# SHOULD C PROGRAMMERS LEARN C++, GO OR RUST?

*2018, Jul 30*

I recently wrote a series of posts called 'Modern C++ for C Programmers'. I mentioned in the introduction: "I hope to convince C programmers to give '2017 era C++' (which is entirely unlike 2003 C++) another good look. (…) My goal is that when you go look for a new language to learn (say, Go or Rust), you will hopefully consider modern C++ as well."

Over the weeks as I posted new parts, I was blown away by the interest. Over 50,000 visitors spent significant time on the site, leading to over 100,000 page views. These numbers are still rising.

Meanwhile, some (former) C programmers informed me they had moved to Go already and were having a lot of fun there. This made me read up on Go, and learn about its design goals and trade-offs. PowerDNS has adopted Go as well for some projects.

Far less polite noises came from the 'Rust evangelism strikeforce' who informed me in no uncertain terms that anything other than immediate and complete adoption of Rust was a dangerous waste of time and that I should stop teaching people C++. As far as I can tell, none of these complaints came from people that ship any software in Rust.

I was originally not planning to opine on the relative merits of C++, Rust and Go, since I am very biased towards C++. C++ is my tool of choice (warts and all) and has been for over two decades. I am not a neutral observer.

But given the feedback from Go users and the exhortations from Rust evangelists, I now do want to weigh in on the question: should C programmers consider C++, Rust or Go?

## Language design

Language design is stupendously hard. Everyone would like to have a language that is fast, easy to learn, simple to implement, powerful, expressive, safe and generically useful to boot.

These things contradict each other of course. Expressiveness requires lambdas, which are complex if you want to compile them, but if you don't, they aren't fast. Easy to use almost

mandates garbage collection, but that collides with ‘fast’ and ‘simple to implement’. Removing pointers makes the language safer, but at a steep loss of generality. Trying to make pointers safe with a ton of language rules makes the language hard to use. I could go on.

The most talented designers are able to craft a language that manages to get “as much in” as possible while keeping things relatively simple. Lesser qualified teams might instead come up with a language that is slow, hard to use AND complex. This has been done many times.

Once designed, it may take half a decade before it is clear how good the language actually is. This, by the way, explains Bjarne Stroustrup’s observation that “There are only two kinds of languages: the ones people complain about and the ones nobody uses”. As long as nobody actually uses the language, we will not have to find out if it is any good or not.

# Some non-language things

Both Go and Rust have innovated in terms of dependency management. Whereas for both of these languages the subject is still hotly debated, solutions abound. C and C++ are still stuck in the dark ages in this respect.

In terms of shipping binaries, we can debate the merits of the Go and Rust approaches. I can however state that for C++, shipping binaries is still a pain. A tool like exodus helps tremendously though.

# Who is it for?

Because language design is so hard and because it takes so long to know ‘how the cake will come out of the oven’, it is tremendously important to start with a clear goal in mind: who or what is this for?

# Go

For Go, the Google designers thought that one through really well. They considered C++ to be too big a language, and I think this is true. My advice to make judicious use of C++ features only works if your whole codebase sticks to it - otherwise you’d have to know about all the features anyhow. This argues strongly for a language that is restrained in features but that still retains sufficient power. A simple language means junior programmers (and there are always more junior programmers than senior programmers!) can get up to speed quickly.

> *The key point here is our programmers are Googlers, they’re not researchers. They’re typically, fairly young, fresh out of school, probably learned Java, maybe learned C or C++, probably learned Python. They’re not capable of understanding a*

> *brilliant language but we want to use them to build good software. So, the language that we give them has to be easy for them to understand and easy to adopt. – Rob Pike*

The Go language does away with a bunch of common features like exceptions, generics, formal object inheritance, assertions, operator overloading, unions and pointer arithmetic, and this indeed left a mostly simpler language. It does have `goto` though!

Regarding the simplicity, a read of the Go Frequently Asked Questions shows that, like entropy, complexity is hard to suppress, and eventually any language finds itself explaining unexpected things.

For Go the question is if their opinionated decisions on what features could be skipped and how things must work will, down the road, turn out to have been the right choices. But consciously making choices is good, and Go users sure seem productive for now.

In terms of performance and generality, Go powers the Google gVisor sandboxed container runtime, where Go filters or even performs all Linux system calls. This clearly means Go is not slow. They do note: "gVisor was written in Go in order to avoid security pitfalls that can plague kernels. With Go, there are strong types, built-in bounds checks, no uninitialized variables, no use-after-free, no stack overflow, and a built-in race detector. (The use of Go has its challenges too, and isn't free.)"

This seems like a fair summary.

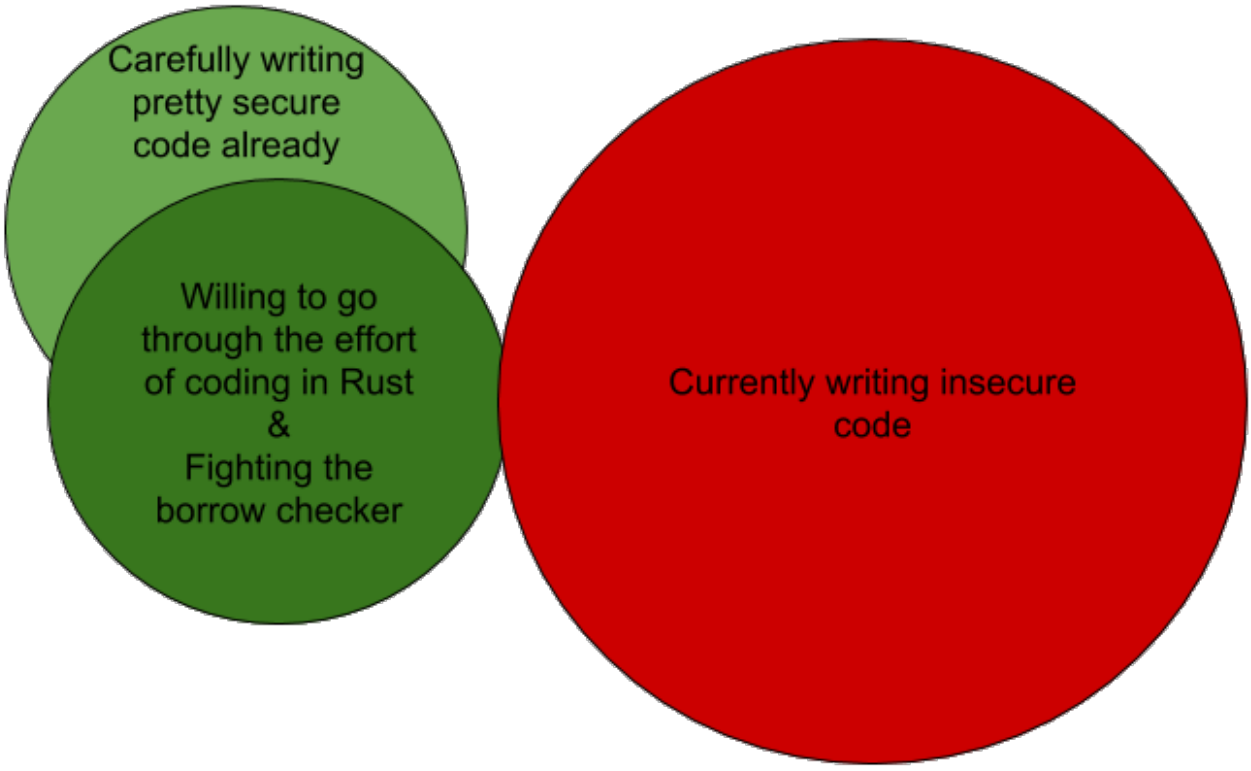
# Rust

Rust also had design goals. Their overriding belief is that any C like language is dangerous, because it allows unsafe memory manipulation. The language should therefore prevent such dangerous memory access. Languages with no pointers and only checked arrays have been done before, but they've never attained the speed or flexibility to be "systems programming languages".

Rust set itself the goal to provide lightning fast access to memory, while guaranteeing safety and correctness mostly at compile time (where this is 'free' in terms of performance). In itself this is a laudable goal, and one that we should all strive towards.

Whereas the Go people spent time on thinking "who is this language for", Rust appears to not have thought this through very well.

The biggest Rust users I know are among the very best and most secure C programmers around. These were not the people writing unsafe code in C.




Rust is not a particularly fun language to use (unless you like puzzles). Programmers spend a lot of time fighting the borrow checker and other language rules in order to placate the compiler that their code really is safe.

The fact that Rust is hard to use scares off the very people it should be protecting. The people currently using Rust were already writing safe code.

Stated differently, Rust is the dream language for people who weren't the problem. Rust is adored by people already spending a lot of time on keeping their code safe.

The people writing unsafe C today will not embrace Rust simply because it does not make their life any easier or more fun. If they were willing to do hard work, their C code would not be as unsafe as it is.

Note that this is no criticism of the laudable goal of creating a safer language. But it is a condemnation of making a safe language designed for people that were already among the safest programmers around!

I too enjoy a language features that make it **impossible** to make certain classes of mistakes. A great example is `const`, which makes sure you don't accidentally change variables. The promise of guaranteed memory safety in Rust is indeed highly seductive.

Most people have a fixed supply of discipline for tasks which are not intrinsically fun but still important. This is the kind of discipline that is needed to create secure code. Robert M. Pirsig

wrote great words on this in Zen and the art of Motorcycle Maintenance where he describes 'Gumption'. Recommended.

However - we can deplete our supply of gumption easily enough by fighting our language. We run out of discipline that way. The result may be code that is memory safe but executes plugins from a globally writable directory.

In this sense, the Rust memory security features may not be a net positive for writing safe code.

# C++

As noted, I'm a big C++ fanboy. Bjarne once replied to an email I sent him, and it was a big moment for me. I do however program in Lua and Javascript as well and I enjoy that too. But know where I come from: C++ is my homebase.

Seen in isolation, if someone invented C++17 today, it would be considered to be way too complex. Writing a C++ compiler from scratch is a daunting challenge. But, as it stands, in 2018, the relevant C++ compilers are (almost fully) compliant with C++17, which only got ratified in December 2017. C++20 features are already starting to appear.

Furthermore, a seasoned C programmer already has a lot of skills relevant to core C++ features. Syntax, pointers, control flow and the trade-offs are already familiar. A C programmer has a running start understanding C++.

In addition, although the language standard is complex, implementations of C++ and its standard library are of high quality. Bugs do happen but it has been years since I ran into one. The complexity is something everyone, including important members of the C++ standardization committee, would like to reduce. But with the exception of error messages that sometimes fill pages, it is possible to simply 'use' C++ and not worry about all the magic that is happening underneath.

This brings me to generality. Almost all the functionality of C++ is also written in C++. The standard library code is on the same footing as your code. The C++ `std::map` or `std::string` implementations for example can easily be swapped out for faster or more specialised versions.

Both Go and Rust decided to special case (or at least 'unsafe') their `map` implementations. Go simply had to since their language does not offer operator overloading and other mechanics required to cook up a native looking map. You can however of course make one that works just as well but does not have the syntactic sugar of a native `map[]` (or works as generically, perhaps).

Rust meanwhile notes that you can't write a performant data structure in safe-mode Rust, so they urge you not to do that. There is also an epic page that lists several convoluted linked list implementations in Rust.

If I measure C++ against the criteria set out earlier:

- Performance
- Powerful
- Expressive
- Generically useful
- Easy to use
- Safe
- Simple to implement

No one can reasonably argue C++ isn't fast, powerful, expressive or generically useful. Simple to implement it definitely isn't though, and a single typo will create pages full of hard to decrypt errors.

This leaves us with 'easy to use' and 'safe'. Easy to use, for C++03, I would definitely say no. For C++11 and beyond, definitely yes. `auto` and lambdas have taken out a lot of the pain we used to experience, typing in stuff like:

```
for(map<string, pair<string,string> >::const_iterator iter = p.begin();
    iter != p.end(); ++p)
```

Something we can now achieve without loss of performance and generality in with range-for:

```
for(const auto& val: p)
```

This brings us to safety. C++ has not made the language intrinsically safer than C. Most (but not all) C bugs compile just fine as C++. All data structures now offer bounds checking access via `at()`, but the default `operator[]` will happily let you touch memory you shouldn't.

More stringent type checking compared to C has helped though, which reduces the chances of not allocating sufficient memory. Meanwhile, smart pointers vastly reduce or eliminate the chances of use-after-free bugs.

With a little bit of discipline, C++ can be used in a very memory safe way, with the actual unsafe operations concentrated in very few well audited places. The easy availability of address space sanitizers and other static checking technologies has also helped tremendously in keeping code secure.

# Conclusions

For a seasoned C programmer, I think Rust is not the way to go. I appreciate the goals of Rust, but I think that in practice they may not be making real life shipped code a lot more secure - also because not that much actual Rust code is shipping. Rust is the dream language for programmers who were already writing very secure code. In Rust their memory access may be safer, but I wonder if the somewhat painful process to achieve this may not have depleted the discipline required for other security aspects.

Go meanwhile looks like a lot of fun and does indeed go a long way towards being a simpler yet still powerful language - if not as powerful or fast as C++. Go pleases a lot of people though, and its users appear to be highly productive. If the limitations of a consciously constrained language do not frighten you, it looks like a good choice.

Finally, based on all of the above, I think a seasoned C programmer would honestly do well to look at modern C++. With a little bit of discipline it too is a highly secure language, and if you are already well versed in C, C++ is an easy step to make in becoming a more productive programmer.

I hope this has been informative to you - please let me know your thoughts on bert.hubert@powerdns.com or @PowerDNS_Bert.

← Previous
*Modern C++ for C Programmers: introduction*