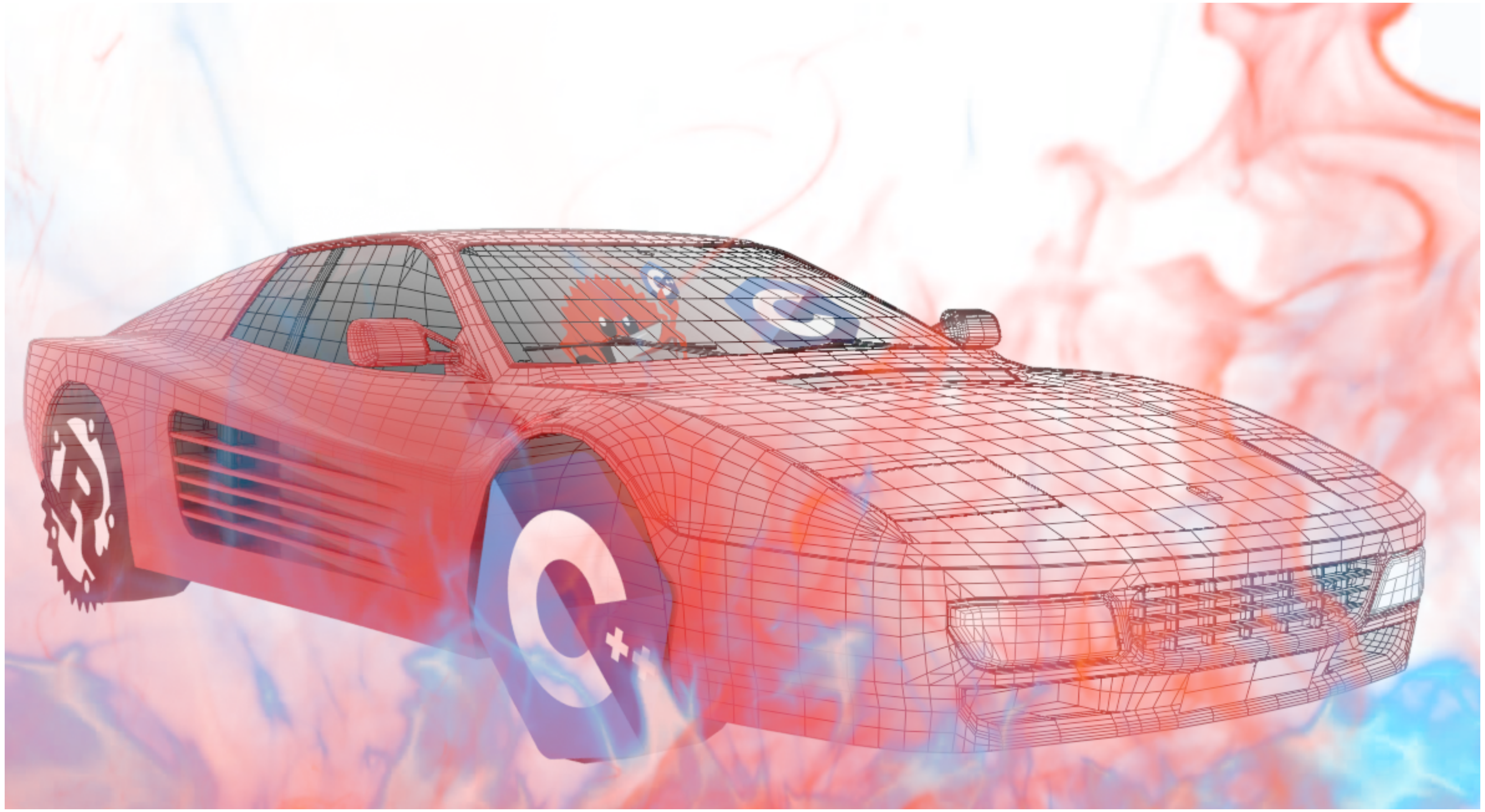
Effizientes Programmieren in C, C++ und Rust

Rust - Introduction and Basics

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024



Recap

1. What did you like about C++?
2. What did you hate about C++?

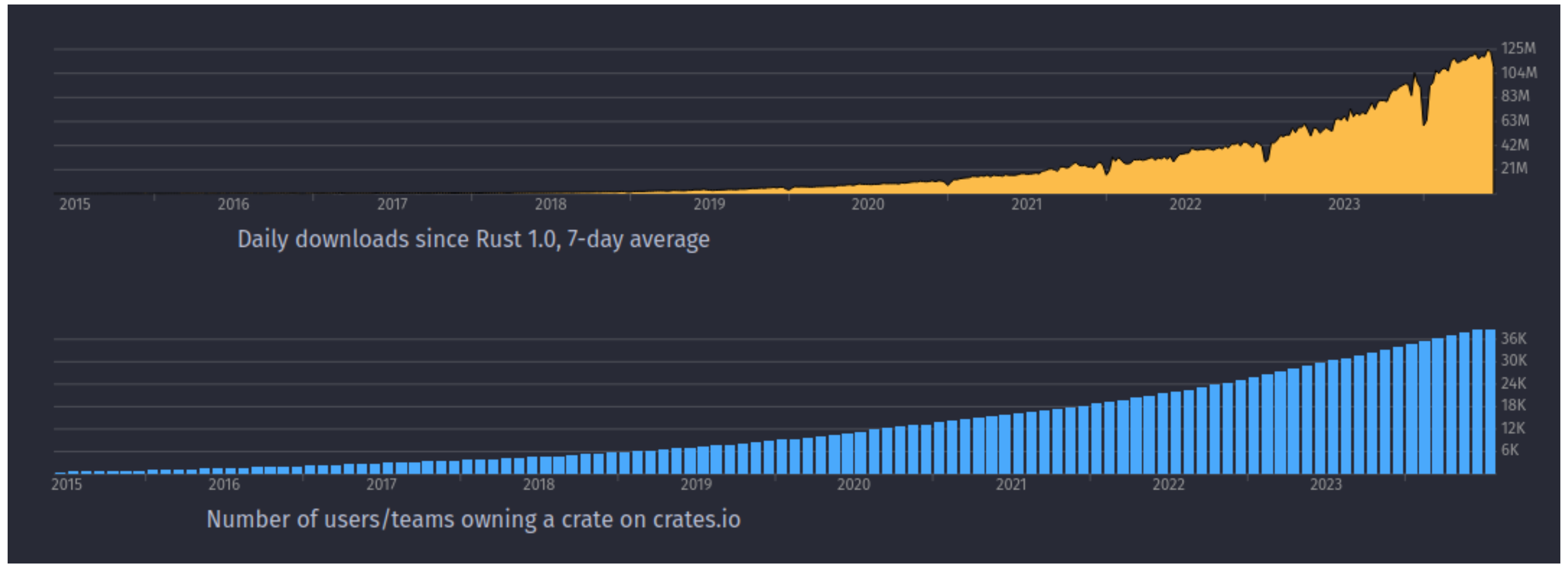
Table of Contents

A New, Hot Systems Programming Language

style="width: 85%" /> style="width: 85%" />

- Probably the most hyped programming language at the moment
- Fast growing
- Serious competition for C++?
- RIIR: rewrite it in Rust
  - *almost a technical term by now*

style="width: 50%;"/>



https://lib.rs/stats"

What is behind the hype?

The Official Website

> "A language empowering everyone to build reliable and efficient software."

"blazingly fast and memory-efficient" "memory-safety and thread-safety"

"top-notch tooling" "friendly compiler with useful error messages"

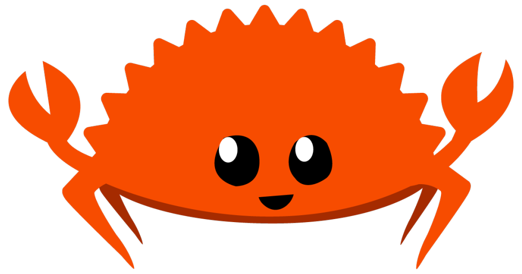
Goals

- Similar to C++: high-level, efficient systems programming

  $\Rightarrow$ Strong abstraction capabilities, same performance as C
- Why can't we just use C++?
  - C++ is full of footguns *(memory safety/undefined behavior)*
  - C++ is burdened with a giant legacy of backwards-compatibility

Safety vs Performance

- C++: we pay for high efficiency with undefined behavior (UB)↗
- Traditional view: either safety or low level control and efficiency
- Rust: Why don't we have both?
  (with some caveats)



Achieving Safety and Performance

- Separation into safe and unsafe Rust

  $\Rightarrow$ All the UB lives exclusively in unsafe
- Unsafe code is wrapped in a safe API
- Powerful type system (borrow checker!) ensures the API can't be misused

> "Rust locks all the footguns in a safe – and un-safe is the key"

$\Rightarrow$ Principle: build safe abstractions upon an unsafe fundament

Rust: Standing on the Shoulders of Giants

- Development started (officially) in 2009 at Mozilla
  - *Hearsay: the developers were too annoyed with the problems caused by C++ in Firefox*
- Rapid evolution, stable 1.0 release in 2015

So what is Rust?

- "C++ done right"

  $\Rightarrow$ Zero-cost abstractions, monomorphization, value/reference-semantics, ...
- Strong functional influences

  $\Rightarrow$ Pattern matching, immutability (default), ...
- Ownership and the borrow checker!

- **Claim 1:** Rust is both object oriented and functional
- **Claim 2:** Rust is neither object oriented nor functional

Getting Started

- Unified build system and dependency management: `cargo`

Rust Hello World

```
ferris@rustbootcamp: cargo new hello-world
ferris@rustbootcamp: cd hello_world && tree
.
├── Cargo.toml
└── src
    └── main.rs
```

main.rs

```rust
fn main() {
    println!("Hello, world!");
}
```

Build and Run

```
ferris@rustbootcamp: cargo build --release
    Compiling hello_world v0.1.0 (/home/.../hello_world)
    Finished release [optimized] target(s) in 0.11s
ferris@rustbootcamp: ./target/release/hello_world
Hello, world!
```

Getting Started

- Recommended reading: the book (doc.rust-lang.org/book/)

- Control flow syntax similar to C++
- Expression oriented: every statement returns a value (e.g. if-else)
- Type inference for local variables
- `double` → `f64`, `int` → `i32`, `size_t` → `usize`
- Immutable by default: **`mut`** required to mark variable as mutable

```rust
fn max_of_slice(vals: &[f64]) -> f64 {
    let mut max = f64::MIN;
    for &val in vals {
        max = if val > max { val } else { max };
    }
    max  // <- implicit return
}
```

- Additional primitive types:
  - Tuples
  - Arrays (fixed size)
  - Slices (variable size, non-owned)

```rust
let array: [i32; 10] = [0; 10];
let slice: &[i32] = &array;  // get slice of array

let result: (i32, i32) = min_max_of_slice(slice);
let (min, max) = result;     // destructuring
println!("input={}, min={}, max={}",
         array, min, max);
```

Getting Started

References and Move Semantics

- Borrowing: use **`&`/`&mut`** to get a shared/unique reference
- The borrow checker ensures only valid references are used!

```rust
let mut my_vec = Vec::<i32>::new();  // *
my_vec.push(42);
let answer = &my_vec[0];  // reference first element
do_something_else(my_vec);
// use after free?!
println!("the answer is {}", answer);
```

- Default assignment moves values
- Can't be used afterwards, invalidates references
- Exception: Some types are copied instead (e.g. numbers, plain structs)

```
error[E0505]: cannot move out of `my_vec` because it is borrowed
  --> src/main.rs:24:23
   |
21 |     let mut my_vec = Vec::<i32>::new();  // <- turbofish!
   |         ---------- binding `my_vec` declared here
22 |     my_vec.push(42);
23 |     let answer = &my_vec[0];  // reference to first element
   |                   ------ borrow of `my_vec` occurs here
24 |     do_something_else(my_vec);
   |                       ^^^^^^ move out of `my_vec` occurs here
25 |     println!("the answer is {}", answer)
   |                                  ------ borrow later used here
```

$\Rightarrow$ More details later

** this syntax is called a turbofish (and avoids the syntactic ambiguity of C++ templates)*

Structs

- Basic purpose analogous to C++

Syntax

- `Self` (upper case) is the type of the current struct
- `self` parameter indicates methods
- Member access only with explicit `self`
  - *Rust generally prefers explicitness*
- Initialization via `Self {...}`

```rust
struct Vector2D { x: f64, y: f64 }

impl Vector2D {
    // "static" function for construction
    pub fn new(x: f64, y: f64) -> Self {
        Self { x, y }  // initialization
        //  ^- shorthand for Vector2D { x: x, y: y }
    }

    // method (shared reference)
    pub fn length(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }

    // method (unique reference)
    pub fn set(&mut self, x: f64, y: f64) {
        self.x = x;
        self.y = y;
    }
}
```

Structs are Not Classes

- No inheritance, no unique address
- No metadata or vtable-pointer (POD)
- Only implicit "constructor"
  - Structs always fully initialized
    (C++: partial initialization observable)
  - Use functions to define different ways of construction

```
struct Vector2D { x: f64, y: f64 }

impl Vector2D {
    pub fn new(x: f64, y: f64) -> Self {
        Self { x, y }
        //  ^- shorthand for Vector2D { x: x, y: y }
    }

    pub fn length(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }
}
```

- Methods are syntactic sugar for functions
  - *Also: automic dereferencing (no `->` operator)*
  - *Dynamic dispatch only via traits↗*

```
let v = Vector2D::new(1.0, 2.0);
assert_eq!(
    v.length(), Vector2D::length(&v)
);
```

- Primitive types also support methods

```
assert_eq!(1.max(2), i32::max(1, 2));
```

⇒ **No difference in behavior & efficiency** between primitives and structs!

Enums and Pattern Matching

- Usually called sum types or tagged unions

  $\Rightarrow$ functional programming!
- Represents one of multiple variants
  - *set theoretic view: member of* $A + B + C$

    for variant sets A, B, C
- Consist of associated data and tag
- Support methods etc. in the same way as structs
- Used via pattern matching (**`match`**)
  - "switch on steroids"

- C++: no language support, `std::variant` as poor substitute

```rust
enum Message {
    Quit,                       // unit variant
    Move { x: i32, y: i32 },    // struct variant
    ChangeColor(u8, u8, u8),    // tuple variant
}

impl Message {
    fn handle(&self) {
        match self {
            Message::Quit => process::exit(0),
            Message::Move { x, y }
                => println!("move to {x}, {y}"),
            Message::ChangeColor(r, g, b)
                => println!("#{r:02X}{g:02X}{b:02X}"),
        }
    }
}
```

Enums and Pattern Matching

- Pattern matching is powerful and versatile
  - Nested patterns, arbitrary conditions
  - Compiler ensures no case is missing
- Example: match slice and contained enum variants at once
- Also usable via destructuring, **`if let`** and **`let ... else`**

```
fn handle_multiple(message_list: &[Message]) {
    match message_list {
        [msg] => msg.handle(),
        [Message::Quit, ..] => process::exit(0),
        [head, tail @ ..] => {
            head.handle();
            handle_multiple(tail);
        }
        _ => { },  // catch-all
    }
}
```

- Take-away: useful tool for data with multiple variants

$\Rightarrow$ But is the resulting code efficient?

```
fn handle_move(msg: Message) {
    let Move{ x, y } = msg else {
        return;
    };
    // do something with the move
}
```

Enums and Pattern Matching

- What are the performance implications? 🚴

Data Layout

- **enum** = data + tag
- Tag must be aligned! ⇒more than one byte
- Data size is defined by the largest variant

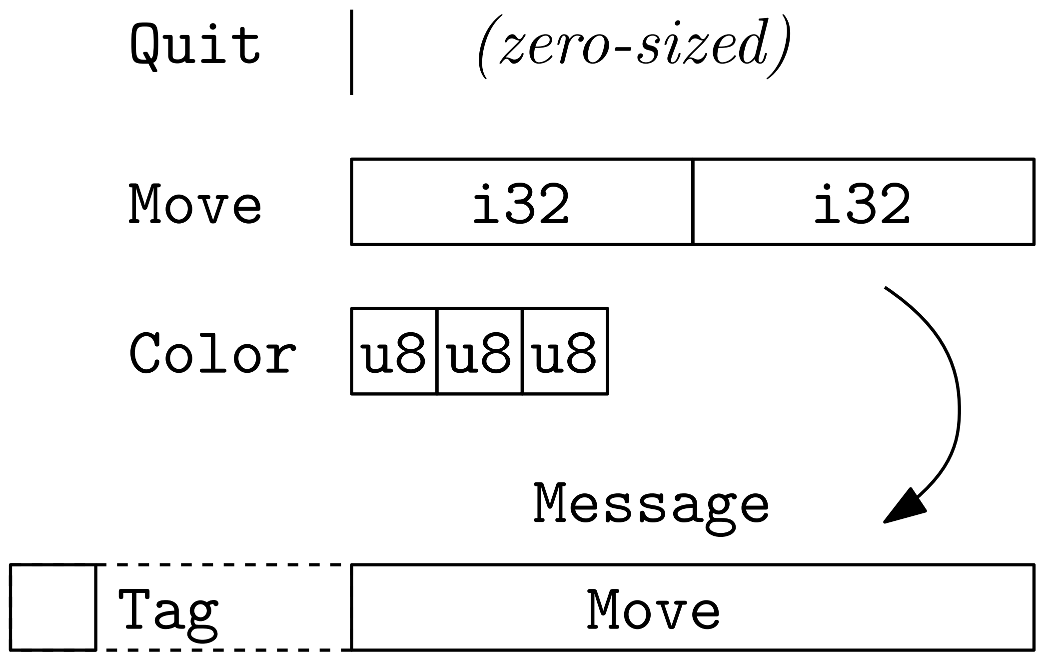
  ⇒ Seldomly used large variants problematic – move to heap?
- Niche optimization: compiler uses invalid states of the data to eliminate the tag

```
enum Niche { A(bool), B, C }

println!("{}", mem::size_of::<Niche>());
// 1
```

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    ChangeColor(u8, u8, u8),
}

println!("{}", mem::size_of::<Message>());
// 12
```



Enums and Pattern Matching

Code Generation

- Naive approach: check every condition one by one (`if else`-chain)
  - *Linear in number of match arms*
  - *Branch mispredictions?*

```
match message_list {
    [msg] => msg.handle(),
    [Message::Quit, ..] => std::process::exit(0),
    [head, tail @ ..] => {
        head.handle();
        handle_multiple(tail);
    }
    _ => { },  // catch-all
}
```

Some test results*

- Simple mapping of variant to constant creates a lookup table
- Compiler tries to generate a jump table starting at 4 arms
- For `if else`-chains, the compiler tends to go top to bottom in order

- In more complicated cases, jump tables are not possible

  $\Rightarrow$ For really hot matches, check generated assembly and try different code layouts

** no reproducability guarantee*

Aside – Lookup Tables and Jump Tables

- First case: map variant to value

```
enum Variants { A, B, C, D }
```

```
match var {
    Variants::A => -1,
    Variants::B => 42,
    Variants::C => 1337,
    Variants::D => 13,
}
```

A Lookup Table

```
# extract tag
movsx   rax, dil

# load base address
lea     rcx, [rip + .lookup_table]

# read table entry
mov     eax, dword ptr [rcx + 4*rax]

.lookup_table:
    .long   4294967295
    .long   42
    .long   1337
    .long   13
```

- Effectively compiled to an array access
- Array is part of binary
- Requires no branch

Aside – Lookup Tables and Jump Tables

- Second case: do something more complicated for each variant (e.g. call a function)

```
enum Variants { A, B, C, D }
```

```
match var {
    Variants::A => callA(),
    Variants::B => callB(),
    Variants::C => callC(),
    Variants::D => callD(),
}
```

- Note: This is actually not optimal
- Could we save one indirection?

A Jump Table

```
movzx   eax, dil     # get jump address via table
lea     rcx, [rip + .address_table]
movsxd  rax, dword ptr [rcx + 4*rax]
add     rax, rcx     # address is relative (ASLR)
jmp     rax          # jump to according address

.address_table:
    .long   .jmp_a-.address_table
    .long   .jmp_b-.address_table
    .long   .jmp_c-.address_table
    .long   .jmp_d-.address_table
.jmp_a:     jmp     callA
.jmp_b:     jmp     callB
.jmp_c:     jmp     callC
.jmp_d:     jmp     callD
```

- Get address of code for current variant from lookup table
- Jump directly to address

Pattern Matching versus Dynamic Dispatch

- Scenario: data with multiple variants
- OOP solution: inheritance

  ⇒ one subclass per variant, polymorphic methods

```cpp
struct Shape {
    virtual double area() = 0;
};

struct Rectangle: public Shape {
    double x, y;
    double area() override { return x * y; };
};

struct Square: public Shape {
    double width;
    double area() override { return width * width; };
};

struct Circle: public Shape {
    double radius;
    double area() override { M_PI * radius * radius; };
};
```

- FP solution: pattern matching

  ⇒ represent data as enum variants

```rust
pub enum Shape {
    Rectangle(f64, f64),
    Square(f64),
    Circle(f64),
}

impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Rectangle(x, y) => x * y,
            Shape::Square(x) => x * x,
            Shape::Circle(r) => PI * r * r,
        }
    }
}
```

Pattern Matching versus Dynamic Dispatch

Pattern Matching

- Method is defined in a single place (static function!)

  ⇒ Can add method without modifying existing code

- Flat data layout

Dynamic Dispatch

- Variant is defined in a single place

  ⇒ Can add variant without modifying existing code

- Requires indirection and a vtable (e.g. via reference or `std::unique_ptr`)

```
impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Rectangle(x, y) => x * y,
            Shape::Square(x) => x * x,
            Shape::Circle(r) => PI * r * r,
        }
    }
}
```

```
struct Rectangle: public Shape {
    double x, y;
    double area() { return x * y; };
};

struct Square: public Shape {
    double width;
    double area() { return width * width; };
};

struct Circle: public Shape {
    double radius;
    double area() { M_PI * radius * radius; };
};
```

Pattern Matching versus Dynamic Dispatch

Benchmark (1k)

- Sum area of 1.000 random shapes
- Input: `&[Shape]` versus `std::vector<std::unique_ptr<Shape>>&`
- Result: ~780ns versus ~4800ns

Reasons?

- Worse data layout
  - *Though `&[Box<Shape>]` still achieves ~900ns*
- Additional vtable indirection
- Inlining! *(without inlining: ~4500 ns)*
- ... what about larger data?
  - *1k fits in L1 cache ⇒ memory access cheap*

```rust
impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Rectangle(x, y) => x * y,
            Shape::Square(x) => x * x,
            Shape::Circle(r) => PI * r * r,
        }
    }
}
```

```cpp
struct Rectangle: public Shape {
    double x, y;
    double area() { return x * y; };
};

struct Square: public Shape {
    double width;
    double area() { return width * width; };
};

struct Circle: public Shape {
    double radius;
    double area() { M_PI * radius * radius; };
};
```

Pattern Matching versus Dynamic Dispatch

Benchmark (1000k)

- Sum area of 1 million random shapes
- Only ~25% of throughput for 1k elements, factor 1.25 faster than dynamic dispatch

$\Rightarrow$ Dominated by memory accesses

Benchmark (1000k, randomized)

- Shuffle resulting list randomly
- More than factor 3 faster than dynamic dispatch

$\Rightarrow$ Allocations have random order, thus the indirection hurts

```
impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Rectangle(x, y) => x * y,
            Shape::Square(x) => x * x,
            Shape::Circle(r) => PI * r * r,
        }
    }
}
```

```
struct Rectangle: public Shape {
    double x, y;
    double area() { return x * y; };
};

struct Square: public Shape {
    double width;
    double area() { return width * width; };
};

struct Circle: public Shape {
    double radius;
    double area() { M_PI * radius * radius; };
};
```

Pattern Matching versus Dynamic Dispatch

- Pattern matching is usually more efficient
- Even better, if possible: avoid both

Software Design Aspects

- Enums are closed to new variants and open to new operations

  $\Rightarrow$ reversed to OO solution

  (see: visitor pattern, expression problem)
- Arguably, simpler and more maintainable in many cases
- Impossible if external code needs to add variants

Efficiency Aspects

- Flat data layout, while dynamic dispatch requires indirection
- Jump table comparable to vtable
- Dynamic dispatch typically not inlined!
- However: large enum variants and long `if else`-chains are performance pitfalls

Nullability, or: Maybe Just Option

- There is no `null` in (safe) Rust
- What to do instead? ⇒ Enums, of course!
- Also known as: `Optional`, the `Maybe` monad

```
enum Option<T> {
    Some(T),
    None,
}
```

```
fn twice(val: Option<u32>) -> u32 {
    val + val
}
```

```
error[E0369]: cannot add `Option<u32>` to `Option<u32>`
  --> src/main.rs:11:9
   |
11 |     val + val
   |     --- ^ --- Option<u32>
   |     |
   |     Option<u32>
```

Correctness

- Nullability encoded in type system
- Compiler enforces checks where necessary

Efficiency

- Niche optimization: `Option<&T>` is represented as nullable pointer
- Might reduce null checks compared to `nullptr` (depends on usage)

Better:

```
match val {
  Some(v) => v + v
  None => None,
}
```

or

```
val.map(|v| v + v)
```

Error Handling in Rust

- By the way: we can also replace exceptions with enums (mostly)
- Differentiate two kinds: bugs/unrecoverable errors and **recoverable** errors

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Panic!

- Mechanism for unrecoverable errors
- Implementation similar to exceptions (stack unwinding)
- Not supposed to be caught!
- Example: bounds checking for arrays

  ⇒ Panic instead of buffer overflow

```
panic!("Oh no! Guess we abort the program");
```

Result

- Enum representing (potentially) recoverable errors
- `match` instead of `try ... catch`
- `?`-operator for simple error propagation

```
fn execute_fallible_operations(
) -> Result<Data, Error> {
    let result: Data = op_that_might_fail()?;
    // ....
    Ok(result)
}
```

Error Handling in Rust

Result vs Exception

Advantages 👍

- Encodes failure conditions in the type system
- Programmer is forced to handle errors
- More composable than checked exceptions
- Error path is much faster than stack unwinding

Disadvantages 👎

- Encodes failure conditions in the type system
- Programmer is forced to handle errors
  - *Note: you can "ignore" errors with* `unwrap` *⇒ panics instead*
- Happy path incurs an additional (easily predictable!) branch

⇒ Very useful for high reliability software

⇒ Can be annoying for prototyping/scripting

Modules

- Modules are a tool for organizing your source code
  - To split code into multiple files, you need modules
  - Allow to trivially find definitions and determine whether imports are used
- Definition via `mod`, usage via `use` *(→ next slide)*
- Visibility management via `pub` and `pub(crate)` *(crate = only within the library)*
- Default "private" visibility is within module

Definitions are Authoritative!

- Source files without corresponding module definition are ignored

  ⇒ Each new file first requires a new module definition!

- Different to most other languages, can be confusing at the beginning

Modules

Most Important Rules

- Modules are a tree with root either in `main.rs` (binary) or `lib.rs` (library)
  - The root is referenced via **`crate::`**
- The `mod` keyword defines a new module as child of the current module
- The `use` keyword imports a module or item from another module

```rust
// import HashMap from the standard library
use std::collections::hash_map::HashMap;

// define a new module, implementation
// lives in child_module.rs
mod child_module;

// No auto-import! We need to import everything
// we want to use from child_module
use child_module::{MyStruct, my_function};
// equivalent (if this is main.rs)
use crate::child_module::{MyStruct, my_function};
```

- Path in module tree defines the file path, with three options:
  - Inline in file of parent module
  - `<module_name>.rs`
  - `<module_name>/mod.rs`

Macros

- Tool for code generation
  ⇒ same basic idea as in C/C++

Not a Footgun

- Unlike C, Rust macros are "hygienic": can not interfere with syntax or local variables
- Operate on the Abstract Syntax Tree
  ⇒ only specific syntax is accepted

Function-like Macros

- Look like functions except for the `!`
  - `println!` – printing with compile-time checked formatting
  - `dbg!` – quick debug output
  - `panic!`
  - `todo!`
  - `assert!`
  - ...

Attribute-like Macros

- Usable to annotate any item
- Syntax: `#[attribute_name]`
- Example: auto-implementation of printing, comparisons, etc. for structs via `#[derive(Debug, Eq, ...)]`

More about the Rust Toolchain

- `cargo` integrates everything related to building, depencies and much more
  - Building:
    `cargo check/build/run [--release]`
  - Dependency management:
    `cargo add/remove/update <dep>`
  - Tests and Benchmarks: `cargo test/bench`
  - Formatting: `cargo fmt`
  - Linting: `cargo clippy`
    `[-- -W clippy::pedantic]`
- Third-party funtionality can be added via `cargo install`
  - Recommendation: `cargo-asm`

Configuration

- Central configuration file: `Cargo.toml`
  - *`[profile.dev]` and `[profile.release]` for compilation options*
- Integrated option for LTO (link time optimization): `lto = true/"thin"`
  - *Often important, since proper inter-crate inlining requires explicit `#[inline]` annotation*
- No good support for architecture-specific compilation yet
  (currently: environment variable `RUSTFLAGS="-C target-cpu=native"`)

Summary

- Rust achieves safety and efficiency via the safe/unsafe separation
- No inheritance or metadata: structs are just plain data
- Pattern matching!

  $\Rightarrow$ Also useful for nullability and error handling

  $\Rightarrow$ Rust requires a different approach than typical OO
- Read the compiler errors. Completely.
  Yes, I'm serious!

Next Lecture

- Ownership and Borrowing
- Generics and Traits
- Is Rust object oriented or functional?