A "BETTER C" BENCHMARK

C is almost 50 years old. A great age for a bottle of wine, not so great for a programming language in the fast-moving industry. Over the past 10 years many new languages have emerged, with different flavours, all trying to become a C replacement to some extent.

When a new language becomes more or less popular – developers start writing benchmarks, showing how performant the software in this language is, how much CPU power and how much memory is uses, how big are the binaries and so on.

Here I would like to carry a little experiment in a different plane - UX of the programming languages, how *productive* the developer is when using this or that language, how easy it is to use them, what common frustrations there are, how people feel when they read the code. I believe that UX of the programming language is as important as their technical characteristics and it contributes a lot to a success of a language.

Warning: the rest of the post is very subjective.

EXPERIMENT

Let's write an app that scans all files recursively in the current directory and prints those lines in the files that match a given wildcard. Something like `ag` or `grep`, but using wildcards rather than regular expressions. Binary files should be ignored.

I find this problem to be a good exercise, because it shows how to implement a very simple wildcard matching algorithm (which works with pure data such as strings and numbers, no need to learn any libraries or APIs). This algorithm should pass a few very simple tests. Then it requires some very common, yet low-level APIs, such as scanning the directory recursively or reading files line by line. All parts of the

problem are very simple, small and well-scoped. I assumed that such utility should be easy to implement in any language, even if there was no previous experience with it.

I wanted to test how “friendly” the language is to a regular “write-compile-run-debug” loop, how easy it is to write tests for the matching algorithm, how easy it is to find required APIs to work with the file system and basic I/O, how friendly is the compiler when pointing out errors, how “intuitive” is the language and so on.

My sample size was rather humble - just myself. But to avoid bias you are encouraged to do the same on your own (it should not take too long) and compare the results. After the utility has been written in different languages – I asked my peer developers (~20 people) to read it and tell what it does, what lines are unclear, what feels “easier” to read and to understand. These developers had no experience with either of the languages, but have been coding a lot in other languages, such as Java, C#, JavaScript, Kotlin, Swift etc.

The languages I tested here are C++, Go, Rust and Zig. The resulting programs I got are available at github.com/zserge/glob-grep, feel free to criticise.

ZIG

It all started because I wanted to see what kind of a language Zig is. I never used it, but heard some good feedback. With no previous experience I opened vim and started coding.

It took me ~1 hour to finish the program. The wildcard matching algorithm (which I knew before, and only had to implement in Zig) took me ~20 minutes. The rest was looking for the APIs to do the directory scanning and the file reading.

TLDR: great, intuitive little language, poor stdlib and docs.

What I liked: the language is surprisingly intuitive to a C coder. Feels very simple, the documentation about the *language* (but not the stdlib) is very clear and friendly.

Vim integration was also pretty good, for a young language (before I enabled vim plugin - I was annoyed with formatting errors, which are compiler errors).

I liked the error handling approach. Liked that test harness comes with the language. Even liked that strings are just the arrays of bytes, like in C.

My first reaction to carrying the allocator around was a shock, but in practice it was not even noticeable. It gives a feel of minimalism that the core of the language is so simple that it does not even use dynamic memory. Again, very close to C.

I had to read lot of the Zig compiler and stdlib sources while writing this, and the code was very clear and concise.

What I disliked: the stdlib documentation is terrible. Everything I learned about directory scanning and file I/O – I got from github search results, which are also pretty scarce.

Compiler messages are also far from being friendly, but to the one who is familiar with C it's not a big deal.

The lack of string handling routines in the stdlib was unexpected, to concatenate strings one has to do everything manually - allocate the buffer, put strings there. Or use formatter and an allocator to print both strings side by side and free the buffer afterwards. It's still very different from `s1+s2`.

Overall, the core language is simple and I enjoyed it, but the stdlib is even more limited than libc. I hope that this is just a sign of an early age of the language.

People who read the resulting Zig code actually mentioned that is was the one that made them realise what the program does. It is a bit verbose, but explicit, predictable and easy to understand. Not surprising, as the language was designed with readability in mind (no hidden control flow, no hidden allocations, no macros, no operator overloading, no metaprogramming etc).

RUST

I had a few failed attempts to learn Rust. It took me over 2 hours to finish this program and I felt frustrated after I finished.

TLDR: complicated.

What I liked: the compiler messages are nice. The documentation to the language is also good. But that's probably it. At least I didn't have to fight too much with lifetime errors this time. The tooling around the language is modern and nice.

What I disliked: compiler messages are too verbose and take the whole screen. No, I don't want to run `rustc --explain` for every mistake. Please, don't punish me. Documentation is also sometimes too verbose. I mean, it's better have more docs than less, but having a TLDR version first would be even better. Same for stdlib, having a brief list of functions and what they do in one sentence would be easier to skim through as a reader. Having `&str`, `Str` and `[u8]` in obviously necessary, but surprises a newcomer.

Overall, coding in Rust feels like puzzle solving to me. Could be fun and exciting, especially when using Rust as a hobby language, but for most tasks I would rather prefer an "ergonomic" language that would be a barely noticeable tool.

People who read the resulting Rust code often raise at least a couple of "wtf" questions in process. They often complain the syntax is unclear and requires paying attention to details. Also, pattern matching is still an unfamiliar thing to the "mainstream" developers.

GO

This was cheating. I obviously had some previous experience with Go, but nevertheless I wanted to try it in this experiment. It took me ~15 minutes to make my complete "glob" utility work, just as I expected.

TLDR: productive, but opinionated.

What I liked: it feels very productive, docs are amazing to my taste - brief, but useful, one can immediately open the related stdlib function sources and investigate further. From the past experience, while writing the app I already envisioned how to make it multithreaded and boost the performance (simple fan-out).

What I disliked: too many things are under the hood (buffered I/O, GC). You don't feel like you are in control of everything (like in Zig). Too opinionated – this was the only language on the list that required me to create 3 separate files to make it work. It

is still possible to make silly mistakes, like accidental variable shadowing, or using defer inside a loop.

People who read the resulting Go code found it clear, some wondered about the inline walker function (which does not have to be inline, they were right). Some wondered about the multiple assignment, i.e. `a, b = c, d`, which also was not needed there and made things more confusing. Ironically, if I was new to Go - I would have written even more straightforward Go code.

C++

Although I have some C experience, I am rather distant from the modern C++, so I decided to give it a try. It took me ~20 minutes to finish, and that was unexpected.

TLDR: good old “frenemy”.

What I liked: feels familiar, like meeting an old friend from the past. I enjoyed the docs, with lots of examples and good readability. I was surprised to see how powerful stdlib is nowadays. Support in text editors and IDEs is also very solid.

What I disliked: poor tooling – no build system, no test harness, no linter. We are used to it from the past, but that’s not what a modern developer expects. Too powerful – for this very task C++ felt very productive, but I can imagine myself with a decision paralysis at some point when there are many different ways to do something and all are equally good (or bad).

People who read the resulting C++ code actually have read C or C++ in the past, at least as part of their university classes. Many complained about the use of `::`, so I should properly use the namespaces, I guess. Overall, as I don’t have a “taste” in C++ code – I’m sure it could have been written more clearly, but I also see how it could have been written much worse without even noticing.

OTHER BENCHMARKS

All languages produced static executables, all about the same size (2..5MB). The smallest one was Zig, the largest one was Rust. All had about the same performance when scanning through my whole `/usr/include` file tree. That’s why I wanted to

highlight that technical characteristics are often not as important as the developer experience.

I would like to mention separately the build times. I ran the whole build+test+clean loop a hundred of times. Go comes the fastest (as expected), the other three are LLVM-based and are ~3x-4x slower.

What does it all mean? The results are not surprising and often follow the cliches about the languages: Go is simple to read, Rust is complicated, C++ is familiar, Zig looks promising but is too young to judge.

If I had to write a new service/utility that does not have to interact with C code a lot - I would definitely choose Go. If I had to call some C or C++ libraries - I would stick to C++ (unfortunately). What place would Rust and Zig take in the modern programming world - only time will tell. I wish Zig had a better documentation to gain popularity before it becomes too niche and obscured. I will definitely pay a closer attention to it, so far it's the first real C replacement I've met, especially when it comes to the low-level coding.

But of course, C is here to stay despite its age. There are still too many areas where C is the only real choice. And I'm glad that C exists.

I hope you've enjoyed this article. You can follow – and contribute to – on Github, Twitter or subscribe via rss.

Mar 18, 2021

See also: Étude in C minor and more.