hacker bits

Issue 11

new bits

Welcome to the December issue of Hacker Bits!

If you're looking for some positivity after a grueling election season, then you've come to the right place.

Brian Gilham recounts a delightful anecdote about the importance of being kind, and we think it's just what everyone needs.

As always, our team of experts is here to offer their insights on how to become a better programmer.

In this issue...

Bill Cruise a.k.a. Bill the Lizard, gives us the lowdown on books programmers read, and ones that they claim to have read.

It's easy to feel old in an industry populated with wide-eyed twenty-somethings, but take heart and read what veteran software engineer Ben Northrop has to say about being an "old" programmer.

And lastly...

For y'all Python enthusiasts out there, Tim Abbott gives us an in depth report on the state of static types in Python.

Enjoy, and we'll see y'all back here next month.

— Maureen and Ray

us@hackerbits.com

content bits

Issue 11

contributor bits

Elliot Chance
Elliot is a data nerd and TDD enthusiast living in Sydney, Australia. He loves exploring new technologies and working on modern ways to solve age old problems. He blogs at elliot.land.

Alex Denning
Alex is a freelance blogger and marketer specialising in WordPress. Alex writes about the challenges in the digital economy every week at alexdenning.com.

Brian Gilham
Brian is a developer, writer, and teacher in Toronto, Canada. By day, he's a Mobile Team Lead at TWG. At night, he writes the Monday Mailer, his weekly newsletter about doing your best work.

Tim Abbott
Tim was the co-founder and CTO of Ksplice; after Oracle's acquisition of Ksplice, he ran the Ksplice engineering team at Oracle for a year before starting Zulip group chat, which was acquired by Dropbox in March 2014. He left Dropbox in Spring 2016 and now spends his time running the Zulip open source project.

Simon Holywell
Simon is a Senior Dev and author working at Aurion in Brisbane, AU with a passion for web app dev and motorcycles. He wrote *Functional Programming in PHP*, which helps the reader to understand functional programming within the context of a popular language that they know. @treffynnon - simonholywell.com

Ben Northrop
Ben is a Distinguished Technical Consultant at Summa. He believes that software development is fundamentally about making decisions, and he writes about this at BenNorthrop.com.

Itamar Turner-Trauring
Itamar is working on a book called *The Programmer's Guide to a Sane Workweek*, and sharing his many mistakes as a software engineer at Software Clown.

Satabdi Das
Satabdi is a software engineer with more than 7 years of experience building software for the semiconductor companies. Currently, she is a Recurser at the Recurse Center where she is busy learning assembly, debugger internals and compiler details and raft.

Henrique Bucher
Henrique owns Vitorian LLC which provides PhD-level expertise in Financial Technology: front-end algorithms (strategies), backend/tick data warehouse, Ultra Low Latency techniques, low level assembly, Modern C++/software and reconfigurable hardware (FPGA).

Bill Cruise
Bill blogs about programming, math, learning, and technology at billthelizard.com. You can follow him @lizardbill.

Ray Li
Curator

Ray is a software engineer and data enthusiast who has been blogging at rayli.net for over a decade. He loves to learn, teach and grow. You'll usually find him wrangling data, programming and lifehacking.

Maureen Ker
Editor

Maureen is an editor, writer, enthusiastic cook and prolific collector of useless kitchen gadgets. She is the author of 3 books and 100+ articles. Her work has appeared in the *New York Daily News*, and various adult and children's publications.

What makes a great software engineer?

By ELLIOT CHANCE

This is something I've personally thought a lot about. Not just for the purposes of finding other engineers to work with, but also to discover a metric that allows me to better myself and my own skill set. After much thought this is what I've come up with...

> *"Given the same amount of time, a better software engineer will write less code. The best engineer will write zero code."* — Elliot Chance

There's three parts to this, let me explain...

1. Simplicity reduces code

It's quite common for software changes to be fast and easy at the start of a project and become more difficult and slow as the project progresses. This is for a variety of reasons but one of those is usually complexity. As the software has more moving parts there's more things to maintain, more potential for bugs, higher build and testing times... the list goes on and that's before we take into account all the other less ideal things like poorly written code, business pressures and myriad of other things that push software inevitably forward.

> *"I didn't have time to write a short letter, so I wrote a long one instead."*
> — Mark Twain

Using more code is actually the lazy fix. It is often not the best solution, for now and in the future. Take more time to find a more simple solution, then hopefully less code is produced.

2. Experience reduces code

I believe software engineers go through three phases:

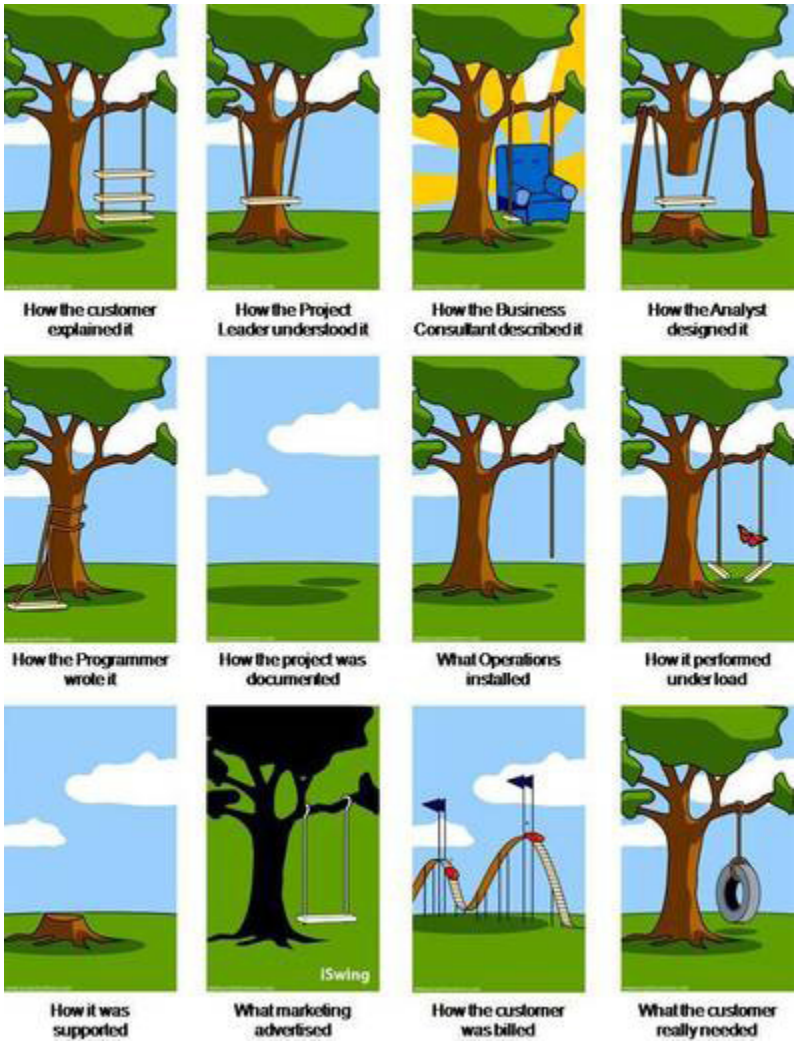
1. *Thrown in the deep end.* We've all been here. This is when you're new and absorbing as much as possible. You probably want to do the right thing by yourself and/or your employers but your lack of experience and time pressures make it hard to produce high quality software because you simply haven't made much of it yet. You will (or at least should) make lots of mistakes, this is how you learn. Hopefully you are surrounded by engineers that give you the right support and guidance.
2. *Too smart for your own good.* This is when you think you have mastered your software stack, language or framework. It tends to produce software that is overly clever with a high focus on trying to write lots of code that reinvents the wheel, contains a lot of "magic" code or goes against the norm because "I can do it better". Perhaps this is a natural over-compensation for all the mistakes made in the past, or a self-denial that allows us to justify that we don't need to learn anything new. I know I am guilty of this more times that I can count.
3. *Engineering maturity.* If you're looking back on your work and saying to yourself things like "Why did I make this so much more complicated than I needed to be?" or "Why didn't I research how others have solved this before I had a crack at it?" and you truly understand these are actual flaws in yourself as an engineer then you can move on and write much better solutions going forward.

> *"Programs must be written for people to read, and only incidentally for machines to execute."*

I think one of the biggest misconceptions with software engineering is that we are coding machines. After all, that's what we do right? Give us a problem and we'll write you some code to fix it. I believe

this is incorrect. Our job is to solve software problems to a level of quality and maintainability that suits the environment we are working in. That doesn't always have to involve writing code. Code is the glue that ties together the rest of our experience and knowledge, it's not the most important thing to focus on.

3. Understanding reduces code



What is the worst by-product of creating and maintaining software? Most people would say 'technical debt'. Technical debt can be many things but a large part of it is the source code. I want to emphasise that *it's not always poorly written code*. Standards, frameworks and even languages change over time. Code that was written very well at the time may simply be unusable or incompatible years later and require large refactoring.

If you have been writing code for several years I'm sure you could look back at an earlier time when all you wanted to do at the start of a problem was jump in and write some code. The drive to find the best technical solution is what makes most of us love what we do as software engineers. At the same time your former self will also admit that you may have dived in without fully understanding what was needed for the solution - this causes a lot more code to be created.

Code cannot easily be undone, both while it's being written and even more so after it has been published. **Even the best written code will become technical debt at some point**, even if it's just the next engineers (or yourself) trying to understand what it does.

Communication is key. Not only between the engineers and the non-technical people driving the project forward, but also between the engineers themselves. If you can spend more time finding a better way that requires less code you have found a better solution.

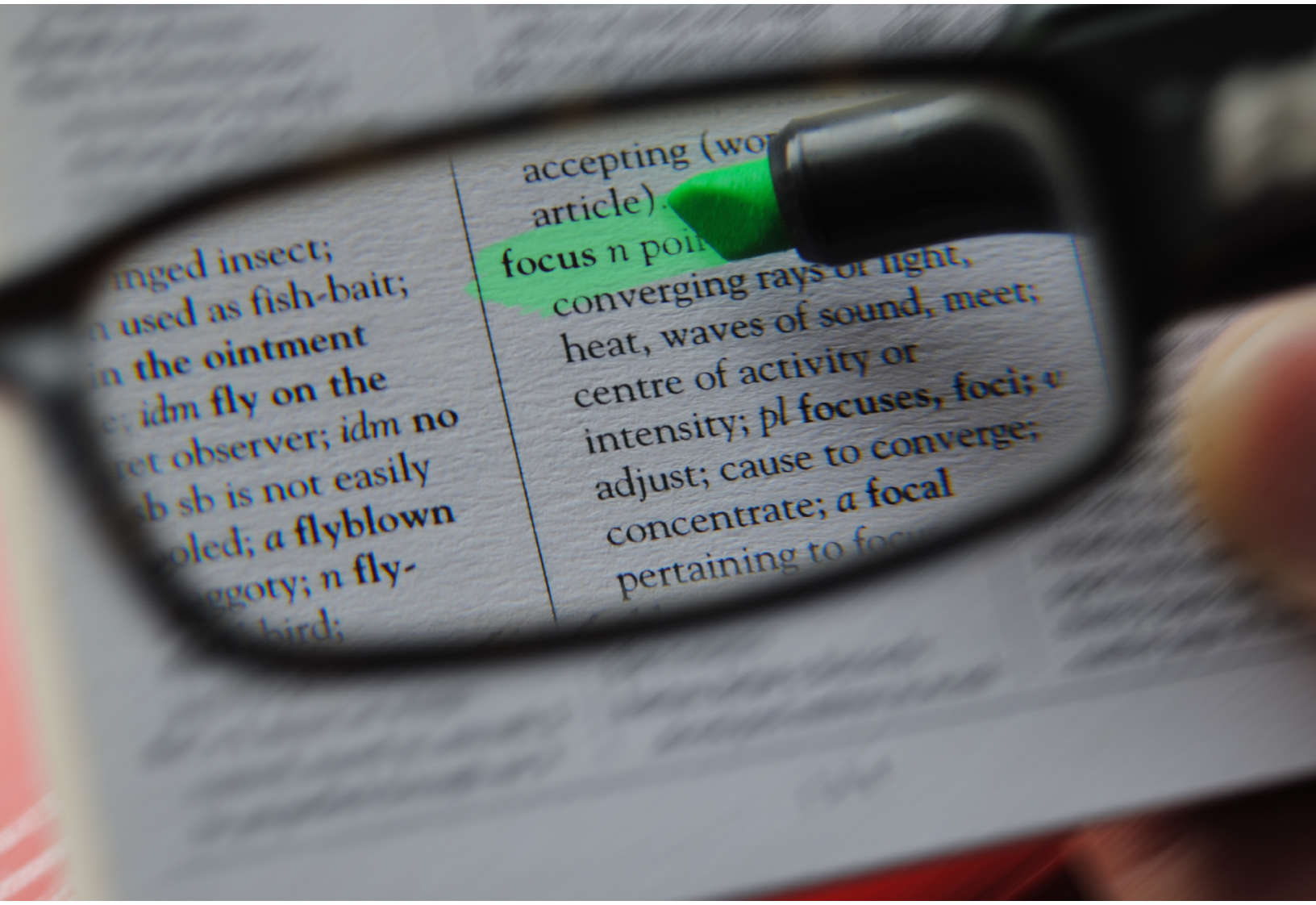
In closing

In one sense you could say the moral of the story is to avoid all your natural instincts as a software engineer. Maybe, but above all, KISS.

I understand that in a lot of this I am over generalising and there are plenty of exceptions to the rule. It's also an unobtainable goal for a software engineer to write very little code in their day-to-day work, but if you can write a little less, that's a little less that can go wrong and lot less that needs to be maintained in the future.

Thank you for reading. I'd love to hear your thoughts and perspective, or any other unrelated feedback. Please leave your comments below or consider subscribing. **Happy coding.** ■

Reprinted with permission of the original author. First appeared at elliot.land.



Deep work in practice

By ALEX DENNING

> *One of the more embarrassing and self-indulgent challenges of our time is the task of relearning how to concentrate. The past decade has seen an unparalleled assault on our capacity to fix our minds steadily on anything. To sit still and think, without succumbing to an anxious reach for a machine, has become almost impossible. - Alain de Botton*

My recent survey of the last 50 days of my five minute journalling showed I have a serious problem with "deep work".

Deep work refers to Cal Newport's thesis (he expands on it vastly in his excellent book) that:

> *[Deep work is] cognitively demanding activities that leverage our training to generate rare and valuable results, and that push our abilities to continually improve… [. Deep work results in] improvement of the value of your work output… [and] an increase in the total quantity of valuable output you produce.*

This is contrasted with "shallow work", the tasks that "almost anyone, with a minimum of training, could accomplish" such as checking emails, planning, social media etc.

I read Deep Work more or less in one sitting on and it summarised much of my pre-held thoughts on productivity, I just hadn't adopted to the same extent as Prof. Newport advocates. I think the basic thesis is very strong and I'm yet to find a better blueprint for work in the modern economy.

So why am I so bad at *doing* deep work? And how can you start using it in practice?

Are we getting more distracted?

Deep work requires prolonged periods of concentration on hard problems. For me that typically means writing of some kind and project planning. For you that may mean something similar; any sort of computer work and the concept stands.

As Alain de Botton commented up top, we've experienced an *unparalleled assault on our capacity to fix our minds steadily on anything* in recent years and this is a problem.

There are no end of op-eds and studies arguing we are hopelessly hooked and most people I know have problems with internet addiction and we're creating a culture of distraction – and the economic data says the same thing: the US output per hour (a standard measure of labour productivity) has decreased from an annual rate of 3% between 1945 and the 1970s to 0.5% since 2010.

The latest annualised productivity growth rate was *minus* 0.4%.

The Internet "is designed to be an interruption system", and we are "addicted to distraction", argues Nicholas Carr. As a 90s kid I've grown up with the internet and all its benefits – but also can't help myself from switching tabs the second I need to think about something difficult.

Social media and apps use the same principles as slot machines: intermittent reinforcement. When you pull to refresh Twitter or Facebook or your email you don't know what you're going to get – it may be nothing or it may be a really cool email. The randomness makes the action of checking addictive. Technology companies know and use this and we are ill-equipped to defend ourselves.

Distraction seems to be a well-established problem but nobody wants to take it seriously or knows how to fix it.

How do I generally reduce distractions, anxiety and noise?

We've made two theses so far:

1. The type of work that is valuable in the modern economy involves long periods of serious work and focus.
2. We are easily distracted and find prolonged period of focus difficult.

Identifying these as problems is important and valuable in itself, but real progress will come from being able to properly put deep work into practice.

Trying to break the "distraction habit" is hard and this is by no means a comprehensive or exhaustive list, but this is what I've been doing and what has worked for me.

Be less distracted in general:

- Delete social media apps from my phone (that aim to be addictive). I can still access from the mobile browser if I want, but its inconvenience puts me off.
- Ban my phone from the toilet. Yup. This is actually a big one.
- Stop keeping my phone near my bed. Check your email before you get out of bed? If you can't reach it, you can't.
- Stop carrying my phone in my pocket. Keeping it in my bag instead makes it less convenient and me less prone to picking it up.

Be less distracted at work:

- Recognise when I'm doing deep work and need to focus (this is useful for stopping yourself when you're tempted to change tab).
- Install Timewarp for Chrome. This doesn't aggressively block websites (although you can if you wish), but puts up a timer for selected sites, showing you how much time you've spent on a site that day. Much more useful than sweary alternatives*.
- Put my phone on do not disturb and keep it off my desk.
- Use full screen mode in app and browser.
- Remove the bookmarks bar from Chrome.
- Close my email app and only open at set times.
- Listen to more classical music.

I've long been a fan of the Pomodoro Technique and relied on it almost exclusively for the last two years of my degree, but I've not found it such a useful ally in the quest for deep work. It's a prop for concentration rather than an outright fix – and I want to be able to fix my focus problem rather than cover it up. The Pomodoro timer (my favourite is Pomotodo) mawy come out again in the future, but for now I'm leaving it.

*I've received lots of recommendations for Chrome extensions after publishing this. A couple of

note: Inbox When Ready hides your inbox by default, so you can compose email without getting distracted. News Feed Eradicator removes the news feed from Facebook (or just use Messenger.com to get chat-only Facebook instead). Finally, Forest offers an intense tree-killing alternative to Timewarp. See what works for you.

How do I get deep work done?

Two more things to think about. Your work schedule and what you're working on.

The former is pretty simple: set aside time for working out what you're working on, so when you sit down to work it's just a case of getting on with the deep work. No time faffing working out what to work on.

The second appears simple but is more complex to fix: you need to be working on projects interesting enough that they can demand your attention. Something to think about.

Embracing deep work

Distraction is a problem. We're probably reliant on or addicted to the internet more than we'd like to admit. Fixing this will be a work in progress, but acting now, recognising the problem and consciously trying to fix it is as good a first step as any.

For your convenience here are some of the key takeaways:

- Deep work requires prolonged focus on hard things.
- We're addicted to the internet and distractions. Certain apps are especially bad for this.
- Reducing distractions in general and at work is helpful.
- Timewarp for Chrome is helpful.
- Plan your schedule in advance.
- This is all a lot easier if you're working on interesting and important things.

There may be a magical fix for this. I'll let you know if and when I find it. ■

Be kind

By BRIAN GILHAM

One Friday afternoon, early in my career, I was wrapping up some new features for the back-end of a client's Rails app. Simple stuff. Confident in my work, I deployed the changes, closed my laptop, and drove out of town for a weekend of camping with friends. I had just arrived when my phone rang. It was my project lead, Kevin.

> *"The client's site is down. What happened?"*

Oh shit. Fuck. I had no idea. I was three hours away with no laptop.

"Don't worry about it," he said. "I'll take care of it. Have a good weekend."

Like that was going to happen. I'd let the team down. I'd ruined someone else's weekend. I beat myself up for days. Come Monday; I walked into the office certain I was about to be fired. The project lead walked over.

> *"Hey, Brian. How was your trip?"*

He was smiling. There wasn't even a hint of frustration or annoyance. "It was okay," I said, waiting for the bad news. "Sorry about Friday. I completely blew it."

"It's okay," he replied. "We've all done it." He paused for a moment. "But what did you learn?"

I talked about the need for proper QA. About thoroughly testing my changes. About taking the time to make sure the job gets done right. After a few minutes, he held up his hand.

> *"Great. It sounds like you get it. I know that you can do better."*

And that was the end of it. Kevin never brought it up again.

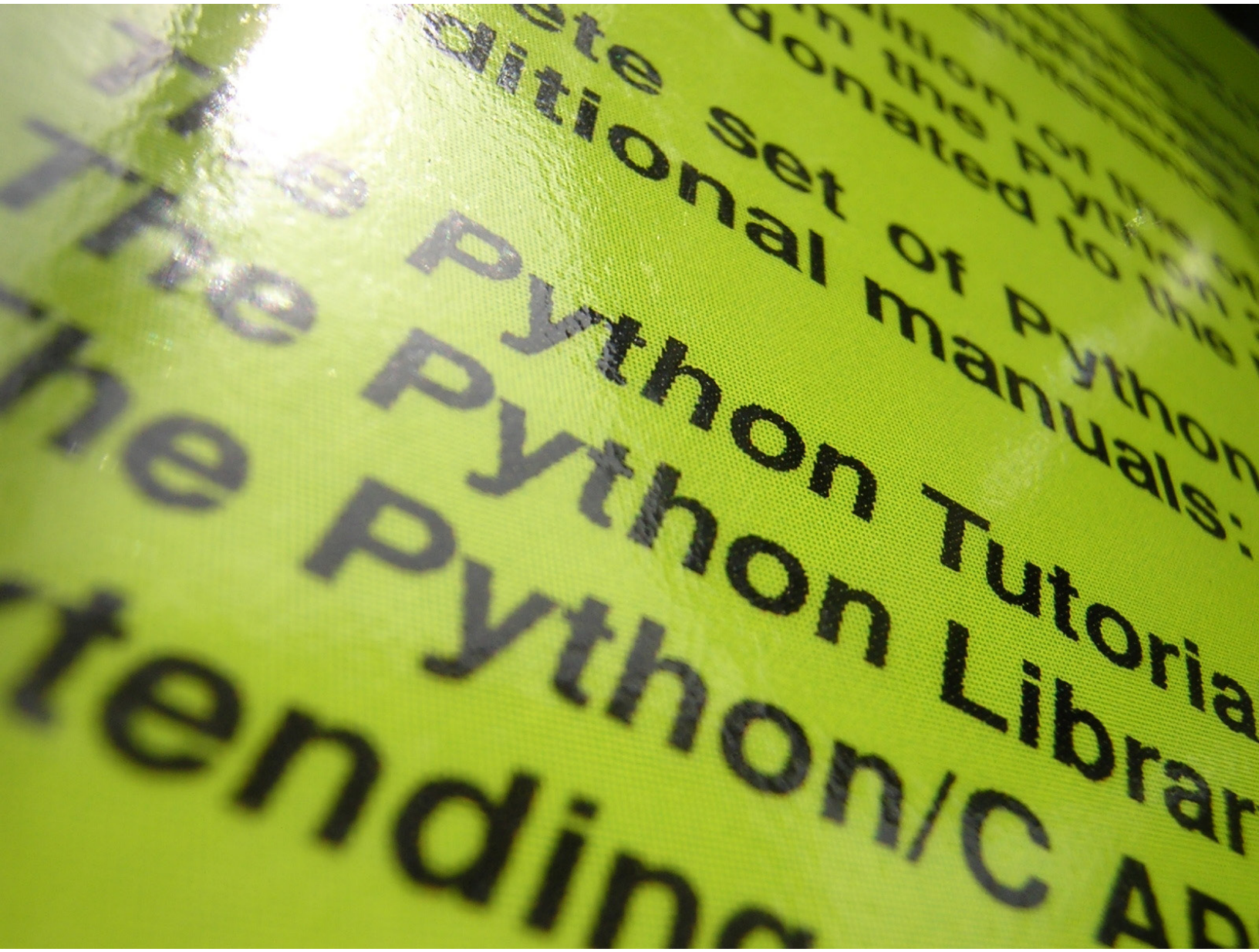
Kevin gave me the space to screw up, as long as I learned from it. He jumped in, with his years of experience, and helped me out when I needed it most. And still believed I was a competent developer, despite my mistake. He saw my potential.

Now that I'm the one leading projects and mentoring junior developers, I often think back to that day. And I remind myself to be kind and see the potential in people. Give them a break.

Just like Kevin did for me. ■

Join the Monday Mailer

I write exclusively for my mailing list. No spam – just my thoughts on shipping side projects, doing your best work, and improving your craft. Each and every week.
https://briangilham.com/newsletter/



Static types in Python

By TIM ABBOTT

Over the last few years, static type checkers have become available for popular dynamic languages like PHP (Hack) and JavaScript (Flow and TypeScript), and have seen wide adoption. Two years ago, a provisional syntax for static type annotations was added to Python 3. However, static types in Python have yet to be widely adopted, because the tool for checking the type annotations, mypy, was not ready for production use… until now!

The exciting news is that over the last year, a team at Dropbox (including Python creator Guido van Rossum!) has led the development of mypy into a mature type checker that can enforce static type consistency in Python programs. For the many programmers who work in large Python 2 codebases, the even more exciting news is that mypy has full support for type-checking Python 2 programs, scales to large Python codebases, and can substantially simplify the upgrade to Python 3.

The Zulip development community has seen this in action during 2016. Zulip is a popular open source group chat application, complete with apps for all major platforms, a REST API, dozens of integrations, etc. To give you a sense of scale, Zulip is about 50,000 lines of Python 2, with dozens of developers contributing hundreds of commits every month. During 2016, we have annotated 100% (!) of our backend with static types using mypy, and thanks to mypy, we are on the verge of switching to Python 3. Zulip is now the largest open source Python project that has fully adopted static types, though I doubt we'll hold that title for long :).

In this post, I'll explain how mypy works, the benefits and pain points we've seen in using mypy, and share a detailed guide for adopting mypy in a large production codebase (including how to find and fix dozens of issues in a large project in the first few days of using mypy!).

A brief introduction to mypy

Here is an example of the clean mypy/PEP-484 annotation syntax in Python 3:

```
def sum_and_stringify(nums: List[int]) -> str:
    """Adds up the numbers in a list and returns the result as a string."""
    return str(sum(nums))
```

And here's how that same code looks using the comment syntax for both Python 2 and 3:

```
def sum_and_stringify(nums):
    # type: (List[int]) -> str
    """Adds up the numbers in a list and returns the result as a string."""
    return str(sum(nums))
```

With this comment syntax, mypy supports type-checking normal Python 2 programs, and programs annotated using mypy will run normally with any Python runtime environment (mypy shares this excellent property with the Flow JavaScript type checker). This is awesome because it means projects can adopt mypy without changing anything about how they run Python.

You run mypy on your codebase like a linter, and it reports errors in a nice compiler-style format. For example, here's what the mypy output would be if I'd incorrectly annotated `sum_and_stringify` as returning a float:

```
$ mypy /tmp/test.py
/tmp/test.py: note: In function "sum_and_stringify":
/tmp/test.py:6: error: Incompatible return value type: expected builtins.float,
got builtins.str
```

If you're curious how to annotate something, check out the mypy syntax cheat sheet (for simple things) and PEP-484 (for complex things); they're great resources. If you want to play with mypy now, you can install it with `pip3 install mypy-lang`.

When mypy has complete type annotations for a module as well as its dependencies, then it can provide a very strong consistency check, similar to what the compiler provides in statically typed languages. Mypy uses the typeshed repository of type "stubs" (type definitions for a module in the style of a header file) to provide type data for both the Python standard library and dozens of popular libraries like requests, six, and sqlalchemy. Importantly, mypy is designed for gradually adding types; if type data for an import isn't available, it just treats that import as being consistent with anything.

Benefits of using mypy

Here are the benefits we've seen from adopting mypy, starting with the most important:

- **Static type annotations improve readability.** Improved readability is probably the most important benefit of statically typed Python. With static types, every function clearly documents the types of its arguments and return value, which saves a ton of time reading the codebase to figure out the precise types a function expects so that you can call it. In many large codebases, developers write docstrings that contain little information other than the types of the arguments and often end up out of date. Type annotations are far better since they are automatically verified for correctness and consistency. It turned out that the code that we found most difficult to annotate was precisely the code where the type annotations improved readability the most: the cases where it was totally unclear from the code alone what all the objects being passed around are.
- **We can refactor with confidence.** Refactoring a Python codebase without breaking things involves a lot of careful manual checking to confirm that you've updated all the code that used the old interface. With a statically typed Python codebase, the type checker can often verify this for you automatically. We've already found this to be super useful when refactoring Zulip.
- **We upgraded to Python 3.** Thanks to mypy, Zulip now supports Python 3. For years, the 2to3 library has been able to do automatically most of the changes required to support Python 3. The hard part of the migration is the unicode issue: Python 3 is much more strict about the distinction between arrays of bytes and strings than Python 2. Mypy annotations made this project much easier, since we could explicitly declare which values were which across the entire codebase, and check those declarations for consistency. Even in Zulip, which has an unusually comprehensive automated test suite, mypy caught many Python 3 issues that the tests did not find.
- **mypy frequently catches bugs.** When we started using mypy, Zulip was a mature web application with unusually high test coverage, so we didn't find any super embarrassing bugs as part of annotating the codebase. However, mypy flagged dozens of latent bugs, and dozens more pieces of confusing code. I'm glad that mypy helped us purge those categories of issues from the codebase. More important for us is that running mypy on code that hasn't yet been reviewed or tested regularly saves time by catching bugs.

- Static types highlight bad interfaces. Annotated functions with messy interfaces (e.g sometimes returning a `List` and sometimes a `Tuple`) really stand out.

Pain points

To provide a complete picture of the experience of adopting mypy, I think it's important to also talk about the pain points with mypy today:

- **Interactions with "unused import" linters:** Currently, linters like `pyflakes` don't read the Python 2 type annotations style (since the annotations are comments, after all!), and thus will report that imports needed for mypy are unused. Since cleaning up unused imports is only marginally useful anyway, we just filtered out those warnings. This problem will go away when we move to the new Python 3 type syntax, since then type annotations are visible "users" of the imports to tools like pyflakes; I also expect that popular Python linters will add an option to check imports in type comments before long.
- **Import cycles:** We needed to create a few import cycles in order to import types that were used as arguments (but not explicitly referenced) in a file. One can work around this (in the Python 2 comment syntax) using e.g.`if False: from x import y` (soon to be the cleaner `if typing.TYPE_CHECKING`), but it won't work when we move to the Python 3 syntax. I'm hopeful that someone will come up with a nice solution to the import cycle creation issue, which may just take the form of recommendations on how to organize one's code to avoid this.

Non-pain points

This section discusses the things that (before trying mypy) I was worried might be problems, but that after adopting mypy I don't think are significant issues:

- **Training.** Zulip has had well over 100 contributors at this point, so the thing I was most worried about when adopting mypy was that it could hurt the project by adding a new obscure technology that contributors need to learn. This has not turned out to be a material problem: new contributors usually annotate their code correctly on the first try. Python programmers know the Python type system, and the PEP-484 type syntax is a pretty intuitive way of writing it down. The mypy cheat sheet helps a lot by providing a single place to look up all the common stuff.
- **False positives.** Mypy is well-engineered and designed around the idea of only reporting things that are actual inconsistencies in a program's types. It has a far lower false positive rate than the commercial static analyzers I've used! And, it has a really nice `type: ignore` system for silencing false positives to get clean output. Finally, we've had good luck with typing even some highly dynamic parts of our system (e.g. our framework for parsing REST API requests).
- **Mypy and typeshed bugs.** Zulip started using mypy in January 2016, when mypy itself was essentially the only open source project using mypy. Because we were the first webapp and the second large project using mypy seriously, we encountered a lot of small bugs at the beginning (in total, Zulip developers have reported ~50 issues in mypy and submitted 16 PRs for typeshed). Essentially all of those issues were fixed upstream (with tests!) months ago, and we rarely encounter regressions. So, I wouldn't expect similar projects to encounter a similar scale of issues; exactly how many issues an individual project encounters will depend a lot on whether that project uses the same corners of Python as other projects that already use mypy and typeshed. Even as a super early adopter, I feel like we spent far more time annotating our code than investigating, working around with `type: ignore` , and reporting bugs.
- **Performance.** Mypy today is really fast compared to other static analyzers I've used. With the mypy module cache, it takes about 3s to check the entire Zulip codebase and thus is about as fast as pyflakes (one of the faster Python linters). However, because mypy detects cross-file is-

sues (unlike most linters), it doesn't have a really fast (e.g. <100ms) way to fully check if a given diff/commit introduced a new issue.

- Community. The mypy and typeshed communities are amazing and are impressively responsive to good bug reports (aka ones with a clean, simple reproducer, which I can almost always generate in under 5 minutes).

Finding bugs in your first few days with mypy

This section details what you need to do in order to start benefiting from mypy in a large codebase. To give you a sense of the scale of work involved, I did everything described in this section in 4 days at a hackathon in January (and back then, mypy wasn't mature, so I spent half the time filing good bug reports for all the issues I found). If you're thinking about using mypy but want more data to help make a decision, I'd recommend completing the steps discussed in this section. It's a pretty good effort/value tradeoff.

Read the mypy cheat sheet. The mypy cheat sheet provides a great overview of the PEP-484 syntax, and you'll be referring to it often as you start writing annotations.

Standardize how you'll run mypy. Write tooling to install and run `mypy` against your codebase, so that everyone using the project can run the type checker the same way. Two features are important in how you run mypy:

- Support for determining which files should be checked (a whitelist/exclude list is useful!).
- Specifying the correct flags for your project at this time. For a Python 2 project, I recommend starting with `mypy --py2 --silent-imports --fast-parser -i <paths>`. You should be able to do this using a mypy.ini file.

Get mypy running on your codebase with clean output. This usually requires just adding type annotations for empty global data structures. Back in January, this took a few hours of work (including reporting bugs with reproducers). It's probably even less work now. By default, mypy will only check the interior of functions that are annotated, so with an unannotated codebase, this is just making sure mypy can analyze your entire codebase.

Check for basic consistency. Add the `--check-untyped-defs` option to your mypy arguments, and get that running with no errors on the codebase. This option causes mypy to check every `def` in the codebase for internal consistency; mypy can detect many classes of bugs and mistakes in your codebase, without your having written a single type annotation!
In most cases, you'll want to fix bugs and bad code as you go, but you can also use `#type: ignore` annotations or exclude files to defer issues. For example, we excluded all of Zulip's tests at first, since they're both lower value to type-check and had most of the monkey-patching and other questionable Python in the project. For Zulip, getting clean `--check-untyped-defs` output took me about 2 days of hard work, including merging fixes for about 40 issues in the Zulip codebase.

I spent another day or two generating nice reproducers for the mypy bugs I'd encountered and improving typeshed. Now that mypy is no longer in its infancy, mypy bugs are increasingly rare. But a large project should expect to encounter and fix typeshed issues (just submit a PR!).

Run mypy in continuous integration. Once `mypy --check-untyped-defs` passes on your codebase, you should lock down your progress by running the `mypy` checker in your CI environment.

Because mypy type annotations are optional, once you've done the setup above, you can start annotating your codebase at whatever pace you like. Over time, you'll get the benefits of static types in the parts of the codebase that you've annotated, without needing to do anything special to the rest of the codebase (the wonders of gradual typing!). In the next section, I'll discuss strategy for how to move the codebase towards being fully annotated.

Fully annotating a large codebase

This section discusses what's involved in going from having mypy set up to having a fully annotated codebase.

The great thing about mypy is that you can do everything gradually. After the initial setup, we actually didn't do anything with mypy for a couple months. That changed when we posted mypy annotations as one of our project ideas for Google Summer of Code (GSoC). We found an incredible student, Eklavya Sharma, for the project. Eklavya did the vast majority of the hard work of annotating Zulip, including upgrading our tooling, annotating the core library, contributing bug reports and PRs to mypy and typeshed upstream, and fixing all the mistakes we made early on. Amazingly, he also found the time during the summer to migrate Zulip to use virtualenvs and then upgrade Zulip to Python 3!

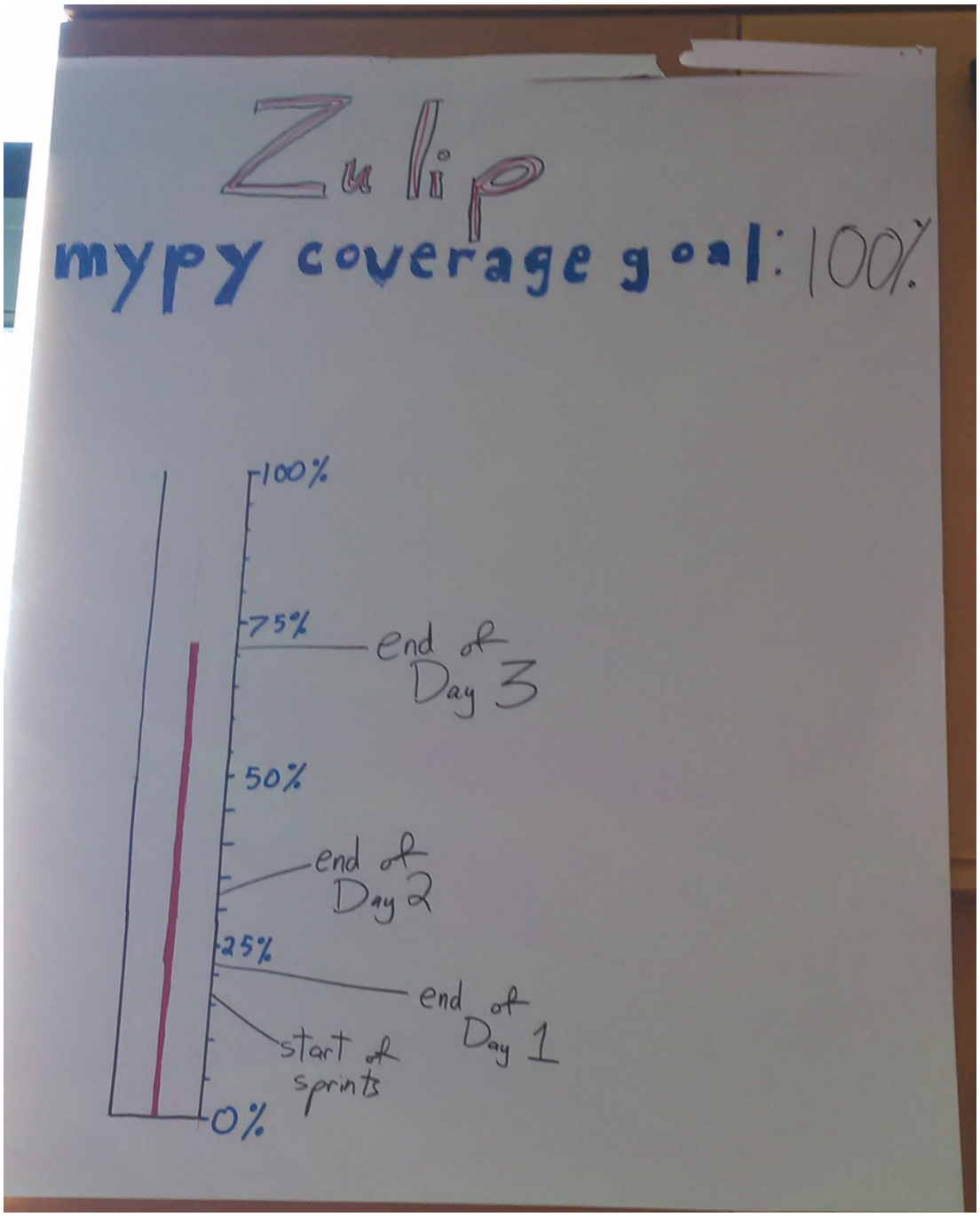
One can divide the work of annotating a large project into a few phases.

Phase 1: Annotate the core library. Strategically, you want to annotate the core library code that is used in lots of other files first. The annotations for these functions impose constraints on the types used in the rest of the codebase, so you'll spend less time fixing incorrect annotations and catch more actual bugs faster if you do these files first. This is also a good time to document how your project is using mypy (and link that doc from mypy failures in the CI system).

Phase 2: Annotate the bulk of the codebase. Many projects will likely do this slowly over many months as developers touch the different parts of the codebase, which is a pretty reasonable strategy.

It also works well to annotate a codebase as a focused effort; the story of how we did this for Zulip is instructive. About halfway through Eklavya's summer of code, we went to the PyCon sprints with the goal of annotating as much of Zulip as possible. The PyCon sprints are my favorite part of PyCon. It's an awesome 4-day party after the main PyCon conference where hundreds of developers work on open source projects together. It's completely free to attend, and is a great opportunity to get involved in contributing to open source projects.

We grabbed a few tables next to the mypy developers, and managed to attract a rotating cast of 5–10 developers to the Zulip mypy annotation project each day. During the PyCon sprints, Zulip went from 17% annotated to 85% (with 25–30 engineer-days of work, mostly by folks new to both Zulip and mypy). We used mypy's coverage support and coveralls.io to track progress, but the more fun progress bar was our giant sheet of paper, pictured here at the start of the last day:



Our PyCon experience is I think the best proof that mypy is accessible to new developers: aside from me, all of the contributors adding annotations were new to both Zulip and mypy. We found that with a good 5-minute demo and good documentation, new contributors were effective within their first hour of working with mypy. I would definitely recommend the mypy hackathon approach to other open source projects — it's a great way for contributors to have meaningful impact on an unfamiliar project.

Phase 3: Get to 100%. Annotating the last several files is relatively difficult work, because this is where you end up debugging all the mistakes you made in Phase 2. While you do this, it's important to lock

down files/directories that reach 100% by adding the `--disallow-untyped-defs` option (which will report any functions missing type annotations) to the `mypy` flags to prevent regressions.

Eklavya brought us from 85% to 96% before his college started up again, and then a few weeks ago we spent the couple hours needed to get to 100%. Now, all new Python code going into Zulip comes with mypy annotations (with the exception of a shrinking list of scripts, settings, and test files).

Phase 4: Celebrate and write a blog post! At least, that was the next step for Zulip :)

Overall, the focused time that went into fully annotating Zulip was a 1-week hackathon, a GSOC project, and a party at the PyCon sprints. In the scheme of things, this is a pretty modest level of effort.

I should mention that even though Zulip is 100% annotated, Zulip's mypy journey is not complete. We will eventually want to add stubs to typeshed for the most important libraries used by Zulip (e.g. Django).

Recommendations for annotating code

We have a few recommendations for annotating that will likely save you a lot of time:

- Make sure to handle `str` vs. `Text` correctly. Bytes vs. str and str vs. unicode errors are the majority of the problem for moving a codebase from Python 2 to Python 2+3. If you do this right as you annotate your codebase, you'll save yourself a lot of time when you upgrade to Python 3; we found the Python 3 upgrade went very quickly once we had the codebase mostly annotated. We ended up adding a couple helper functions to do casts between `str` , `Text` , and `bytes` correctly (and readably!) in our codebase on both Python 2 and Python 3.
- Remember to annotate class variables! We neglected to do this when we first annotated our Django models file, and ended up having to fix a number of incorrect annotations (primarily around Text vs. bytes) that got into the codebase because they weren't being cross-checked against anything.
- Avoid using a guess-and-check approach when adding annotations. In a partially annotated codebase, mypy will still detect many classes of errors in annotations, but it can't detect every error (since the inconsistency might be with code you haven't annotated yet). So you should make sure people writing annotations are actually understanding/tracing the code, and that you code review the annotations just like actual code. We found writing one commit per large file (or collection of related smaller files) to be a good commit discipline.
- Use precise types where possible (e.g. avoid using `Any`).
- When using `type: ignore` to work around a potential mypy or typeshed bug, I recommend using the following style to record the original GitHub issue: bad_code # type: ignore # https://github.com/python/typeshed/issues/372
- That way, it's easy for future you to verify whether the issue that required that use of `type: ignore` has since been fixed upstream. If a file would require a lot of `type: ignore` annotations, you can always add it to the exclude list (another feature of our `run-mypy` wrapper) and plan to come back to it later.

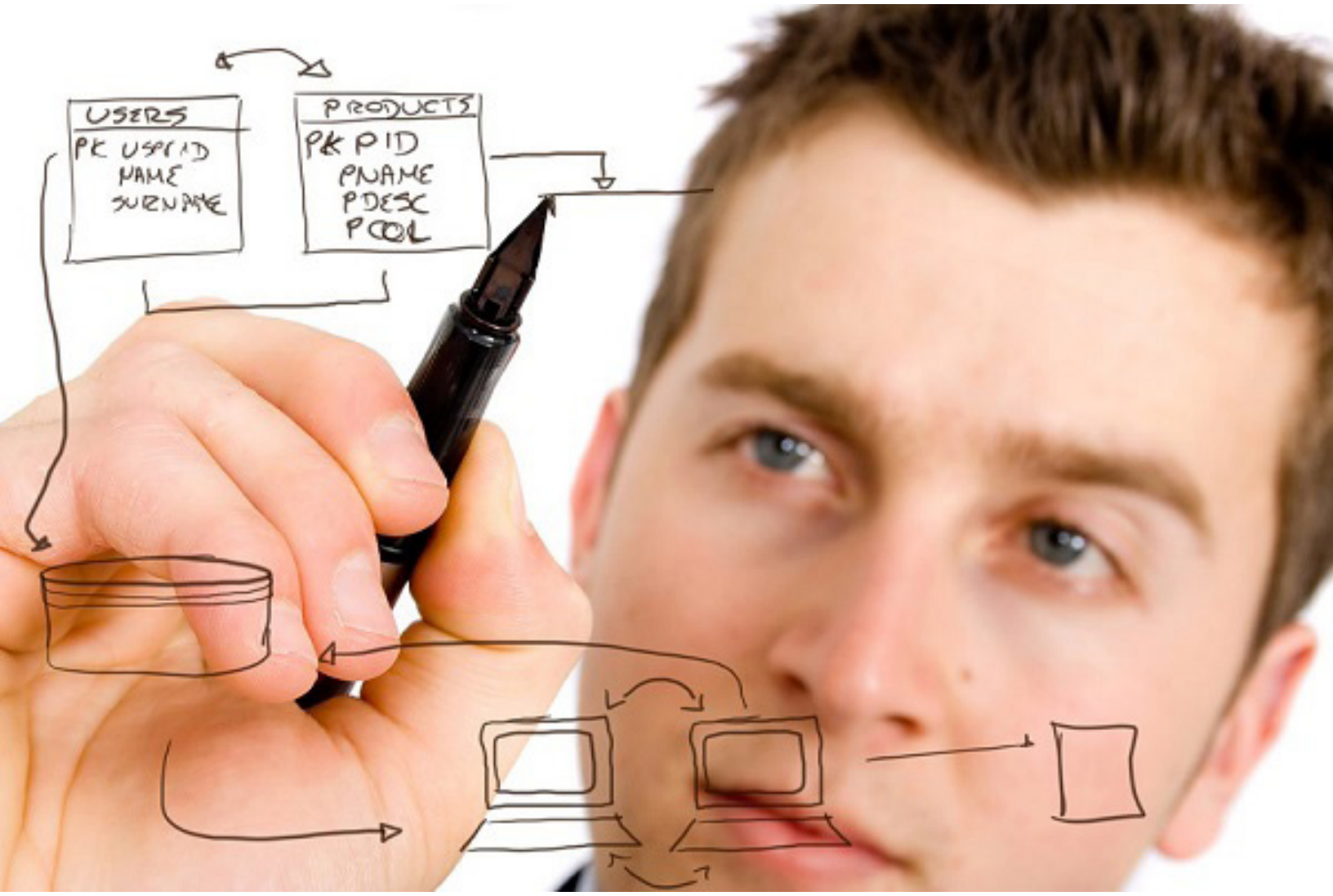
Conclusion

Overall, the experience of using mypy (and the PEP-484 type system) has been awesome, and we feel that adopting mypy has been a big advance for the Zulip project. It improves readability, catches bugs in code without running it, has very few false positives, and hasn't come with significant downsides. Deploying mypy in a large codebase required relatively little investment on our part, and annotating the codebase had the side benefit of making our Python 3 migration feel easy.

If you have a large Python codebase and would like to make working in your codebase better, you should find a week to get started with mypy!

Finally, if you're excited to see what static types in Python look like in a large codebase, check out the Zulip server project on GitHub. We welcome new contributors!

Huge thanks to Guido van Rossum, Alya Abbott, Steve Howell, Jason Chen, Eklavya Sharma, Anurag Goel, and Joshua Simmons for their feedback on this post. ■



SQL style guide

By SIMON HOLYWELL

Overview

You can use this set of guidelines, fork them or make your own - the key here is that you pick a style and stick to it. To suggest changes or fix bugs please open an issue or pull request on GitHub.

These guidelines are designed to be compatible with Joe Celko's SQL Programming Style book to make adoption for teams who have already read that book easier. This guide is a little more opinionated in some areas and in others a little more relaxed. It is certainly more succinct where Celko's book contains anecdotes and reasoning behind each rule as thoughtful prose.

It is easy to include this guide in Markdown format as a part of a project's code base or reference it here for anyone on the project to freely read—much harder with a physical book.

General

Do

- Use consistent and descriptive identifiers and names.
- Make judicious use of white space and indentation to make code easier to read.
- Store ISO-8601 compliant time and date information (`YYYY-MM-DD HH:MM:SS.SSSSS`).
- Try to use only standard SQL functions instead of vendor specific functions for reasons of portability.
- Keep code succinct and devoid of redundant SQL—such as unnecessary quoting or parentheses or WHERE clauses that can otherwise be derived.
- Include comments in SQL code where necessary. Use the C style opening /* and closing */ where possible otherwise precede comments with -- and finish them with a new line.

```
SELECT file_hash  -- stored ssdeep hash
  FROM file_system
 WHERE file_name = '.vimrc';
/* Updating the file record after writing to the file */
UPDATE file_system
   SET file_modified_date = '1980-02-22 13:19:01.00000',
       file_size = 209732
 WHERE file_name = '.vimrc';
```

Avoid

- CamelCase—it is difficult to scan quickly.
- Descriptive prefixes or Hungarian notation such as `sp_` or `tbl`.
- Plurals—use the more natural collective term where possible instead. For example `staff` instead of `employees` or `people` instead of `individuals`.
- Quoted identifiers—if you must use them then stick to SQL92 double quotes for portability (you may need to configure your SQL server to support this depending on vendor).
- Object oriented design principles should not be applied to SQL or database structures.

Naming conventions

General

- Ensure the name is unique and does not exist as a reserved keyword.
- Keep the length to a maximum of 30 bytes—in practice this is 30 characters unless you are using multi-byte character set.
- Names must begin with a letter and may not end with an underscore.
- Only use letters, numbers and underscores in names.
- Avoid the use of multiple consecutive underscores—these can be hard to read.
- Use underscores where you would naturally include a space in the name (first name becomes `first_name`).
- Avoid abbreviations and if you have to use them make sure they are commonly understood.

```
SELECT first_name
  FROM staff;
```

Tables

- Use a collective name or, less ideally, a plural form. For example (in order of preference) `staff` and `employees`.
- Do not prefix with `tbl` or any other such descriptive prefix or Hungarian notation.
- Never give a table the same name as one of its columns and vice versa.
- Avoid, where possible, concatenating two table names together to create the name of a relationship table. Rather than `cars_mechanics` prefer `services`.

Columns

- Always use the singular name.
- Where possible avoid simply using `id` as the primary identifier for the table.
- Do not add a column with the same name as its table and vice versa.
- Always use lowercase except where it may make sense not to such as proper nouns.

Aliasing or correlations

- Should relate in some way to the object or expression they are aliasing.
- As a rule of thumb the correlation name should be the first letter of each word in the object's name.
- If there is already a correlation with the same name then append a number.
- Always include the `AS` keyword—makes it easier to read as it is explicit.
- For computed data (`SUM()` or `AVG()`) use the name you would give it were it a column defined in the schema.

```
SELECT first_name AS fn
  FROM staff AS s1
  JOIN students AS s2
    ON s2.mentor_id = s1.staff_num;
SELECT SUM(s.monitor_tally) AS monitor_total
  FROM staff AS s;
```

Stored procedures

- The name must contain a verb.

- Do not prefix with `sp_` or any other such descriptive prefix or Hungarian notation.

Uniform suffixes

- The following suffixes have a universal meaning ensuring the columns can be read and understood easily from SQL code. Use the correct suffix where appropriate.
- `_id`—a unique identifier such as a column that is a primary key.
- `_status`—flag value or some other status of any type such as `publication_status`.
- `_total`—the total or sum of a collection of values.
- `_num`—denotes the field contains any kind of number.
- `_name`—signifies a name such as `first_name`.
- `_seq`—contains a contiguous sequence of values.
- `_date`—denotes a column that contains the date of something.
- `_tally`—a count.
- `_size`—the size of something such as a file size or clothing.
- `_addr`—an address for the record could be physical or intangible such as `ip_addr`.

Query syntax

Reserved words

Always use uppercase for the reserved keywords like `SELECT` and `WHERE`.

It is best to avoid the abbreviated keywords and use the full length ones where available (prefer `ABSOLUTE` to `ABS`).

Do not use database server specific keywords where an ANSI SQL keyword already exists performing the same function. This helps to make code more portable.

```
SELECT model_num
  FROM phones AS p
 WHERE p.release_date > '2014-09-30';
```

White space

To make the code easier to read it is important that the correct compliment of spacing is used. Do not crowd code or remove natural language spaces.

Spaces

Spaces should be used to line up the code so that the root keywords all end on the same character boundary. This forms a river down the middle making it easy for the reader's eye to scan over the code and separate the keywords from the implementation detail. Rivers are bad in typography, but helpful here.

```
SELECT f.average_height, f.average_diameter
  FROM flora AS f
 WHERE f.species_name = 'Banksia'
    OR f.species_name = 'Sheoak'
    OR f.species_name = 'Wattle';
```

Notice that `SELECT`, `FROM`, etc. are all right aligned while the actual column names and implementation specific details are left aligned.

Although not exhaustive always include spaces:

- before and after equals `(=)`
- after commas `(,)`
- surrounding apostrophes (`'`) where not within parentheses or with a trailing comma or semicolon.

```
SELECT a.title, a.release_date, a.recording_date
  FROM albums AS a
 WHERE a.title = 'Charcoal Lane'
    OR a.title = 'The New Danger';
```

Line spacing

Always include newlines/vertical space:

- before `AND` or `OR`
- after semicolons to separate queries for easier reading
- after each keyword definition
- after a comma when separating multiple columns into logical groups
- to separate code into related sections, which helps to ease the readability of large chunks of code.

Keeping all the keywords aligned to the righthand side and the values left aligned creates a uniform gap down the middle of query. It makes it much easier to scan the query definition over quickly too.

```
INSERT INTO albums (title, release_date, recording_date)
VALUES ('Charcoal Lane', '1990-01-01 01:01:01.00000', '1990-01-01
01:01:01.00000'),
       ('The New Danger', '2008-01-01 01:01:01.00000', '1990-01-01
01:01:01.00000');
UPDATE albums
   SET release_date = '1990-01-01 01:01:01.00000'
 WHERE title = 'The New Danger';
SELECT a.title,
       a.release_date, a.recording_date, a.production_date -- grouped dates to-
gether
  FROM albums AS a
 WHERE a.title = 'Charcoal Lane'
    OR a.title = 'The New Danger';
```

Indentation

To ensure that SQL is readable it is important that standards of indentation are followed.

Joins

Joins should be indented to the other side of the river and grouped with a new line where necessary.

```
SELECT r.last_name
  FROM riders AS r
       INNER JOIN bikes AS b
       ON r.bike_vin_num = b.vin_num
          AND b.engines > 2
```

```
       INNER JOIN crew AS c
       ON r.crew_chief_last_name = c.last_name
          AND c.chief = 'Y';
```

Subqueries

Subqueries should also be aligned to the right side of the river and then laid out using the same style as any other query. Sometimes it will make sense to have the closing parenthesis on a new line at the same character position as it's opening partner—this is especially true where you have nested subqueries.

```
SELECT r.last_name,
       (SELECT MAX(YEAR(championship_date))
          FROM champions AS c
         WHERE c.last_name = r.last_name
           AND c.confirmed = 'Y') AS last_championship_year
  FROM riders AS r
 WHERE r.last_name IN
       (SELECT c.last_name
          FROM champions AS c
         WHERE YEAR(championship_date) > '2008'
           AND c.confirmed = 'Y');
```

Preferred formalisms

- Make use of `BETWEEN` where possible instead of combining multiple statements with `AND`.
- Similarly use `IN()` instead of multiple `OR` clauses.
- Where a value needs to be interpreted before leaving the database use the `CASE` expression. `CASE` statements can be nested to form more complex logical structures.
- Avoid the use of `UNION` clauses and temporary tables where possible. If the schema can be optimised to remove the reliance on these features then it most likely should be.

```
SELECT CASE postcode
       WHEN 'BN1' THEN 'Brighton'
       WHEN 'EH1' THEN 'Edinburgh'
       END AS city
  FROM office_locations
 WHERE country = 'United Kingdom'
   AND opening_time BETWEEN 8 AND 9
   AND postcode IN ('EH1', 'BN1', 'NN1', 'KW1')
```

Create syntax

When declaring schema information it is also important to maintain human readable code. To facilitate this ensure the column definitions are ordered and grouped where it makes sense to do so.

Indent column definitions by four (4) spaces within the `CREATE` definition.

Choosing data types

- Where possible do not use vendor specific data types—these are not portable and may not be

available in older versions of the same vendor's software.

- Only use `REAL` or `FLOAT` types where it is strictly necessary for floating point mathematics otherwise prefer `NUMERIC` and `DECIMAL` at all times. Floating point rounding errors are a nuisance!

Specifying default values

- The default value must be the same type as the column — if a column is declared a `DECIMAL`, do not provide an `INTEGER` default value.
- Default values must follow the data type declaration and come before any `NOT NULL` statement.

Constraints and keys

Constraints and their subset, keys, are a very important component of any database definition. They can quickly become very difficult to read and reason about though so it is important that a standard set of guidelines are followed.

Choosing keys

Deciding the column(s) that will form the keys in the definition should be a carefully considered activity as it will affect performance and data integrity.

1. The key should be unique to some degree.
2. Consistency in terms of data type for the value across the schema and a lower likelihood of this changing in the future.
3. Can the value be validated against a standard format (such as one published by ISO)? Encouraging conformity to point 2.
4. Keeping the key as simple as possible whilst not being scared to use compound keys where necessary.

It is a reasoned and considered balancing act to be performed at the definition of a database. Should requirements evolve in the future it is possible to make changes to the definitions to keep them up to date.

Defining constraints

Once the keys are decided it is possible to define them in the system using constraints along with field value validation.

General

- Tables must have at least one key to be complete and useful.
- Constraints should be given a custom name excepting `UNIQUE`, `PRIMARY KEY` and `FOREIGN KEY` where the database vendor will generally supply sufficiently intelligible names automatically.

Layout and order

- Specify the primary key first right after the `CREATE TABLE` statement.
- Constraints should be defined directly beneath the column they correspond to. Indent the constraint so that it aligns to the right of the column name.
- If it is a multi-column constraint then consider putting it as close to both column definitions as possible and where this is difficult as a last resort include them at the end of the `CREATE TABLE` definition.
- If it is a table level constraint that applies to the entire table then it should also appear at the end.

- Use alphabetical order where `ON DELETE` comes before `ON UPDATE`.
- If it makes sense to do so, align each aspect of the query on the same character position. For example all `NOT NULL` definitions could start at the same character position. This is not hard and fast, but it certainly makes the code much easier to scan and read.

Validation

- Use `LIKE` and `SIMILAR TO` constraints to ensure the integrity of strings where the format is known.
- Where the ultimate range of a numerical value is known it must be written as a range `CHECK()` to prevent incorrect values entering the database or the silent truncation of data too large to fit the column definition. In the least it should check that the value is greater than zero in most cases.
- `CHECK()` constraints should be kept in separate clauses to ease debugging.

Example

```
CREATE TABLE staff (
    PRIMARY KEY (staff_num),
    staff_num      INT(5)       NOT NULL,
    first_name     VARCHAR(100) NOT NULL,
    pens_in_drawer INT(2)       NOT NULL,
                   CONSTRAINT pens_in_drawer_range
                   CHECK(pens_in_drawer >= 1 AND pens_in_drawer < 100)
);
```

Designs to avoid

- Object oriented design principles do not effectively translate to relational database designs—avoid this pitfall.
- Placing the value in one column and the units in another column. The column should make the units self-evident to prevent the requirement to combine columns again later in the application. Use `CHECK()` to ensure valid data is inserted into the column.
- EAV (Entity Attribute Value) tables—use a specialist product intended for handling such schema-less data instead.
- Splitting up data that should be in one table across many because of arbitrary concerns such as time-based archiving or location in a multi-national organisation. Later queries must then work across multiple tables with `UNION` rather than just simply querying one table. ■

Reflections of an "old" programmer

By BEN NORTHROP

I'm a programmer, a few months shy of his 40th birthday. It's Saturday morning, my kids are home with my wonderful wife (who is pulling my weight on the domestic front), and I'm at a tech conference. It's a session on React Native, and the presenter is convincing us why it's truly the "next big thing" for mobile development. To me, it seems a bit like JSPs of 15 years ago, with all the logic in the presentation code, but I'm "old", so I assume I just don't "get it".

The presenter blows through some slides, dazzles us with some impressive live coding, and then makes his way to the "name dropping" portion of the presentation, where he rattles off about a half a dozen supporting tools that I never knew existed, including something called Pepperoni (seriously). As someone who just recently got the hang of Angular, it all makes me feel a little disheartened. "Here we go again", I think.

Of course I'm not really surprised. Over the past 20 years, I've taken seats on a number of band wagons, and have generally enjoyed the rides. The buzz that comes with a new "disruption" in programming can be exciting – feeling a part of a community of technical innovators, championing something that will make things a little easier, quicker, cleaner, better. It can be fun. But on this particular morning, at the cusp of 40, I have to admit I feel a little drained. I know this is part of the job – if I want to stay relevant (and well paid), I know that every so often I need to cast out some of the knowledge that I've so dutifully absorbed, and gear up up for the next journey. It's just how it is.

As I think about it though, this regular ritual of my programming career doesn't seem to be a way of life for other professionals. The doctor at 40 doesn't seem to be worried about discovering that all his knowledge of the vascular system is about to evaporate in favor of some new organizing theory. The same goes for the lawyer, the plumber, the accountant, or the English teacher. While there are certainly unappealing aspects to these professions, it's safe to say that for each of them, mid-way through their career, the knowledge they've accumulated is relatively stable, and has afforded them with some increased measure of respect and compensation. In programming though, 20 years of experience does not seem to confer those same advantages.

Two main forces

Of course not all is so dismal in our profession – there are so many things to love about being a programmer – but in terms of the never-ending struggle to "keep up", it is an interesting feature that seems more or less unique to our field. Am I right though? Is programming really different in this regard? And if it is, then why? And what does it mean for our career trajectory? I'd like to try to answer *all* of this (because, why not) in terms of two concepts.

The first is **knowledge decay**. Everything we know, not just about programming, has an expiration date; a point at which it is no longer useful. I learned how to drive a car when I was 16, and for that most part, that knowledge still serves me well. This piece of knowledge could be said to have a long half-life. For many professionals, their domain knowledge also has a relatively long half-life. Sure, new discoveries in medicine may displace some existing procedures, but likely there will not be a major overhaul in our understanding of our biology. When the expiration is long like this, knowledge can effectively be considered cumulative. The doctor is *more* knowledgeable than he was last year, because everything he learned in the past 12 months built on all that he knew before.

In programming, for good or bad, I'd assert that this is not exactly the case. Putting a (rather arbitrary) stake in the ground, I'd say that:

> *Half of what a programmer knows will be useless in 10 years.*

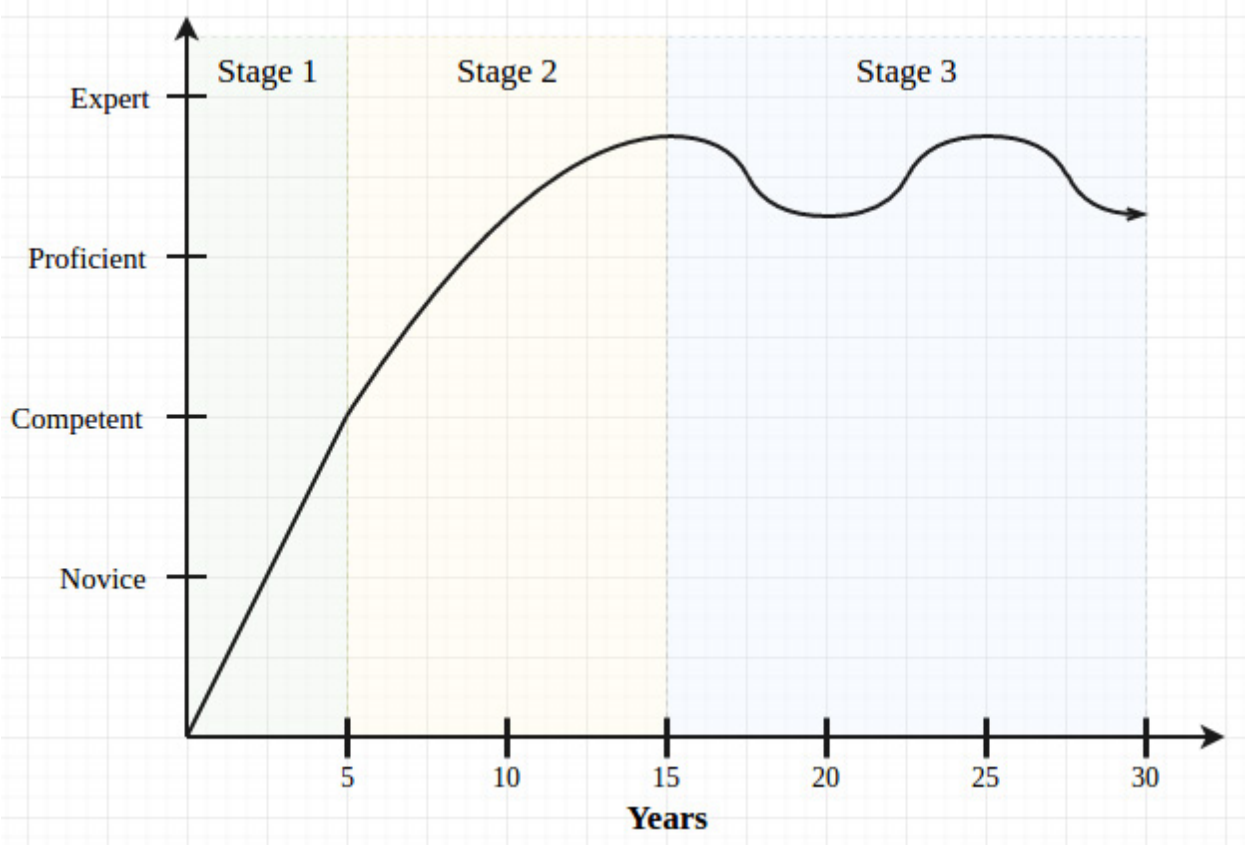
This could be way off (and there are many caveats of course – read on!)...but it seems about right for me. If I learned nothing else from this point forward, I bet that only about a half of my knowledge could

I still use in 2026 (long live SQL!), and the other half would probably be of no use (React Native, perhaps?). Now of course I *will* be gaining new knowledge to replace the dead stuff, but will it be enough? Will I know more (useful) knowledge in 2026 than I do now?

This brings me to the second concept, **knowledge accumulation rate** – the pace at which we add new things to our knowledge corpus. In every field, there is a certain threshold of knowledge that must be met in order to be "certified" (or at least hireable), and the early portion of a career is typically dedicated to acquiring this knowledge. In programming, however, because of the fast decay of knowledge, it seems like we never really transcend the "student" period. We know we must always be learning, and this makes the stages of our career a bit atypical.

The three stages

If I were to graph an average programmer's knowledge over the course of their career, keeping in mind knowledge decay and accumulation rate, I think it might look like something this:



In the beginning of our careers, in what we could call the **eager apprentice** stage, accumulating knowledge is relatively easy. Everything is new, and so each experience is a vehicle to gain more knowledge. Moreover, since we're younger, we often have fewer hard obligations, and so we probably don't mind spending a few nights and weekends picking up new languages and frameworks. Lastly, and importantly, the expectations on us from our employers is lower. Everyone understands that we're junior, and so more easily than our colleagues, we can carve out a little time during the work day to fill in holes in our knowledge. This is a fun stage, but there's this persistent feeling that there's so much we don't know.

At some point though we cease to be novices, and we establish ourselves as productive, self-sufficient

developers. For the first time, the gap between us and our colleagues (even the ones 10 years our senior!) does not seem so large. This fills us with vim and vigor, and so this is the **rising star** stage. The investment we made in learning has paid off, and just about everything we know is still useful – i.e. none of our knowledge has noticeably decayed. With this reservoir full of relevant knowledge, we begin to earn the respect of clients, peers, and managers, and with this respect comes titles, salary, and opportunities. Though we don't necessarily see it at the time, this is also an important point of inflection.

It's at this point that two things happen. First, that promotion to "senior" comes with something more than just money: greater expectations. Employers need their star programmers to be leaders – to help junior developers, review code, perform interviews, attend more meetings, and in many cases to help maintain the complex legacy software they helped build. All of this is eminently reasonable, but it comes, subtly, at the expense of our knowledge accumulation rate. The time we used to have to read tech blogs: gone. Second, it's also at this point that we first experience (or at least recognize) a little knowledge decay. Some of what we learned early in our career is now outdated. All that time "we" (read: I) spent learning GWT? Lost! Essentially, both forces, knowledge decay and knowledge accumulation rate, begin to work against us.

It's at this point where we enter the third and final stage, the ebb-and-flow of the **steady veteran**. We are knowledgeable and productive, yes, but we also understand that we may actually know fewer (useful) things than we did at a prior point in our career. A non-trivial amount of our knowledge has decayed, and we may not have had the time to accumulate enough new knowledge to compensate. This can be frustrating, and I think it's why it's at this point that so many of us bail for other pastures – management, sales, testing, or (my dream) farming. We realize that it'll require real effort to just maintain our level proficiency – and without that effort, we could be *worse* at our jobs in 5 years than we are today. There is no coasting.

Humble advice

This is where I'm at. I still love to learn, but I appreciate that without some herculean effort, I will probably always remain in an equilibrium state hovering around the lower boundary of "expert". I'm ok with this, because I enjoy my personal life more than I want to be the next Martin Fowler (although I bet Martin has a kick-ass personal life too – that guy is amazing). Thinking about my career in terms of knowledge decay and accumulation though has changed my perspective a little.

First, I try to **take the long view**. I'm more wary of roles with excessively taxing expectations and few opportunities for novel experiences. I've seen quite a few colleagues take the bigger pay check at an employer where there'll be little opportunity to work with new things and learn. In 5 years, they realize that much of their valuable knowledge has evaporated and their pay is way out of whack with their *actual* worth. In some cases, I think making less money in the short term (at a better employer) will yield more money (and stability) over the course of a long career.

Second, given that time is limited, I try to **invest most in knowledge that is durable**. My energy is better spent accumulating knowledge that has a longer half-life – algorithms, application security, performance optimization, and architecture. Carving out niches in these areas, I hope, will better bullet-proof my career than learning the newest, flash-in-the-pan Javascript library.

In the end, perhaps I haven't really forged any new ground here, but it's been useful for me to think about my career in terms of these two things: knowledge decay and knowledge accumulation. I'd love to hear any thoughts you have! ■

Reprinted with permission of the original author. First appeared at bennorthrop.com.

The programmer's guide to a sane workweek

By ITAMAR TURNER-TRAURING

I'm working on a book: ***The Programmer's Guide to a Sane Workweek****, a guide to how you can achieve a saner, shorter workweek. If you want to get a free course based on the the book signup in the email subscription at the end of the post. Meanwhile, here's the first excerpt from the book:*

- *Are you tired of working evenings and weekends, of late projects and unrealistic deadlines?*
- *Do you have children you want to see for more than just an hour in the evening after work?*
- *Or do you want more time for side projects or to improve your programming skills?*
- *In short, do you want a sane workweek?*

A sane workweek is achievable: for the past 4 years I've been working less than 40 hours a week.

Soon after my daughter was born I quit my job as a product manager at Google and became a part-time consultant, writing software for clients. I wrote code for 20-something hours each week while our child was in daycare, and I spent the rest of my time taking care of our kid.

Later I got a job with one of my clients, a startup, where I worked as an employee but had a 28-hour workweek. These days I work at another startup, with a 35-hour workweek.

I'm not the only software engineer who has chosen to work a saner, shorter workweek. There are contractors who work part-time, spending the rest of their time starting their own business. There are employees with specialized skills who only work two days a week. There are even entrepreneurs who have deliberately created a business that isn't all-consuming.

Would you like to join us?

If you're a software developer working crazy hours then this book can help you get to a saner schedule. Of course what makes a schedule sane or crazy won't be the same for me as it is for you. You should spend some time thinking about what exactly it is that *you* want.

How much time do you want to spend working each week?

- 40 hours?
- 32 hours?
- 20 hours?
- Or do you never want to work again?

Depending on what you want there are different paths you can pursue.

Some paths to a saner workweek

Here are some ways you can reduce your workweek; I'll cover them in far more detail in later chapters of the book:

Normalizing your workweek

If you're working a lot more than 40 hours a week you always have the option of unilaterally normalizing your hours. That is, reducing your hours down to 40 hours or 45 hours or whatever you think is fair. Chances are your productivity and output will actually increase. You might face problems, however, if your employer cares more about hours "worked" than about output.

Reducing overhead

Chances are that the hours your employer counts as your work are just part of the time you spend on your job. In particular, commuting can take another large bite out your free time. Cut down on commuting and long lunch breaks and you've gotten some of that time back without any reduction in the hours your boss cares about.

Negotiating a shorter workweek at your current job

If you want a shorter-than-normal workweek you can try to negotiate that at your current job. Your manager doesn't want to replace a valued, trained employee: hiring new people is expensive and risky. That means you have an opening to negotiate shorter hours. This is one of the most common ways software engineers I know have reduced their hours.

Find a shorter workweek at a new job

If you're looking for a 40-hour workweek this is mostly about screening for a good company culture as part of your interview process. If you want a shorter-than-normal workweek you will need to negotiate a better job offer. That usually means your salary but you can sometimes negotiate shorter working hours. This path can be tricky; I've managed to do it, but have also been turned down, and I know of other people who have failed. It's easier if you've already worked for the company as a consultant, so they know what they're getting. Alternatively if your previous (ideally, your current) job gave you a shorter workweek you'll have better negotiating leverage.

Long-term contracts

Instead of working as an employee you can take on long-term contract work, often through an agency. The contract can specify how many hours you will work, and shorter workweeks are sometimes possible. You can even get paid overtime!

Consulting

Instead of taking on long-term work, which is similar in many ways to being an employee, you go out and find project work for yourself. That means you need to spend something like half your time on marketing. By marketing well and providing high value to your clients you can charge high rates, allowing you to work reasonable hours.

Product business

All the paths so far involved exchanging money for time, in one form or another. As a software engineer you have another choice: you can make a product once and easily sell that same product multiple times. That means your income is no longer directly tied to how many hours you work. You'll need marketing and other business skills to do so, and you won't just be writing code.

Early retirement

Finally, if you don't want to work ever again there is the path of early retirement. That doesn't mean you can't get make money afterwards; it means you no longer *have* to make a living, you've earned enough that your time is your own. To get there you'll need very low living expenses, and a high saving rate while you're still working. Luckily programmers tend to get paid well.

Which path will you take?

Each of these paths has its own set of requirements and trade-offs, so it's worth considering which one fits your needs. At different times of your life you might prefer one path, and later you might prefer another. For example, I've worked as both a consultant and a part-time employee.

What kind of work environment do you want right now?

- Do you want to work from your spare bedroom?
- Do you like having co-workers?
- Do you want to start your own business?
- Do you want to just code, or do you want to expand your skills beyond that?

A later chapter will cover choosing your path in more detail. For now, take a little time to think it through and imagine what your ideal job would be like. Combine that with your weekly hours goal you should get some sense of which path is best for you.

It won't be easy

Working a sane workweek is not something corporate culture encourages, at least in the US. That means you won't be following the default, easy path that most workers do: you're going to need to do some work to get to your destination. In later chapters I'll explain how you can acquire the prerequisites for your chosen path, but for now here's a summary:

- You'll need to get your engineering skills to a place where you're both productive and can work independently. As an employee this will help you negotiate with your employer. As a contractor or consultant it will help get you work.
- You'll need to reduce your living expenses. You can then afford to work fewer hours, and the larger your savings in the bank the more time you can take to look for a new job. Plus it makes for a better negotiating position.
- You'll need to be able to negotiate successfully, whether it's with your employer or with clients.
- Finally, you'll need to have the self-confidence or stubbornness to choose and stick to a path that most people don't take.

How much do you really want to work a sane workweek? Do you care enough to make the necessary effort?

It won't be easy, but I think it's worth it.

Shall we get started? Sign up here to get a free course that will take you through the first steps of your journey. ■

How breakpoints are set

By SATABDI DAS

I am fascinated by the debuggers. I love them so much that I wrote a small and very basic debugger as one of my projects recently. In this post I am going to write down what I've learned about how can a debugger set a breakpoint.. This post can be divided into these following sections.

1. What's a breakpoint?
2. What's a debugger?
3. What does the debugger need to do to set a breakpoint?
4. How can the debugger make the debuggee process halt?

What's a breakpoint?

A breakpoint makes your program stop whenever a certain point in the program is reached.

What's a debugger?

You can consider your debugger to be a program which forks() to create a child process and then calls execl() to load the process we want to debug. I used execl() in my code, but any of the system calls from the exec family of functions can be used.

```
20    pid = fork();
21    if (pid == 0) {
22      // child process
23      run_child(argv[1]);
24    } else if (pid >= 1) {
25      // parent process
26      run_debugger(pid);
```

The debugger process

And here's the run_child() function which calls the execl() with the debuggee process's executable name and path.

```
34
35  void run_child(const char* childProg) {
36    if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
37      perror("Error in ptrace\n");
38      return;
39    }
40    execl(childProg, childProg, NULL);
41  }
```

We see a call to ptrace() in run_child() function before calling execl(). Let's, for the moment, not go into what ptrace() is, even though it's very important to understand how does a debugger work. We will eventually come to it.

Now we have two processes running:

1. The debugger as the parent process.
2. And the debugee as the child process.

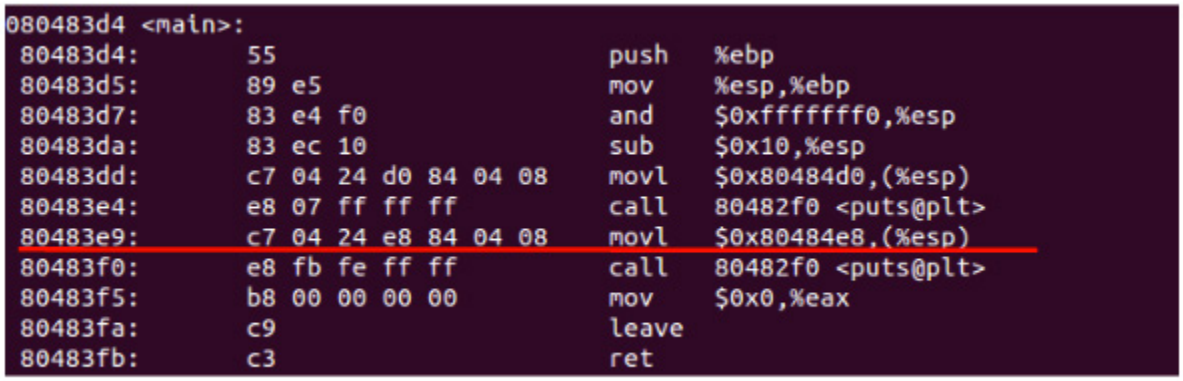
Let's now try to think abstractly in a sort of hand-wavy manner what does a debugger need to do to set a breakpoint in the child process. The debugger needs the child process to stop where the breakpoint has been set. But how?

What does the debugger need to do to set a breakpoint?

Let's examine the phrase "setting a breakpoint" a little closely. We say a process is running when the instructions of the process are being executed by the processor. **Where are those instructions located?** In the text/code section of the process's virtual memory.

By setting a breakpoint we expect the debuggee process to halt at a certain point. Which means that we expect the debuggee process to stop just before executing some instruction. What can that instruction be? Well, if the user has set a breakpoint at the beginning of a function, it's the first instruction at the start of the function. If the user has set a breakpoint at some line number in some file, the instruction is the first instruction in the series of instructions that correspond to that line.

Therefore the debugger needs to make the process halt right before executing that instruction.
In my project, I chose the instruction underlined in the following screenshot.



How can the debugger make the debuggee process halt right before executing a particular instruction?

The debugger replaces the instruction (at least part of the instruction) at the start of the debuggee process with an instruction that generates a software interrupt. Hence, when this modified instruction is executed by the processor, the SIGTRAP signal is delivered and that is all that is needed to make the process stop. I have skipped a lot of details here, but we will discover them as we go.

But let's first discover **how does the debugger modify an instruction?**

The instructions reside at the process's text section which is mapped to the virtual memory when a process is loaded. Hence to modify the instruction the debugger needs to know the address of that instruction.

How does the debugger find out the address of an instruction?

If you are compiling a C/C++ program, you pass "-g" option to the compiler to generate some extra information. Those extra information contain these mappings and they are stored in a format known as DWARF in the object file. On Linux, the DWARF format is used to store the debug information inside the ELF file. Isn't that cool! ELF stands for Executable Linkable Format. It's the format to represent object files, executable files and shared libraries.

What is the instruction that the debugger puts in place of the original instruction?

The debugger overrides the first byte of the location where the original instruction was with the instruction "int 3". "int 3" has 1 byte opcode "0xcc", hence the debugger touches only the first byte of the memory address.

What's an "int 3" instruction?

"int n" instructions generate a call to the exception handler specified by the destination operand. "int 3" generates a call to the debug exception handler. The exception handler is a part of Kernel code and it delivers the signal SIGTRAP to the process, in our example to the process we are debugging.

How and when does the debugger change an instruction of the debugee process?

And now the most exciting part of this post has arrived. Using a very powerful system call ptrace().

Let's understand ptrace.

What does ptrace() do? ptrace is a system call by which our debugger controls the execution of the process we are debugging. The debugger uses ptrace() to examine and change the memory and registers of the process that we are debugging.

If you look at the , before execl()'ing the debuggee process we called ptrace() with PTRACE_TRACEME which indicates that this process is to be traced by its parent process i.e. the debugger.

The man page for ptrace mentions this:

If the PTRACE_O_TRACEEXEC option is not in effect, all successful calls to execl(2) by the traced process will cause it to be sent a SIGTRAP signal, giving the parent a chance to gain control before the new program begins execution.

Which simply means that the debugger gets notified via the wait()/waitpid() system call just before the debuggee process starts. And there the debugger has its golden chance to modify the text/code section of the process being debugged.

It also means that while being traced the debuggee process will stop each time a signal is being delivered and the tracer (the debugger) will be notified at its next call of waitpid(). Hence, when the signal SIGTRAP is delivered to the debugee process, it will stop and the debugger will be notified, which is exactly what we want. We want the debuggee process to stop and the debugger gets notified when the debuggee process executes the "int 3" instruction. The debugger gets the information about the cause of the stopping of the debuggee process from the status returned by waitpid().

The default behaviour of SIGTRAP is to dump core and abort the process but we can not debug a process if it's killed. Can we? Hence the debugger ignores the SIGTRAP signal and causes the debuggee process to continue.

Here's the code to set "int 3" instruction at the first byte of the address that contained the original instruction. As you can see, the same old ptrace() function is used to first get the original instruction at the address so that we save the original instruction to restore it later. And then the same old ptrace() function is used with a different flag PTRACE_POKETEXT to set the new instruction which has "int 3" at its first byte.

```
60
61  unsigned set_breakpoint(pid_t pid, unsigned addr) {
62    unsigned data = ptrace(PTRACE_PEEKTEXT, pid, (void *) addr, 0);
63    unsigned old_data = data;
64
65    printf("Setting a breakpoint at %x\n", addr);
66    printf("Value at the address %x %x\n", addr, data);
67    data = (data & ~0xff) | 0xcc;
68    ptrace(PTRACE_POKETEXT, pid, (void *)addr, data);
69    printf("Value at the address %x after setting the int 3 opcode %x\n", addr, data);
70    printf("Done setting the breatpoint at %x\n\n", addr);
71    return old_data;
72  }
73
```

What does the debugger do now that the "breakpoint has been hit"?

- First, the debugger needs to restore the original instruction at the address where the breakpoint has been set.
- Second, once it has been restored that original instruction should be executed once the debugger lets the debuggee process restart its execution.

How does the debugger restore the original instruction?

Just the way the debugger sets "int 3" to set a breakpoint. Here's the code. While setting the breakpoint we saved the original instruction, now all we need to do is to set it back at the given address.

```
104
105   /* Restore the code at the address - 80483e9 */
106   ptrace(PTRACE_POKETEXT, pid, (void *)addr, old_data);
107   printf("Restored the old value %x at the address %x\n", old_data, addr);
108
```

How is the original instruction in the debuggee process then executed?

The program counter of the debuggee now points to the next instruction now that it has already exe-

cuted the instruction at the address where we had out “int 3”.

To make the processor execute the original instruction in the debuggee process, we need to set the value of the program counter – %eip (in case of x86 machines) or %rip (in case of x86 64 machines) of the debuggee process to the address again.

And how can we set the instruction pointer of the debuggee process?

Using ptrace()! ptrace() has this super awesome capability of letting us “change the tracee’s memory and registers.” PTRACE_GETREGS makes ptrace copy the general purpose registers of the debuggee process into a struct. And PTRACE_SETREGS modifies the debuggee process’s general purpose registers. Here’s the code that does that:

```
73
74  void set_rip(pid_t pid, unsigned addr) {
75    struct user_regs_struct regs;
76
77    printf("Setting the instruction pointer to address %x\n", addr);
78    memset(&regs, 0, sizeof(regs));
79    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
80    regs.eip = addr;
81    ptrace(PTRACE_SETREGS, pid, NULL, &regs);
82    printf("Done setting the instruction pointer\n\n");
83  }
84
```

Once the debugger restored the debuggee process’s program counter it let’s the process continue and the way to do that is following:

```
109   /* Set the correct address at the eip/rip to resume the execution */
110   /* after the breakpoint has been hit. */
111   set_rip(pid, addr);
112   ptrace(PTRACE_CONT, pid, NULL, NULL);
113
```

And that’s how a debugger can set breakpoints.

The full code of the debugger can be found here.

I presented on this topic at the Thursday evening presentation at RC. Here’s the link to the slides. ■

Why I don't spend time with Modern C++ anymore

By HENRIQUE BUCHER

Henrique owns Vitorian LLC which provides PhD-level expertise in Financial Technology: front-end algorithms (strategies), backend/tick data warehouse, Ultra Low Latency techniques, low level assembly, Modern C++/software and reconfigurable hardware (FPGA).

Modern C++, what is it, where does it come from?

For the past 10 years with the advent of C++11, and before that its respective initiatives (TR1, Boost), there was a big movement within the C++ development community towards moving projects to the new standard: Modern C++. That meant using features like auto, closures (lambda), variadic templates and many more. C++ certainly became a rich platform for experimentation, and several "Modern C++" libraries appeared. Those lucky to grasp the new idioms as SFINAE, tag dispatching, CRTP, type generator, safe bool and others, or at least those who can recite them, were hailed with the status of Gurus.

After 1970 with the introduction of Martin-Löf's Intuitionistic Type Theory, an inbreed of abstract math and computer science, a period of intense research on new type languages as Agda and Epigram started. This ended up forming the basic support layer for functional programming paradigms. All these theories are taught at college level and hailed as "the next new thing", with vast resources dedicated to them. It also formed an ecosystem of speakers that make a living out of hyping out this "next new thing" into Corporate America. It is no surprise that those formed under this environment come out of college and drive the current C++ committee developments.

The performance-to-functional POV switch

After enough time, C++ became from the "fast language" to the "functional language" and performance was put aside. It is currently well known that several systems within C++ are inherently slow like iostreams, strings and also lacking basic features like networking, and very basic ones like a simple string splitting routine. Ask anyone in the committee about why this has not been resolved in almost two decades and the response is always the same: nobody has ever cared to submit a paper or proposal.

With SG14, which is a branch within the ISO Working group 21 (WG21) dedicated to games and high frequency tranding, the committee intends to give a channel for discussion from the general "low latency industry". However, given the discussions that there have been in the discussion groups, or the Michael Wong's Standard (as funny as it sounds), there is visibly no appetite to implement the radical reforms that C++ needs in order to compete with "C++ with Classes" for the low latency applications.

Why is it a problem really?

It is my personal experience reviewing code as a performance/optimization consultant, that "Modern C++" leads to unsatisfactory implementations in general. Not to say that it is not possible to create low latency, fast platforms based on Modern C++ but it creates many stumbling blocks that cast the team in paralysis or generates extremely hard-to-optimize graphs that C++ compilers have trouble with. It is with extreme satisfaction to see Chandler Karruth's series of videos bring back the lost link (and a bit of sanity) between the C++ language and reality.

Performance loss

The first stumbling block is, as mentioned, the potential loss in performance due to more complex IR structures and passes in the compiler. This is where Fortran excels – because of the much simpler language, compilers are able to optimize it much more efficiently than the respective C/C++ code. The ubiquitous understanding that templates and variadic templates will produce much faster assembly because the C++ code will automagically vanish during compilation *cannot be corroborated* by my per-

sonal experience, by no reputable peer-reviewed publications nor by anyone with some knowledge of compiler optimization. *This is an ongoing fallacy.* In fact, it is the opposite: smaller bits of code can be compiled, debugged and optimized by looking at the generated assembly much more efficiently than with templates and variadics.

Compilation times

The next stumbling block refers to compiling time. Templates and variadics notoriously increase the compilation time by orders of magnitude due to recompilation of a much higher number of files involved. While a simple, traditional C++ class will only be recompiled if the header files directly included by it are changed, in a Modern C++ setup one simple change will very often trigger a global recompilation. It is not rare to see Modern C++ applications taking 10 minutes to compile. With traditional C++, this number is counted in low seconds for a simple change.

The time between when the developer is reasonably sure the code works and the time he gets to test the generated assembly is directly proportional to the quality of the code generated. If the developer can quickly try and test its application, it is more likely he will be able to nail all the possible edge cases he is worried about. When a recompilation takes ten minutes, not so much.

Maintainability

The final stumbling block has to do with complexity. As Edger Djikstra once said,

> *Simplicity is prerequisite for reliability.*

If you have to hire a guru/expert to understand the code, either you have the wrong code or the wrong language.

If one reads with attention, the very fundamentals of Djkstra's arguably most well known paper, "Go-to Statements Considered Harmful" can be directly applied to templates and variadics: it is not that the feature itself is bad but its inherent structure leads to complex code that eventually brings down the most desired characteristic of a source code: easiness of understanding.

When one is programming trading systems that can send 100,000 firm orders per second over the wire from platforms with strategies being created and put in production over a two-day production cycle, you ***want*** simplicity. You ***need*** simplicity. This leads to my one minute trading code rule:

> *If you cannot figure out in one minute what a C++ file is doing, assume the code is incorrect.*

The real reason

But the real motivation why I do not spend much time with Modern C++ anymore, despite being pretty good at it at some point, is just that there is much more being developed in the industry that needs my attention.

C++ today is like Fortran: it reached its limits. Nowadays there are interfaces so fast that the very notion of "clock cycle" ceases to exist in certain parts of the system. The speed has gone so fast that at those speeds if two bits are sent at the exact same time through two adjacent wires, they will likely get out of sync less than a meter away.

To handle the type of speed that is being delivered in droves by the technology companies, C++ cannot be used anymore because it is inherently serial, even in massively multithreaded systems like GPUs.

Today the "Modern Technologist" has to rely on a new set of languages: Verilog, VHDL. The new technologists have to be able to build their own CPUs, create their own virtual motherboards, otherwise they will be drowned by the avalanche of technological advancements in the coming years. Not because FPGAs are faster – they are not. They actually run 10x slower in frequency than top CPUs.

However, all sorts of reconfigurable hardware are being deployed in the field. Intel is currently shipping Xeon CPUs with embedded FPGAs. IOT is going to become a multi-billion dollar market in the next five years, and it is largely propelled by little $10 FPGAs that will require an astounding amount of qualified technologists to program them. And programming on RTL is hundreds of times more complex than coding C++, believe me. If C++ has a learning curve, try to become an expert fpga programmer (Altera's transceiver toolkit alone is 700 pages long, Quartus is another 1,000 pages, not to mention all Xilinx tools and devices).

But it is worth. Once you dominate these new ways to code, it opens up a huge avenue to express your ideas – and that is where the unicorns are born: by people that can see on both sides of the fence, higher than anyone else.

Conclusion

In my personal view for what's worth, C++ is a tool that like Fortran, it is showing its age, accelerated by inaction from the ISO committee. Investing too much in being excellent in Modern C++ is becoming akin to excelling in Cobol or Fortran – a thoughtful exercise but with very marginal gain. One has to learn C/C++ but cannot find the language in itself be sufficient.

> ***The new technologist cannot be restricted in such serial environment anymore.***

C++ is awesome, my main tool, but just not enough anymore. At this point, looking at the mailing lists for C++17 it is clear that C++ is going at 10 m.p.h. while the world is going supersonic – and this is normal! Technologies level up to their efficiency maximum.

Technologists have to be open-minded to take advantage of new tools and new techniques and tools in order to handle the new generation of technologies to be delivered in the next decades. And there is just not enough time to learn! ■

Books programmers don't really read

By BILL CRUISE

Mark Twain once said that a classic novel is one that many people want to have read, but few want to take the time to actually read. The same could be said of "classic" programming books.

Periodically over on Stack Overflow (and in many other programming forums) the question comes up about what books are good for programmers to read. The question has been asked and answered several times, in several different ways. The same group of books always seems to rise to the top, so it's worth it to take a look at these books to see what everyone is talking about.

Books most programmers have actually read

1. Code Complete
2. The Pragmatic Programmer
3. C Programming Language (2nd Edition)
4. Refactoring: Improving the Design of Existing Code
5. The Mythical Man-Month
6. Code: The Hidden Language of Computer Hardware and Software
7. Head First Design Patterns
8. Programming Pearls
9. Effective Java (2nd Edition)
   or Effective C++
10. Test Driven Development: By Example

I've read all of these books myself, so I have no difficulty believing that many moderately competent programmers have read them as well. If you're interested enough in programming that you're reading this blog, you've probably read most, if not all of the books in this list, so I won't spend time reviewing each one individually. I'll just say that each of the books on the list in an exceptional book on its respective topic. There's a good reason that many software developers who are interested in improving their skills read these books.

Among the most commonly recommended programming books there is another group that deserves special consideration. I call the next list "Books Programmers Claim to Have Read". This isn't to say that no one who recommends these books has actually read them. I just have reason to suspect that a lot more people claim to have read the following books than have actually read them. Here's the list.

Books programmers claim to have read

1. Introduction to Algorithms (CLRS)
   This book may have the most misleading title of any programming book ever published. It's widely used at many universities, usually in graduate level algorithms courses. As a result, any programmer who has taken an algorithms course at university probably owns a copy of CLRS. However, unless you have at least a Masters degree in Computer Science (and in Algorithms specifically), I doubt you've read more than a few selected chapters from Introduction to Algorithms.

   The title is misleading because the word "Introduction" leads one to believe that the book is a good choice for beginning programmers. It isn't. The book is as comprehensive a guide to algorithms as you are likely to find anywhere. Please stop recommending it to beginners.

2. Compilers: Principles, Techniques, and Tools (the Dragon Book).
   The Dragon Book covers everything you need to know to write a compiler. It covers lexical anal-

ysis, syntax analysis, type checking, code optimization, and many other advanced topics. Please stop recommending it to beginning programers who need to parse a simple string that contains a mathematical formula, or HTML. Unless you actually need to implement a working compiler (or interpreter), you probably don't need to bring the entire force of the Dragon to bear. Recommending it to someone who has a simple text parsing problem proves you haven't read it.

3. The Art of Computer Programming (TAOCP)
   I often hear TAOCP described as the series of programming books "that every programmer should read." I think this is simply untrue. Before I'm burned at the stake for blasphemy, allow me to explain. TAOCP was not written to be read from cover to cover. It's a reference set. It looks impressive (it is impressive) sitting on your shelf, but it would take several years to read it through with any kind of retention rate at all.

   That's not to say that it's not worthwhile to have a copy of TAOCP handy as a reference. I've used my set several times when I was stuck and couldn't find help anywhere else. But TAOCP is always my reference of last resort. It's very dense and academic, and the examples are all in assembly language. On the positive side, if you're looking for the solution to a problem in TAOCP (and the appropriate volume has been published) and you can't find it, the solution probably doesn't exist. It's extremely comprehensive over the topic areas that it covers.

4. Design Patterns: Elements of Reusable Object-Oriented Software (Gang of Four)
   Design Patterns is the only book on this list I've personally read from cover to cover, and as a result I had a hard time deciding which list it belongs on. It's on this list not because I think that few people have read this book. Many have read it, it's just that a lot more people claim to have read it than have actually read it.

   The problem with Design Patterns is that much of the information in the book (but not enough of it) is accessible elsewhere. That makes it easy for beginners to read about a few patterns on Wikipedia, then claim in a job interview that they've read the book. This is why Singleton is the new global variable. If more people took the time to read the original Gang of Four, you'd see fewer people trying to cram 17 patterns into a logging framework. The very best part of the GoF book is the section in each chapter that explains when it is appropriate to use a pattern. This wisdom is sadly missing from many of the other sources of design pattern lore.

5. The C++ Programming Language
   This book is more of a language reference than a programming guide. There's certainly plenty of evidence that someone has read this book, since otherwise we wouldn't have so many C++ compilers to choose from.

   Beginning programmers (or even experts in other languages) who want to learn C++, though, should not be directed to The C++ Programming Language. Tell them to read C++ Primer instead.

As I said before, I know there are a few of you who have actually read these books. This post isn't intended for you, it's intended for the multitudes who are trying to appear smarter by pretending to have read them. Please stop recommending books to others that you haven't read yourself. It's counterproductive, as often there is a better book (more focused on a specific problem domain, easier to understand, geared more toward a specific programming language or programming skill level) that someone more knowledgeable could recommend. Besides that, you may end up embarrassing yourself when someone who has actually read TAOCP decides to give you a MMIX pop quiz (if you don't know what I'm talking about, then this means you). ■

Reprinted with permission of the original author. First appeared at billthelizard.com.

food bit *

Gingerbread folks and houses

Ah, it's that time of the year again where gingerbread people seem to peek out of every shelf and box. But before you dismiss these sweet-natured folks as mere holiday indulgences, consider their long storied history. These hard, spiced cookies have been around since the Middle Ages and were enormously popular in Germany, England, France and Holland.

Over the centuries, gingerbread dough have been used to make everything from love tokens and Christmas ornaments to elaborate edible houses (shh…don't tell Hansel and Gretel). Even royalty got a little gingerbread-mad – Queen Elizabeth I enjoyed these treats so much that she had her bakers fashion the cookies in the likeness of visiting dignitaries.

* FOOD BIT is where we, enthusiasts of all edibles, sneak in a fun fact about food.