**ECE459: Programming for Performance** Winter 2020

# Lecture 36 — Programming Languages and P4P

*Patrick Lam* *2019-10-22*

In ECE 459 we've used C/C++ as systems languages:

- C: "portable assembly language";
- C++: can be much higher-level; we've seen C++ abstractions like threads, algorithms, STL, etc.

The focus of this course hasn't been on security. But in many cases, writing insecure fast code isn't the right thing.

**Question.** Is it possible to write secure C/C++?

*problems:* memory errors, race conditions

*tools:* Valgrind tools memcheck, helgrind; code review; other static tools e.g. Coverity.

Here are some points of view on this question.

**Robert O'Callahan.**

"I cannot consistently write safe C/C++ code.[1]" (17 July 2017)

(Holds a PhD in CS from Carnegie Mellon University; was Distinguished Engineer at Mozilla for 10 years; etc.)

**Stephen Kell.** "Some Were Meant for C" (Onward '17) [**?**]: One could have safe C implementations; C is good for interoperating with low-level sytems (e.g. reproduce systems' memory layouts).

**Chrome.** March 2019: disclosure of Chrome use-after-free vulnerability[2]; 0-day attacks observed in the wild. Google implements best practices, and has all the tools and developers that money can buy!

## Rust

Enough of talking about the failings of C and C++. We'll talk about Rust, an alternative to C/C++. It is a new-school secure systems programming language used by Mozilla's Project Quantum. This material is based on *The Rust Programming Language* by Steve Klabnik and Carol Nichols[**?**] and I'll make references as appropriate.

Here's some Rust code.

```
fn main() {
  let x = 42; // NB: Rust infers type "s32" for x.
  println!("x␣is␣{}", x);
}
```

By default, Rust variables are *immutable*.

```
fn main() {
  let x = 42; // NB: Rust infers type "s32" for x.
  x = 17; // compile-time error!
}
```

Let's consider two examples that look similar but have drastically different meanings.

```
let x = 1729;
let x = 88;
println!("shadowed␣x␣is␣{}", x);
```

```
let mut x = 33; // mutable
x = 5;
println!("mutated␣x␣is␣{}", x);
```

[1] https://robert.ocallahan.org/2017/07/confession-of-cc-programmer.html

[2] https://security.googleblog.com/2019/03/disclosing-vulnerabilities-to-protect.html

In the first case, old "x" still exists but is inaccessible under the name "x". In the second case, the storage cell for "x" used to contain 33 and then contains 5. The difference matters, for instance, when there are references to "x".

[NB: Rust also has compile-time `const`s.]

**Rust vs C++: immutability.** In C, you can cast away `const`-ness; not so in Rust. If something is not mutable in Rust, you can't cast it into mutability. Also, in Rust, structs or tuples are either all mutable or all immutable. (Although interior mutability is a thing in Rust. We're not talking about it.)

**Perf implications.** We mentioned immutability in Lecture 7. The best way to avoid having to use locks (even read/write locks still require writes to acquire the read lock): have no writes. However, there's a tradeoff. If your data structure is immutable but you want to update it (as we often do with data structures), you need to copy the data structure, at least partially. That can be slow.