hand side to complete the expression. We call **next_expression** one more time, passing in **None** as the previous expression, because we want to find a separate expression to use at the right-hand side, and put the answer into a variable called **nxt**. Now we combine **nxt** with the information we already have, then return a tuple representing the whole operation: **("operation", value, prev, nxt)**. This is our syntax tree for this expression.

The next **elif** part checks for **"("**, which means we are calling a function. The **prev** variable should already contain the name of the function, so we just need to find the arguments we want to pass in. To find the arguments, we call **self.multiple_expressions**, which is shown in listing 4. Once we have the arguments, we can build a syntax tree of type **"call"** and pass it on into another call to **next_expression**..

By calling **next_expression** again, we allow multiple function calls to be stuck together, allowing us to write functions that return other functions and call them immediately. For example, **divide_by(3)(12);** might return **4**, because **divide_by(3)** could return a function that divides whatever you pass in by 3.

The **multiple_expressions** method parses several expressions separated by tokens of the type we provided (**"sep"**) and finishing when we get to another token (**"end"**). In the example we have seen so far, the separator was **","** and the end token was **")"** because we are looking for the arguments being passed to a function.

The code of **multiple_expressions** itself creates a new instance of the **Parser** class for every expression it looks for, telling it to stop when it hits the separator or the end, and stops looking when it hits the end.

Switching back to the big **if**/**elif** block from listing 3, the last two significant parts check for **"{"** and **"="** tokens. **"{"** means we are defining a function, so we use **multiple_expressions** again to find the statements inside the function, and to find the names of the arguments, it uses the **parameters_list** function, which is like a simplified version of **multiple_expressions** that just looks for the names of the arguments to the function (we skip it here for brevity).

The **"="** sign means we are defining a variable, which is quite simple – we just check that the previous token was a symbol, and then make an **"assignment"** syntax tree with that symbol and whatever is on the right-hand side.

If we get to the **else** part, we have encountered tokens in an order we can't recognise, and we raise an exception, which prints a (very unfriendly) error for the user.

If you've managed to follow so far, you have seen all the interesting parts of Cell's parser – why not try adapting it or writing your own language that works the way you want it to?

A note on operator precedence

If you were watching closely, you might have noticed one of the quirks of Cell's parser that makes it different from most similar-looking languages: the order in which expressions are grouped when they contain multiple terms.

Most languages follow rules inspired by mathematical expressions, so that e.g. multiplications are grouped together before additions, meaning **"3*4+1"** evaluates to 12.

Cell is different. Because we parse 'everything else' and use it as the right-hand side in the operation, we group things on the right before things on the left, and we treat all operators the same, so **3*4+1** evaluates to 15.

Cell works this way because it means have to write less code, but doing things the more normal way would be perfectly possible – we simply need to collect the full list of chained expressions before we start grouping them according to some precedence rules.

Summary

Parsing is an odd programming task, because we want to handle it piece by piece, but we sometimes need to soak up several tokens before we know what we are dealing with, and we need to produce a nested structure as our output. By using recursion (calling **next_expression** from inside itself) we can get the nested structure almost for free.

The code we looked at here is more complicated than the lexer we saw in the last article, but I think you'll agree there is no magic here. The whole of Cell's parser is just 81 lines of code (including empty lines). You can find it on Cell's GitHub site [Balaam] along with more explanations (including some videos).

While Cell's parser does work for a real, working language (if a toy one), it is a very simple example, and there is a huge amount you can learn about different types of parser, as well as tools that automatically build the code of a parser from some higher-level description of the language. A good place to start is the Wikipedia page 'Parsing' [Wikipedia].

You can find the whole source code for Cell on Github [Balaam], along with articles and videos explaining more about how it works.

Next time, we'll get to the real point: we'll look at the evaluator, which takes in the nice structured syntax tree produced by the parser and actually does things, turning our code into behaviour. ■

References

[Balaam] https://github.com/andybalaam/cell

[Wikipedia] 'Parsing', https://en.wikipedia.org/wiki/Parsing

Compile-time Data Structures in C++17: Part 1, Set of Types

Compile time data structures can speed things up at runtime. Bronek Kozicki details an implementation of a compile time set.

The standard collection **std::set** is known to every C++ programmer. The runtime complexity of its **find** and **insert** operations O(N ln N) is one of the reasons why it is often replaced with **std::unordered_set**, which has amortised complexity of O(1). But what if we had access to a similar data structure with a guaranteed complexity of 'O(0)', that is one where no computation is performed during runtime at all, and calls to member functions (such as **insert** or **find**) are directly replaced, during the compilation, by the results of the call? That also means that such data structure could be used inside compile-time expressions, such as **std::conditional** or **std::enable_if**, whose sample implementations are shown in Listing 1.

```
template <bool S, typename T, typename F> struct
  conditional;
template <typename T, typename F> struct
  conditional<true, T, F> { using type = T; };
template <typename T, typename F> struct
  conditional<false, T, F> { using type = F; };

template <bool S, typename T> struct enable_if {};
template <typename T> struct enable_if<true, T> {
  using type = T; };
```

Listing 1

Of course, there must be a drawback to such a data structure: since we expect the lookup operation to work during compilation, it follows that such a set must also be fully populated during compilation. That might not seem very useful in a small program; however, design concerns in large programs often drive to more decoupling, where such ability might be useful. For a trivial example, see the **if constexpr** expression in the body of the **Foo** constructor in Listing 2.

The sample code in Listing 2 demonstrates another important point about compile-time data structures: the 'value' looked for is a tag type (named **Bar** in this example). Such types provide us with symbolic, unique names without the need for a central repository (e.g. an **enum** type), hence avoiding accidental coupling inside such a repository, while at the same time minimising the risk of clashes if there is no such central location (e.g. from **extern const int** or C-style **#define**). Examples of tag types in the C++ standard include **std::nothrow_t** or types defined for various overloads of **std::unique_lock** constructor.

```
struct Bar {};
struct Foo {
  int i = 0;
  template <typename Set> constexpr explicit
  Foo(Set) {
    if constexpr ((bool)Set::template test<Bar>)
      i += 1;
  }
};
```

Listing 2

About O(0)

There is no such thing as 'O(0)', hence quotes. That is because in big-O notation, we disregard the constant component, and the difference between O(1) and 'O(0)' is only the constant component (equal to C, for some unspecified fixed value C). In practice, the notion of big-O applies only to runtime complexity, and, as explained on the left, we are seeking to remove the runtime cost entirely. Hence our choice is to either disregard the big-O notation as not applicable or use a dummy 'O(0)'.

What would be the interface of such a 'set of types' utility? One member has already been presented above; it is the **test** template variable (NB, **(bool)** cast is a workaround for a bug in MSVC 15.7, to be fixed in version 15.8 and superfluous for other modern compilers). We also want the ability to populate the set and compare it to other sets. But how can we populate a compile-time construct? It is immutable, after all. The solution is to apply the principles of functional programming – in the compile time. That is, we can create a new container instance (or, in our case, a new container type) as a copy of the existing container with the addition of a new element. That sounds expensive, but let's not forget that the set must be filled during the compilation, which means that the runtime complexity of such an operation is still going to be 'O(0)', at the cost of compilation time. In other words, to populate our set we are going to use a pure function, to be entirely evaluated during the compilation, which returns either a type or an immutable value: a meta-function. As we will see, quite often meta-functions are not functions at all – they can be immutable template values or template types. A simplistic example is presented in Listing 3.

The implementation of **test** presented in Listing 3 is indeed simplistic (we are going to improve it later). It is encapsulated as an implementation detail **struct contains** (in the **set_impl** namespace), which is a variadic template just like **set**, but dedicated to the calculation of the value **contains**, with the help of standard type trait **std::is_same_v** and two template specialisations, acting the role of a recursive meta-function. Specialisation **contains<T, L...>** performs equality check between **U** and **T**, while **contains<>** returns the terminating condition 'not found'. Note how we used immutable template value **template <typename U> constexpr static bool value = ...** to pass the result of the nested meta-function to its caller. Such recursive calls are a common pattern in functional programming because they work well with immutable data. For the same reason, they work well with meta-functions, and we will employ recursion often. However, in meta-programming, we

Bronek Kozicki developed his lifelong programming habit at the age of 13, teaching himself Z80 assembler to create special effects on his dad's ZX Spectrum. BASIC, Pascal, Lisp and a few other languages followed, before he settled on C++ as his career choice. In 2005, he moved from Poland to London, and promptly joined the BSI C++ panel with a secret agenda: to make C++ more like Lisp, but with more angle brackets. Contact him at brok@incorrekt.com

```
namespace set_impl {
  template <typename ... L> struct contains;
  template <> struct contains<> {
    template <typename>
    constexpr static bool value = false;
  };
  template <typename T, typename ... L> struct
    contains<T, L...> {
    template <typename U>
    constexpr static bool value =
    std::is_same_v<T, U> ||
      contains<L...>::template value<U>;
  };
}

template <typename ... L>
struct set {
  constexpr explicit set() = default;
  template <typename T>
  constexpr static bool test =
    set_impl::contains<L...>::template value<T>;
  template <typename T>
  using insert = set<T, L...>;
};

int main() {
  struct Fuz; struct Baz;
  constexpr static auto s1 =
    set<>::insert<Baz>::insert<Fuz>{};
  constexpr static auto s2 =
    decltype(s1)::insert<Bar>{};
  constexpr static Foo foo2{ s2 };
  std::cout << foo2.i << std::endl;
}
```

Listing 3

have always to remember to provide a terminating specialisation – which is an equivalent of a recursion terminating execution branch in functional programming.

The implementation of the `insert` meta-function is trivial, but sadly also incorrect, as we will see below. That is because of two outstanding, and so far ignored, properties of most sets, which are:

- Uniqueness of the elements.
- Constraints on the type of elements.

It is the presence of both which gives as a 'set of something'. In `std::set`, the first guarantee is provided by quietly ignoring duplicate inserts, while the template parameter captures the later one. For our 'set of types', we are going to follow the lead of `std::set` and ignore duplicates. For the later guarantee we can, for the time being, apply some hardcoded set of constraints. For example:

- Disallow qualifiers (const or volatile)
- Disallow reference types (lvalue or rvalue)
- Disallow pointer types

The proposed constraints do not impact the uses of our set where tag types are passed directly, for example, hardcoded. If they are deduced, or coded in a remote location, they only require additional use of `std::decay_t` to remove qualifiers or reference. The constraint to disallow pointers is meant to protect against user errors – either a typo or deduced type which unexpectedly turns out to be a pointer. Additionally, since our elements are types, we need to consider how to handle `void`. One useful solution is to handle it as an element of an empty set – that is, a non-element. In other words, we are going to use `void` as a placeholder for an element, where no element is available (the similarity to the role of `void` in C and C++ becomes obvious when 'element' is replaced by 'type').

Back to our code example, one can easily spot that the `insert` meta-function is not doing anything to enforce uniqueness of added types. Similarly, nothing is preventing the user from instantiating, e.g. `set<int, int>`. These two problems aside, yet another limitation of our set is that `set<int, long>` and `set<long, int>` are both distinct types but they are one set (because the order of elements does not matter to mathematical sets). There is probably no robust solution for the last problem, but a workable compromise would be a `unique` member type inside `set` to hold unique types, and creation of an `is_same` member meta-function to test set equality with no regard to order. That might look like Listing 4.

Storing actual values

What if we wanted to store actual values, known at compilation time? There are a few options:

- Implement a set of values separately, following the ideas presented here
- Store values wrapped into instances of `std::integral_constant`
- Assign unique, immutable values to tag types

```
template <typename ... L> class set;
namespace set_impl {
  template <typename T> struct check {
    static_assert(std::is_same_v<T,
      std::decay_t<T>>);
    static_assert(not std::is_pointer_v<T>);
    using type = T;
   };

  template <typename ...L> struct contains_detail;
  template <typename U> struct
    contains_detail<U> {
    using type = typename check<U>::type;
    constexpr static bool value =
      std::is_void_v<type>;
    };
    template <typename U, typename T,
      typename ... L> struct
      contains_detail<U, T, L...> {
      using type = typename check<U>::type;
      using head = typename check<T>::type;
      constexpr static bool value =
        std::is_void_v<type>
        || std::is_same_v<type, head>
        || contains_detail<U, L...>::value;
  };

  template <typename ... L> struct insert_detail;
  template <typename T, typename ... L> struct
    insert_detail<T, set<L...>> {
    using head = typename check<T>::type;
    using type = set<head, L...>;
  };

  template <typename ... L> struct
    unchecked_list {};
  template <typename T> struct cracker_list;
  template <typename ... L> struct
    cracker_list<set<L...>> {
    using type = unchecked_list<L...>;
  };
  template <typename T> struct cracker;
  template <typename ... L> struct
    cracker<set<L...>> {
    using list = typename cracker_list<
      typename set<L...>::unique>::type;
    constexpr static size_t size =
      set<L...>::unique::size;
};
```

Listing 4

```
  template <typename ... L> struct
    intersect_detail;
  template <typename ... T> struct
    intersect_detail<unchecked_list<>,
      unchecked_list<T...>> {
    constexpr static size_t size = 0;
  };
  template <typename V, typename ... L,
    typename ... T> struct
    intersect_detail<unchecked_list<V,L...>,
      unchecked_list<T...>> {
    constexpr static size_t size =
      intersect_detail<unchecked_list<L...>,
        unchecked_list<T...>>::size
        + (not std::is_void_v<V>
          && contains_detail<V, T...>::value);
  };
  template <typename ... L> struct unique;
  template <> struct unique<> {
    using type = set<>;
    constexpr static size_t size = 0;
    template <typename , size_t Size>
    constexpr static bool is_same = Size == 0;
    template <typename U>
    constexpr static bool test =
      contains_detail<U>::value;
  };
  template <typename T, typename ... L> struct
    unique<T, L...> {
    using type = typename std::conditional<
      contains_detail<T, L...>::value
      , typename unique<L...>::type
      , typename insert_detail<T,
          typename unique<L...>::type
        >::type
      >::type;
    constexpr static size_t size =
      contains_detail<T, L...>::value
      ? unique<L...>::size
      : unique<L...>::size + 1;
    template <typename U, size_t Size>
    constexpr static bool is_same =
      size == Size
      && size == intersect_detail<
        unchecked_list<T,L...>, U>::size;
      template <typename U>
      constexpr static bool test =
        contains_detail<U, T, L...>::value;
  };
}
```

Listing 4 (cont'd)

```
template <typename ... L>
class set {
  using impl = set_impl::unique<L...>;
public:
  constexpr explicit set() = default;
  using unique = typename impl::type;
  constexpr static size_t size = impl::size;
  constexpr static bool empty = size == 0;
  template <typename T>
  constexpr static bool is_same =
    impl::template is_same<typename set_impl
        ::cracker<T>::list, set_impl::cracker<T>
        ::size>;
  template <typename T>
  constexpr static bool test =
    impl::template test<T>;
  template <typename T>
  using insert =
    typename set_impl::unique<T, L...>::type;
};
```

Listing 4 (cont'd)

The code may appear complex, but that is only because of the amount of typing necessary in C++ to code each simple meta-function. In fact, it is quite simple, although there are few things to note. Again we are delegating the work required to individual meta-functions, and we also have a **`unique`** type to build the unique set of types (this ensures that e.g. **`set<int, int>`** will be recognised as having only one element since repeated elements do not count). Inside this **`unique`** type we delegate tasks to individual meta-functions **`insert_details`**, **`contains_details`** and **`intersect_details`**. The last one calculates the size of the intersection of sets, which is required to test set equality using a simple formula: sets are equal if they have equal size and their intersection is equal to that size as well. We could reuse **`instersect_details`** to implement two more useful checks **`is_cross`** to check whether two sets share a non-empty intersection and **`is_super`** to check whether one set is a subset of another (in reverse, since our 'superset' will be on the left side and 'subset' on the right). In the first case, we only need to know that the size of the set intersection is greater than 0. In the latter, the size of the intersection will be equal to the size of a subset. The opportunity for reuse of **`intersect_details`** is obvious, and to do so we are going to introduce a higher order meta-function **`compare`**, to replace **`unique::is_same`**. In functional programming, a higher order function is a function which consumes (or produces, or both) a function. In our case, a meta-function is a template (whereas types take the role of values) which means that higher-order meta-function **`compare`** is going to consume a template template parameter (yes, that is the word 'template' twice in a row) which encapsulates the meta-function to perform the size comparison required. See Listing 5.

```
namespace set_impl {
  template <size_t Mine, size_t Cross,
    size_t Theirs> struct is_cross {
    constexpr static bool value = Cross > 0;
  };
  template <size_t Mine, size_t Cross,
    size_t Theirs> struct is_super {
    constexpr static bool value =
      Cross == Theirs;
  };
  template <size_t Mine, size_t Cross,
    size_t Theirs> struct is_same {
    constexpr static bool value =
      Mine == Cross && Cross == Theirs;
  };
  template <typename ... L> struct unique;
  template <> struct unique<> {
    // ...
    template <template <size_t, size_t, size_t>
      typename How, typename U, size_t Size>
    constexpr static bool compare =
      How<0, 0, Size>::value;
  };
  template <typename T, typename ... L>
    struct unique<T, L...> {
    // ...
    template <template <size_t, size_t, size_t>
      typename How, typename U, size_t Size>
    constexpr static bool compare =
      How<size,
      intersect_detail<unchecked_list<T, L...>,
      U>::size, Size>::value;
  };
}
```

Listing 5

```
template <typename ... L>
class set {
  using impl = set_impl::unique<L...>;
public:
  // ...
  template <typename T>
  constexpr static bool is_same =
    impl::template compare<set_impl::is_same,
    typename set_impl::cracker<T>::list,
    set_impl::cracker<T>::size>;
  template <typename T>
  constexpr static bool is_cross =
    impl::template compare<set_impl::is_cross,
    typename set_impl::cracker<T>::list,
    set_impl::cracker<T>::size>;
  template <typename T>
  constexpr static bool is_super =
    impl::template compare<set_impl::is_super,
    typename set_impl::cracker<T>::list,
    set_impl::cracker<T>::size>;
};
```

Listing 5 (cont'd)

Right at the end, it is time to revisit the set of constraints imposed on types stored in our set of types. As we have just seen, a template template parameter can be used to implement a higher order (meta-) function, which is an ideal way for the user to select their own set of constraints, and perhaps even type transformations. The final program (Listing 6) makes use of this ability.

```
#include <cstdio>
#include <utility>
#include <type_traits>

template <template <typename> typename Check,
typename ... L> class set;

namespace set_impl {
  template <template <typename> typename Check,
    typename ... L> struct contains_detail;
  template <template <typename> typename Check,
    typename U> struct contains_detail<Check, U> {
    using type = typename Check<U>::type;
    constexpr static bool value =
      std::is_void_v<type>;
  };
  template <template <typename> typename Check,
    typename U, typename T, typename ... L>
    struct contains_detail<Check, U, T, L...> {
    using type = typename Check<U>::type;
    using head = typename Check<T>::type;
    constexpr static bool value =
      std::is_void_v<type>
      || std::is_same_v<type, head>
      || contains_detail<Check, U, L...>::value;
    };
  template <template <typename> typename Check,
    typename ... L> struct insert_detail;
  template <template <typename> typename Check,
    typename T, typename ... L>
    struct insert_detail<Check, T, set<Check,
    L...>> {
    using head = typename Check<T>::type;
    using type = set<Check, head, L...>;
  };
  template <typename ... L> struct unchecked_list
  {};
```

Listing 6

```
  template <typename T> struct cracker_list;
  template <template <typename> typename Check,
    typename ... L> struct cracker_list<set<Check,
    L...>> {
    using type = unchecked_list<L...>;
  };
  template <typename T> struct cracker;
  template <template <typename> typename Check,
    typename ... L> struct cracker<set<Check,
    L...>> {
    using list = typename
      cracker_list<typename set<Check,
      L...>::unique>::type;
    constexpr static size_t size = set<Check,
      L...>::unique::size;
  };
  template <typename T> struct dummy_check {
    using type = T;
  };
  template <typename ... L> struct
    intersect_detail;
  template <typename ... T> struct
    intersect_detail<unchecked_list<>,
    unchecked_list<T...>> {
    constexpr static size_t size = 0;
  };
  template <typename V, typename ... L,
    typename ... T> struct
    intersect_detail<unchecked_list<V, L...>,
    unchecked_list<T...>> {
    constexpr static size_t size =
      intersect_detail<unchecked_list<L...>,
      unchecked_list<T...>>::size +
      (not std::is_void_v<V>
        && contains_detail<dummy_check, V,
        T...>::value);
  };
  template <size_t Mine, size_t Cross,
    size_t Theirs> struct is_cross {
    constexpr static bool value = Cross > 0;
  };
  template <size_t Mine, size_t Cross,
    size_t Theirs> struct is_super {
    constexpr static bool value =
      Cross == Theirs;
  };
  template <size_t Mine, size_t Cross,
    size_t Theirs> struct is_same {
    constexpr static bool value = Mine == Cross
      && Cross == Theirs;
  };
  template <template <typename> typename Check,
    typename ... L> struct unique;

  template <template <typename> typename Check>
    struct unique<Check> {
    using type = set<Check>;
    constexpr static size_t size = 0;

    template <template <size_t, size_t, size_t>
      typename How, typename U, size_t Size>
    constexpr static bool compare =
      How<0, 0, Size>::value;

    template <typename U>
    constexpr static bool test =
      contains_detail<Check, U>::value;
  };
```

Listing 6 (cont'd)

```
  template <template <typename> typename Check,
    typename T, typename ... L>
    struct unique<Check, T, L...> {
    using type = typename std::conditional<
      contains_detail<Check, T, L...>::value
      , typename unique<Check, L...>::type
      , typename insert_detail<Check, T,
        typename unique<Check, L...>::type>::type
    >::type;
    constexpr static size_t size =
      contains_detail<Check, T, L...>::value
      ? unique<Check, L...>::size
      : unique<Check, L...>::size + 1;
    template <template <size_t, size_t, size_t>
      typename How, typename U, size_t Size>
    constexpr static bool compare =
      How<size, intersect_detail
      <unchecked_list<T, L...>, U>::size,
      Size>::value;
    template <typename U>
    constexpr static bool test =
      contains_detail<Check, U, T, L...>::value;
  };
}

template <template <typename> typename Check,
   typename ... L>
class set {
  using impl = set_impl::unique<Check, L...>;
public:
  constexpr explicit set() = default;
  using unique = typename impl::type;

  constexpr static size_t size = impl::size;
  constexpr static bool empty = size == 0;

  template <typename T>
  constexpr static bool is_same =
    impl::template compare<set_impl::is_same,
    typename set_impl::cracker<T>::list,
    set_impl::cracker<T>::size>;

  template <typename T>
  constexpr static bool is_cross =
    impl::template compare<set_impl::is_cross,
    typename set_impl::cracker<T>::list,
    set_impl::cracker<T>::size>;

  template <typename T>
  constexpr static bool is_super =
    impl::template compare<set_impl::is_super,
    typename set_impl::cracker<T>::list,
    set_impl::cracker<T>::size>;

  template <typename T>
  constexpr static bool test =
    impl::template test<T>;

  template <typename T>
  using type =
    typename std::enable_if<(not std::is_void_v<T>
    && test<T>), typename Check<T>::type>::type;

  template <typename T>
  using insert = typename
    set_impl::unique<Check, T, L...>::type;
};
```

Listing 6 (cont'd)

```
struct Baz { Baz() = delete; };
struct Bar { Bar() = delete; };
struct Fuz { Fuz() = delete; };

struct Foo {
  int i = 0;

  template <typename Set> constexpr explicit
  Foo(Set) {
    if constexpr ((bool)Set::template test<Bar>)
      i += 1;
  }
};

template <typename T> struct PlainTypes {
  static_assert(std::is_same_v<T,
std::decay_t<T>>);
  static_assert(not std::is_pointer_v<T>);
  using type = T;
};

int main() {
  constexpr static set<PlainTypes> s0{};
  constexpr static auto s1 =
    decltype(s0)::insert<Baz>::insert<Fuz>{};
  constexpr static Foo foo1{s1};
  std::printf("foo1.i=%d\n", foo1.i);

  constexpr static auto s2 =
    decltype(s1)::insert<Bar>{};
  constexpr static Foo foo2{s2};
  std::printf("foo2.i=%d\n", foo2.i);
}
```

Listing 6 (cont'd)

Building the programs

Presented programs should build without warnings under C++17 compliant compilers. The following have been tested:

- GCC 8.1
- Clang 6.0
- Visual Studio 2017 (compiler version 15.7)

A sample CMake file which works for all these compilers is presented below:

```
cmake_minimum_required(VERSION 3.8)
project(metaset)
set(CMAKE_CXX_STANDARD 17)
if(MSVC)
  set(CMAKE_CXX_FLAGS
   "${CMAKE_CXX_FLAGS} /permissive-")
  set(CMAKE_CXX_FLAGS_DEBUG
   "${CMAKE_CXX_FLAGS_DEBUG} /D_DEBUG")
else()
  set(CMAKE_CXX_FLAGS
   "${CMAKE_CXX_FLAGS} -Wall -Wextra -Wpedantic")
  set(CMAKE_CXX_FLAGS_RELEASE "-O3 -DNDEBUG")
  set(CMAKE_CXX_FLAGS_DEBUG "-O0 -g")
endif()
set(SOURCE_FILES main.cpp)
add_executable(${PROJECT_NAME} ${SOURCE_FILES})
```

Afterwood

Much ado about nothing. Chris Oldwood considers what we have when we have nothing.

There is a classic puzzle that goes:

> The poor have it, the rich need it, and if you eat it you'll die. What is it?

If you haven't come across this before and Google is out of reach because you're reading the printed edition going through a tunnel or an internet blackspot, the answer is 'nothing'. I think it would be fairly easy to come up with a programming specific version of this particular puzzle as there appears to be quite a few variants of 'nothing' in our world, many of which seem to occupy a significant amount of our time due to their cunning camouflaged outfits.

It probably seems strange to us now but once upon a time there was no zero. Essentially you had something which was countable or you had nothing. There were no negative numbers either and nothing preceded one (no pun intended, for once). I recently read *The Nothing that Is: A Natural History of Zero* and it got me thinking about the ways we represent nothingness in programming and the problems it causes.

Zero is almost certainly our 'go to choice' for representing a lack of something as it's been with us since our early days of schooling and support for integer values has also been around in programming pretty much since the beginning too whether you're using assembly or a high-level language. Even most children could tell you `numApples = 0` means you don't have any apples.

Where it starts to go astray is when we have to squeeze the concept of 'none' into a domain that doesn't really support it because you are no longer representing countable things. A classic example here is representing dates where day zero represents some epoch like 1st January 1970 or 1st January 1900 and negative values stretch backwards in time. Here every value in the domain represents a valid value and so if we want something to accommodate the notion of 'no value' we probably have to re-purpose one or other end of the spectrum, e.g. `INT_MIN` or `INT_MAX`.

What about if we're searching an array for some value or object and there is nothing to be found? If our array starts at index 1 (e.g. Visual Basic) we could use the value 0, but many languages have adopted the 0-based approach and Stan Kelly-Bootle's suggestion of using 0.5 has never really received any uptake. For languages like Java and C# that are inherently based on signed integers they can return any negative value for the 'none found' answer. In C++ where unsigned integers are the preferred choice we have no such luck and instead have concocted a magical value by the name of `npos` for strings which (implementation-wise) sits within the valid range of values but on the precipice such that you'd probably run out of memory long before it could ever be a valid result.

Sadly the use of -1 (in either of its signed or unsigned guises) as both a perfectly good response to a question and as a way of signalling an error has only succeeded in muddying the waters further. The Windows API for example uses the constants `LB_ERR` (and `CB_ERR`) in this way which means you often stumble across code that initialises variables with it, e.g. `index = LB_ERR`, because it allows us to exploit a duality of semantics ('no value, yet' and 'not found') and write less code, irrespective of whether it makes comprehending it any easier. (You might argue not finding it is an error; either way you still have the same type representing two different domains – index and error.)

With enumerated types we often walk right into the same trap with our eyes wide open thinking we're being clever by adding an explicit value called `None` or `Default` (usually with a value of 0 in languages that zero-initialise values and references for 'safety's sake').

Of course when you're forced to abuse the type system it will get its own back. By masquerading two different result values within the normal course of events you will trick the caller into believing it's safe to simply compose functions when in reality they're just storing up a world of pain in the form of an `IndexOutOfBounds` exception, access violation or, if really unlucky, undefined behaviour and subsequent data corruption.

The `NullPointerException`, or 'NPE' as it is commonly referred to in the Java world, is a blight on modern programming caused in part by our use of programming languages which embrace the use of reference types over value types meaning that all of our objects can potentially exist or not exist. Unless the use is entirely local it can be difficult to reason about any object's existence and therefore null checks can easily dominate a codebase in an act of overly defensive programming. The introduction of the `?.` operator in some languages might reduce the noise but it's just a case of treating the symptoms, not the disease.

This particular foe likes to disguise themselves by changing their name too, but whether they be `null`, `NULL`, `nullptr`, `nil`, `0`, `end`, `-1`, `npos`, `""`, `NaN`, `None`, etc. we should be on our guard and be ready to banish them to computing history or at the very least quarantine them.

But what can be done, surely we can't undo the past? Well, maybe we can. Over the years the awareness of the Optional (Option, Maybe, etc.) type has grown so that it's no longer just a niche technique used by Comp Sci purists. The desire to right this wrong is so strong in some circles that there is currently a preview of C# [Github] where reference types have been given the `Nullable` makeover thereby allowing us to finally consider deleting our own homebrew variants and deprecating our static analysis annotations in favour of a kosher type annotation. Surprised?

One of Shakespeare's most famous comedies is titled *Much Ado About Nothing*. Given the amount of time we've lost over the years debugging issues caused by our inability to express 'nothing' in a way that is obvious to our fellow programmers I'd say it was no laughing matter. We need to realise that failure can indeed be an option and that the type system should be there to help us, nothing more nothing less. ■

Reference

[Github] https://github.com/dotnet/csharplang/wiki/Nullable-Reference-Types-Preview

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

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Design: Pete Goodliffe

GET MOORE

£634.99

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

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

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner

To find out more about Intel products please contact us:

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