The Rust programming language

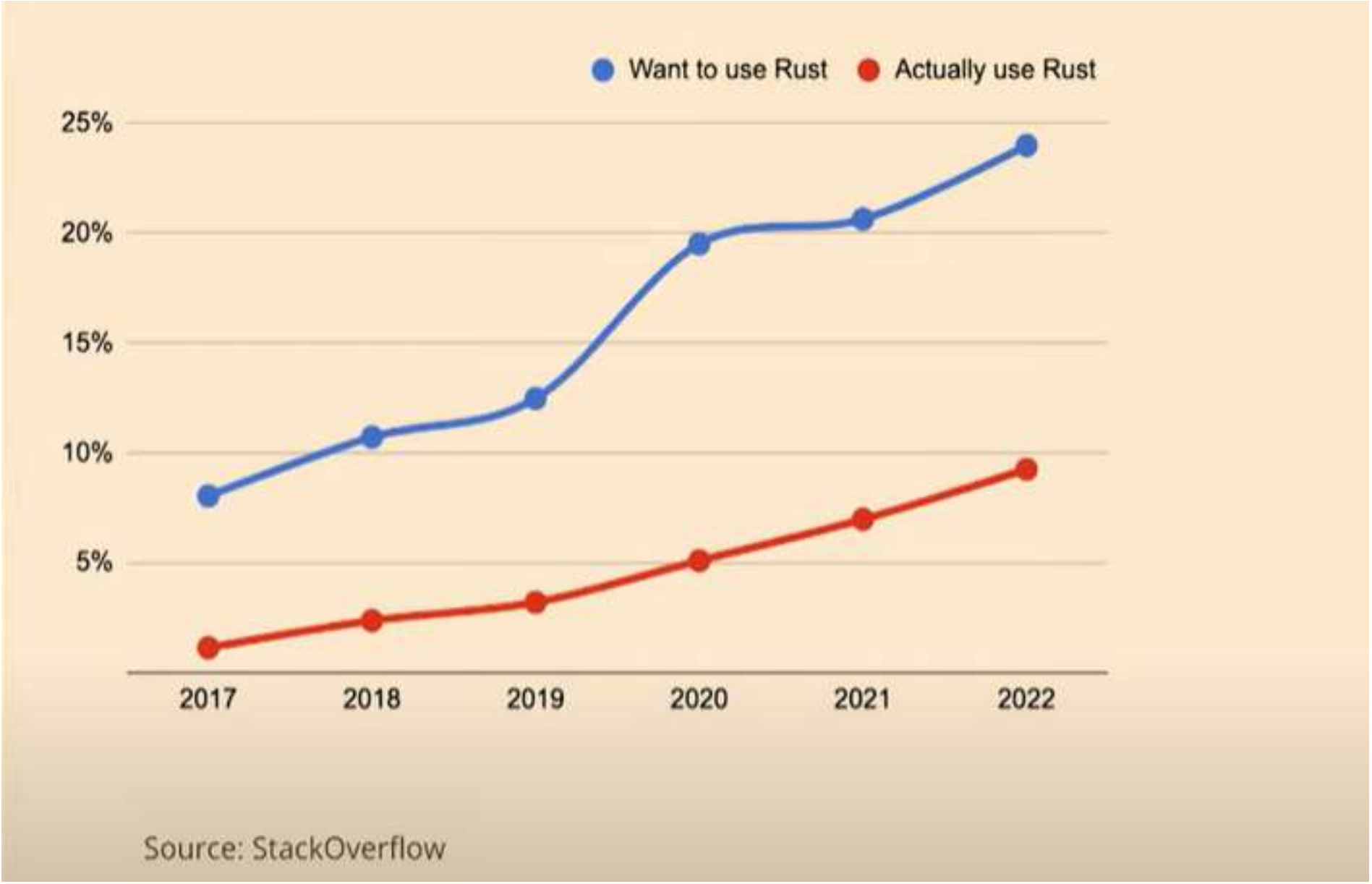
Gary Clynch

School of Enterprise Computing and Digital Transformation

Code samples in https://github.com/gclynch/rust

The Rust Programming Language

- Rust is a multi-paradigm general purpose programming language:
  - V1 by Graydon Hoare, Mozilla Research in 2010
  - Imperative, structured, functional, generic, not object-oriented
  - Cross platform, open-source compiler (MIT and Apache dual permissive licenses) https://github.com/rust-lang
  - Now managed by the Rust Foundation https://foundation.rust-lang.org
    - Mozilla, Google (Android), AWS, Microsoft, Meta, Huawei
  - Most “loved “programming language on Stackoverflow surveys for 7 years in a row to 2022 (75K developers)
  - Top 20 TIOBE index https://www.tiobe.com/tiobe-index/
  - Rust playground https://play.rust-lang.org/


Want to use Rust
Actually use Rust
25%
20%
15%
10%
5%
2017
2018
2019
2020
2021
2022
Source: StackOverflow

The Rust Programming Language

- New stable version every 6 weeks (nightly, beta, stable channels) (currently 1.67.x)
  - https://github.com/rust-lang/rust
- Objective: memory safety, efficiency, concurrency
  - "Speed and Safety"
  - Enforces memory safety without using a garbage collector through its "borrow checker"
  - Systems programming rather than application programming e.g. CLI tools, embedded, WASM, networking
    - e.g. Android, Firecracker (micro VMs), npm, Firefox, Discord read states service (originally written in Go), DropBox sync engine, GNU coreutils, Linux kernel language #2 after C
- Strong static typing

The Rust Programming Language

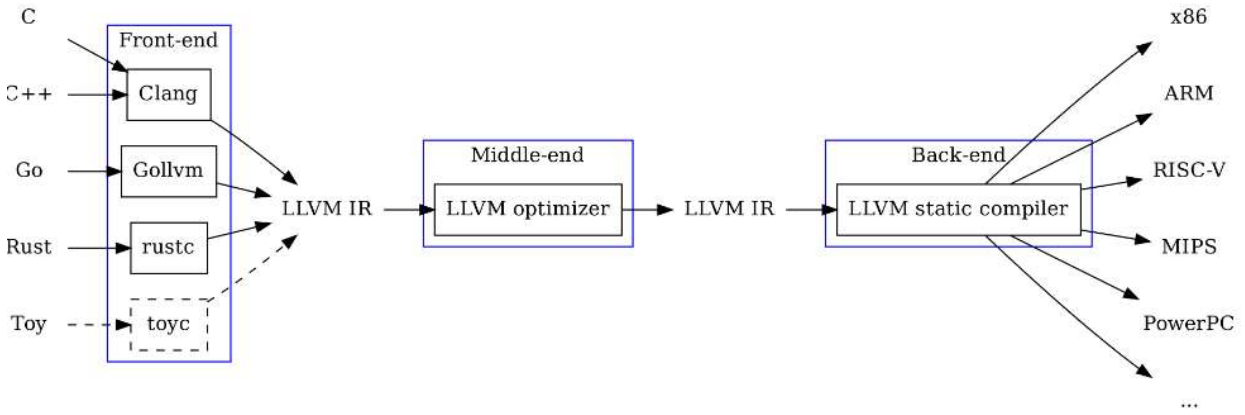
- Compiled to machine code, not managed
- Similar syntax and performance to C++, motivated by Haskell/Ocaml/Scala
- No garbage collection
  - Uses “ownership”, “borrowing”, and “lifetimes” - OBRM
  - No “null” value – `Option<T>`
- Expression oriented language
  - (nearly) every statement is an expression and therefore returns a value; these expression statements can themselves form part of larger expressions
- Community driven – Rust RFCs https://rfcbot.rs

The Rust Programming Language

- Traits
  - Ad-hoc polymorphism ("the ability to substitute one object for another if they both have certain characteristics")
- Macros – for last resort code re-use - metaprogramming
- Structs / Enums (i.e. user-defined types)
  - Can contain implementations
- Rust does not have exceptions
  - Recoverable errors: `Result<T, E>`
  - Unrecoverable errors: `panic!`

Rust tooling

- `rustc` compiler originally written in Ocaml
  - In 2011 it compiled itself (self-hosted)
  - Target: LLVM IR (which is then optimised and compiled to target machine code)



- IDE support usually via Rust analyzer
- clippy, rustfmt

Rust datatypes

- Built-in types, tightly integrated into the language
  - Scalar primitive types
    - Integers (signed and unsigned) (`i8` to `i128`, `u8` to `u128`)
    - floating points numbers (`f32` and `f64`)
    - characters (`char`) (Unicode 4 bytes) (whereas a `String` uses UTF 8)
    - booleans (`bool`)
  - Sequence types
    - Arrays (static size, on stack, same type) `[1, 2, 3]`
    - Tuples (different types) `[1, "hello", '!']`
    - Slices (reference to a contiguous sequence of elements (e.g. array or string) in a collection)

Rust datatypes

- Built-in types (more)
  - Function types
  - Trait types (inspired by Haskell)
  - Pointer types (reference, smart pointer)
- User-defined types (which have “move semantics” by default)
  - structs
  - enums (disjoint unions, containing variants)

Packaging – Cargo + Crates.io

- Cargo is the package manager and build tool for Rust:
  - Package = set of crates – `cargo.toml` manifest
  - Crates are binary crates or libraries (`src/main.rs`, `src/lib.rs` convention)
    - max 1 library crate in a package
  - Crates can be structured into modules
  - Can call a build script (e.g. make) before cargo build
  - Crates.io – centralised package repository/registry for the Rust community, dependencies in `cargo.toml`
  - `rustdoc` for crate documentation (`///` and `//!`)
  - `rustc` compiles a crate

Rust standard library

- In the `std` crate
  - Core types, operations on primitives, threading, I/O, and standard macros etc.
  - It is "small" – no date type, no random number generators, no regular expressions – use a 3rd party crate for these
  - Available to all crates by default (no need for `.toml` to define `std` crate as a dependency)
    - to import (so it can be used using a shorten name) `use std::env;`
    - `env` is a module in the `std` crate
  - Things in the Rust prelude do not need to be imported (e.g. `String`, `Vec<T>` etc.)

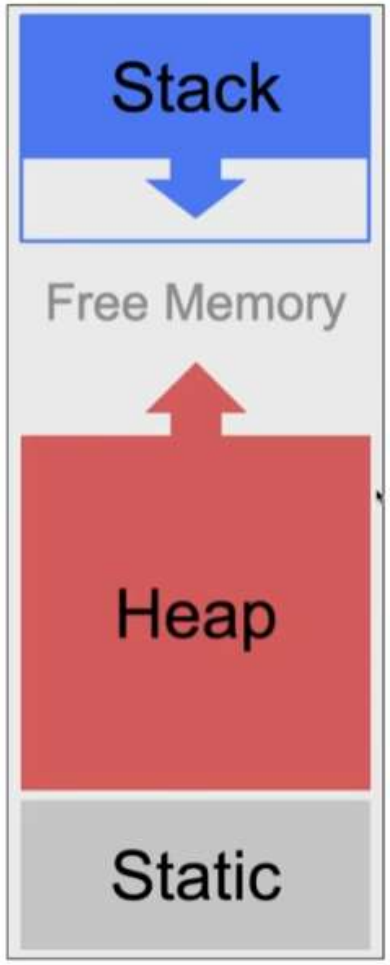
Rust Functional Programming

- Functional programming
  - programs are constructed by applying and composing functions
- Rust is imperative ("how") but has functional features:
  - Expressions (which return values, "what") rather than statements
  - Functions are first-class citizens
  - Immutability
    - immutable types/immutable references/persistent data structures
  - Anonymous functions and closures
  - Pattern matching
  - Recursion

Rust Functional Programming

- Functional aspects:
  - Iterators
    - Iterate forwards over a sequence of elements taking one element at a time using `next()`
    - Sequence – array, string, vector, hash map, graph etc.
    - Doesn't matter how the elements are stored
    - are lazy (do nothing until consumed)
    - Apply higher-order functions (`map`, `fold`, `filter`, `any`, `find` etc.) to an iterator
      - Iterator adaptors and consumers

Memory in Rust



Stack v Heap in Rust

- In general in Rust fixed size data (i.e. size known at compiler time) is stored on stack, dynamically sizing data (e.g. strings, vectors) are stored on heap
  - Stack is automatically managed by the operating system (rather than Rust itself)– all values in Rust are stack allocated by default i.e. including user-defined structs and enums
  - Heap for dynamically sizing data (by explicit boxing using `Box::new<T>` which returns a pointer to the value on heap)
    - Rust's ownership system automatically results in a "drop" of the memory on heap once the owner goes out of scope

Memory Safety in Rust

- Memory safety is the property of a program where memory pointers used always point to valid memory, i.e. allocated and of the correct type/size i.e. avoid
  1. Dangling pointers (pointer that points to data which has been deallocated)
  2. Double frees (trying to free the same memory twice)
- Rust compiler will not compile code with any of the above bugs i.e. we cannot write memory unsafe code

Ownership (OBRM)

- Rules:
  1. Each value (on stack or heap) has an owner (its variable)
  2. Only one owner at a time
  3. When owner goes out of scope the value will be dropped by freeing the memory
- The owner is responsible for freeing memory if it is on heap
  - It is dropped implicitly unless it has a destructor which is run instead (i.e. `drop` is called automatically on the variable if it implements the `Drop` trait (e.g. string or vector))
- The ownership rules are enforced at compile-time

Moving ownership

- By default all types have a “move semantics” i.e. ownership is transferred to another variable (a “move”) on assignment
- Types have a “copy semantics” if they implement the `Copy` trait (and as a result they cannot have destructors i.e. implement `Drop` trait)
  - All primitive types implement the `Copy` trait (it is an empty trait that just means duplicate by copying the bytes)
  - Structs do not implement the `Copy` trait by default
    - If any part of a struct is on heap it cannot implement `Copy`
    - `Copy` trait can only be implemented for values stored on stack
  - A “move” == pass by value (i.e. value is moved to a new owner)
  - A “borrow” == pass by reference

Borrowing

- A reference ("borrow") `(&)` (immutable) can be given to a value (on stack or heap) but that does not transfer ownership, cannot modify something we have an immutable reference to
  - A reference is a non owning pointer type
- A reference goes out of scope after the line of code it was last used
- A variable owns its resource unless it is a reference
- A mutable borrow `(&mut)` must end before the owner can access it again

Smart pointers

- Smart pointers usually own the data they point to (unlike references which borrow data)
- Provide additional features and metadata over references and are implemented as `structs`
- Smart pointer types:
  - for allocating values on heap `Box<T>` e.g. for recursive types
  - "Reference counting" smart pointer type `Rc<T>`
    - Multiple owners, track number of owners, when no owners remain clean up data
  - For enforcing borrowing rules at runtime instead of compile time `Ref<T>` and `RefMut<T>`

References

- The Rust programming language book https://doc.rust-lang.org/book/
- The Rust foundation https://foundation.rust-lang.org/
- The Rust analyzer https://rust-analyzer.github.io/
- Crates https://crates.io/
- Stack Overflow survey 2022 https://survey.stackoverflow.co/2022/