overload 155

FEBRURARY 2020 £4.50

The Path of Least Resistance

Python imports can be overwhelming. We demystify the process with a walk through.

A Secure Environment for Running Apps?

We look into “app confinement’ as an OS-level mechanism to improve security

Quick Modular Calculations

We continue to investigate performant techniques for maths operations

A Line-to-Line Conversion from Node.js to Node.cpp

How a relatively simple conversion can improve software performance

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 155

February 2020

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 156 should be submitted by 1st March 2020 and those for Overload 157 by 1st May 2020.

The ACCU

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

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

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

4 A line-to-line conversion from Node.js to Node.cpp
Dmytro Ivanchykhin, Sergey Ignatchenko and MaximBlashchuk show how we can get a 5x improvement in speed.

8 The Path of Least Resistance
Steve Love helps us with a walk through Python imports.

14 Quick Modular Calculations (Part 2)
Cassio Neri presents alternatives to the minverse algorithm.

18 A Secure Environment For Running Apps?
Alan Griffiths considers the security aspects of apps from the app store.

20 Afterwood
Chris Oldwood shows us that 'the centre half' is more than a sporting term.

Members and Remembrances

Remembering things can be difficult. Frances Buontempo reminisces on routes to erudition.

I had a busy Christmas doing almost nothing, which was nice, but meant I forgot to write an editorial. We played backgammon, which I hadn't played for ages now, and moved on to charades. The randomness of backgammon means you can't learn a sequence of moves off by heart to try to win, but some sensible moves given, say, the initial setup and a certain dice roll do stick in the mind. Certain pairs of numbers together with the distance apart of pieces gives ways to keep things safe, or small risks worth taking, Playing charades is another matter, but again certain visual clues are memorable. Making a 'T' shape with your hands to indicate the word 'the', for example. As the game progresses, once a word has been guessed from a silly action that becomes a new way to express a word. The sillier, the better. Or maybe that's just me.

Certain properties, such as silliness, make ideas and actions more memorable. Children are taught songs that have repetition, rhymes and rhythm. The structure and sound means a novice can mimic or guess and often get things right. People can manage to 'jam' together, extemporizing and ad-libbing music or other types of performance, without sheet music or a script by knowing the rules of the game. Playing in a given key means certain notes will work together. Having a time signature indicates when to play a note. To the uninitiated, it probably looks like magic. To those on the inside, the magic is more a combination of experience and known rules or guidelines. Some jams start with a known tune, but then go on the change and bend the sequence creating something new.

All kinds of activities follow this kind of pattern, along the lines of the Shu Ha Ri idea from martial arts. Meaning 'embracing the kata' it describes three steps in learning:

- Shu: Student follows instructions precisely.
- Ha: Student starts to learn principles and theory (i.e. can break the rules).
- Ri: student becomes master and 'is' the rules, creating new rules.

Martin Fowler wrote a brief blog about this [Fowler14], comparing it to the Dreyfus model of learning, though Alistair Cockburn originally introduced it as a way of approaching the learning of software development. Starting with known patterns and rules, once you can remember them, you can try out new things. This helps you understand what and why. Then you can try completely new things.

Do you know the rules? Perhaps software development doesn't really have rules. Sometimes people have memorised algorithms, perhaps in order to succeed at interviews, or maybe just for fun. This may start by following some pseudocode in a book or on a website, or even copying and pasting from a website. This will not be sufficient to learn the technique properly. More practice is required. It can be illuminating to find several different ways of coding up the same thing, or trying it in various languages. As you go deeper, you may start to understand some underlying principles. Sometimes you can build up an instinct, and spot what might be making code slow, but an expert knows to always measure as well. Sometimes your instinct is wrong, nonetheless being at a point where you do have instincts, even if it's which websites or books to turn to for help, indicates a level of knowledge. If you can then explain what you are noticing to someone else, you are building expertise. If they can follow what you said, then you are doing even better.

Sometimes, you might be trying to learn on your own. If you try to learn a new language or to drive a new framework, where do you start? I find myself constantly looking things up, until I bore myself and get determined to remember the basics. Sometimes practising a few small functions over and over until I can use something like muscle memory to let the code happen helps. Some people prefer books, others videos or classes, either in person or on line. In some ways the medium is immaterial. The process will be similar. Try, try, try again. And then either succeed or give up.

Do you remember how you learnt to code? Or how to find your way around an unfamiliar code base? On several occasions when I've started a new contract, team mates will sit with me, often in a meeting room, and sketch things out on a white board. Or sit me by their desk, pressing the 'go to definition' key in their IDE. Even when I desperately try to make notes, I never feel like I know my own way round until I have explored by myself, or try to actually add new features. Certainly, a short introduction can help you find your way more quickly. We went on holiday to Morocco a while ago. It's very easy to get lost in the medina, but there's a Mosque with a large tower on the edge of the market square. The simple instruction, "Look for the tower" helps us not get completely lost. If you visit a new town, and someone shows you around, you will not remember exactly which routes you took, even if you take extensive notes. When you next venture out by yourself, you will almost certainly have unsure moments of which way to turn. If you at least spot one landmark to get you back to the car park, bus station or other significant place, you have more chance of finding your own way round. You can't remember every detail, but having one or two important things in mind makes so much difference. Introducing a new starter to your setup is difficult, don't get me wrong. You need to find a balance between pointing out an important signpost and allowing your new colleague to explore. Talking at someone for several hours, and having to look things up yourself as you go isn't the best approach. Let the new person look things up. Pair with them for a bit. Ask what they need to 'hit the ground running', as the phrase goes.

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.

Can you remember your best onboarding experiences? What about your best learning experiences? I could tell war stories about some lecturers at University, or some teachers at school. I'm sure we all could. When things didn't go well, those who were keen to learn would often form small self-help groups. Students can often figure stuff out between themselves, despite the teacher. I'm allowed to be negative about teachers because I used to be one. It's not an easy job. I have a lot of respect for people who manage to teach well, and am saddened when I hear of the hard time some lecturers are having in various institutions, trying to find time to help their students. This is straying dangerously close to a political opinion piece now, which would count as an editorial. I have avoided this so far, so must change subject immediately.

People have ways of remembering salient points, and so do machines. In order to remember state between function calls, object oriented programming stores values in member variables for future use in member function calls. Without something like the single responsibility principle, the class can end up more like a school with hundreds of members chucked in to save having to think about how to share state between various objects. An alternative would be lots of global variables. A more functional style might chain calls together, never having state, *per se*, but rather results to operate on. There are also various approaches to asynchronous code. C++20's coroutines can stop for a bit and resume later. They need to 'remember' what they were doing when they kick off again. There are various approaches to this, however 'Coroutines are stackless: they suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack.' [cppreference]. Perhaps you are a full heap developer, rather than a full stack developer, if you are using C++. See what I did there?

What else involves remembering or learning? The first thing that springs to mind, for me, is machine learning. Almost all machine learning algorithms start with a random setup, often several random numbers, and iterate until a criterion is fulfilled, nudging the numbers up or down a bit, based on feedback, in the form of a 'fitness' or scoring function. When complete, they have a new set of numbers. Describing this as learning seems a bit odd at first glance. When we learn, we tend to feel like we can then remember something new afterwards. Does a machine 'remember' anything? It might cache or serialise the new numbers. Perhaps this is like us holding something in memory for a while, or resorting to taking notes. The machine won't be able to explain its new knowledge to anyone though. Don't get me wrong, I love machine learning, but words like 'learn', 'train', and 'intelligence' need some consideration when used in a new context.

What about other new things? How do you get a talk at a conference accepted? Or an article published, or even write a book? Well, if you don't try, it won't happen. It can be helpful to talk to people who have done the thing you want to do. When I consider which of my talks have been accepted and which haven't been, I'm not certain of the main differences. Though proposals with a clearer main topic did better, but there's more to it than that. Getting an article accepted is another matter. It varies between publications. Some more academic journals can take months, if not years, to finally accept or reject an idea. *Overload* has a much quicker turn round, and we'll help you improve an initial idea if required. As for getting a book published, asking for published authors to read your pitch will help. They have been through it before. When I first joined ACCU, I was blown away by people who talked at conferences, wrote articles and had even had books published. I got used to this, making it seem less unachievable. If you are an ACCU member, well done. Use this opportunity to network and join in. If you're not, you're missing out. Just saying.

Another angle of getting published or a talk accepted, is considering how your idea will be assessed. If you've never read someone else's proposal, you're missing out. Many organisations will give you a chance to be on a review board or committee. Several publishing companies let you join their review team, and the ACCU members' magazine, *CVu*, runs a book review section. Review other people's work: you may not get paid for doing so, but will gain valuable experience. You'll start to spot ways of saying things, how to make yourself clearer, and how to develop an idea. Insider knowledge is invaluable. Join ACCU, and take part in the Code Critique Competition. You might win a book token, or at least see your name in print. Write for *Overload*. Volunteer for the review team. Or volunteer to be a guest editor. I'm serious; if you would like a go, then get in touch. I'll help you through the process, from encouraging people to write, to coordinating with the review team and getting the magazine to the production editor. Learn what happens behind the scenes. It's not magic, but is rewarding.

I joined ACCU many years ago. I've met so many people, some I only know by name but others I have met in person at meetups or the conference. Inevitably, people come and go. Last year, a long standing member, Hubert Matthews, died. He was our chairperson for a time, and I went to several of his conferences talks. Many of us paid tribute via the accu-general email list. It's always a shock when someone dies, but having fond and positive memories is wonderful, even when initially tinged with sadness.

I've spent a long time talking about how to learn new things, as well as remembering what's gone before. Whether remembering people or experiences, taking a moment to think back and reflect is useful. Maybe history and tradition give us the Shu – some kind of patterns or 'instructions'. However, to learn from past mistakes and glories, we should avoid cargo cult stuff, mindlessly following a recipe. Don't do something because that's the 'right way', 'best practice', or 'most appropriate', or it worked for someone else. Bend the rules once or twice. Move on to Ha and then Ri. Join ACCU, get involved and lay down good memories and make friends. Bring on the New Year.

References

[cppreference] Coroutines (C++20) at https://en.cppreference.com/w/cpp/language/coroutines

[Fowler14] Martin Fowler, ShuHaRi, published 22 August 2014, available at https://martinfowler.com/bliki/ShuHaRi.html

A line-to-line conversion from Node.js to Node.cpp

Dmytro Ivanchykhin, Sergey Ignatchenko and Maxim Blashchuk show how we can get a 5x improvement in speed.

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 translators and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

It is pretty well known that, computation-wise, C++ code is substantially faster than Node.js (even with all the efforts spent on the v8 engine); for details we can refer to [Debian], where for different calculation-oriented algorithms the results were in favour of C++, with differences in wall-clock time ranging from 1.2x to 6x.

However, most of the benchmarks out there (including [Debian]) are concentrating on pure computations. This means that differences in the ability of different frameworks to handle requests (which forms a basis for handling real-world interactive loads) are not addressed. This article aims to start covering this gap.

Node.cpp

At this point, we want to compare good old Node.js with our own new kid on the block, which we named Node.cpp.

The idea behind Node.cpp is to make a framework which will allow us to write C++ code in Node.js style while benefiting from C++ goodies (including significantly improved performance). Moreover,

> *It should be possible to take existing Node.js code and convert it into Node.cpp with line-to-line correspondence between the two.*[1]

As of now, Node.cpp is still very much in its infancy, but we have already managed to write enough code to run some benchmark tests Let's take a look at the code of the http 'echo' server which is along the lines of the sample from [Ostinelli11] (see Listing 1).

As we can see, there is a direct correspondence between Node.JS code and Node.cpp code; sure, there are quirks related to the nature of C++ (and some more due to still-missing APIs in Node.cpp – which will be fixed before release), but overall the whole thing looks similar enough

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

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

Maxim Blashchuk Maxim Blashchuk has substantial development experience, most of it with embedded programming. Recently he joined a team performing research on low-level C++ libraries providing properties such as determinism and memory safety.

```
//Node.js
http.createServer(function(request, response) {
  if ( request.method == "GET"
       || request.method == "HEAD" ) {
    response.writeHead(200,
      {"Content-Type":"text/xml"});
    var urlObj = url.parse(request.url, true);
    var value = urlObj.query["value"];
    if (value == ''){
      response.end("no value specified");
    } else {
      response.end("" + value + "");
    }
  } else {
    response.writeHead(405,
      "Method Not Allowed");
    response.end();
  }
}).listen(2000);

//Node.cpp
srv = net::createHttpServer<ServerType>(
   [](net::IncomingHttpMessageAtServer& request,
      net::HttpServerResponse& response){
      if ( request.getMethod() == "GET"
           || request.getMethod() == "HEAD" ) {
        response.writeHead(200,
          {{"Content-Type", "text/xml"}});
        auto queryValues =
          Url::parseUrlQueryString(
          request.getUrl() );
        auto& value = queryValues["value"];
        if (value.toStr() == ""){
   response.end("no value specified");
 } else {
   response.end( value.toStr() );
 }
      } else {
        response.writeHead( 405,
          "Method Not Allowed");
        response.end();
      }
  });
srv->listen(2000, "0.0.0.0", 5000);
```

Listing 1

> *to enable manual but more or less mechanistic rewriting from Node.js into Node.cpp*

1 At this point, we're talking about a manual rewrite; whether automated conversion will be possible, and how efficient it will be, is currently beyond the scope of this article.

differences in the ability of different frameworks to handle requests (which forms a basis for handling real-world interactive loads) are not addressed

Sure, after rewriting into Node.cpp the code will be still a bit more verbose, but what is important is that such a rewrite should be feasible without any changes to the *essence* of the Node.js code.

What's the point?

Ok, we can see that it IS possible to convert some rudimentary Node.js code into Node.cpp without breaking the structure of existing Node.js code. But what is the point of going through such an exercise?

In the not so distant future, we're planning to add some ultra-useful features to Node.cpp – such as deterministic recording/replay (which in turn will allow production post-mortem debugging (sic!)); however, for the time being, we'll concentrate on one advantage of Node.cpp – namely, on performance.

Test setup

When testing our Node.cpp against Node.js, our plan was to:

- Exclude testing of computations (there are too many computations out there, and they are addressed by other benchmarks such as [Debian])
- Have a test which is as close to real-world conditions as possible

To achieve this, we used **`httperf`** along the lines described in [Ostinelli11], with some changes intended (a) to reflect the real world better (most importantly, we ran our tests between two separate boxes), and (b) restricting the programs under test to use one single core (multi-core tests using *cluster* module are coming, but they're not a part of this particular article).

Hardware

Unlike [Ostinelli11], we ran our client and server on two different boxes:

- **Server**

  HP DL380eG8 (12xLFF), CPU: 2x Intel Xeon E5-2420 (6 cores, 12 threads, 15M Cache, 1.90 GHz, 7.20 GT/s Intel® QPI), RAM 32 GB, Disks 4x3TB SATA, Network card: 10GE UTP card

  Overall, our test server is pretty much a typical workhorse 2S server, which has tended to dominate data centres for at least for last 20 years.

- **Client**

  Dell R630, CPU: 2x Intel Xeon E5-2630v4, 128G RAM, Disks 2x480GB SSD

  Honestly, client hardware doesn't matter much and is mentioned here merely for the sake of completeness.

Client and server boxes were interconnected directly via a 10Gbit switch.

Software

Both client and server boxes were running stock Ubuntu 19.10 (Eoan).

On the client, we ran **`httperf`** patched to use an increased number of file descriptors as discussed in [Stackoverflow].

We used this command line for **`httperf`**:

```
httperf --timeout=5 --client=0/1
--server=10.32.36.3 --port=2000  --rate=XXX
--send-buffer=4096 --recv-buffer=16384
--num-conns=8000 --num-calls=70
```

where the (session) **`rate`** parameter went from 100 to 2000 in steps of 100.

On the server, we ran either Node.js, which is available for eoan (10.15.2), or the open-source code for [node.cpp] compiled with Clang 9. App-level code used for Node.js and Node.cpp is shown in Listing 1 above. Another test we ran (just as a sanity check and to put things into perspective) was the raw performance of the single-core **`nginx`** serving static files (which are requested by the same **`httperf`**); in a sense, **`nginx`** results represent 'The Holy Grail' of http dynamic processing, something we can *try* to reach.

Results

The results of our testing are shown in Figure 1, overleaf.

Here, along the lines of [Ostinelli11], the 'desired' response rate **`is`** calculated as *rate* × *num-calls* (as specified in the command line for httperf), and the 'real' response rate is the actual response rate as measured by **`httperf`**.

As we can see, with a lower load, all the servers behave similarly until the capacity of a particular server is reached; but after that limit is reached, however much we increase the load, the response rate doesn't really improve and stays more or less stable.

As such, we can conclude (at least within the limitations of the current test setup), that Node.js can handle a maximum of 13,000 responses/sec per core, while Node.cpp can handle around 70,000 responses/sec (that's over a 5× advantage performance-wise(!!)). Static-serving **`nginx`**, as expected, goes well above both dynamic handlers (at 100,000 responses/sec), but we have to note that (i) a 30% difference between Node.cpp and **`nginx`** is not THAT bad, and (ii) we will try to bring node.cpp MUCH closer to the performance of static-serving **`nginx`**.

Important notes about the results:

- All the results are currently for a single core only. Tests for multiple cores (using *cluster* API) are coming soon.
- Node.cpp has a 5× performance advantage over Node.js

  5× is a Damn Lot™. It is also interesting to note that other frameworks competing with Node.js (such as Erlang/Elixir and golang) *seem* to have performance in the same ballpark as Node.js, so Node.cpp can become a competitive advantage of the Node.* ecosystem (more on this below). We're planning to come to this question in our next article, after running multi-core tests adding Erlang, Elixir, and golang to the mix.
- The big fat question is where Node.js (as well as competing frameworks) manage to lose that 5× performance improvement (we DO realize that it is not Node.cpp which performs well, it is rather

Node.cpp is still in its early infancy, and is not ready (yet) for production use... If you want to help, please feel free to contribute

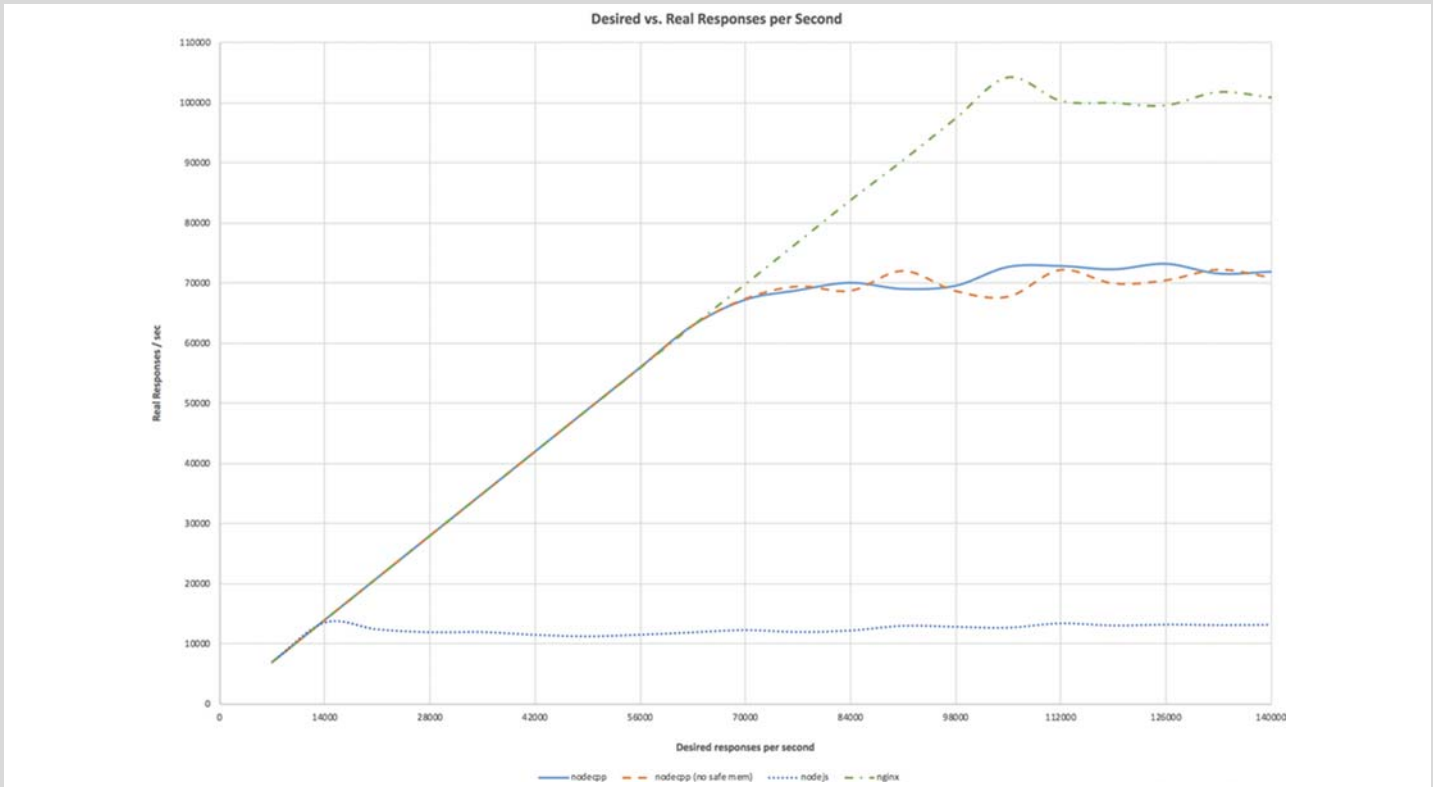


Figure 1

Node.js which performs poorly). More investigation would be needed to find out *why* it is slower.

- Node.cpp is merely 30% behind the statically-serving nginx.

  This number looks even more impressive as Node.cpp didn't even start with any optimizations of its code (it is not pessimized, but that's it); in particular, Node.cpp is currently using generic `poll()` rather than Linux-specific `epoll()` – and is still merely 30% behind The Holy Grail of http performance.

- Our tests were intentionally run for an as-small-as-feasible piece of code; we did NOT try to measure the performance of the computations within the language (this is covered by [Debian] and other benchmarks); instead, we tried to test the maximum performance of the respective frameworks.

  This means that for this particular test where we're implementing an `echo` http server, using C++ plug-ins for Node.js wouldn't bring any observable benefit (what would we do within a C++ plug-in to implement `echo`? Copy a string from input to output?)

- Within these tests, the performance of Node.cpp with runtime memory safety enabled was not that much different from its performance with runtime memory safety checks turned off (NB: this applies ONLY to this particular test; in general, such results depend heavily on the memory structures used, and may vary greatly).

Great! When can we start conversion?

As noted above, the whole point of Node.cpp is to allow conversion of (those 5% of performance-critical Nodes which warrant such an effort) from Node.js into Node.cpp.

The only tiny problem on the way is that Node.cpp is still in its early infancy, and is not ready (yet) for production use. In particular, the set of supported APIs is still extremely limited, and package management is not there yet. If you want to help, please feel free to contribute (or contact the authors); we feel that Node.cpp is a project with wonderful prospects that can help make Node.* ecosystem an indisputable leader at least performance-wise (and a common point of view is that Node loses performance-wise both to Erlang/Elixir and to Golang [Christensen16] [Peabody] [Stressgrid20][2]).

2. We do know that at least some of these tests are unfair to Node.js (by ignoring cluster module), and that Node is generally rather competitive to Erlang/Elixir and Golang, but 5x performance improvement would clearly blow all the competition out of the water.

the Node.cpp server outperforms the Node.js one by a factor of 5x. We feel that this opens up wonderful opportunities

Conclusions and future work

Over the course of this article, we took a very rudimentary Node.js http server, and converted it more or less line-by-line into Node.cpp. Then we ran a bunch of http tests to compare performance of both versions, and found that the Node.cpp server outperforms the Node.js one by a factor of 5×. We feel that this opens up wonderful opportunities and are going to continue our work to enable high-performance Node.* programming for those Nodes where performance is critical. ■

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

References

[Christensen16] Jack Christensen (2016) 'Websocket Shootout: Clojure, C++, Elixir, Go, NodeJS, and Ruby' at: https://hashrocket.com/blog/posts/websocket-shootout, posted 1 September 2016

[Debian] The Computer Language Benchmarks Game: 'Node js versus C++ g++ fastest programs' https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/node-gpp.html

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

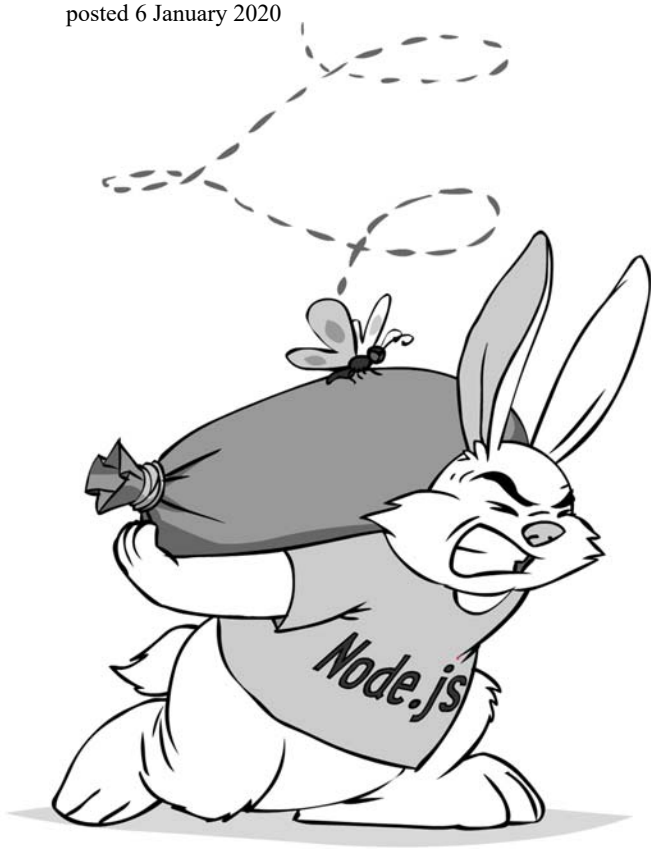
[Ostinelli11] Roberto Ostinelli (2011) 'A comparison between Misultin, Mochiweb, Cowboy, NodeJS and Tornadoweb' at: http://www.ostinelli.net/a-comparison-between-misultin-mochiweb-cowboy-nodejs-and-tornadoweb/

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

[Peabody] Brad Peabody 'Server-side I/O Performance: Node vs. PHP vs. Java vs. Go' at: https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go

[Stackoverflow] lawnmowerlatte 'Changing the file descriptor size in httperf' at: https://stackoverflow.com/a/16449853/4947867

[Stressgrid20] Stressgrid (2020) 'Benchmarking Go vs Node vs Elixir' at: https://stressgrid.com/blog/benchmarking_go_vs_node_vs_elixir/, posted 6 January 2020



The Path of Least Resistance

Python's modules and imports can be overwhelming. Steve Love attempts to de-mystify the process.

There is more to Python than scripting. As its popularity grows, people naturally want to do more with Python than just knock out simple scripts, and are increasingly wanting to write whole applications in it. It's lucky, then, that Python provides great facilities to support exactly that need. It can, however, be a little daunting to switch from using Python to run a simple script to working on a full-scale application. This article is intended to help with that, and to show how a little thought and planning on the structure of a program can make your life easier, and help you to avoid some of the mistakes that will certainly make your life harder.

We will examine modules and packages, how to use them, and how to break your programs into smaller chunks by writing your own. We will look in some detail at how Python locates the modules you import, and some of the difficulties this presents. Throughout, the examples will be based on the tests, since this provides a great way to see how modules and packages work. As part of that, we'll get to explore how to test modules independently, and how to structure packages to make using them straightforward.

A Python program

We begin with that simple, single file script, because it introduces some of the ideas that we'll build on later. Listing 1 is a simple tool to take CSV data and turn it into JSON.

```
import csv
import json
import sys

parser = csv.DictReader( sys.stdin )
data = list( parser )
print( json.dumps( data, sort_keys=True,
  indent=2 ) )
```

Listing 1

This program takes its input from `stdin`, and parses it as comma separated values. The `DictReader` parser turns each row of the CSV data into a `dict`.

Next, the entire input is read by creating a list of the dictionaries.

Lastly, the list is transformed into JSON, and the result (pretty) printed to `stdout`.

The code itself is not really what this article is about. What's more interesting are the `import` statements at the top of the program. All 3 of these modules are part of the Python Standard Library. The `import` statement tells the Python interpreter to find a module (or a package, but we'll get to that) and make its contents available for use.

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

The fact that these are standard library modules means they're always available if you have Python installed. The implication here is that you can share this program with anyone, and they'll be able to use it successfully if *they* have Python installed (and it's the right version, and you've told them *how* to use it).

That's all well and good for such a simple program, but sooner or later (sooner, hopefully!) someone will ask "Where are the tests?", followed quickly by "And what about error handling?" Error handling is left as an exercise, but testing allows us to explore more features of Python's support for modules.

Modularity 0.0

The basic Python feature for splitting a program into smaller parts is the function. It may seem overkill for our tiny script with only 3 lines of working code, but one of the side-effects (bumph-tish!) of creating a function is that with a little care, you can make a unit-test to exercise it independently. 'Independently' has multiple levels of meaning here: independent of other functions in your program, independent of the file system or other external components like databases, independent of user input – and, by the way – screen output. Your unit-tests should work *as if* the computer is disconnected from the outside world – no keyboard, no mouse, no screen, no network, no disk.

Why is this important? Partly because disk and network access is slow and unreliable, and you want your tests to run quickly, and partly because your tests might be run by some automated system like a Continuous Integration system that might not have access to the same things you do on your workstation.

Our program so far doesn't lend itself to being easily and automatically tested, so we'll start with factoring out the code that parses CSV data into a list of dictionaries. Listing 2 shows an example.

The function still uses a `DictReader`, but instead of directly using `sys.stdin`, it passes the argument it receives from the calling code.

The calling code now passes `sys.stdin` to the function and captures the result. The printing to screen remains the same.

A simple first test for this might be to check that the function doesn't misbehave if the input is empty. Although Python has a built-in unit-testing facility, there are lightweight alternatives, and the examples in this article

```
import csv
import json
import sys
def read_csv(input):
  parser = csv.DictReader(input)
  return list(parser)

data = read_csv(sys.stdin)
print(json.dumps(data, sort_keys = True,
  indent = 2))
```

Listing 2

it's the import that's interesting, because it shows that any old Python script is also a module that can be imported

```
from csv2json import read_csv
def test_read_csv_accepts_empty_input():
  result = read_csv(None)
  assert result == []
```

Listing 3

```
==================== ERRORS ====================
____ ERROR collecting test_csv2json.py ____

test_csv2json.py:1: in <module>
    from csv2json import read_csv
csv2json.py:9: in <module>
    data = read_csv( sys.stdin )
csv2json.py:7: in read_csv
    return list( parser )
...

    "pytest: reading from stdin while output is
captured!  Consider using `-s`."
E   OSError: pytest: reading from stdin while
output is captured!  Consider using `-s`.
```

Figure 1

all use pytest. Pytest will automatically discover test functions with names prefixed with `test`, in files with names prefixed with `test_`. For this example, assume the program is in a file called `csv2json.py`, and the test file is `test_csv2json.py`, containing Listing 3.

The first line imports the **`read_csv`** function from our `csv2json` script. The import name **`csv2json`** is just the name of the file, without the `.py` extension.

To start with, we write a test that we expect to *fail*, just to ensure we're actually exercising what we *think* we're exercising. **`csv.DictReader`** works with *iterable* objects, but passing **`None`** should certainly cause an error.

Once again, it's the import that's interesting, because it shows that any old Python script is also a module that can be imported. There is nothing special about code in Listing 2 to make it a module. A Python module is a namespace, so names within it must be unique, but can be the same as names from other namespaces without a clash. The syntax for importing shown here indicates that we only want one identifier from the csv2json module. Alternately, we could use

```
import csv2json
```

and then explicitly qualify the use of the function with the namespace:

```
result = csv2json.read_csv( None ).
```

We have a function, and a test with which to exercise it. Running that test couldn't be easier. From a command prompt/shell, in the directory location of the test script, run **`pytest`** [pytest].

But wait! What's this? (Figure 1 shows the output from pytest.)

It looks like the test failed, but not in the way we expect. It demonstrates another aspect of Python's import behaviour: importing a module *runs* the module. That's obviously not what we intended. What we need is some way to indicate the difference between *running* a python program, and *importing* its contents to a different program. Sure enough, Python provides a simple way to do this.

Each Python module in a program has a unique name, which you can access via the global **`__name__`** value. Additionally, if you invoke a program, then *its* name is always **`__main__`**. Listing 4 shows this in action.

Importing a module runs all the top-level code – which includes the *definition* of functions. The code within a function (or class) is syntax checked, but not invoked

When the script is run, the test for **`__name__`** will fail if it's being run as a result of an import of the module.

The function is invoked explicitly if the script is being run rather than imported.

Running the test now produces the output shown in Figure 2 (overleaf).

This time, the test has failed in the way we expected: **`DictReader`** is expecting an iterator. We can now alter the test to something we expect to *pass*, as shown here:

```
def test_read_csv_accepts_empty_input():
  result = read_csv( [] )
  assert result == []
```

And sure enough, it does.

```
==== test session starts ====
platform win32 -- Python 3.7.2, pytest-5.3.2,
py-1.8.0, pluggy-0.13.1
rootdir:
collected 1 item

test_csv2json.py .      [100%]

==== 1 passed in 0.04s ====
```

The next steps would be to add more tests, and some error handling, and factoring out the JSON export with its own tests, too. These are left as exercises.

```
import csv
import json
import sys
def read_csv(input):
  parser = csv.DictReader(input)
  return list(parser)
if __name__ == '__main__':
  data = read_csv(sys.stdin)
  print(json.dumps(data, sort_keys = True,
    indent = 2))
```

Listing 4

Python provides another facility to bundle several modules together into a package

```
================== FAILURES ==================
____ test_read_csv_accepts_empty_input() ____

    def test_read_csv_accepts_empty_input():
>       result = read_csv( None )

test_csv2json.py:4:
 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
csv2json.py:6: in read_csv
    parser = csv.DictReader( input )
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <csv.DictReader object at
0x000001E882CD94A8>, f = None, fieldnames = None,
restkey = None, restval = None, dialect = 'excel'
args = (), kwds = {}

    def __init__(self, f, fieldnames=None,
                 restkey=None, restval=None,
                 dialect="excel", *args, **kwds):
        self._fieldnames = fieldnames
          # list of keys for the dict
        self.restkey = restkey
          # key to catch long rows
        self.restval = restval
          # default value for short rows
>       self.reader = reader(f, dialect, *args,
                     **kwds)
E       TypeError: argument 1 must be an iterator
```

Figure 2

What have we learned?

1. Simple Python modules are just files with Python code in them
2. Importing a module runs all the top-level code in that module
3. Python modules all have a unique name within a program, which is accessed via the value of **`__name__`**
4. The entry point of the program is a module called **`__main__`**
5. Unit tests are a great way of exercising your code, but they're also great for exercising the *structure* of the code.

Packages

Having satisfied the requirement for tests, we may wish to embellish our little library to be a bit more general. One obvious thing might be to extend the functionality to handle other data formats, and perhaps to be able to convert in any direction. It wouldn't necessarily be unreasonable to just add a new method to the module called **`json2csv`**[1]. However, if we want to add new text formats, the combinations become unwieldy, and would introduce an unnecessary amount of duplication.

Namespaces

You may need to explicitly qualify function calls with namespacing, if, for example, you were importing another module or package that defined a **`read_csv()`** function. A full qualification includes the name of the package:

```
import textfilters.csv2json
def test_read_csv_accepts_empty_input():
  result = textfilters.csv2json.read_csv( [] )
  assert result == []
```

It may be, however, that just the module name provides sufficient uniqueness in the name, and so a compromise is possible:

```
from textfilters import csv2json

def test_read_csv_accepts_empty_input():
  result = csv2json.read_csv( [] )
  assert result == []
```

Don't confuse this general idea with Python's Namespace Packages [NamespacePkg], which are used to split packages up across several locations.

The basis of the library is to take some input and parse it into a Python data structure, which we can then turn into an export format. Having gone to the trouble of separating the inputs and outputs, we can extend the idea and use a different module for each text format. Python provides another facility to bundle several modules together into a package.

As we've already explored, a Python module can be just a file containing Python code. A Python package is a module that gathers sub-modules (and possibly sub-packages) together in a directory[2], along with a special file named `__init__.py`. For the time being, this can be just an empty file, but we will revisit this file in a later article.

We'll begin by just adding our existing module to a package called **`textfilters`**. Our source tree now should look something like this:

```
root/
  |__ textfilters/
        |__ __init__.py
        |__ csv2json.py
  |__ test_csv2json.py
```

The **`import`** statement in `test_csv2json.py` now needs to change as shown here:

```
from textfilters.csv2json import read_csv

def test_read_csv_accepts_empty_input():
  result = read_csv( [] )
  assert result == []
```

The **`import`** statement line at the top says 'import the **`read_csv`** definition from the **`csv2json`** module in the **`textfilters`** package'.

1. It doesn't always make sense to do this conversion due to the possibility of nesting in the JSON input, of course, but just for the exercise.

2. This is a little simplistic, because there is no intrinsic requirement for modules or packages to exist on a file-system, but it suffices for now.

Importing a module (and remember, packages are modules too) requires the Python interpreter to locate the code for that module, and make its code available to the program

Note that the portion of the import line *after* the **`import`** keyword defines the namespace. In this example, the namespace consists only of the function name, and needs no further qualification when used.

So far, so good. Whichever method of importing the function you use, running the test works, and (hopefully!) your tests still pass.

The next step is to split the functions into their separate modules. So far, we have two text formats to deal with: CSV and JSON. It makes some sense, therefore, to handle all the CSV functionality in a module called **`csv`**, and all the JSON in a module called **`json`**. Our original CSV function – **`read_csv`** – now has a name with some redundancy, duplicating as it does the name of the new module. I also decided that 'read' and 'write' weren't really accurate names, implying some kind of file-system access, and decided upon **`input`** and **`output`**. Listing 5 shows the content of the new **`csv`** module called `csv.py`.

As previously, it's not really the implementation that's interesting about this, it's that first **`import`** statement. Remember, this is a file called `csv.py`, and so *its* module name is **`csv`**. It should be clear from the code that the intention is to import the built-in **`csv`** module, and that is indeed what will *probably* happen, due to a number of factors. It's time to talk about how Python imports modules.

The search for modules

Importing a module (and remember, packages are modules too) requires the Python interpreter to locate the code for that module, and make its code available to the program by **binding** the code to a name. You can't usually give an absolute path directly, but Python has a number of standard places in which it attempts to locate the module being imported.

Python keeps a cache of modules that have already been imported. Where an imported module is a sub-module, the parent module (and its parents) are also cached. This cache is checked first to see if a requested module has already been imported.

```
import csv
from io
import StringIO
def input(data):
  parser = csv.DictReader(data)
  return list(parser)
def output(data):
  if len(data) > 0:
    with StringIO() as buffer:
      writer = csv.DictWriter(buffer, data[
        0].keys())
      writer.writeheader()
      for row in data:
        writer.writerow(row)
      return buffer.getvalue()
  return ""
```

Listing 5

```
from textfilters
import csv
def test_read_csv_accepts_empty_input():
  result = csv.input([])
  assert result == []
```

Listing 6

Next, if the module was not found in the cache, the paths in the system-defined **`sys.path`** list are checked in order. This sounds simple enough, but this is probably where the consequences of giving our own module the same name as a built-in one will become apparent. The contents of **`sys.path`** are initialized on startup with the following:

1. The directory containing the script being invoked, *or* an empty string to indicate the current working directory in the case where Python is invoked with no script (i.e. interactively).
2. The contents of the environment variable **`PYTHONPATH`**. You can alter this to change how modules are located when they're imported.
3. System defined search paths for built-in modules.
4. The root of the **`site`** module (more on this in a later article).

Let's consider the directory layout containing our package:

```
root/
  |__ textfilters/
        |__ __init__.py
        |__ csv.py
        |__ ...
  |__ test_filters.py
```

If you invoke a python script, or run the Python interpreter, in the `root/textfilters/` directory, the statement **`import csv`** will find the **`csv`** module in that directory first. Note that the current working directory is *not* used if a Python script is invoked. Correspondingly, invoking a Python script, or running the Python interpreter in any other location would result in **`import csv`** importing the built-in **`csv`** module.

Back to our `csv.py` module, the statement **`import csv`** would indeed be recursive if you were to invoke the script directly, or any other script in the same directory that imported it. However, the usual purpose of a package is for it to be imported into a program, and the reason for making a directory for a package is to keep the package's contents separate from other code.

To demonstrate this, let's have a look at how the test script looks now it's using the **`textfilters`** package in Listing 6.

The **`import`** statement is explicitly requesting the **`csv`** module from the **`textfilters`** package.

As long as the **`textfilters`** package is found, then this script will use the **`csv`** module within it, and will never import the built-in module of the same name. In this instance, the invoked Python script is `test_filters.py`, and the search path will have its directory at the head of **`sys.path`**. The **`textfilters`** package is found as a child of that directory, and all is good with the world.

It needs to be said that deliberately naming your own modules to clash with built-in modules is a bad idea

If the **textfilters** package were located somewhere else, away from your main program, you could add its path to the **PYTHONPATH** environment variable to ensure it was found. As previously mentioned, you don't *directly* import modules using absolute paths, but the **PYTHONPATH** environment variable is one *indirect* way of specifying additional search paths. Requiring all users of your module(s) to have an appropriate setting for **PYTHONPATH** is a bit heavy-handed[3], but can be useful during development.

It needs to be said that deliberately naming your own modules to clash with built-in modules is a bad idea, because just relying on the search behaviour to ensure the right module is imported is flying close to the wind, to say the least. However, things are never that simple: you cannot know what future versions of Python will call new modules, and you cannot know what other 3rd party libraries your users have installed. This is one reason why it's a good idea to partition your code with packages, and be *explicit* about the names you import.

What have we learned?

1. A Python package is a module that has sub-modules. Standard Python packages contain a special file called `__init__.py`.
2. The package name forms part of the namespace, and needs to be unique in a program.
3. Python looks for modules in a few standard places, defined in a list called `sys.path`.
4. You can modify the search path easily by defining (or changing) an environment variable called **PYTHONPATH**.
5. You should partition your code with packages to minimize the risk of your names clashing with other modules.

Relativity

One of the reasons for packaging up code is to make it easy to share. At the moment, we have a package – **textfilters**, and the test code for it lives *outside* the package. If we share the package, we should also share the tests, and having them *inside* the package means we can more easily share it just by copying the whole directory.

Look back at Listing 6, and note the **import** statement directly names the package. While this is fine (the tests should pass), it seems redundant to have to explicitly name it. Since this test module is now part of the same package as the **csv** module, what's wrong with just **import csv**?

The problem with that is we will get caught out (again) by the Python search path for modules; **import csv** will just import the built-in **csv** module. Is there an alternative to explicitly having to give a full, absolute, name to modules imported within a package?

We previously learned that you can't give an absolute path to the **import** statement, but you *can* request a *relative* path when importing within a module, i.e. one file in a package needs to import another module *within the same package*. Listing 7 shows the needed invocation of import.

```
from . import csv

def test_read_csv_accepts_empty_input():
  result = csv.input( [] )
  assert result == []
```

Listing 7

The test file is now part of the **textfilters** package, and so uses **.** instead of the package name to indicate 'within my package directory'.

As we've already noted, Python modules have a special value called **__name__**. We've looked at how this is set to **__main__** when a module is run as a script. When a module is imported, this value takes on the namespace name, which is essentially the path to the module relative to the application's directory, but separated by **.** instead of **** or **/**.

Python modules have another special value which is used to resolve relative imports within it. This is the **__package__** value, which contains the name of the package. If the module is *not* part of a package, then this value is the empty string. This is discussed in detail in PEP 366 [PEP366], but the important thing to note here is that, since the testing module is now part of a package, the relative import path shown in Listing 7 uses the value of **__package__** to determine how to find the csv module to be imported.

The package so far is a basic outline, with a simple API to translate one data format to another. We can extend this idea with facilities to transform the data as it passes through, perhaps to rename fields, or select only those fields we need. We could clearly just make a new package for these general utilities, but the intention is that they're used in conjunction with the functions we've already created, so let's instead make the new package a child of the existing one.

```
root/
 |__ textfilters/
        |__ __init__.py
        |__ csv.py
        |__ ...
        |__ transformers/
            |__ __init__.py
            |__ change.py
        |__ test_filters.py
```

Relative imports also work for sub-packages. This is easily demonstrated with more tests, this time for the `transformers/change.py` module, as shown in Listing 8.

The test module imports the code under test from `change.py` using a relative import path prefixing the package and module names.

In this case, it's exactly as if we'd written:

```
from textfilters.transformers.change import
change_keys.
```

By now, we're collecting a few test modules in the base of the package directory, and we might want to think about more tidying up to gather all the tests together away from the actual package code. We can do this by creating a new sub-package called **tests** and moving all the test code into

3. You can also modify sys.path in code, since it's just a list of paths, but your users will probably not thank you for it

```
# textfilters / transformers / change.py
def change_keys(data, fn):
  return {
    fn(k): data[k]
    for k in data
  }
# textfilters / test_change.py
from .transformers.change
import change_keys
def test_change_keys_transforms_input():
  d = {
    1: 1
  }
  res = change_keys(d, lambda k: 'a')
  assert res == {
    'a': 1
  }
```

Listing 8

it. This must be a sub-package, and so requires an `__init__.py` of its own, and the relative imports need to change as shown in Listing 9.

The relative module imports now have an extra dot to indicate the *parent* package location.

This may look a bit like relative paths on a file-system, where `../` is the parent directory, it's not quite the same. Packages can be arbitrarily nested, and to indicate the grand-parent directory, on Linux you'd say `../../`, whereas in Python you just add another dot: `...`

What have we learned?

1. Modules in a package can use relative imports to access code within the same module.
2. Packages have a special value **`__package__`** which is used to resolve relative imports.
3. Packages can have sub-packages, and these can be deeply nested – if you really want!

All together now

We've demonstrated how to write test code in the package to exercise our little library, so now for completeness, it's time to demonstrate a little running program[4]. Listing 10 shows how it might be used.

This short program essentially does what the code in Listing 1 did, with the added bonus of taking the column-names and changing them to upper case. Importing the required functions is the job of the first 2 lines, and for now, the **`textfilters`** package needs to be on Python's search path. The simplest way to do *that* is to have it as a sub-directory of the main application folder.

Have we achieved what we set out to achieve? We had three goals in mind at the start:

1. to be able to test independent code independently
2. to be able to split our programs into manageable chunks
3. to be able to share those chunks with others.

We have created a package, with its own set of tests. Those tests not only test each part of the package code independently of the others, but also – and importantly – independently of the code that uses it in the main

```
# textfilters / tests / test_change.py
from ..transformers.change
  import change_keys
def test_change_keys_transforms_input():
  d = { 1: 1 }
  res = change_keys(d, lambda k: 'a')
  assert res == { 'a': 1 }
```

Listing 9

4. Ok, not actually completeness, because some of the functionality that this uses was left as exercises. You did do the exercises?

```
from textfilters
import csv, json
from textfilters.transformers.change
import change_keys
import sys
if __name__ == '__main__':
  def key_toupper(k):
    return k.upper()
  data = csv.input(sys.stdin)
  result = [change_keys(row, key_toupper) for row
    in data
  ]
  print(json.output(result, sort_keys =
    True, indent = 2))
```

Listing 10

application. This makes sharing the code with others much easier: the package is a small, self-contained parcel of functionality.

The import statements are a little verbose, due to the use of sub-packages, and the need to make the package directory a direct child of the main application is a little unwieldy.

In the next installment, we will explore how to improve on both of those things so that your fellow users will really like using your library. ■

References and resources

[NamespacePkg] Python Namespace Packages: https://packaging.python.org/guides/packaging-namespace-packages/

[PEP366] 'PEP 366, Main module explicit relative imports' https://www.python.org/dev/peps/pep-0366/

[pytest] 'A Python testing framework' https://docs.pytest.org/en/latest/

And the winners are...

In the last *Overload* (and *CVu*) we invited our readers to vote for their favourite articles of 2019 in *CVu*, which is our sister magazine for members, and in *Overload*.

For CVu:

1st place:	Francis Glassborow for 'The Early Days of C++ in UK C User Groups' in *CVu* 31.3 (July 2019)
2nd place:	Pete Goodliffe for 'Effective Software Testing' in *CVu* 30.6 (January 2019) *and*
	Spencer Collyer for 'Who Are You Calling Weak?' in CVu 31.3 (July 2019)

For Overload:

1st place:	Cassio Neri for 'Quick modular calculations (Part 1)' in *Overload* 154 (December 2019)
2nd place:	Anders Modén for 'The Duality…' in *Overload* 150 (April 2019)

Thank you to everyone who took time to vote, and for those who wrote. We can't offer a prize to these winners, just the mention here. A number of other writers got a vote – so be assured if you wrote for us someone probably thoroughly enjoyed what you had to say. Keep up the good work.

The article titles above link to the articles if you are reading this as a PDF. *Overload* articles are publicly available, but you must be a member (and logged in) to access the *CVu* ones. If you're not a member yet, why not join?

Quick Modular Calculations (Part 2)

The minverse algorithm previously seen is fast but has limitations. Cassio Neri presents two alternatives.

The first instalment of this series [Neri19] introduced the *minverse* algorithm and a C++ library called **qmodular** [qmodular] that implements it. We have seen that in certain cases *minverse* performs better than the algorithm currently implemented by major compilers when evaluating expressions of the form `n % d == r`. Unfortunately, *minverse* does not work when `==` is replaced by `<`, `<=`, `>` or `>=`. This article presents two algorithms that do not have this restriction, namely, *Multiply and Shift* (*mshift*) and *Multiply and Compare* (*mcomp*). They are very similar to one another and also to the *Remainder by Multiplication and Shifting Right* (*RMSR*) covered in *Hacker's Delight* [Warren13].

It is important to remember that the intention here is not to 'beat' the compiler. On the contrary, this series is an open letter addressed to compiler writers presenting some algorithms that, potentially, could be incorporated into their product for the benefit of all programmers. Performance analysis shows that the alternatives discussed in this series are often faster than built-in implementations.

Recall and warm up

We start by looking at Figure 1[1] which displays the time taken to check whether each element of an array of 65,536 uniformly distributed unsigned 32-bits dividends in the interval $[0, 10^6]$ leaves remainder 3 when divided by 10. Bars *built_in* and *minverse* correspond to algorithms presented in [Neri19]. (*Built-in* is simply[2] `n % 10 == 3`.) Bars *mshift* and *mcomp* refer to algorithms covered by this article. A simple variant of each, *mshift_promoted* and *mcomp_promoted*, are also shown.

All measurements include the time to scan the array of dividends which is used as unit of time. Timings[3] are 2.14 for *built-in*, 1.71 for *minverse*, 1.47 for *mshift* and *mcomp* and 1.30 for promoted variants. Subtracting the scanning time and taking results relatively to *built-in*'s yields 0.71/1.14 ≈ 0.62 for *minverse*, 0.47/1.14 ≈ 0.41 for *mshift* and *mcomp* and 0.30/1.14 ≈ 0.26 for promoted algorithms. These numbers, however, depend on the divisor as will be made clear later on.

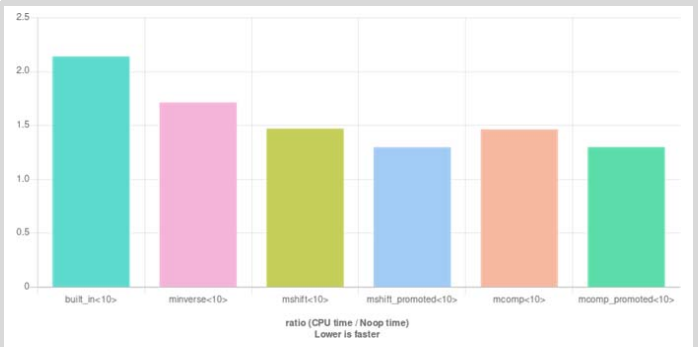


Figure 1

```
built_in
   0: mov    %edi,%eax
   2: mov    $0xcccccccd,%edx
   7: mul    %edx
   9: shr    $0x3,%edx
   c: lea    (%rdx,%rdx,4),%eax
   f: add    %eax,%eax
  11: sub    %eax,%edi
  13: cmp    $0x3,%edi
  16: sete   %al
  19: retq

minverse
   0: sub    $0x3,%edi
   3: imul   $0xcccccccd,%edi,%edi
   9: ror    %edi
   b: cmp    $0x19999999,%edi
  11: setbe  %al
  14: retq

mshift
   0: imul   $0x1999999a,%edi,%edi
   6: shr    $0x1c,%edi
   9: cmp    $0x4,%edi
   c: sete   %al
   f: retq

mcomp
   0: sub    $0x3,%edi
   3: imul   $0x1999999a,%edi,%edi
   9: cmp    $0x1999999a,%edi
   f: setb   %al
  12: retq
```

Listing 1

Listing 1 contrasts the code generated by GCC 8.2.1 with `-O3` for some of these algorithms.

Cassio Neri has a PhD in Applied Mathematics from Université de Paris Dauphine. He worked as a lecturer in Mathematics before moving to the financial industry. He can be contacted at cassio.neri@gmail.com.

1. Powered by quick-bench.com. For readers who are C++ programmers and do not know this site, I strongly recommend checking it out. In addition, I politely ask all readers to consider contributing to the site to keep it running. (Disclaimer: apart from being a regular user and donor, I have no other affiliation with this site.)
2. We are using bold fixed-width font for code (as is usual in *Overload*), so 98 / 10 = 9.8 is maths and `98 / 10 == 9` is code.
3. YMMV, reported numbers were obtained by a single run in quick-bench.com using GCC 8.2 with `-O3` and `-std=c++17` [QuickBench]. I do not know details about the platform it runs on, especially the processor.

The ellipses serve as reminders that those equalities hold in the beautiful world of infinite precision which our CPUs do not belong. Rounding is necessary and error is unavoidable.

Recall that we are concerned with the evaluation of modular expressions where **the divisor is a compile time constant and the dividend is a runtime variable. The remainder can be either. They all have the same unsigned integer type which implements modulus 2^w arithmetic.** (Typically, $w = 32$ or $w = 64$.) We focuses on GCC 8.2.1 for x86_64 target but some ideas here might also apply to other platforms.

The 'Remainder by Multiplication and Shifting Right' algorithm

This section loosely follows [Warren13] and covers the basics of *RMSR* by means of an example. We shall see that *RMSR*, *mshift* and *mcomp* have the same underlying idea.

Let the divisor be $d = 10$. By Euclidean division, any integer n can be uniquely written as $n = 10 \cdot q + r$, where q and r are integers, $r \in [0, 9]$. Written in code, `q = n / 10` and `r = n % 10`. Using this expression we obtain

$$\begin{aligned}\lfloor 16 \cdot (n / 10) \rfloor &= \lfloor 16 \cdot ((10 \cdot q + r) / 10) \rfloor = \lfloor 16 \cdot q + (16 \cdot r) / 10) \rfloor \\ &= 16 \cdot q + \lfloor 16 \cdot (r / 10) \rfloor .\end{aligned}$$

It follows that $\lfloor 16 \cdot (n / 10) \rfloor \equiv \lfloor 16 \cdot (r / 10) \rfloor \pmod{16}$.

[Warren13] observes that for $r = 0, 1, 2, 3, 4, 5, 6, 7, 8$ and 9, the quantity $\lfloor 16 \cdot (r / 10) \rfloor$ takes values on 0, 1, 3, 4, 6, 8, 9, 11, 12 and 14, respectively. Since the latter numbers are all distinct they can be mapped back to the former. Therefore, we efficiently obtain r provided we can reasonably evaluate:

1. $\lfloor 16 \cdot (n / 10) \rfloor \bmod 16$; and
2. the inverse of φ defined by $\varphi(r) = \lfloor 16 \cdot (r / 10) \rfloor$, for all $r \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

We consider the evaluation of $\lfloor 16 \cdot (n / 10) \rfloor \bmod 16$ by looking at the binary representation of intermediate results, namely, $x_1 = n / 10$, $x_2 = 16 \cdot x_1$, $x_3 = \lfloor x_2 \rfloor$ and $x_4 = x_3 \bmod 16$. To get x_2 we multiply x_1 by 16, which shifts bits 4 positions to the left. In particular, the first 4 bits of x_1 on the right of the binary point move to its left. To get x_3, we apply the integer part function discarding fractional bits. Finally, to get x_4 we take modulo 16 zeroing all but the 4 leftmost bits. For example, for $n = 98$ we have (relevant bits are emphasised):

$x_1 = 98 / 10 = 9.8 = (1001.\mathbf{1100}1100\ldots)_2$,

$x_2 = 16 \cdot x_1 = 156.8 = (1001\mathbf{1100}.1100\ldots)_2$,

$x_3 = \lfloor x_2 \rfloor = 156 = (1001\mathbf{1100})_2$,

$x_4 = \text{x3} \bmod 16 = 12 = (\mathbf{1100})_2$.

The ellipses above serve as reminders that those equalities hold in the beautiful world of infinite precision which our CPUs do not belong. Rounding is necessary and error is unavoidable. Caution is required to avoid trashing the bits that we are interested in. Hence, we look for an approximation of $x_1 = n / 10$ known to be correct at least up to 4 bits after the binary point or, equivalently, an approximation of $x_2 = (2^4 / 10) \cdot n$ correct, at least, on its integer part. There is an efficient way to tackle this problem. It is a classical idea widely used by compilers to evaluate division by compile time constants. For the sake of concreteness, we restraint this discussion to 32-bits CPUs.

We have $x_2 = (2^4 / 10) \cdot n = (2^{32} / 10) \cdot n / 2^{28}$. Here the rounding comes: number $(2^{32} / 10) = 429{,}496{,}729.6$ is rounded up to $M = 429{,}496{,}730$ ([Warren13] rounds it down) and we get $x_2 \approx M \cdot n / 2^{28}$. The error of this approximation, $(M - (2^{32} / 10)) \cdot n / 2^{28}$, grows with n and thus, for n large enough, it becomes such that x_2 and $M \cdot n / 2^{28}$ do not have the same integer part. For such values $x_3 \neq \lfloor M \cdot n / 2^{28} \rfloor$ and the algorithm breaks. For instance, for $n = 2^{27}$, we have $x_2 = (2^4 / 10) \cdot 2^{27} = 2^{31} / 10 = 214{,}748{,}364.8$ whereas $M \cdot n / 2^{28} = M \cdot 2^{27} / 2^{28} = M / 2 = 214{,}748{,}365$.

Despite the last issue, there exist a maximal interval $[0, N]$ such that $x_3 = \lfloor M \cdot n / 2^{28} \rfloor$ for any integer $n \in [0, N]$. Finishing part 1 of the *RMSR* requires evaluating $x_4 = x_3 \bmod 16$. However, by yet another practical aspect of CPUs, this step becomes unnecessary. Indeed, $\lfloor M \cdot n / 2^{28} \rfloor$ is evaluated as `(M * n) >> 28` in modulus 2^{32} arithmetic. Hence all but the 32 leftmost bits of `M * n` are discarded. After the 28-bits right shift only the 4 leftmost bits of the result might be non-zero. Hence, the subsequent mod 16 operation has no effect.

The consideration for $n = 2^{27}$ above implies $N < 2^{27}$. In particular, this method does not work on the whole range of 32-bits unsigned integers. [Warren13] tackles this issue at the same time as he deals with the evaluation of φ^{-1}. (Part 2 of *RMSR*.) This is not straightforward and he covers it only for a handful of divisors. His approach uses bitwise tricks obtained, as he puts it, by 'a lot of trial and error' (which is not generic) and look-up tables (which does not scale well). Details can be seen in [Warren13].

The mshift algorithm

Luckily, *mshift* and *mcomp* seek to make remainder comparisons without knowing it. For this reason they do not need to evaluate φ^{-1} and only take into consideration the fact that φ is strictly increasing.

We have seen that if r is the remainder of the division of n by 10, then $\lfloor 16 \cdot (n / 10) \bmod 16 = \varphi(r)$. Since φ is strictly increasing, we have $r = 0$ if, and only if, $\varphi(r) = \varphi(0) = 0$, ie, $\lfloor 16 \cdot (n / 10) \rfloor \bmod 16 = 0$. Therefore, the latter equality is equivalent to divisibility by 10. Finally, using the efficient evaluation of $\lfloor 16 \cdot (n / 10) \rfloor \bmod 16$, we get that for $n \in [0, N]$, `n % 10 == 0` is equivalent to `(M * n) >> 28 == 0`. The same applies to, say, $r = 3$: if $n \in [0, N]$, then `n % 10 == 3` if, and only if, `(M * n) >> 28 == (M * 3) >> 28`. (This is *mshift* in Figure 1 and Listing 1.)

Summarising, let `uint_t` be an unsigned integer type with modulus 2^w arithmetic. Given any integer $d \in [2, 2^w[$, let p be the smallest integer such that $d \leq 2^p$. Set $M = \lceil 2^w / d \rceil$ and $s = w - p$. Then, there exists an integer $N \in [0, 2^w[$ such that for integers $n \in [0, N]$, $r \in [0, d - 1]$ and any relational operator $\lesseqgtr$ (i.e., $\lesseqgtr$ is any of `==`, `!=`, `<`, `<=`, `>` or `>=`), we have `n % d` $\lesseqgtr$ `r` is equivalent to `phi(n)` $\lesseqgtr$ `phi(r)`, where `phi` is defined by

```
uint_t phi(uint_t n) {
  return (M * n) >> s;
}
```

A lower bound for N is straightforward to calculate but its derivation is outside the scope of this article. (See [qmodular].) Most often, we have $N < 2^w - 1$ and, consequently, *mshift* cannot be applied to the whole range of `uint_t`. We shall see later how to deal with this limitation.

The mcomp algorithm

The underlying idea of *mcomp* is similar to *mshift*'s, the difference being a simple observation. The essential point of *mshift* is looking at a few bits of n / d after the binary point which are obtained through `phi`. This function introduces a rounding error that grows with n and, eventually, gets large enough to trash the important bits. At this point `phi` returns a wrong value and *mshift* breaks. Looking at more bits on the right of the binary point enables *mcomp* to carry on.

Consider the example $d = 10$ again. As in *mshift*, the first step of *mcomp* is evaluating the approximation $\lceil 2^{32} / 10 \rceil \cdot n$, of $(2^{32} / 10) \cdot n$, under modulus 2^{32} arithmetic. Table 1 shows both products, in binary, for a few values of n. For easy of presentation, it only shows the most and least significant bytes of the 32 bits on the left of the binary point (ellipses account for the other 16 bits). (The 4 bits that are important to *mshift* are emphasised.)

When n divides 10 the second column shows 0 since $(2^{32} / 10) \cdot n$ is a multiple of 2^{32}. The third column only approximates the second one: it starts showing 0, for $n = 0$, and steadily diverges from this value as n increases. Indeed, from n to $n + 10$ the error goes up by 4 (or $(100)_2$). (This is due to $\lceil 2^{32} / 10 \rceil \cdot 10 - (2^{32} / 10) \cdot 10 = \lceil 2^{32} / 10 \rceil \cdot 10 - 2^{32} = 4$.) Therefore, starting at 0, after 2^{26} steps the error accrues to $4 \cdot 2^{26} = 2^{28}$ for $n = 10 \cdot 2^{26}$. This is large enough to trash the 4 leftmost bits of $M \cdot n$, which become $(0001)_2$. Notice the latter is the bit pattern expected for remainder $r = 1$ and not for $r = 0$ and then, *mshift* breaks at this point.

However, looking at all bits of the result, $(0001\ 0000\ \ldots\ 0000\ 0000)_2$, we realise it is still far below $(0001\ 1001\ \ldots\ 1001\ 1010)_2$ which is the result for $n = 1$, that is, $M = \lceil 2^{32} / 10 \rceil \cdot 1 = \lceil 2^{32} / 10 \rceil$. Therefore, if that n is not large enough for $M \cdot n$ to reach the barrier M, then it is still possible to detect that n is multiple of 10. In other words, `n % 10 == 0` is equivalent to `M * n < M`.

In general, for any remainder $r \in [0, d - 1]$, the result of $M \cdot$ `n` mod 2^{32} starts, for $n = r$, with a small error with respect to $(2^{32} / 10) \cdot r$. The error steadily increases in steps of size 4 as n takes successive values with the same remainder. If n is not large enough for $M \cdot n$ to reach the next barrier, $M \cdot (r + 1)$, then it is still possible to detect that n has remainder r. This means that `n % 10 == r` is equivalent to `(M * r <= M * n) && (M * n < M * (r + 1))`. Well, this is not entirely true and some details must be considered.

n	$(2^{32} / 10) \cdot n$	$\lceil 2^{32} / 10 \rceil \cdot n$
0	**0000** 0000 … 0000 0000	**0000** 0000 … 0000 0000
10	**0000** 0000 … 0000 0000	**0000** 0000 … 0000 0100
20	**0000** 0000 … 0000 0000	**0000** 0000 … 0000 1000
30	**0000** 0000 … 0000 0000	**0000** 0000 … 0000 1100
40	**0000** 0000 … 0000 0000	**0000** 0000 … 0001 0000
⋮	⋮	⋮
$10 \cdot (2^{26} - 4)$	**0000** 0000 … 0000 0000	**0000** 1111 … 1111 0000
$10 \cdot (2^{26} - 3)$	**0000** 0000 … 0000 0000	**0000** 1111 … 1111 0100
$10 \cdot (2^{26} - 2)$	**0000** 0000 … 0000 0000	**0000** 1111 … 1111 1000
$10 \cdot (2^{26} - 1)$	**0000** 0000 … 0000 0000	**0000** 1111 … 1111 1100
$10 \cdot 2^{26}$	**0000** 0000 … 0000 0000	**0001** 0000 … 0000 0000
$10 \cdot (2^{26} + 1)$	**0000** 0000 … 0000 0000	**0001** 0000 … 0000 0100
⋮	⋮	⋮
1	**0001** 1000 … 1001 1001	**0001** 1001 … 1001 1010

Table 1

For $r = 9$, we have $M \cdot (r + 1) = M \cdot 10 = 2^{32} + 4 \geq 2^{32}$. Hence, regardless of n, the second operand of `&&` above should be `true`. Nevertheless, its evaluation can yield `false` since $M \cdot (r + 1)$ overflows under modulus 2^{32} arithmetic. To get the right result when $r = 9$, we should test only the left hand side of `&&` above.

Regarding `&&`, this operator creates a branch that can immensely degrade performance. To address this issue, notice that by subtracting $M \cdot r$ from $M \cdot r \leq M \cdot n < M \cdot (r + 1)$ we get that these inequalities are equivalent to $0 \leq M \cdot (n - r) < M$. However, under modulus 2^{32} arithmetic, the subtraction overflows when $n < r$. Although the correct derivation in modulus 2^{32} arithmetic is not that straightforward, the previous result remains valid. Furthermore, $0 \leq M \cdot (n - r)$ is always true and then, we can evaluate `n % 10 == r` as `M * (n - r) < M`, except when $r = 9$. In this case, the upper bound `M` should be replaced by `M - 4` to account for the fact that `M * 10` overflows by an excess of `4`.

There are outstanding issues to be addressed but we evasively refer to [qmodular] for details.

We now summarise the *mcomp* algorithm. Let `uint_t` be an unsigned integer type with modulus $2w$ arithmetic. Given any integer $d \in [2, 2^w[$, set $M = \lceil 2^w / d \rceil$ and $m = M \cdot d - 2^w$. If $m < M$, then there exists an integer $N \in [0, 2^w[$ such that for all integers $n \in [0, N]$ and $r \in [0, d - 1]$, we have:

1. `n % d == r` is equivalent to
   a) `M * (n - r) < M`, if $r \neq d - 1$;
   b) `M * (n - r) < M - m`, if $r = d - 1$;
   c) `M * n >= M * r`, if $r = d - 1$;
4. `n % d < r` is equivalent to `M * n < M * r`.

From these results we can derive similar equivalences for other relational operators.

The unbounded case

Like *minverse*, $r \in [0, d - 1]$ is a precondition for *mshift* and *mcomp*. To extend these algorithms to larger remainders, the same approach used for *minverse* can be applied. (See [Neri19].)

Increasing the range of applicability

Both *mshift* and *mcomp* have a range of applicability, (expressed by the hypothesis $n \in [0, N]$) which, most often, is not the whole domain of the unsigned integer types they work on. (In contrast, *built-in* and *minverse* do not have this limitation.)

An easy way of increasing these algorithm's applicability is by promoting the type that they work on. Specifically, to evaluate a modular expression with `uint32_t` operands, we can promoted `n`, `d` and `r` to `uint64_t` and use the algorithm for this type. This is what the suffix *_promoted* in Figure 1 refers to.

Although GCC provides 128-bit unsigned integer types, on x86_64 platforms these types only have partial support from the CPU and must be synthesised by software. Therefore, the promotion strategy might be rather expensive for `uint64_t` and using *mshift* or *mcomp* for this type of operand might not be the best option if performance and full range of applicability are simultaneously needed.

Performance analysis

As in the warm up, all measurements shown in this section concern the evaluation of modular expressions for 65,536 uniformly distributed unsigned 32-bits dividends in the interval $[0, 10^6]$. Remainders can be either fixed at compile time or variable at runtime. Charts show divisors on the x-axis and time measurements, in nanoseconds, on the y-axis. Timings are adjusted to account for the time of array scanning.

For clarity we restrict divisors to [1, 50] which suffices to spot trends. (Results for divisors up to 1,000 are available in [qmodular].) In addition, we filter out divisors that are powers of two since the bitwise trick is

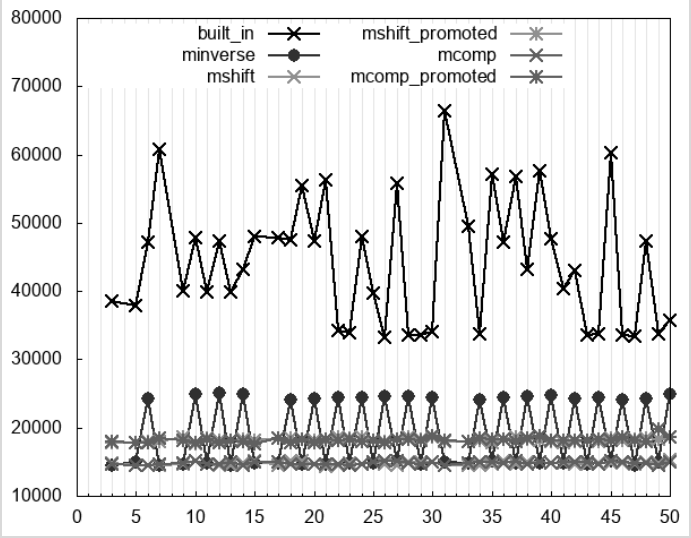


Figure 2

undoubtedly the best algorithm for them. The timings were obtained with the help of Google Benchmark [Google] running on an AMD Ryzen 7 1800X Eight-Core Processor @ 3600Mhz; caches: L1 Data 32K (x8), L1 Instruction 64K (x8), L2 Unified 512K (x8), L3 Unified 8192K (x2).

Figure 2 concerns divisibility tests, that is, evaluation of `n % d == 0`. As already seen in [Neri19], built-in is slower than *minverse* which zigzags as divisors changes from odd (faster) to even (slower) values. The new algorithms, *mshift* and *mcomp*, perform for all divisors as good as *minverse* does for odd divisors. In contrast, their promoted variants perform better than *minverse* for even divisors. This makes the *minverse* preferable for odd divisors since it does not have the limitation on the range of `n` whereas either of the promoted variants is preferable for even divisors.

Figure 3 covers `n % d == r` where `r` is variable and uniformly distributed in [0, *d* - 1]. In this example, `r < d` always holds true but the compiler does not know it and adds a precondition check before calling *mshift*, *mcomp* and their promoted variants. For clarity, we do not included *minverse* but it is worth remembering (see [Neri19]) that it is slower than the built-in algorithm except for the few divisors for which the latter spikes. More or less the same happens here for *mshift* and its promoted flavour. The good news is that both variations of *mcomp* are faster than the built-in algorithm.

Finally, Figure 4 considers the expression `n % d > 1`. (Recall that *minverse* cannot evaluate this expression.) The built-in algorithm performs worse than all others and the promoted variations perform worse than their regular counterparts.

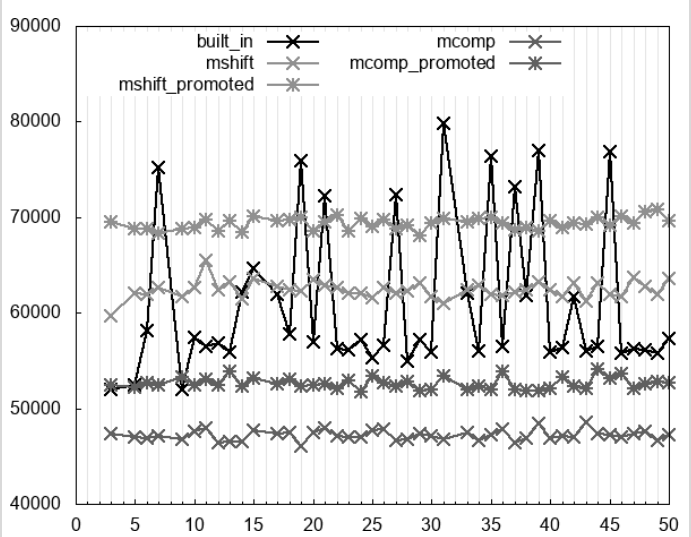


Figure 3

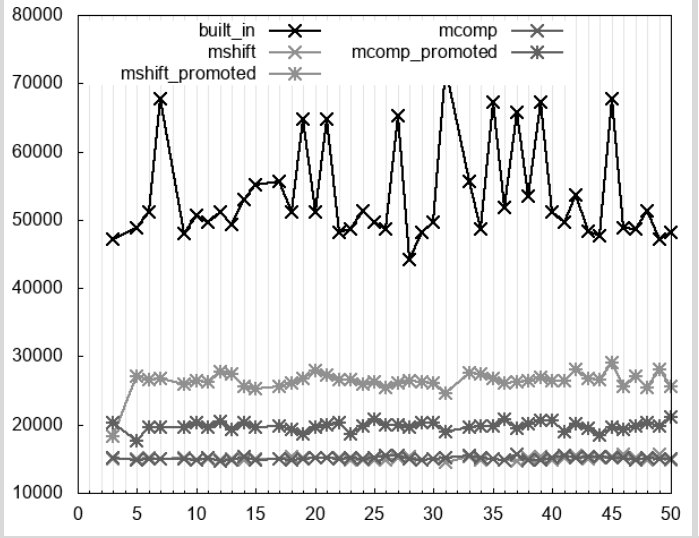


Figure 4

Related work

As we have seen, *mshift* and *mcomp* follow the idea of *RMSR* as presented in [Warren13]. Furthermore, a recent work [Lemire19] presents a slight variation of promoted *mcomp* for `uint32_t`. However, [Lemire19] restricts its discussion and analysis to divisibility tests, that is, to expressions of the form `n % d == 0`.

Conclusion

This article presents the *mshift* and *mcomp* algorithms, which make remainder comparisons without knowing its value. In contrast to *minverse*, seen in the previous instalment of this series, they allow the usage of any comparison operator. However, they have a limitation not shared by *minverse*: in general, their range of applicability is not as large as the range of the inputs' type. To address this issue, this article presents the promoted variants which take `uint32_t` inputs and make intermediate calculations on `uint64_t` values. We have seen situations where these algorithms can perform better than *minverse* and the built-in algorithm currently implemented by major compilers.

Although we have worked around the limitation on inputs by promoting `uint32_t` values to `uint64_t`, a similar idea is not viable when the input is already of the latter type. More precisely, there is no efficient promoted variant when inputs are already of type `uint64_t`. This is the greatest limitation of *mshift* and *mcomp*. In Part 3 of this series we shall see yet another algorithm that does not have this issue. ■

Acknowledgements

I am deeply thankful to Fabio Fernandes for the incredible advice he provided during the research phase of this project. I am equally grateful to the *Overload* team for helping improve the manuscript.

References

[Google] https://github.com/google/benchmark

[Lemire19] Daniel Lemire, Owen Kaser and Nathan Kurz, Faster Remainder by Direct Computation: Applications to Compilers and Software Libraries, *Software: Practice and Experience* 49 (6), 2019.

[Neri19] Cassio Neri, 'Quick Modular Calculations (Part 1)', *Overload* 154, pages 11–15, December 2019.

[qmodular] https://github.com/cassioneri/qmodular

[QuickBench] Quick C++ Benchmark: http://quick-bench.com/u3y2EZt_F8eAtWmFimt0MrwfHS8

[Warren13] Henry S. Warren, Jr., *Hacker's Delight*, Second Edition, Addison Wesley, 2013.

A Secure Environment for Running Apps?

Getting apps from the app store is easy. Alan Griffiths considers this from a security perspective.

What does 'app confinement' mean?

When you run an application on a computer you are giving it, and by extension, the developers of that application access to your computer. Unless you take precautions, it gets access to everything you can access.

Historically, there's been a high cost of entry to application development and distribution meaning that developers have had to establish a reputation and trust. While some have suspicions of what, say, Microsoft Office or Chromium does, there's no realistic fear that it will steal from you or hold information on your computer for ransom.

But the barrier to entry has become low, writing an app and getting it into the app store has never been easier and, as a result, application development is no longer the preserve of a few well-known organizations. The basis for trust that used to exist has been eroded.

At the same time, computers are being trusted with more and more sensitive information. We carry pocket computers with us everywhere and trust them to hold personal information including access to bank accounts, credit cards and medical details.

When the computer has access to your bank accounts, running code from developers that are essentially unknown to you beyond a picture of their app on the app store is risky.

Taking precautions to mitigate the risk posed by untrusted code is where app confinement comes into play. By confining the app at the operating system level it is possible to restrict its access to your computer to only those things that are needed for it to work.

How does 'app confinement' work?

As developers, we all know that something that sounds simple in the user domain can involve some serious work in the solution domain. App confinement is no exception: we need to consider what the operating system needs to do to confine an app; how that can be controlled; how the user can review and configure the confinement; and how to write and package applications so they work with restricted access to the system.

The discussion that follows talks about some specific Linux technologies for app confinement. That's for the convenience of having concrete examples that I'm familiar with, but the principles involved can be, and have been, applied with other technologies and on other operating systems.

Kernel and userspace

The code running on a computer can be divided into 'kernel' and 'userspace'. The kernel is that part of the operating system that mediates all interaction with hardware and between processes. The userspace is everything that runs within a normal app. (I know this isn't the whole story, but software development is about useful abstractions and this separation is useful for this article.)

If we write a "hello world" application, the code we write runs in userspace. And so does the output function from the library we use (maybe `operator<<()`, or `printf()` or …) but at some point it writes to the console and at that point the kernel takes over and, eventually (there may well be further userspace and kernel code executed), some pixels are lit on the screen.

While code can run without invoking the kernel it cannot produce significant effects without doing so. It can't access your files, it can't access the internet, it can't access your keyboard, mouse, touchpad, interact with other processes, etc.

That makes the interface between userspace and kernel a useful place to restrict the activities of a program.

AppArmor

The kernel enhancement I'm familiar with for implementing the confinement of apps is AppArmor. This intercepts calls to the kernel and checks to see if the app is permitted to make them. It does this based on an 'AppArmor profile' that has been applied to the app.

Like much of Linux configuration, these profiles are based on text files. These contain rules for matching resources on the system and specify the access that is permitted. For example:

```
owner /run/user/[0-9]*/wayland-[0-9]* rw,
```

allows read and write access to any files matching the pattern that have the same owner (i.e. user) as the app's process. The app cannot access files or resources unless they are allowed by a rule. (Not even if it is running as root.)

While AppArmor profiles are readable, they are not at a very convenient level of abstraction. Usually, one is concerned with, for example, enabling the playing of DVDs not with listing the various logical devices that may be needed to do so. Profiles can easily run into hundreds of lines, to take an example I am working with:

```
$ cat
/var/lib/snapd/apparmor/profiles/snap.mir-kiosk-
kodi.mir-kiosk-kodi | wc -l
1199
```

Lists of rules that are this long for each and every application are not easy to maintain nor review.

Snaps, snapcraft and snapd

AppArmor is an implementation detail of 'snap confinement', which is a component of Canonical's 'Snap' packaging format. Snaps make use of lists of AppArmor rules called 'interfaces', each of which covers identifiable capabilities. These interfaces are reviewed by the Snap developers and can be enabled (or disabled) by the end user.

Listing 1 is an example corresponding to the 1200-line AppArmor profile mentioned above:

Alan Griffiths Alan has delivered working software and development processes to a range of organizations, written for a number of magazines, spoken at several conferences, and made many friends. He can be contacted at alan@octopull.co.uk

```
$ snap connections mir-kiosk-kodi
Interface         Plug                                 Slot                Notes
alsa              mir-kiosk-kodi:alsa                  :alsa               manual
audio-playback    mir-kiosk-kodi:audio-playback        :audio-playback     -
avahi-observe     mir-kiosk-kodi:avahi-observe         :avahi-observe      manual
hardware-observe  mir-kiosk-kodi:hardware-observe      :hardware-observe   manual
locale-control    mir-kiosk-kodi:locale-control        :locale-control     manual
mount-observe     mir-kiosk-kodi:mount-observe         :mount-observe      manual
network-observe   mir-kiosk-kodi:network-observe       :network-observe    manual
opengl            mir-kiosk-kodi:opengl                :opengl             -
pulseaudio        mir-kiosk-kodi:pulseaudio            :pulseaudio         -
removable-media   mir-kiosk-kodi:removable-media      :removable-media    manual
shutdown          mir-kiosk-kodi:shutdown              :shutdown           manual
system-observe    mir-kiosk-kodi:system-observe        :system-observe     manual
wayland           mir-kiosk-kodi:wayland               :wayland            manual
```

Listing 1

The owner of the computer is in charge of the interfaces a snap connects to. Some, carefully curated, interfaces will 'auto-connect' on installation, most require the user to explicitly enable them. (There are both graphical and command-line ways to manage the connections.)

This means that, provided an app is packaged and confined as a snap, you can install it and be sure that it isn't accessing parts of your computer you do not choose to share. Instead of trusting each and every application, you just have to trust 'snap confinement'. Trusting the well-known company that provides the operating system is less of a risk than trusting 'Jo' who uploaded some interesting looking game to the app store.

Writing apps for confined environments

In principle, there is nothing very special about writing apps for confined environments. Your app will need to 'do stuff' and that implies having the permissions needed to do that stuff. In the above example, Kodi, a media player needs access to various sources of media and the devices needed for audio and video playback.

A side-effect of Snap confinement is that some directories are not in the 'expected' place and applications must respect the environment variables that locate them. For example, each snap will have its own `$HOME` directory (something like `/home/alan/snap/mir-kiosk-kodi/51`) which it can use without restrictions. So long as the application uses `$HOME` (and not something like `/home/$USER`) it can 'just work'.

Although it has its own `$HOME` a confined app has no access to the user's home directory unless the 'home' interface is connected. Even connecting this interface does not give unfettered access: it only allows access to 'normal' files and directories, it does not provide any access to hidden ones or those associated with other snapped applications.

While the details of this treatment of `$HOME` are specific to Snaps something similar is needed by any system of confinement to allow applications to work without changes. There are a few other environment variables that are adjusted for Snap confinement but (possibly with a bit of tweaking to the packaging 'recipe') most applications 'just work'.

The main thing application developers need to do is avoid requiring unnecessary capabilities for the application to run. And to be aware that some capabilities may not be enabled so that can be handled gracefully.

The Kodi media centre is a good example of this: options that rely on access to resources that are unavailable (because the user hasn't enabled those interfaces) do not appear on the menus. I don't think this is intentional support for confinement by the Kodi developers, just a by-product of it being possible to install Kodi on devices with a wide range of capabilities.

Computers are everywhere

Computers are being used in increasing numbers of internet connected devices. As well as the familiar desktops, laptops, tablets and phones there are all sorts of smart devices that are getting both an internet connection and the ability to install apps. Securing the operation of these is important for both users and developers.

As a user, it may seem cute to install a game for the kids on your car infotainment system, but you really want to be sure that it cannot misbehave and interfere with the satnav! Or, inside the home, adding apps for some local shops to the latest smart fridge could expose traffic on your home WiFi.

As developers, we have a responsibility to ensure the systems we deploy are properly protected against bad actors. Fulfilling that responsibility while opening the system to extension by, for example, installing third-party applications from a 'store' needs care.

I hope I've given a flavour of how, if the operating system is secure by design, this is possible. ■

Further reading

- **AppArmor**

  AppArmor (https://en.wikipedia.org/wiki/AppArmor) is common on Debian based distros including Ubuntu. Android and some Redhat based distros use SELinux (https://en.wikipedia.org/wiki/Security-Enhanced_Linux).

- **Snaps**

  Snaps (https://en.wikipedia.org/wiki/Snappy_(package_manager)) provide a way to package Linux applications so they can be installed and run on a wide range of distros.

  The containment tool described here is specific to snaps and differs from competing distribution-agnostic packaging technology such as AppImage (https://en.wikipedia.org/wiki/AppImage) and Flatpak (https://en.wikipedia.org/wiki/Flatpak).

Afterwood

The centre half is more than a sporting term. Chris Oldwood shows us why.

As a child, I loved playing football (nay, soccer). Every waking moment was an opportunity to get outside and play football. At school, lessons and lunch were merely a chance to catch my breath before heading back outside again at break or lunch time to play another game. The school holidays were a Utopia where football could be played from dawn to dusk with no pointless interruptions, like lessons.

The first real distraction to put a significant dent in my football time was when the father of a school friend built himself a ZX81 from a kit and we got to play on it one summer. The discovery that my next-door-but-one neighbour had a ZX80 sitting idle in their attic nibbled into more of that time, while the remainder slowly became a victim to an increase in swimming training until football became the exception instead of the norm. It wasn't until starting my first proper job at a software house where they played football together after-hours that I got my boots down from the attic and rekindled my love for the beautiful game.

I never played as a striker, even as a child, so I resumed my previous position in the defensive line, sometimes as a left- or right-back, but mostly as a centre-half or sweeper. This position tends to be frequented by players who are tall and reasonably well built and hence it can earn them the somewhat dubious title of 'Big Man at the Back'. Naturally, from such a central position you have a good view of the field of play and can coordinate the surrounding defenders and mid-fielders. Each player acts with a high degree of autonomy, adapting to the local conditions as the opposing team surges forward, but the centre-half also has one eye on the rest of the field so that the defence can adjust itself to the changing face of an attack.

Having never played (adult-wise) at anything more than Sunday League football (i.e. 'pub' football) I have no idea if this is how a real centre-half plays or is even supposed to play; it just felt natural to play that way. As a professional programmer, I have found myself gravitating towards a similar role with the various teams I've worked in over the last couple of decades. If we consider the goal to be the delivery of bug fixes and new features – the crowd-pleasing items – while the opponent is the background noise of complexity, tooling, deployment, documentation, process, etc. then you'll usually find me tackling the latter rather than ramming home the former. Naturally, I'll surface around the 6-yard box every now and then for corners and free-kicks but I'm just as happy with an assist as a name on the score-sheet.

While in the early years my own skills were growing in many directions as I uncovered the mountain of things to learn, once I had a grasp of the basic mechanics of programming I found more time to comprehend many of the more peripheral duties which make the sustained delivery of software hard. While it was never a conscious decision at first it always seemed more important to help colleagues out, where possible, rather than plough on with my own assigned work uninterrupted. Before I'd even heard of pair or mob programming I was in no doubt that I had a far more enjoyable time working with other people and so to be more knowledgeable in those areas that others were less interested in gave me an opportunity to work more closely with other people. Naturally, it's a two-way street and, much as I enjoy reading, learning from other people feels far more effective.

While I'm happy to 'watch their backs' and help the team and, by extension, the company reach a global maxima it's not a position that sits well within many large organisations due to the way they assess progress and performance. When working in a small team that sits together where the management can see you physically wandering around the desks or standing debating with colleagues across the partition it gives a very concrete view of collaboration which no doubts helps promote your value to the team, even if your personal goal tally for the season is a little on the low end.

When working entirely remotely this advantage disappears – out of sight, out of mind. With no physical presence to rely on, a management team that relies solely on metrics, perhaps one metric in particular, like the number of JIRA tickets closed, is likely to form a less-than-flattering picture of your productivity or 'value' to the team. Lest you think an organisation would be foolish to consider this approach any more valid than counting lines of code or number of commits; trust me, it happens. Hearing phrases like 'they have their own work to do' should be considered a free-kick on the edge of the penalty box. Ultimately such times of madness demand us to summon Charles Goodhart and game the system.

As a proponent of the Russian proverb 'trust, but verify' I do not expect anyone to blindly trust me any more than I expect someone to be watching my every move. Trust is also a two-way street and I expect any verifier to look past arbitrary metrics and find the fingerprints that the team leaves behind in the commit log, meetings, documentation, chat channels, etc. If someone is having an impact, their presence will be observable despite the best efforts of some products to cling to the outdated notion of one person one task.

I might have finally reached an age where nature has convinced me to hang up my boots again for good but it's not changed my position. Software teams run more smoothly when they have someone that can read the game well and provide backup and support when necessary to help them get the results they're after. ■

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 also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG