Introduction to programming in Rust

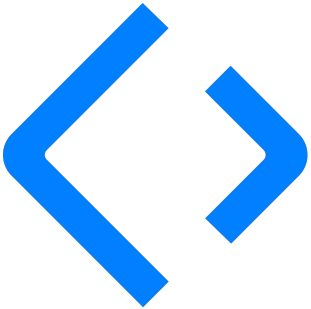
Pascal van Dam

February 12, 2026

Introduction

About the Trainer

- Pascal van Dam, living in Nieuw Bergen (Limburg/NL)
- Owner of Poortier Management B.V / PASCALVANDAM.COM
- Trainer & Consultant Open-Source Solutions:
  - Kubernetes & Containers
  - Virtualization & Cloud
  - Go, Rust, NodeJS, C, C++, Perl
  - Cloud Automation & Orchestration
  - CI/CD Argo, Flux, Gitlab
  - Linux Kernel Internals



Pascal Van Dam

“Let us orchestrate your success!" #K8SMastery

Introduction

Trainer & Student Introduction

About the Student

- Introduce yourself shortly
- Do you have any experience with
  - Rust
  - Go, Zig, Haskell
  - Other programming languages
  - Containers/Kubernetes
  - Linux

About this Course

This course:

- Is developed for professionals that are already familiar with a programming language and would like to get up & running with Rust.
- It will introduce you to the essentials of the Rust programming language
- Enables you to write programs in an idiomatic way in Rust
- Introduces you into the rites and habits of a Rustling/Rustacean

Certification

Currently there is no official certification program for Rust
If you want to standout in the crowd, solve a non trivial problem (OpenSource) and publish it on github or gitlab

Coverage of Rust versions

- Minimal Rust version required: 1.88.0 dd 2025-06-26
- Course updated up to version: 1.88.0 dd 2025-06-26

AGENDA DAY 1/4 PART 1

- Introduction to the course
- Introduction to Rust
- Philosophy behind Rust
- Install and configure your Rust environment
- Anatomy of a Rust program

AGENDA DAY 1/4 PART 2

- My first Rust program
- Variables
- Constants
- Primitive datatypes
- Strings

AGENDA DAY 1/4 PART 3

- Arrays
- Ownership & Borrowing
- Conditionals
- Loops
- Functions

AGENDA DAY 2 PART 1

- Tuples
- Structs
- Enums
- Vectors
- HashMaps
- Options
- Results

AGENDA DAY 2 PART 2

- Errorhandling in Rust
- Structs & OOP
- Traits

AGENDA DAY 3/4

- Closures
- Lifetimes
- Generics
- Rust std library
- Crates
- Building and publishing your own crates

AGENDA DAY 4/4

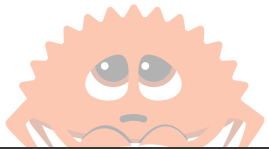
- Cross compiling
- Concurrency in Rust
- Rust in containers
- Webservers in Rust
- Rust and JSON
- Templating in Rust
- Smart pointers

Behaviour Categories in C/C++

Three Categories of "Non-Portable" Behaviour

The C and C++ standards define three categories of behaviour that are not fully specified:

- Implementation-defined behaviour
  - The compiler **must** pick a behaviour and **document** it
  - Consistent and predictable on a given platform
- Unspecified behaviour
  - The compiler picks from a set of allowed options
  - **No obligation** to document or be consistent
- Undefined behaviour (UB)
  - **No requirements whatsoever**
  - The compiler may assume UB **never happens**
  - Can break your entire program



Key Differences at a Glance

How do they compare?

	Documented?	Consistent?	Dangerous?
Implementation-defined	Yes	Yes	Low
Unspecified	No	Not necessarily	Medium
Undefined	No	No	**Extreme**

The crucial insight: with UB, the compiler uses the **assumption that UB never occurs** to optimize away entire code paths.

Implementation-Defined - Trivial 1/3

Size of fundamental types

</> code/impl-defined-1/src/main.c </>

```c
#include <stdio.h>

int main() {
    printf("sizeof(int)  = %zu\n", sizeof(int));
    printf("sizeof(long) = %zu\n", sizeof(long));
    // Could be 4 or 8 for long
    // The compiler documents what it chose
    return 0;
}
```

- The standard only mandates **minimum** sizes
- Your compiler documents the exact sizes it uses

Implementation-Defined - Trivial 2/3

Signedness of char

</> *code/impl-defined-2/src/main.c* </>

```c
#include <stdio.h>

int main() {
    char c = 200;
    printf("c = %d\n", (int)c);
    // Could print 200 (unsigned char)
    // Could print -56 (signed char)
    return 0;
```

- Is char signed or unsigned? The standard doesn't say!
- On x86 Linux (GCC): typically **signed**
- On ARM: typically **unsigned**

Implementation-Defined - Trivial 3/3

Byte order (endianness)

</> *code/impl-defined-3/src/main.c* </>

```c
#include <stdio.h>

int main() {
    int x = 0x01020304;
    unsigned char *p = (unsigned char *)&x;
    printf("First byte: 0x%02x\n", p[0]);
    // Little-endian: 0x04
    // Big-endian:    0x01
    return 0;
}
```

- How multi-byte values are stored in memory
- x86/x64: little-endian, ARM: configurable
- Consistent and documented per platform

Implementation-Defined - Intermediate 1/2

Right-shifting a negative signed integer (before C++20)

</> *code/impl-defined-4/src/main.c* </>

```
1  #include <stdio.h>
2
3  int main() {
4      int x = -8;
5      int y = x >> 1;
6      printf("y = %d\n", y);
7      // Arithmetic shift: y = -4
8      // Logical shift: y = large positive nr
9      return 0;
```

- Before C++20: implementation-defined whether arithmetic or logical
- Most compilers use arithmetic shift (preserving sign)
- C++20 mandates two's complement and arithmetic right shift

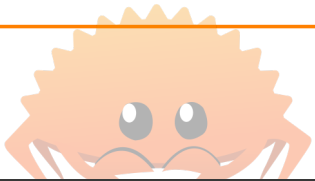
Implementation-Defined - Intermediate 2/2

Struct padding and alignment

</> *code/impl-defined-5/src/main.c* </>

```c
#include <stdio.h>

struct Example {
    char  a;  // 1 byte
    int   b;  // 4 bytes
    char  c;  // 1 byte
};

int main() {
    printf("sizeof = %zu\n",
           sizeof(struct Example));
    // Typically 12, not 6!
    return 0;
}
```

- The compiler inserts padding for alignment
- Exact layout is implementation-defined



Unspecified Behaviour - Trivial 1/3

Order of evaluation in expressions

</> *code/unspecified-1/src/main.cpp* </>

```cpp
#include <iostream>

int a() { std::cout << "a "; return 1; }
int b() { std::cout << "b "; return 2; }

int main() {
    int result = a() + b();
    // Could print "a b" or "b a"
    // Both WILL be called
    // Order is unspecified
```

- The compiler can evaluate `a()` or `b()` first
- This is **not** UB — both are evaluated, just in unknown order
- No obligation to document or be consistent

Unspecified Behaviour - Trivial 2/3

Order of function argument evaluation

</> *code/unspecified-2/src/main.cpp* </>

```cpp
#include <cstdio>

void foo(int a, int b, int c) {
    printf("%d %d %d\n", a, b, c);
}

int main() {
    // Which argument is evaluated first?
    // Unspecified! May vary per build
    foo(f(), g(), h());
```

- Arguments to a function may be evaluated in any order
- May vary between optimization levels

Unspecified Behaviour - Trivial 3/3

Value of a moved-from object (C++)

</> *code/unspecified-3/src/main.cpp* </>

```cpp
#include <string>
#include <iostream>

int main() {
    std::string a = "hello world";
    std::string b = std::move(a);

    // 'a' is in a valid but unspecified state
    std::cout << a.size() << std::endl;
    // Could be 0, could be something else
```

- After `std::move`, the source is in a “valid but unspecified” state
- You can call methods on it, but the value is unknown

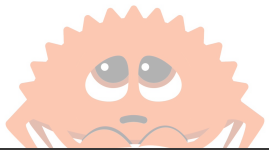
Unspecified Behaviour - Intermediate 1/2

Static initialization order across translation units

</> *code/unspecified-4/src/main.cpp* </>

```cpp
// file_a.cpp
int compute_a();
int global_a = compute_a();

// file_b.cpp
extern int global_a;
int global_b = global_a + 1;
// global_a might not be initialized yet!
// Order across TUs is unspecified
```

- Known as the “Static Initialization Order Fiasco”
- Within one file: top-to-bottom (defined)
- Across files: unspecified! Could change between builds



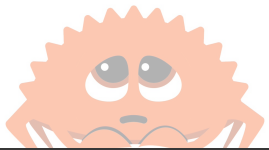
Unspecified Behaviour - Intermediate 2/2

Heap allocation placement

code/unspecified-5/src/main.cpp

```cpp
#include <iostream>

int main() {
    int *a = new int(1);
    int *b = new int(2);

    // Are a and b adjacent in memory?
    // Is a < b or b < a?
    // Completely unspecified
    std::cout << (a < b) << std::endl;

    delete a;
```

- The allocator places objects anywhere on the heap
- Result may vary between runs



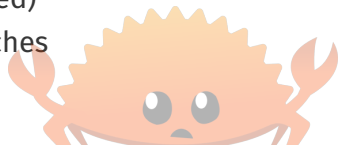
Undefined Behaviour - Trivial 1/3

Signed integer overflow

</> *code/ub-1/src/main.c* </>

```c
#include <limits.h>
#include <stdio.h>

int main() {
    int x = INT_MAX;
    int y = x + 1;
    printf("y = %d\n", y);
    // UB! Compiler assumes this never happens
    // May optimize away overflow checks
    return 0;
```

- Signed overflow is UB (unsigned wraps — well-defined)
- Compilers **exploit** this to remove “impossible” branches



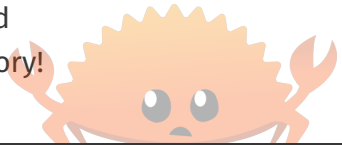
Undefined Behaviour - Trivial 2/3

Dereferencing a null pointer

</> code/ub-2/src/main.c </>

```c
#include <stdio.h>

int main() {
    int *p = NULL;
    *p = 42;
    // UB! Anything can happen:
    // - Segfault (if you're lucky)
    // - Silent corruption
    // - Compiler removes surrounding code
    return 0;
```

- The most classic form of UB
- On most OSes you get a segfault, but not guaranteed
- In embedded systems, address 0 may be valid memory!



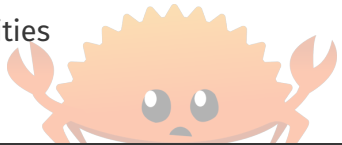
Undefined Behaviour - Trivial 3/3

Out-of-bounds array access

code/ub-3/src/main.c

```c
#include <stdio.h>

int main() {
    int arr[3] = {10, 20, 30};
    printf("%d\n", arr[5]);
    // UB! Reading beyond the array
    // Could return garbage, crash, or
    // compiler removes the function
    return 0;
```

- No bounds checking in C/C++ arrays
- Buffer overflows: the #1 cause of security vulnerabilities



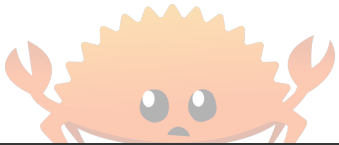
Undefined Behaviour - Intermediate 1/2

Use-after-free and dangling pointers

</> *code/ub-4/src/main.cpp* </>

```cpp
#include <iostream>
#include <string>

std::string* create() {
    std::string s = "hello";
    return &s; // address of local variable!
}

int main() {
    std::string *p = create();
    std::cout << *p << std::endl;
    // UB! 's' destroyed when create() returned
```

- Dangling pointers: pointing to freed memory
- May “work” in debug builds, crash in release builds



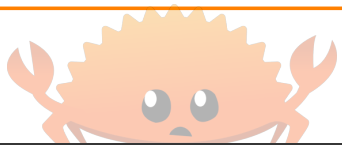
Undefined Behaviour - Intermediate 2/2

Data race in multithreaded code

</> *code/ub-5/src/main.cpp* </>

```cpp
#include <thread>
#include <iostream>

int counter = 0;

void increment() {
    for (int i = 0; i < 100000; i++)
        counter++; // UB! unsynchronized write
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join(); t2.join();
    std::cout << counter << std::endl;
```

- Two threads writing without synchronization = UB
- Fix: use `std::atomic<int>` or `std::mutex`



What does Rust do about it? - Overview

Rust's philosophy: make the **compiler** catch these bugs

- **Implementation-defined** → Fixed-size types
  - `i32`, `u64`, `f32` — no ambiguity
- **Unspecified** → Defined evaluation order
  - Function arguments: left to right
  - No "static initialization order fiasco"
- **Undefined** → Safe Rust has **no UB by design**
  - Ownership prevents use-after-free and data races
  - Bounds checking on array/slice access
  - No null pointers (`Option<T>` instead)
  - Integer overflow: panic in debug, wrap in release

Rust vs C/C++ - Implementation-defined

Rust eliminates ambiguity with explicit types

</> *code/rust-impl-1/src/main.rs* </>

```rust
fn main() {
    let x: i32 = 42;    // Always 32-bit signed
    let y: u8 = 200;    // Always 8-bit unsigned
    let z: f64 = 3.14;  // Always 64-bit float

    // No implicit type conversions!
    // let bad: i32 = y; // Does not compile
    let good: i32 = y as i32; // Explicit
    println!("{x} {good} {z}");
```

- No guessing: i8, i16, i32, i64, i128
- Platform-specific sizes are explicit: isize, usize
- Struct layout controllable with #[repr(C)]

Rust vs C/C++ - No Null Pointers

Null pointer dereference → impossible in safe Rust

</> *code/rust-ub-1/src/main.rs* </>

```rust
fn main() {
    // There is no null in Rust!
    // Instead, use Option<T>:
    let maybe_value: Option<i32> = None;

    // You MUST handle the None case:
    match maybe_value {
        Some(v) => println!("Got: {}", v),
        None    => println!("No value"),
    }
```

- Tony Hoare called null his “billion-dollar mistake”
- Rust’s `Option<T>` makes absence explicit and checked

Rust vs C/C++ - Bounds Checking

Buffer overflow → caught at runtime in Rust

code/rust-ub-2/src/main.rs

```rust
fn main() {
    let arr = [10, 20, 30];
    // println!("{}", arr[5]);
    // Panics at runtime:
    // "index out of bounds: len is 3
    //  but the index is 5"

    // Safe iteration - no index needed:
    for val in &arr {
        println!("{}", val);
    }
}
```

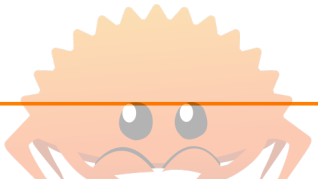
- A panic is **not** UB — it's a controlled, defined crash
- Iterators avoid the need for manual indexing

Rust vs C/C++ - Data Races

Data races → prevented at compile time

</> *code/rust-ub-3/src/main.rs* </>

```rust
use std::thread;
use std::sync::atomic::{AtomicI32, Ordering};
use std::sync::Arc;

fn main() {
    let counter = Arc::new(AtomicI32::new(0));
    let mut handles = vec![];
    for _ in 0..2 {
        let c = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            for _ in 0..100_000 {
                c.fetch_add(1, Ordering::Relaxed);
            }
        }));
    }
    for h in handles { h.join().unwrap(); }
    println!("{}", counter.load(Ordering::Relaxed));
```



But what about unsafe Rust?

Rust does allow UB — but only inside `unsafe` blocks

</> *code/rust-unsafe-1/src/main.rs* </>

```rust
fn main() {
    let x: i32 = 42;
    let p: *const i32 = &x;

    // This is safe:
    println!("{}", x);

    // This requires unsafe:
    unsafe {
        println!("{}", *p);
    }
}
```

- unsafe is an explicit opt-in to manual memory management
- Marks exactly where to look when things go wrong
- The vast majority of Rust code does not need unsafe

Summary

C/C++ gives you power but trusts you completely. Rust verifies.

C/C++ Problem	Category	Rust Solution
`sizeof(int)` varies	Impl-defined	Fixed-size types
`char` signedness	Impl-defined	`u8`/`i8` explicit
Evaluation order	Unspecified	Left-to-right
Moved-from state	Unspecified	Move = transfer
Null dereference	UB	`Option<T>`
Buffer overflow	UB	Bounds checking
Use-after-free	UB	Ownership
Data races	UB	`Send`/`Sync`
Signed overflow	UB	Panic or wrap

Introduction to Rust

Introduction to Rust 1/2

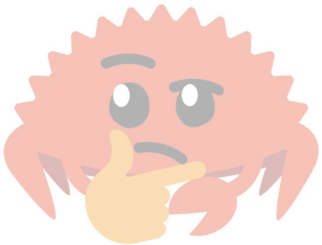
- Started out in 2006 as a personal project by Graydon Hoare
- Sponsored by Mozilla since 2009
- Opened up to the world in 2010
- Stable 1.0 version in 2015
- Est. of Rust Foundation in 2021
- Named after the fungus: Rust
- Moniker of choice: Rustaceans
- Mascot: Ferris the Crab

Introduction to Rust 2/2

- Tobe PCI: 33rd in 2019, 18th in 2020
- Stack Overflow: Most loved language since 2016
- Systems programming language
- Influenced by: SML, OCaml, C++, Cyclone, Haskell, Lisp and Erlang
- Used by companies like: Amazon, ARM, Discord, Dropbox, Google, Meta and Microsoft

What Rust promises

- Safety -> guaranteed @ compiletime
- Fearless concurrecy
- Blazingly fast speed



Rust is ...

- a language with a steep learning curve
- a friendly language
- a language with a large ecosystem
- a so called expression language

Projects realized in Rust

- Firefox/Mozilla
- Linux kernel ext. Linux 6.x
- Redox (Microkernel OS)
- Tauri
- OpenEthereum
- Ruffle (flash emulator)
- Rustdesk

Can I do ... in Rust?

- Procedural programming? -> Yes
- Object Oriented Programming> -> Yes
- Functional Programming? -> Yes
- Concurrent Programming? -> Yes

Cons to Rust

Cons to Rust

- No formal language description...

Cons to Rust

- No formal language description...
- Compile times can take very long...

Cons to Rust

- No formal language description...
- Compile times can take very long...
- Learning curve can be steep...

Cons to Rust

- No formal language description...
- Compile times can take very long...
- Learning curve can be steep...

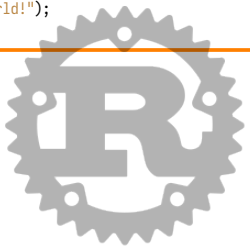
Go Show

Simple helloworld in Rust

- package declaration
- `fn main()`
- semicolons
- `println!`

code/rust-helloworld/src/main.rs

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



Rust Show - http-client example

```rust
use std::collections::HashMap;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let client = reqwest::Client::builder()
        .build()?;

    let res = client
        .get("https://httpbin.org/ip")
        .send()
        .await?;

    let ip = res
        .json::<HashMap<String, String>>()
        .await?;

    println!("{:?}", ip);
    Ok(())
}
```

Rust Show - http-server example

```rust
use warp::{Filter};
use gethostname::gethostname;
use local_ip_address::local_ip;

#[tokio::main]
async fn main() {
    let hello = warp::path!("hello" / String)
        .map(|name| format!("
           Ferris says: hello {} from: {:?} - {:?}\n\n\n",name,gethostname(),local_ip().unwrap()));

    let routes =
        warp::
        get()
            .and(hello);

    let (host , port) = ([0,0,0,0], 3030);
    println!("Starting server on: {}:{}", host.map(|a| a.to_string()).join("."), port);
    warp::serve(routes)
        .run((host, port))
        .await;
}
```

Rust Show - concurrency example

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("Number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
    handle.join().unwrap();
}
```

Install and configure your Rust environment

Installing Rust on Linux, UNIX or OSX

Installation Options for Linux, UNIX and OSX

- Install from distro repos (apt, yum, dnf)
- Download latest using rustup tool from http://rustup.rs

</> *code/install-linux.sh* </>

```
# Install Rust using the rustup tool on Linux, UNIX and Mac OS/X
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Installing Rust on Windows

Installation on Windows

- Download `rustup-init.exe` from https://static.rust-lang.org/rustup/dist/x86_64-pc-windows-msvc/rustup-init.exe
- And follow the instructions

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version

rustup update Updates the Rust toolchain to the latest available version

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version

rustup update Updates the Rust toolchain to the latest available version

rustup self update Updates `rustup` tool

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version
rustup update Updates the Rust toolchain to the latest available version
rustup self update Updates `rustup` tool
rust up check Checks if there are updates

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version
rustup update Updates the Rust toolchain to the latest available version
rustup self update Updates `rustup` tool
rust up check Checks if there are updates
rustup help Provides help to the rustup (sub-commands)

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version
rustup update Updates the Rust toolchain to the latest available version
rustup self update Updates `rustup` tool
rust up check Checks if there are updates
rustup help Provides help to the rustup (sub-commands)
rustup show Shows current toolchain versions

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version
rustup update Updates the Rust toolchain to the latest available version
rustup self update Updates `rustup` tool
rust up check Checks if there are updates
rustup help Provides help to the rustup (sub-commands)
rustup show Shows current toolchain versions
rustup doc Shows documentation of current toolchain

Rustup commands

The `rustup` command gives us access to various functionality of the Rust SDK

rustup -V Shows the current `rustup` version
rustup update Updates the Rust toolchain to the latest available version
rustup self update Updates `rustup` tool
rust up check Checks if there are updates
rustup help Provides help to the rustup (sub-commands)
rustup show Shows current toolchain versions
rustup doc Shows documentation of current toolchain
rustup self remove Removes rustup and Rust installation

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory
cargo init Initializes a new Rust project in existing directory

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory
cargo init Initializes a new Rust project in existing directory
cargo run Builds and runs current Rust project

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory
cargo init Initializes a new Rust project in existing directory
cargo run Builds and runs current Rust project
cargo build Compiles/builds current Rust project

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory
cargo init Initializes a new Rust project in existing directory
cargo run Builds and runs current Rust project
cargo build Compiles/builds current Rust project
cargo clippy Run's clippy/linter against current project

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory
cargo init Initializes a new Rust project in existing directory
cargo run Builds and runs current Rust project
cargo build Compiles/builds current Rust project
cargo clippy Run's clippy/linter against current project
cargo test Run tests from current project

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory
cargo init Initializes a new Rust project in existing directory
cargo run Builds and runs current Rust project
cargo build Compiles/builds current Rust project
cargo clippy Run's clippy/linter against current project
cargo test Run tests from current project
cargo bench Run benchmarks from current project

Rustup commands

The `cargo` command gives us access to various functionalities of the Rust toolchain

cargo new Initializes a new Rust project in a new directory
cargo init Initializes a new Rust project in existing directory
cargo run Builds and runs current Rust project
cargo build Compiles/builds current Rust project
cargo clippy Run's clippy/linter against current project
cargo test Run tests from current project
cargo bench Run benchmarks from current project
cargo clean Clean up project's target directory

IDE and Editors for Rust

- VScode: (or short code with Rust extensions)
- IntelliJ/CLion: add OpenSource Rust plugin
- Vim/Neovim: with vim-rust plugins

Rust Editions

Rust so far has had 3 Editions

- Rust 2015 - The original 1.0
- Rust 2018
- Rust 2021
- Rust 2024 - Current edition

Rust releases or channels

Rust has three channels providing Rust releases:

- Nightly
- Beta - Every 6 weeks
- Stable - 6 weeks after first Beta

My first Rust program

First steps in Rust 1/2

Simple helloworld in Rust

- The entrypoint of every Rust program is the `main()` function
- Reaching the end of main will exit the program
- Standard modules are in the `Prelude`
- Other modules are imported with `use`
- Semi columns are mandatory
- Prelude contents: https://doc.rust-lang.org/stable/std/prelude/index.html

code/rust-helloworld/src/main.rs

```
1  fn main() {
2      println!("Hello, world!");
3  }
```

First Steps in Rust 2/2

Using a module

code/rust-rnd/src/main.rs

```rust
use rand::Rng;

fn main() {
    let mut range = rand::thread_rng();

    let num: i32 = range.gen();

    println!("Random: {}", num);
}
```

Rust general syntactics

- The language is case sensitive
- `main()` is a reserved function
- Blocks are enclosed in curly braces {}
- Semicolons are mandatory
- Except for `tail expressions`
  - With semicolon -> statement
  - Without semicolon -> expression

Setting up a new Rust project

- Use |cargo new <prjname>| to setup a new Rust project
  - The <prjname> directory will be created
  - Cargo.toml will be created
  - Diverse GIT files *.gitignore*
  - The `src` directory
  - In `src/main.rs` a template Rust program

Running a Rust program

- To run, use `cargo run`
- and check the output in a terminal

Compiling/building a Rust program

- To build a Rust program for test/debug, use `cargo build`
- To build for release use `cargo build --release`
- The resulting binary will be in the target directory

Cross compiling

Two options:

- Using `rustup`
- Using `cargo cross`

Cross compiling using Rustup

- Add Rust target toolchain with `rustup`
- Add cross compiler toolchain to OS setup
- Use `cargo` to compile to the new target

```
1  rustup target add aarch64-unknown-linux-gnu
2  sudo apt-get install gcc-aarch64-linux-gnu g++-aarch64-linux-gnu
```

Cross compiling using Cargo cross

- External cargo package
- Uses `docker` or `podman` for cross compiling
- Can use `qemu` for testing

```
cargo install cross

sudo apt install podman -y

cd march
cross build --target riscv64gc-unknown-linux-gnu
```

Comments in Rust

- Comments in Rust serve 2 purposes:
  - Improving readability & clarity of source code
  - As input for Rust doc as integrated documentation

</> code/rust-comment/src/main.rs </>

```rust
// Single line comment

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

</> code/rust-ml-comment/src/main.rs </>

```rust
/*
 * This is a multiline comment
 *
 */

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

Variables in Rust

- Rust is a statically typed language
- Variable declarations reserve memory for a specific type and value
- The memory location is identified by the name of the variable

Variables in Rust

- Variables are by default immutable after assignment of a value
- Immutable variables cannot get a new value assigned
- Variables can be made mutable using the `mut` prefix

```
1  fn main() {
2
3      let color = "red";
4      let mut n = 0 ;
5
6      n = 42 ;
7
8      println!("n == {}\n",n) ;
9  }
```

Constants in Rust

- Immutable variables are NOT constants
- Consts are literals and are inlined
- Statics do have a known memory location
- Choose `const` over `static`

```
1   const MY_NR: usize = 10 ;
2   static PI2: f64  = 6.28 ;
3
4   fn main() {
5
6       let mut i = MY_NR ;
7
8       println!("{}",PI2) ;
9
10      while i>0 {
11          println!("{}", i) ;
12          i -= 1 ;
13      }
14  }
```

Constants in Rust: const

A `const`

- Represents a value that will get inlined
- Always immutable
- Has not lifetime *inlined*
- Initialization at compile-time *constantexpression*
- Can be of any type (evaluated at compile time)
- No dereferencing possible
- Naming convention => `SCREAMING_SNAKE_CASE`

Constants in Rust: static

A `static`

- Represents a memory location
- Immutable by default
- Static lifetime => the lifetime of the program
- Initialization at run-time with constant expression
- Can be of any type even references
- Dereferencing IS possible
- Naming convention => `SCREAMING_SNAKE_CASE`

Rust primitive datatypes

Rust Primitive Datatypes

Primitive datatypes

- Integers
- Floats
- Bool
- Char
- String literals

Integers

- Signed integers (i8, i16, i32, i64, i128, isize)
- Unsigned integers (u8, u16, u32, u64, u128, usize)

Floats

- f32
- f64

Bool

- true
- false

Char

- Represents a unicode character
- Always 4 bytes in size

String Literal

- Type: &str
- Slice (&[u8]) that always points to a valid UTF-8 sequence

Ownership

Memory management concepts 1/4

Stack vs Heap

- Stack
  - Fast
  - Size of variable must be known
  - LIFO

Memory management concepts 2/4

Stack vs Heap

- Heap
  - Slow
  - Size of variable can be unknown
  - Who frees up allocated memory?

Memory management concepts 3/4

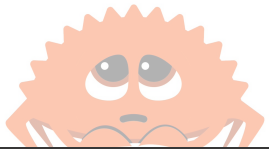
Explicit or implicit memory management:

- Explicit
  - Explicit memory allocation for compound types
  - Lot's of control
  - Lot's of bugs
  - Fast
  - Used by C, C++

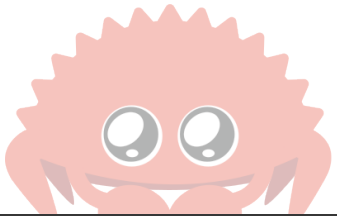
Memory management concepts 4/4

Explicit or Implicit memory management:

- Implicit
  - Garbage Collection
  - Limited control
  - No bugs
  - Freezes/lag spikes
  - Java, JS, Go etc



Can’t we have both? Fast and safe?
Yes we can! In Rust, By taking ’ownership’!!



The Three Rules of Ownership in Rust

Three ownership rules

- Each variable is the owner of it's initialized value
- A variable can only have 1 owner at a time
- When the owner goes out of scope, the value will be dropped

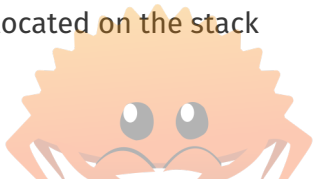
Ownership rules - 1/3

Each variable is the owner of it's initialized value

code/ownership-1/src/main.rs

```
1  // ownership_1
2
3  fn main() {
4
5      let a: i64 = 42 ;
6      let b = a;
7
8      println!("a = {0} and b = {1}",a,b) ;
9  }
```

- Var 'a' and 'b' are primitive data types and as such allocated on the stack
- Both 'a' and 'b' own their own values



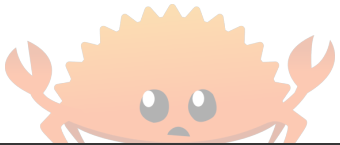
Ownership rules - 2/3

A variable can only have 1 owner at a time

</> code/ownership-3/src/main.rs </>

```
1  // ownership_3
2
3  fn main() {
4
5      let a = String::from("one");
6      let b = a ;
7
8      println!("a == {0} and b == {1}",a,b) ;
9  }
```

- Var 'a' and 'b' are compound datatypes and as such allocated on the heap
- Ownership 'a' => 'b'
- Var 'a' is not valid anymore
- Does not compile!



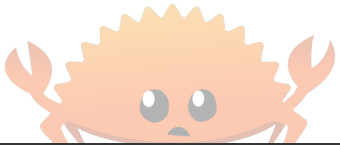
Ownership rules - 3/3

When the owner goes out of scope, the value will be dropped

</> *code/ownership-3c/src/main.rs* </>

```
1  // ownership_3
2
3  fn main() {
4
5      let a = String::from("one");
6
7      {
8          let b = a;
9          println!("b = {0}",b) ;
10     }
11     println!("b = {}",b) ;
12 }
```

- Var 'b' is defined in a new scope
- Var 'b' is dropped when it goes out of scope
- Does not compile!
- Is var 'a' available?



How to solve ownership move problems?

For primitive types the `copy` trait is implemented and automatically called upon assignment

- All integers
- All floats
- Booleans
- Char
- String literals *&str*

How to solve ownership move problems?

For compound types like `String`, `Vec` etc. there are two solutions:

- Using the `clone` method if implemented
- Use borrowing

</> *code/ownership-4/src/main.rs* </>

```
1  // ownership_3
2
3  fn main() {
4
5      let a = String::from("one");
6      let b = a.clone();
7
8      println!("a = {0} and b = {1}",a,b) ;
9  }
```

Manually implementing the Clone trait

We can implement the Clone() trait manually:

</> *code/ownership-6/src/main.rs* </>

```
#[derive(Debug)]
struct Person {
    first_name: String,
    nr_of_children: i32,
    childrens_ages: Vec<i32>,
}

impl Clone for Person {
    fn clone(&self) -> Self {
        Person {
            first_name: self.first_name.clone(),
            nr_of_children: self.nr_of_children,
            childrens_ages: self.childrens_ages.clone(),
        }
    }
}
```

Deriving the clone method on a datatype

For compound types like `structs` etc we can use the `#[derive(Clone)]` macro

code/ownership-5/src/main.rs

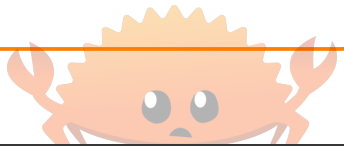
```
#[derive(Clone, Debug)]
struct Person {
    first_name: String,
    nr_of_children: i32,
    childrens_ages: Vec<i32>,
}

fn main() {
    let person1 = Person {
        first_name: "Pascal".to_string(),
        nr_of_children: 4,
        childrens_ages: vec![8, 17, 19, 20],
    };

    let person2 = person1.clone();
    println!("{:?}", person2);
    println!("{:?}", person1);
}
```

Insight: compound type of primitives

How to solve the move issue for a compound type consisting only of primitives?

</> code/ownership-7/src/main.rs </>

```rust
#[derive(Debug)]
struct Complex {
    real_part: f64,
    img_part: f64,
}

fn main() {
    let c1 = Complex {
        real_part: 0.3,
        img_part: 2.0,
    };

    let c2 = c1;
    println!("{:#?} {:#?}", c1, c2);
}
```

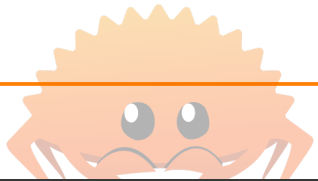


Insight: compound type of primitives

Solution; implement/derive the Copy trait:

</> *code/ownership-7a/src/main.rs* </>

```rust
#[derive(Debug, Copy, Clone)]
struct Complex {
    real_part: f64,
    img_part: f64,
}

fn main() {
    let c1 = Complex {
        real_part: 0.3,
        img_part: 2.0,
    };

    let c2 = c1;
    println!("{:#?} {:#?}", c1, c2);
}
```



Concept of Borrowing

Borrowing is the concept of trying to ease the burden of ownership movement.

- Avoiding the move of ownership
- Providing a reference to the data
- A receiver of a reference can temporary use the value without taking ownership of it
- A reference is passed using by using the & prefix

</> *code/borrowing-1/src/main.rs* </>

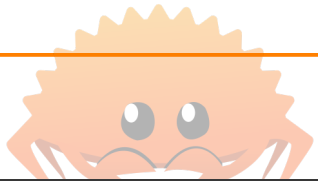
```
1  // borrowing_1
2
3  fn main() {
4
5      let a = String::from("one");
6      let b = &a ;
7
8      println!("a == {} and b == {}", a,b) ;
9  }
```

Borrowing Rules - 1/4

At any time there maybe any nr of immutable references at the same time als longs as there is no mutable reference.

code/borrowing-2/src/main.rs

```
1  // borrowing_2 - multi immutable borrows
2
3  fn main() {
4
5      let a = String::from("Hello ");
6      let r1 = &a ;
7      let r2 = &a ;
8      let r3 = &a ;
9
10     println!("a == {} and r1 = {}, r2 = {}, r3 = {}", a,r1,r2,r3) ;
```



Borrowing Rules - 2/4

A reference will always at all times point to a valid value

</> code/borrowing-3/src/main.rs </>

```rust
// borrowing_2 - borrow should point to valid data

fn main() {
    let s = String::from("Hello ");
    let b = &s;

    println!("s = {}", s);
    println!("b = {}", s);

    drop(s);
    println!("b = {}", b);
}
```

- This does not compile

Borrowing Rules - 3/4

References cannot live longer than their owners

</> code/borrowing-4/src/main.rs </>

```
1  // borrowing_4 - a reference cannot outlive the owner's data
2
3  fn main() {
4
5      let r;
6
7      {
8          let x = 5;
9          r = &x;
10     }
```

- This does not compile

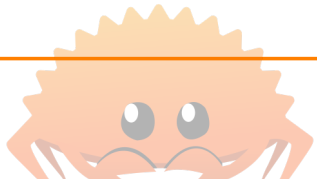
Borrowing Rules - 4/4

There can only be one mutable reference to a variable at the same time

code/borrowing-5/src/main.rs

```rust
// borrowing_5 - there can only be one mutable borrow

fn main() {
    let mut a = [1, 2, 3, 4];
    println!("{:?}", a);
    {
        let b = &mut a[0..2];
        // let c = &mut a[3];
        println!("b: {:?}", b); // [1, 2]
        b[0] = 42;
        println!("b: {:?}", b); // [42, 2]
        println!("a: {:?}", a);
    }
    println!("a: {:?}", a); // [42, 2, 3, 4]
}
```

- This does not compile



Rust Tuples

Rust Tuples

Tuples

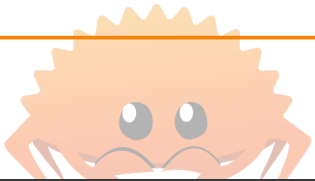
- Compound type
- Sequences of elements
- Heterogenous
- Fixed length
- Printed using debug trait

Usage of tuples

How to use tuples in Rust?

</> *code/tuples-1/src/main.rs* </>

```rust
// Example of the use of tuples in Rust

fn main() {
    let person_data = ("Hector", 48, "72kg", "181cm");
    println!("{} is aged {}",person_data.0,person_data.1);

    let solutions: (f64, f64) = (-1.0,2.0);
    let (x1, x2) = solutions;
    println!("x1 = {}, x2= {}",x1,x2);
}
```



Rust Arrays

Rust Arrays

Arrays

- Homogenous sequence of elements
- Fixed length
- Elements accessed by index nr
- First element is zero

Usage of Arrays

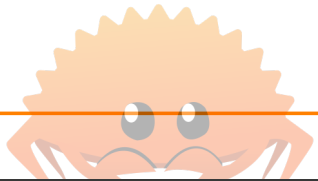
How to use arrays in Rust?

- Element addressing using [n]
- Printing using debug trait
- Printing by iterating over array
- Use .len() to get nr of elements

Usage of Arrays

code/arrays-1/src/main.rs

```rust
fn main() {

    let mut my_array: [i32;4] = [0,1,2,3];
    let days = ["Mo","Tu","We","Th","Fr","Sa","Su"];

    println!("{:?}",days);
    println!("Day 1: {}",days[0]);

    for (i, d) in days.iter().enumerate() {
        println!("element {} {} ",i,d);
    }

    for n in my_array.iter_mut()  {
        *n = *n * 2;
    }

    println!("{:?}",my_array);

    println!("A week has {} days.",days.len());
}
```



Rust Strings

Rust Strings

String literals `&str`

- Primitive type
- Fixed size
- Have the copy trait
- Are borrowed
- Value of string is known at compile-time
- Immutable
- Non-zero terminated

Usage of &str

How to use literal strings \&str in Rust?

- Created by assigning a literal string to a variable

</> code/str-1/src/main.rs </>

```
1  fn main() {
2
3      let my_str = "This is a literal string";
4      println!("{}",my_str);
5
6  }
```

Rust Strings

String objects `String`

- Strings are growable
- Strings have no copy trait
- Strings are owned
- Encoded in UTF-8
- Allocated on the heap

Creating String objects

How to create object Strings in Rust?

- Create an empty String
- Creating an initialized `String` object
- Converting from a string literal \&str

code/string-1/src/main.rs

```rust
fn main() {
    // Creating an empty String object
    let mut s1 = String::new();
    s1.push_str("Hello ");
    s1.push_str("Rustaceans");
    println!("{}",s1);
    // Creating an initialized String object
    let s2 = String::from("My initialized string");
    // Creating a string object by converting a &str
    let str = "Another string";
    let s3 = str.to_string();
}
```

Additional methods on String objects

There are quite a few more methods on `String` objects we can use:

- `s.capacity()` Returns the capacity of the String.
- `s.length()` Returns the capacity of the String.
- `s.trim()` Remove leading/trailing whitespaces
- `s.contains()` Find substring in String
- `s.replace()` Replaces substring in String

</> *code/string-2/src/main.rs* </>

```
1  fn main() {
2      let s1 = String::from("Hello Gophers");
3      let look_for = "Gophers";
4      let replace_with = "Rustaceans";
5      let r1 = s1.replace(look_for,replace_with);
6      println!("{} len = {}, cap = {}",s1,s1.len(),s1.capacity());
7      println!("{} len = {}, cap = {}",r1,r1.len(),r1.capacity());
8  }
```

Iterating over String objects 1/3

Iterate over words in a String:

code/strings-3/src/main.rs

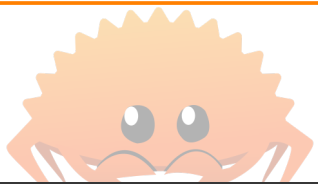
```
1  fn main() {
2      let s1 = String::from("In Rust we trust");
3      for word in s1.split_whitespace() {
4          println!("- {}",word);
5      }
6  }
```

Iterating over String objects 2/3

Iterate using a separator:

</> *code/strings-4/src/main.rs* </>

```
1  fn main() {
2      let s1 = String::from("Perlmonks,Pythonistas,Gophers,Rustaceans");
3      for word in s1.split(',') {
4          println!("- {}",word);
5      }
6  }
```

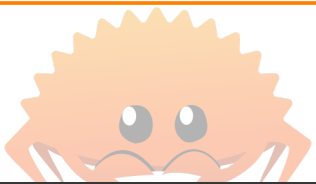


Iterating over String objects 3/3

Iterate over individual characters in a String:

</> *code/strings-5/src/main.rs* </>

```
1  fn main() {
2      let s1 = String::from("Hello Rustaceans!");
3      for c in s1.chars() {
4          print!("{} ",c);
5      }
6  }
```



Updating / Growing Strings

One can update/grow a string by:

- `s.push` Adds a UNICODE character to the string
- `s.push_str)` Concatenates a string to the string
- `+ operator` Adds a string slice (\&str) to the string
- `format!` This macro returns a formatted string slice

</> *code/strings-6/src/main.rs* </>

```
1  // Updating / growing Strings
2
3  fn main() {
4      let mut s1 = String::from("Welcome");
5      println!("{} {:p}",s1.capacity(),&s1);
6      let mut s3 = String::from("BMW");
7      println!("{} {:p}",s3.capacity(),&s3);
8
9      s1.push(' ');
10     println!("{} {:p}",s1.capacity(),&s1);
```

Slicing Strings

One can slice strings:

</> *code/strings-7/src/main.rs* </>

```
1  fn main() {
2      let s1 = String::from("Learning Rust");
3      let slice1 = &s1[9..];
4      println!("{}",slice1);
5  }
```

Rust Conditionals

Rust Conditionals

Rust has 2 conditionals in 4 variants:

- `if/else`
- `let/if`
- `match`
- `match/if`

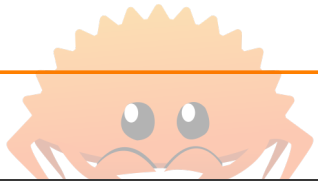
If conditionals

The if and if/else conditional does not differ much from other C like languages

- No parentheses needed
- Tail expressions are possible/recommended

</> *code/conditionals-1/src/main.rs* </>

```rust
fn main() {
    let lang = "Rust";

    if lang == "Rust" {
        println!("We are learning Rust");
    } else {
        println!("We are learnning something else");
    }
}
```



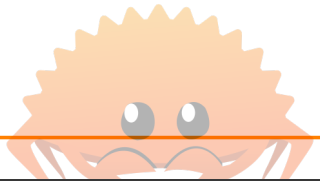
If let expressions

Remember that in Rust it's either an expression or a statement

- `let` expressions allow conditional assignment to variables
- Blocks {} can also contain a tail expression

</> *code/conditionals-2/src/main.rs* </>

```
1  use chrono::{Timelike, Utc};
2
3  fn do_something() {
4      println!("Busy")
5  }
6
7  fn main() {
8      let now = Utc::now();
9      let (is_pm, hour) = now.hour12();
10     let min = now.minute();
11
12     // If expression to determine AM/PM string
13
14     let md = if is_pm { "PM" } else { "AM" };
15
16     println!("It is {}:{} {} UTC", hour, min, md);
```



Match expressions

The match expression in rust ressembles a switch statement in C

- Match expressions are much flexible
- Here "_" ressembles the default clause

</> *code/conditionals-3/src/main.rs* </>

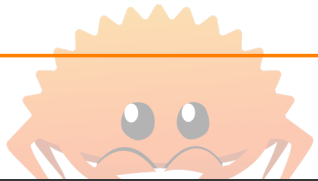
```
1  // Conditionals-3 Match Expression
2
3  fn main() {
4      let code = 3;
5
6      match code {
7          0..=4 => println!("All good"),
8          5 => println!("OSI Layer 8 problem"),
9          6 | 10 => println!("Printer on fire"),
10         _ => println!("Unidentified Error {}", code),
11     }
12 }
```

Match let expressions

The match expression also has it's variant

</> code/conditionals-4/src/main.rs </>

```rust
fn main() {
    let lang = "Rust";

    // return value of match expression in a variable
    let programmer = match lang {
        "Rust" => "Rustacean",
        "Go" => "Gopher",
        "Python" => "Pythonista",
        "Perl" => "Perl Monk",
        _ => "Unknown",
    };
    println!("Some one programming {} is called a {} ", lang, programmer);
}
```



Match guards

code/match-3/src/main.rs

```rust
struct Item {
    weight: f64,
    fragile: bool,
}

fn determine_shipping_cost(item: &Item) -> f64 {
    match item.weight {
        w if w < 1.0 && item.fragile => 5.0, // Fragile light items
        w if w < 1.0 => 3.0,                 // Non-fragile light items
        _ if item.fragile => 20.0,           // Fragile heavy items
        _ => 15.0,                           // Non-fragile heavy items
    }
}

fn main() {
    let light_fragile_item = Item {
        weight: 0.5,
        fragile: true,
    };
    println!(
        "Light Fragile Item Shipping Cost: ${}",
        determine_shipping_cost(&light_fragile_item)
    );
}
```

Rust Loops

Rust Loops

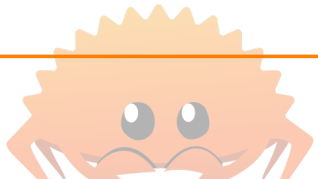
Rust has 4 looping expressions

- The `while` loop
- The `while let` loop
- The indefinite `loop`
- The `for .. in ..` loop

The `while` loop

code/loops-1/src/main.rs

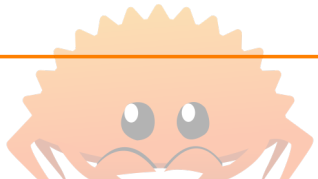
```
1  use std::{thread, time};
2  fn main() {
3      let mut n = 10;
4      let delay = time::Duration::from_secs(1);
5
6      while n ≥ 0 {
7          println!("{}",n);
8          thread::sleep(delay);
9          n-=1;
10     }
11     println!("\n\nLift off!");
12 }
```



The `while let` loop

</> code/loops-2/src/main.rs </>

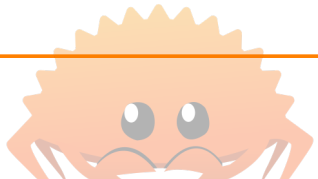
```rust
fn main() {
    let mut optional = Some(0);

    while let Some(i) = optional {
        if i > 9 {
            println!("Greater than 9, quit!");
            optional = None;
        } else {
            println!("`i` is `{:?}`. Try again.", i);
            optional = Some(i + 1);
        }
    }
```



The indefinite `loop`

</> *code/loops-3/src/main.rs* </>

```
1  fn main() {
2      let mut n = 0;
3      loop {
4          n += 1;
5          if n == 4 {
6              println!("Skip");
7              continue;
8          }
9          println!("{}", n);
10         if n == 8 {
11             println!("Break out after {}",n);
12             break;
```



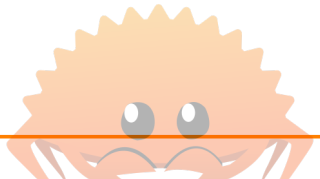
Rust loops - break and continue

Rust loops with breaks

- Also works with `while` and `for .. in ..` loops
- Since Rust 1.65 and later works also for any code block

</> *code/loops-4/src/main.rs* </>

```rust
fn main() {
    let mut i = 0;
    loop {
        let mut j = 0;
        loop {
            if j == 3 {
                break;
            }
            j+=1;
            println!("i = {}, j = {}",i,j);
        }
        if i == 2 {
            break;
        }
        i+=1;
    }
```



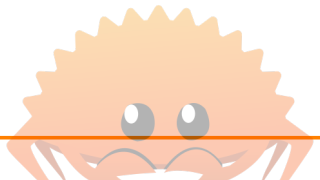
Rust loops - break and continue with labels

Controlled breaking out and continuing with labels:

- Also works with `while` and `for .. in ..` loops
- Since Rust 1.65 and later works also for any `code` block

code/loops-5/src/main.rs

```
1  fn main() {
2      let mut i = 0;
3      'outer: loop {
4          let mut j = 0;
5          'inner: loop {
6              if j == 3 {
7                  break 'outer;
8              }
9              j+=1;
10             println!("i = {}, j = {}",i,j);
11         }
12         if i == 2 {
13             break;
14         }
15         i+=1;
16     }
```



Rust Loops - the for loop

The `for .. in ..` loop

</> *code/loops-6/src/main.rs* </>

```
1  use std::{thread, time};
2  fn main() {
3      let delay = time::Duration::from_secs(1);
4      for n in 0..=10 {
5          println!("{}", 10 - n);
6          thread::sleep(delay);
7      }
8      println!("\n\nLift off!");
9  }
```

Rust Loops - the for loop with breaks

The `for in ..` loop with `break`s

- This also works for `continue` of course

</> *code/loops-7/src/main.rs* </>

```
1  use std::{thread, time};
2  fn main() {
3
4      let delay = time::Duration::from_secs(1);
5      for n in 0..=10 {
6          println!("{}", 10 - n);
7          thread::sleep(delay);
8          if n == 7 {
9              println!("Launch abortion at t-{}s", 10 - n);
10             break;
11         }
12         if n == 10 {
13             println!("Lift off!");
14         }
15     }
16 }
```

Rust Enums

Rust Enums

In rust Enums

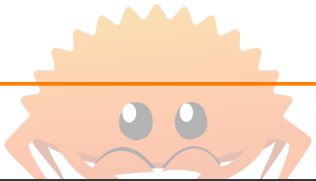
- Are custom type
- Composed of variants
- Used to enumerate values
- Used a lot in Rust (Option, Result)
- Naming convention: `UpperCamelCase`

How to use Enums?

To declare, initialize and use Enums in Rust:

</> code/enums-1/src/main.rs </>

```
1  // enums-1
2  //
3  #[derive(Debug)]
4  use WeekDays::*
5     enum WeekDays {
6          Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday
7     }
8
9  fn main() {
10
11    let d1 = Monday ;
12    let d2= Wednesday ;
13
14    println!("{:?} {:?}",d1, d2);
15 }
```



Type of Enums in Rust

Enum types in Rust:

- Basic enums
- Enums with data
- Option enums
- Result enums

Basic enums

Use case: well defined set of possible values without data:

- Days of week
- Suite of cards (Spades, Hearts, Diamonds Clubs)

</> *code/enums-10/src/main.rs* </>

```
1  // This brings the variants of the enum into scope
2  use Direction::*;
3
4  #[derive(Debug)]
5  enum Direction {
6      North,
7      South,
8      East,
9      West,
10 }
11
12 fn main() {
13     println!("{:?}", Direction::North);
14     println!("{:?}", South);
15     println!("{:?}", West);
16     println!("{:?}", East);
17 }
```

Enums with data

Use case: Representing different kind of actions/data structures etc.

- Different kind of messages in a messaging system

</> code/enums-11/src/main.rs </>

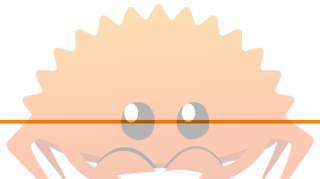
```rust
enum Shape {
    Circle(f64),         // f64 represents the radius
    Rectangle(f64, f64), // Two f64 values represent width and height
}

fn main() {
    let circle = Shape::Circle(5.0); // A circle with radius 5.0
    let rect = Shape::Rectangle(4.0, 6.0); // A rectangle with width 4.0 and height 6.0

    // Calculate and print area for each shape
    print_area(circle);
    print_area(rect);
}

fn print_area(shape: Shape) {
    match shape {
        Shape::Circle(radius) => println!("Circle area: {}", 3.14 * radius * radius),
        Shape::Rectangle(width, height) => println!("Rectangle area: {}", width * height),
    }
}
```

Option enums

Use case: return values for functions representing both present as well absent (useful) data

</> *code/enums-12/src/main.rs* </>

```rust
fn find_element(arr: &[i32; 5], value: i32) -> Option<i32> {
    for &item in arr.iter() {
        if item == value {
            return Some(item);
        }
    }
    None
}

fn main() {
    let numbers = [1, 2, 3, 4, 5];

    match find_element(&numbers, 3) {
        Some(val) => println!("Found: {}", val),
        None => println!("Not found"),
    }
}
```



Result enums

Use case: return values for functions representing valid return value or error message

</> *code/enums-13/src/main.rs* </>

```rust
fn divide(numerator: f64, denominator: f64) -> Result<f64, &'static str> {
    if denominator == 0.0 {
        Err("Cannot divide by zero!")
    } else {
        Ok(numerator / denominator)
    }
}

fn main() {
    match divide(10.0, 2.0) {
        Ok(result) => println!("Result: {}", result),
        Err(e) => println!("Error: {}", e),
    }

    match divide(10.0, 0.0) {
        Ok(result) => println!("Result: {}", result),
        Err(e) => println!("Error: {}", e),
    }
}
```

Rust Structs

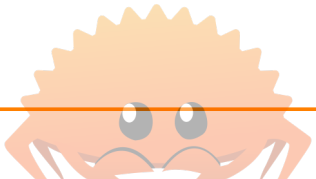
Rust Structs

Structs are like tuples, but in a struct the 'fields' are named and typed;

- Structs belong to the custom-types.
- Naming convention: PascalCase / `UpperCamelCase`

</> *code/structs-1/src/main.rs* </>

```
1  // structs-1
2  fn main() {
3      struct Rectangle {
4          width: f64,
5          length: f64
6      }
7      struct DevLang {
8          name: String,
9          mascot: String,
10         moniker: String,
11         year: u16
12     }
13 }
```

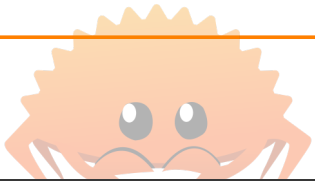


How to initialize and access Structs?

Structs must be declared and initialized in one step like this:

</> *code/structs-2/src/main.rs* </>

```
1  // structs-1
2  fn main() {
3      struct Rectangle {
4          width: f64,
5          length: f64
6      }
7
8      let my_rect = Rectangle { width: 2.0, length: 6.0 };
9      println!("Rectangle perimeter is: {}",my_rect.width*2.0 + my_rect.length*2.0) ;
10 }
```



Nested Structs

In Rust Structs can also be nested in Structs

</> *code/structs-3/src/main.rs* </>

```
1  fn main() {
2      struct Coord { x: u16, y: u16, z: u16 }
3      struct Pixel {
4         c: Coord,
5         color: String
6      }
7
8      let c1 = Coord { x: 1, y: 1, z: 1 };
9      let my_color = String::from("blue");
10
11     let p = Pixel{c: c1, color: my_color};
12
13     println!("{}-{}-{} in {} color",p.c.x, p.c.y, p.c.z,p.color) ;
14 }
```

Tuple Structs

In Rust we use have so called truple structs. Use cases:

- To Give Meaning to Primitive Types: `struct Point(f64,64)`
- To differentiate between NewTypes
- To implement traits on Tuples

</> *code/structs-5/src/main.rs* </>

```rust
struct MyPair(i32, i32);

trait PairSum {
    fn pair_sum(&self) -> i32;
}

impl PairSum for MyPair {
    fn pair_sum(&self) -> i32 {
        self.0 + self.1
    }
}

fn main() {
    let my_pair = MyPair(5, 7);
    println!("Sum: {}", my_pair.pair_sum());
}
```

Tuple Structs

Unit structs can be used for type-level distinction without holding any data:

</> *code/structs-6/src/main.rs* </>

```rust
struct Guest;
struct User;
struct Admin;

fn can_access_dashboard(user: &User) -> bool {
    true
}

fn can_access_admin_panel(user: &Admin) -> bool {
    true
}

fn can_access_public_content(_user: &Guest) -> bool {
    true
}

fn main() {
    let guest = Guest;
    let user = User;
    let admin = Admin;

    println!(
        "Guest can access public content: {}",
```

Rust Vectors

Rust Vectors

Vectors in Rust are the resizable variant of Arrays

- Size is unknown
- Vectors can grow and shrink
- Pointer to data
- Length (nr of items)
- Capacity

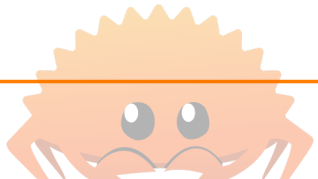
How to create Vectors?

Vectors in Rust can be created in two ways:

- Using the `Vec:new()` method
- Using the `vec!` macro

code/vectors-1/src/main.rs

```rust
// vectors-1

fn main() {
    let mut v1: Vec<&str> = Vec::new();
    let v2 = vec!["apples", "oranges", "mangos"];

    v1.push("Rust");
    v1.push("Go");

    println!("{:?}", v2);
```



How to access data in Vectors?

Data in vectors in rust can be accessed;

- Array-like using an index
- Using the v.pop() method
- Using the v.get() method
- Using an iterator

How to access data in Vectors?

code/vectors-2/src/main.rs

```
// vectors-2

fn main() {
    let mut v2 = vec!["apples", "oranges", "mangos"];

    println!("Using index, element 2 = {}",v2[1]);
    let popped_fruit = v2.pop() ;
    println!("Popped {}",popped_fruit.unwrap());

    for (i, f) in v2.iter().enumerate() {
        println!("In position {} we have the fruit {}", i, f);
    }
}
```

Protection against out-of-bounds: index

When trying to access out-of-bounds data, Rust will PANIC. To prevent this:

- Use the v.get() method
  - If there is data, it will return `Some(data)`
  - If there is no data, it will return `None`

Protection against out-of-bounds: index

code/vectors-3/src/main.rs

```
// vectors-3 -- with out-of-bounds handling

fn main() {
    let v1 = vec!["apples", "oranges", "mangos"];

    for n in 0..=4 {

        match v1.get(n) {
        Some(x) => println!("Value at given index {} {}", n,x),
        None => println!("Sorry, can't do -> index {} is out-of-bounds",n)
        }
    }
}
```

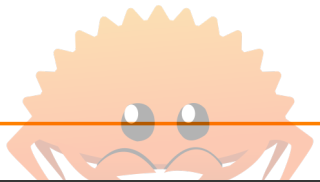
Finding out the vector is empty : pop()

When we try to pop beyond the last element in the vector, Rust will not PANIC. But, how to find out there is no data anymore?

- Using the v.pop() method
  - If there is data, it will return `Some(data)`
  - If the vector is empty, it will return `None`
- This is a common pattern

</> *code/vectors-4/src/main.rs* </>

```
1  // vectors-4 -- Popping till the vector is empty
2
3  fn main() {
4      let mut v1 = vec!["apples", "oranges", "mangos", "kiwis"];
5
6      while let Some(value) = v1.pop() {
7          println!("Popped fruit: {}", value);
8      }
9  }
```



Adding elements to a vector

To add elements to a vector:

- Using the v.pop() method
  - If there is data, it will return `Some(data)`
  - If the vector is empty, it will return `None`
- This is a common pattern

Adding elements to a vector

code/vectors-5/src/main.rs

```
1  // vectors-5 -- Adding items to a vector
2
3  fn main() {
4      let mut v1 = vec!["apples", "oranges", "mangos"];
5
6      v1.push("Bananas");
7      v1.push("Ananas");
8      v1.push("Jackfruit");
9
10     for f in v1.iter() {
11         println!("Item: {}",f);
12     }
13 }
```

Removing elements to a vector

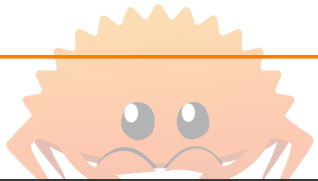
To remove elements from a vector:

- `v.pop()` removes the last element
- `v.remove()` removes the element at given index nr

Removing elements to a vector

code/vectors-6/src/main.rs

```
1  // vectors-6 -- Removing elements
2
3  fn main() {
4      let mut v1 = vec!["apples", "oranges", "mangos", "Bananas", "Jackfruit"];
5      println!("{:?}",v1) ;
6
7      v1.pop();
8      v1.remove(0);
9
10     println!("{:?}",v1) ;
11
12     println!("Vector v1; length = {}, capacity = {}",v1.len(), v1.capacity());
13
14 }
```



Iterating and mutating

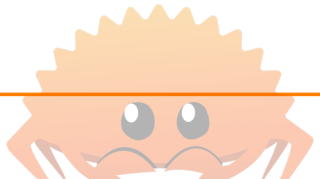
To iterate and mutate elements in a vector:

- User `iter.mut()` instead of `iter()`

Iterating and mutating

</> code/vectors-7/src/main.rs </>

```
1  // vectors-7 -- Iterate & Mutate
2
3  fn main() {
4      let mut v1 = vec!["apples", "oranges", "mangos", "Bananas", "Jackfruit"];
5
6      for i in v1.iter_mut() {
7          *i = "Sold out";
8      }
9
10     for j in v1.iter() {
11         println!("- {}",j);
12     }
13
14     let mut v2 = vec![1,2,3,4,5,6,7,8,9,10];
15     for i in v2.iter_mut() {
16         *i = *i * *i ;
17     }
18     for j in v2.iter() { println!("- {}",j); };
```



Slicing Vectors

We can slice Vectors like we slices Arrays in Rust

</> *code/vectors-8/src/main.rs* </>

```
1  // vectors-8 -- Slicing vectors
2
3  fn main() {
4      let v1 = vec!["apples", "oranges", "mangos", "bananas", "jackfruit"];
5
6      let slice1 = &v1[3..];
7
8      for i in slice1.iter() {
9          println!("{}",i);
10     }
11 }
```

Rust HashMaps

Rust HashMaps

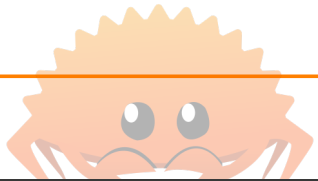
HashMaps in Rust;

- Store values by key
- Keys can be booleans, integers, strings
- Needs `Eq` and `Hash` traits
- Are growable and shrinkable

How to create and initialize HashMaps?

</> code/hashmaps-1/src/main.rs </>

```rust
use std::collections::HashMap;

fn main() {
    let mut pet_owners = HashMap::new();
    pet_owners.insert("Pascal", "Tarja");
    pet_owners.insert("Jarmo", "Nibbit");
    pet_owners.insert("Sill", "Bowser");

    for (k, v) in &pet_owners {
        println!(" {} {}", k, v);
    }

    let res = pet_owners.insert("Jarmo", "Apoe");
    if res.is_some() {
        println!("Replacing {} with Apoe...", res.unwrap());
    }
```



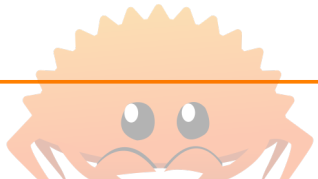
How to access data in HashMaps?

Data in HashMaps in Rust can be accessed;

- Using the m.get() method
- Using an iterator

</> *code/hashmaps-2/src/main.rs* </>

```
1  use std::collections::HashMap;
2
3  fn main() {
4      let mut pet_owners = HashMap::new();
5      pet_owners.insert("Pascal", "Tarja");
6      pet_owners.insert("Jarmo", "Nibbit");
7      pet_owners.insert("Sill", "Bowser");
8      // pet_owners.insert("Kjell", "Tingelfantje");
9
10     let res = pet_owners.get("Sill");
```



Removing data from HashMaps

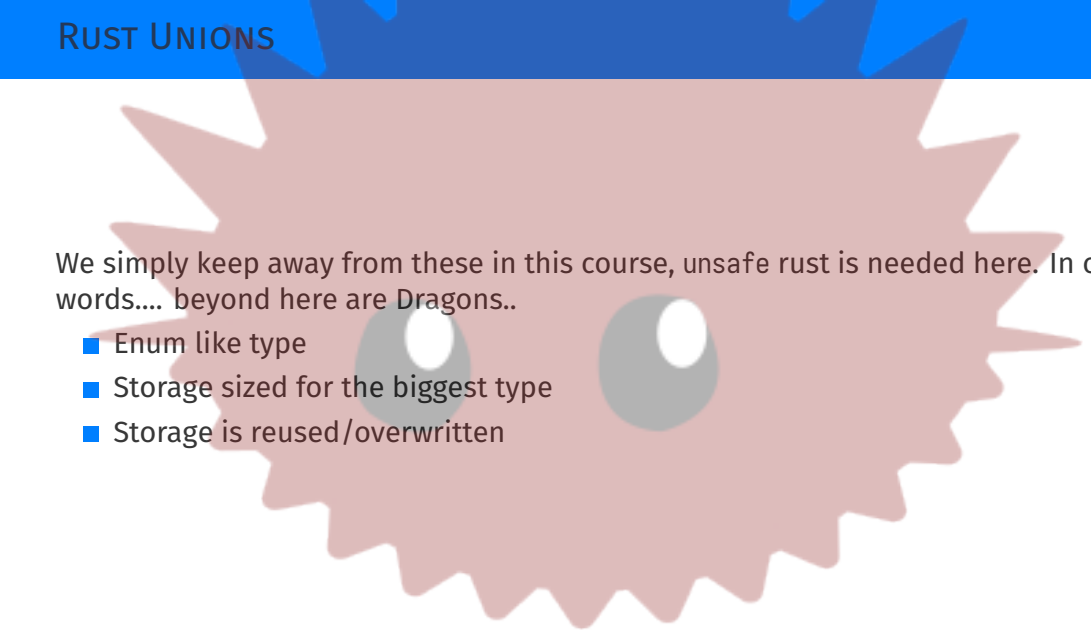
Removing data by key from HashMaps

</> code/hashmaps-3/src/main.rs </>

```rust
use std::collections::HashMap;

fn main() {
    let mut pet_owners = HashMap::new();
    pet_owners.insert("Pascal", "Tarja");
    pet_owners.insert( "Jarmo","Nibbit");

    if let Some(v) = pet_owners.remove("Jarmo") {
        println!("Removed Jarmo from hashmap, who owned {}",v);
    } else {
```

Rust Unions

Rust Unions



We simply keep away from these in this course, unsafe rust is needed here. In other words.... beyond here are Dragons..

- Enum like type
- Storage sized for the biggest type
- Storage is reused/overwritten

Rust Functions

Rust Functions

Functions in Rust

- Can return one or multiple return values
- Pass arguments by value or by reference
- Naming convention is: `snake_case`

Passing by value

Function which passes argument by value

</> *code/functions-1/src/main.rs* </>

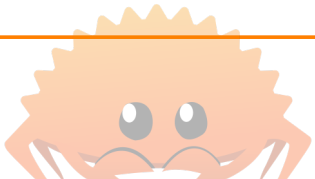
```
1  fn print_square(side: f32) {
2      println!("The square of {} is {}", side, side * side);
3  }
4
5  fn print_str(a: &String) {
6      println!("String is: {}", a);
7  }
8
9  fn main() {
10     let s = 12.0;
```

Passing by value

Function which passes argument by reference

</> *code/functions-2/src/main.rs* </>

```
1  fn scale(side: & mut f32, factor: f32) {
2      *side *= factor;
3  }
4
5  fn main() {
6      let factor = 2.5;
7      let mut side: f32 = 12.0;
8
9      print!("Scaling {} ",side);
10     scale(& mut side,factor);
```



Returning a single value

</> *code/functions-3/src/main.rs* </>

```
1  fn divide(a: f32, b: f32) -> f32 {
2      a/b
3  }
4
5  fn main() {
6      let a: f32 = 8.0 ;
7      let b: f32 = 2.0 ;
8      println!("{} divided by {} gives {}",a,b,divide(a,b));
9
10 }
```

Returning multiple values

code/functions-5/src/main.rs

```rust
// Returns the real solutions for the quadratic equation defined by aX²+bX+c
//
fn solve(a: f64, b: f64, c: f64) -> (f64, f64) {
    let discriminant = b * b - 4.0 * a * c;
    if discriminant < 0.0 {
        println!("Does not have any real solutions");
        return (0.0, 0.0);
    };
    let d = discriminant.sqrt();
    let solution1 = (-b + d) / (2.0 * a);
    let solution2 = (-b - d) / (2.0 * a);
    (solution1, solution2)
}

fn main() {
    let (sol1, sol2) = solve(1.0, 3.0, 2.0);
    println!("sol1 = {}, sol2 = {}", sol1, sol2);
}
```

Rust idomatic Error Handling

In Rust the concept of `null` or `nil` is not part of the language. Also failure is not an Option, it's a Result:

</> *code/functions-4/src/main.rs* </>

```rust
fn divide_deluxe(a: f32, b: f32) -> Result<f32,String> {
    if b == 0.0 {
        Err("!! Division by zero".to_string())
    } else {
        Ok(a/b)
    }
}

fn main() {
    let a: f32 = 8.0;
    let b: f32 = 2.0;

    let res = divide_deluxe(a,b);

    match res {
        Ok(v) =>  { println!("{}",v); }
        Err(msg) => {  println!("Error: {}",msg) ; }
    }
}
```

Rust Methods

Rust Methods

Rust doesn't have classes, in Rust methods are based upon it's `struct` data types.

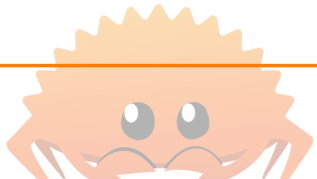
- Methods are functions enclosed in an `impl` code block
- This `impl` codeblock is referencing the `struct` the method(s) belong too
- A `self` parameter refers to the `struct` the methods belong too.
- Methods that consult the struct
- Methods that change the struct
- Static methods
- Needed for traits later
- Naming convention: `snake_case`

Methods that consult the Struct

Method that only reads the underlying `struct`

</> *code/methods-1/src/main.rs* </>

```
1  use std::f64::consts::PI;
2  struct Circle { radius: f64 }
3
4  impl Circle {
5      fn perimeter(&self) -> f64 {
6          self.radius*2.0*PI
7      }
8  }
9
10 fn main() {
11     let c = Circle{radius: 2.0};
12     println!("Perimeter of circle with radius {} is {}",c.radius,c.perimeter());
13 }
```

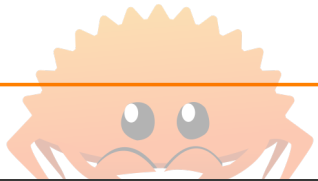


Methods that mutate the struct

Method that changes the underlying struct

</> *code/methods-2/src/main.rs* </>

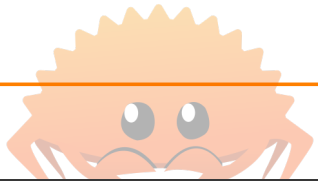
```
1  use std::f64::consts::PI;
2  struct Circle { radius: f64 }
3
4  impl Circle {
5      fn perimeter(&self) -> f64 {
6          self.radius*2.0*PI
7      }
8      fn scale(&mut self, factor: f64) { self.radius *= factor; }
9  }
10
11 fn main() {
12     let mut c = Circle{radius: 2.0};
13     c.scale(2.0);
14     println!("Perimeter of circle with radius {} is {}",c.radius,c.perimeter());
15 }
```



Static method

code/methods-3/src/main.rs

```rust
use std::f64::consts::PI;
struct Circle {
    radius: f64,
}

impl Circle {
    fn new(r: f64) -> Circle {
        Circle { radius: r }
    }
    fn perimeter(&self) -> f64 {
        self.radius * 2.0 * PI
    }
}

fn main() {
```



Rust Traits

Rust Traits

Rust does not do class inheritance, but uses composition instead.

- Inheritance is about 'is'; the horse is an animal.
- Composition is about 'has'; the horse has 4 legs.
- Composition is implemented using nested `structs`
- Generics/polymorphism is implemented using `traits`

Implementing Traits

A sphere is a mathematical body. It is defined by a radius and has a volume and an area. A Cube is also a mathematical body. Traits define the common interface between different type of objects. So let's define the methods that are required for a Body. This bill of requirements (the interface) is called a `trait` in Rust.

</> *code/traits-1/src/main.rs* </>

```
8
9   struct Pyramid {
10      side: f64,
11      height: f64,
12  }
```

Implementing Traits

Now we have defined the interface, it's now time to write the individual methods on the structs. These methods 'implement' the methods as required by the Trait 'Body'.

</> *code/traits-1/src/main.rs* </>

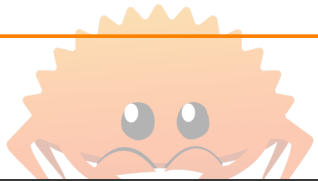
```
13
14  trait Solid {
15      fn volume(&self) -> f64;
16  }
17
18  impl Pyramid {
19      fn volume(&self) -> f64 {
20          ((self.side * self.side) * self.height) / 4.0
21      }
22  }
23
24  impl Sphere {
```

Implementing Traits

The goal is now to write a function that is able to handle both Sphere and Cube projects and in the future probably other objects that satisfy the 'Body' `trait`. We are using Generics for this:

</> *code/traits-1/src/main.rs* </>

```
25      fn volume(&self) -> f64 {
26          (self.radius * self.radius * self.radius) * PI * (4.0 / 3.0)
27      }
28  }
29  impl Cube {
30      fn volume(&self) -> f64 {
```

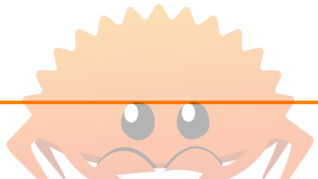


Implementing Traits

Last part is calling the generic function on structs/objects that satisfy the Body interface:

</> *code/traits-1/src/main.rs* </>

```
31              self.side * self.side * self.side
32          }
33      }
34
35      // Generic function for Structs implementing the Body trait
36
37      fn volume(Solid: B) -> f64 {
38          B.volume()
39      }
40
41      fn main() {
42          let s = Sphere { radius: 10.0 };
43          println!(
44              "Volume of Sphere with radius {} is {}",
45              s.radius,
```



Rust and OOP

The four major principles of OOP

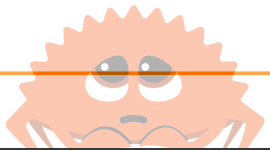
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Enapsulation in Rust

- Use `structs` to create custom data types *classes*
- Use `pub` keyword to make fields `public`
- Methods can be associated with struct using `impl`

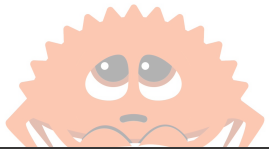
</> *code/oop-1/src/main.rs* </>

```rust
struct Person {
    pub name: String,
    age: u8,
}

impl Person {
    pub fn new(name: String, age: u8) -> Self {
        Person { name, age }
    }

    pub fn say_hello(&self) {
        println!(
```



Inheritance in Rust

- Rust doesn't support inheritance like in traditional OOP languages
- Rust favours composition over inheritance
- In Rust you can use embedded structs for composition
- Trait based generics enable polymorphism and code re-use
- Rust supports trait inheritance

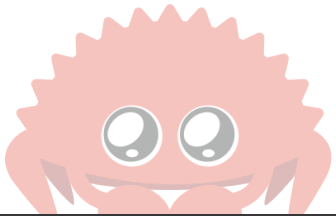


Polymorphism in Rust

- Polymorphism is achieved by using Rust's trait system
- Define a trait
- Implement that trait for various types

Abtraction

- Using `structs`, `enums` and traits only to expose relevant data
- Using `pub` and the module system to hide implementation details



Default implemtation of a trait

- Implementation on `trait` level
- Can be overridden by specific types

Operator overloading

- Operator overloading use specific traits like
  - Add
  - Sub
  - Mul
  - Div
- Covered by: `std::ops`

code/oop-5/src/main.rs

```rust
// Overloading the * operator for scalar multiplication
impl<'a> Mul<f64> for &'a Matrix {
    type Output = Matrix;

    fn mul(self, scalar: f64) -> Matrix {
        let result = self
            .data
            .iter()
            .map(|row| row.iter().map(|val| val * scalar).collect::<Vec<f64>>())
            .collect::<Vec<Vec<f64>>>();

        Matrix::new(result)
    }
}
```

Rust Lifetimes

Rust Lifetimes

Rust lifetimes are a need extension to Ownership and Borrowing. They help the Rust compiler to ensure memory safety. We need `lifetime annotations` when the compiler can't decide on it's own the lifetime of variables and would risk access to invalidated data

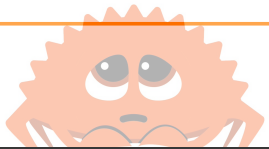
Case 1 - The Rust compiler can decide

In the following case the Rust compiler has all the needed info and can derive the lifetimes of the variables from the program code itself. No annotations needed.

</> *code/lifetimes-1/src/main.rs* </>

```
1  struct Circle { radius: f64 }
2
3  fn main() {
4    let c1: Circle;
5    {
6        let c2 = Circle{ radius: 5.0};
7    }
8
9    // c2 dropped out of scope, so won't compile
10
11   c1 = &c2;
12 }
```

Does not compile.

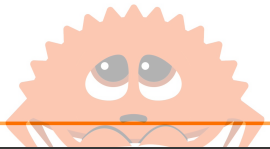


Case 2 - Compiler cannot make a decission

In the following case the Rust compiler can't decide on the lifetimes and needs hints from us. Hence `lifetime` annotations will be needed. This Rust program will refuse to compile

</> *code/lifetimes-2/src/main.rs* </>

```
1  struct Circle { radius: f64, color: String }
2
3  fn largest_circle(c1: &Circle, c2: &Circle)->&Circle{
4    if c1.radius > c2.radius {
5       c1
6    }
7    else {
8       c2
9    }
10 }
11
12 fn main(){
13   let c1 =  Circle{ radius: 2.0, color: "yellow".to_string() };
14   let c2 = Circle { radius: 1.0, color: "red".to_string() };
15   let c3 = largest_circle(&c1, &c2);
16   println!("Largest is the {} circle",c3.color);
17 }
```



Case 2 - What is the issue?

</> code/lifetimes-3/src/main.rs </>

```
3  fn largest_circle<'a>(c1: &'a Circle, c2: &'a Circle)->&'a Circle{
4    if c1.radius > c2.radius {
5       c1
6    }
7    else {
8       c2
9    }
10 }
```

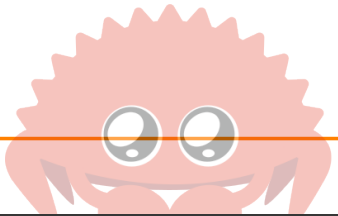
- The function has 2 reference arguments and returns one of them
- It is unknown at compile time which one will be returned
- Normally both references get individual lifetimes assigned
- Here, they need to have the same lifetime assigned
- And so needs the return reference value

Case 2 - The solution

We need to assign the same lifetime for both arguments AND the return value:

code/lifetimes-3/src/main.rs

```
1  struct Circle { radius: f64, color: String }
2
3  fn largest_circle<'a>(c1: &'a Circle, c2: &'a Circle)->&'a Circle{
4    if c1.radius > c2.radius {
5      c1
6    }
7    else {
8      c2
9    }
10 }
11
12 fn main(){
13   let c1 =  Circle{ radius: 2.0, color: "yellow".to_string() };
14   let c2 = Circle { radius: 1.0, color: "red".to_string() };
15   let c3 = largest_circle(&c1, &c2);
16   println!("Largest is the {} circle",c3.color);
17 }
```



Important notes about lifetime annotations

Lifetime annotations:

- do not change the lifetime of references!
- specify the relation of lifetimes between references
- are never needed for owned variables
- naming convention is `'` followed by a single lowercase character `<'a>`
- can also be needed in `structs` containing references

Lifetime elision

Lifetime elision is the process in which the Rust compilers tries to determine the lifetimes of variables automatically.

- If `lifetime elision` is possible, no lifetime annotations are required
- If `lifetime elision` is not possible, we need to annotate these variables
- There are three cases/rules where `lifetime elision` is possible

Lifetime elision rules

- Each elided lifetime in input position becomes a distinct lifetime parameter
- If there is exactly one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes
- If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of self is assigned to all elided output lifetimes.
- Otherwise, it is an error to elide an output lifetime

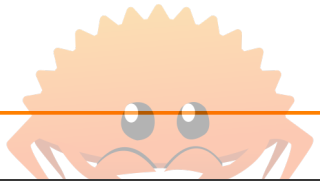
In the latter case we need to help the Rust compile by providing lifetime annotations.

Lifetime elision rules - example 1

Each elided lifetime in input position becomes a distinct lifetime parameter

</> *code/lifetimes-6/src/main.rs* </>

```
1  // Each elided lifetime in input position becomes a distinct lifetime parameter.
2  struct Circle { radius: f64, color: String }
3
4  fn largest_circle<'a, 'b>(c1: &'a Circle, c2: &'b Circle)->& Circle{
5    if c1.radius > c2.radius {
6       c1
7    }
8    else {
9       c2
10   }
11 }
12
13 fn main(){
14   let c1 =  Circle{ radius: 2.0, color: "yellow".to_string() };
15   let c2 = Circle { radius: 1.0, color: "red".to_string() };
16   let c3 = largest_circle(&c1, &c2);
17   println!("Largest is the {} circle",c3.color);
18 }
```



Lifetime elision rules - example 2

If there is exactly one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes

</> code/lifetimes-5/src/main.rs </>

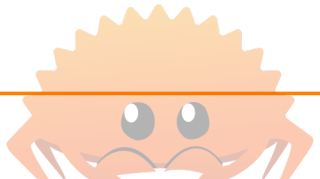
```
1  // If there is exactly one input lifetime position (elided or not),
2  // that lifetime is assigned to all elided output lifetimes.
3  fn last_word (s: &str) -> &str {
4      let b = s.as_bytes();
5      for (n,&c) in b.iter().rev().enumerate() {
6          if c == b' ' {
7              return &s[s.len()-n..];
8          }
9      }
10     s
11 }
12
13 fn main() {
14     let my_sentence = "Famous last words";
15     println!("Last word in '{}' is '{}'",my_sentence,last_word(my_sentence));
16 }
```

Lifetime elision rules - example 3

If there is exactly one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes

</> *code/lifetimes-4/src/main.rs* </>

```
1  fn last_item (s: &str, ch: char) -> &str {
2      let b = s.as_bytes();
3      for (n,&c) in b.iter().rev().enumerate() {
4          if c as char == ch {
5              return &s[s.len()-n..];
6          }
7      }
8      s
9  }
10
11 fn main() {
12     let my_sentence = "apples,oranges,bananas and ananas";
13     println!("Last item in '{}' is '{}'",my_sentence,last_item(my_sentence,','));
14 }
```

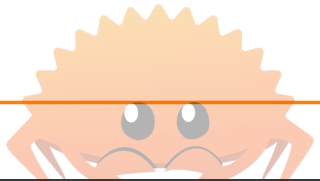


Lifetime elision rules - example 4

If there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of self is assigned to all elided output lifetimes.

</> *code/lifetimes-7/src/main.rs* </>

```
1  // If there are multiple input lifetime positions, but one of them is &self or &mut self
2  // the lifetime of self is assigned to all elided output lifetimes.
3  struct Circle { radius: f64, color: String }
4
5  impl Circle {
6      fn colorize (&self, color: &str) -> &str {
7          &self.color
8      }
9
10 }
11
12 fn main(){
13   let c1 =  Circle{ radius: 2.0, color: "blank".to_string() };
14   println!("{}",c1.colorize("red"));
15 }
```



Lifetime elision rules - example 5

If the compiler can't make a decission, we need to provide lifetime annotations

</> *code/lifetimes-3/src/main.rs* </>

```
1  struct Circle { radius: f64, color: String }
2
3  fn largest_circle<'a>(c1: &'a Circle, c2: &'a Circle)->&'a Circle{
4    if c1.radius > c2.radius {
5      c1
6    }
7    else {
8      c2
9    }
10 }
11
12 fn main(){
13   let c1 =  Circle{ radius: 2.0, color: "yellow".to_string() };
14   let c2 = Circle { radius: 1.0, color: "red".to_string() };
15   let c3 = largest_circle(&c1, &c2);
16   println!("Largest is the {} circle",c3.color);
17 }
```

Rust Error Handling

Rust error handling

Error types:

- Non-recoverable errors
- Recoverable errors

Non-recoverable errors

Errors that cannot be recovered from:

- Out-of-bounds access
- Integer division by zero
- Assertion failure
- Calling `.expect` or |.unwrap|on an `err`

Please note that `panics` happen on a per `thread` basis.

code/error-handling-1/src/main.rs

```
1  use std::fs;
2
3  fn main() {
4    let content = fs::read_to_string("./Cargo.toml").unwrap();
5    println!("Contents:\n{}", content);
6
7    let _content2 = fs::read_to_string("./Cargo.tonl").expect("Can't read Cargo.tonl");
8    println!("{}", content);
9  }
```

The panic! macro

The `panic!` macro is there when you want to `panic` out of a situation in your program:

code/error-handling-2/src/main.rs

```
1  fn main() {
2      let world_needs_reboot: bool = true;
3
4      if world_needs_reboot {
5          panic!("Unable to comply, world needs rebooting...");
6      }
7  }
```

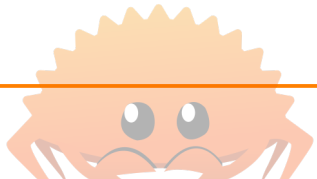
Error is not an option, it's a Result

`Result` Enums are the Rust way to communicate on errors

- Compare them with the `Option` enum
- Result Enum contains:
  - OK(v) if all went ok
  - Err(e) in case of unwanted error

</> *code/error-handling-7/src/main.rs* </>

```rust
fn halves_if_even(i: i32) -> Result<i32, String> {
    if i % 2 == 0 {
        Ok(i / 2)
    } else {
        Err("That's odd".to_string())
    }
}
```

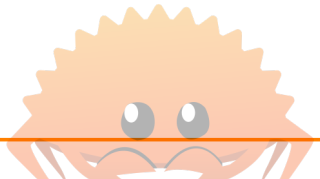


Testing for the kind of error

In Rust `ErrorKind` can be tested for the kind of error

code/error-handling-8/src/main.rs

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let myfile = "tst.txt";
    let f = File::open(myfile);

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create(myfile) {
                Ok(fc) => fc,
                Err (e) => panic!("Unable to create file: {:?}",e),
            },
            other_error => {
                panic!("Unable to open file: {:?}",other_error)
            }
        }
    };
}
```

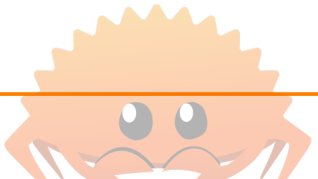


Mixing in closures with unwrap_or_else

The `unwrap_or_else()` method invokes a closure to handle the error.

code/error-handling-9/src/main.rs

```
1  use std::fs::File;
2  use std::io::ErrorKind;
3
4  fn main() {
5
6      let myfile = "tst.txt";
7      let f = File::open(myfile).unwrap_or_else(|error| {
8          if error.kind() == ErrorKind::NotFound {
9              File::create(myfile).unwrap_or_else(|error| {
10                 panic!("Issue creating the file {}: {}",myfile,error);
11             })
12         } else {
13             panic!("Problem opening the file {}: {}",myfile,error);
14         }
15     });
16 }
```



Matching Results

One can use `match` to test for the results

- Ok(v)
- Err(e)

</> *code/error-handling-10/src/main.rs* </>

```
1  use std::fs::File;
2
3  fn main() {
4      let myfile = "myfile2.txt";
5
6      let f = File::open(myfile);
7
8      let f = match f {
9          Ok(file) => file,
10         Err(e) => panic!("Issue opening file {}: {}",myfile,e),
11     };
12 }
```

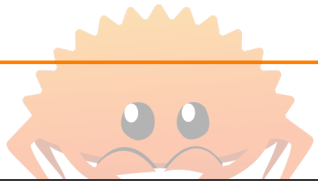
Unwrap or Panic

The unwrap method simply:

- Returns the Result value wrapped in Ok if no error occured
- Panics if the Result was an error

</> code/error-handling-11/src/main.rs </>

```
1  use std::fs::File;
2
3  fn main() {
4      let myfile = "myfile2.txt";
5
6      let f = File::open(myfile).unwrap();
7
8  }
```

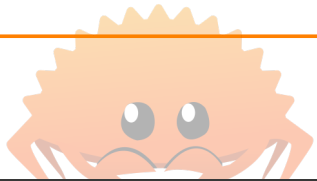


Unwrap or Panic with custom message

The expect method provides an extra option to supply an accompanying message to explain the error from the program's perspective:

</> *code/error-handling-12/src/main.rs* </>

```
use std::fs::File;

fn main() {
    let myfile = "myfile2.txt";

    let f = File::open(myfile).expect("Unable to open file");

}
```

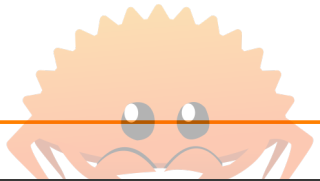


Using let match expressions

Example using multiple `let match` expressions to handle the error(s)

code/error-handling-13/src/main.rs

```rust
use std::fs::File;
use std::io;
use std::io::Read;

fn read_name_from_file() -> Result<String, io::Error> {
    let f = File::open("myfile3.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```



Test for errors using is_error method

code/error-handling-13/src/main.rs

```
23
24      if s.is_err() {
25          panic!("We ran into an error reading name from file");
26      } else {
27          println!("We read the following name: {}",s.unwrap());
28      }
29  }
```

Simplyfying with the ? operator

The ? operator:

- Returns directly with Err(e) if it's an error case
- Unwraps and continues if there's no error
- Idiomatic Rust for error propagation

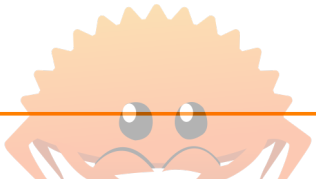
</> *code/error-handling-14/src/main.rs* </>

```rust
use std::fs::File;
use std::io;
use std::io::Read;

fn read_name_from_file() -> Result<String, io::Error> {
    let mut f = File::open("myfile3.txt")?;
    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

More simplification

code/error-handling-15/src/main.rs

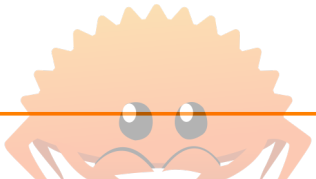
```rust
use std::fs::File;
use std::io;
use std::io::Read;

fn read_name_from_file() -> Result<String, io::Error> {
    let mut f = File::open("myfile3.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}

fn main() {
    let s = read_name_from_file();

    if s.is_err() {
        panic!("We ran into an error reading name from file");
    } else {
        println!("We read the following name: {}",s.unwrap());
    }
}
```



More simplification

code/error-handling-15/src/main.rs

```
1  use std::fs::File;
2  use std::io;
3  use std::io::Read;
4
5  fn read_name_from_file() -> Result<String, io::Error> {
6      let mut f = File::open("myfile3.txt")?;
7      let mut s = String::new();
8      f.read_to_string(&mut s)?;
9      Ok(s)
10 }
11
12 fn main() {
13     let s = read_name_from_file();
14
15     if s.is_err() {
16         panic!("We ran into an error reading name from file");
17     } else {
18         println!("We read the following name: {}",s.unwrap());
19     }
20 }
```

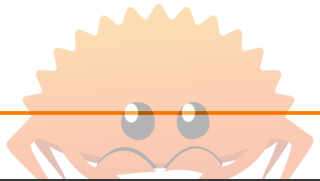


Ultimate simplification

Chaining can provide even more simplification:

</> *code/error-handling-16/src/main.rs* </>

```
1  use std::fs::File;
2  use std::io;
3  use std::io::Read;
4
5  fn read_name_from_file() -> Result<String, io::Error> {
6      let mut s = String::new();
7      File::open("myfile3.txt")?.read_to_string(& mut s)?;
8      Ok(s)
9  }
10
11 fn main() {
12     let s = read_name_from_file();
13     if s.is_err() {
14         panic!("We ran into an error reading name from file");
15     } else {
16         println!("We read the following name: {}",s.unwrap());
17     }
18 }
```



Rust Closures

Closures in Rust

Closures in Rust are:

- closes over it's environment
- anonymous function
- Used a lot in other functions, iterators and concurrency

Closures

Closures:

- If the body consists of a single expression, no {} needed
- Return types are not mandatory
- Data types are not mandatory
- Can capture the environment. Each used external variable is borrowed

The need for closures

Trying to sort an array. Using ascending order works out-of-the-box:

</> *code/closures-2/src/main.rs* </>

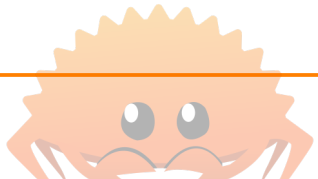
```
1  fn main() {
2      let mut a = [21, 4, 28, 45, 99, 5, 9];
3
4      println!("{:?}", a);
5      a.sort();
6      println!("{:?}", a);
7  }
```

Sorting in a descending order - solution 1

We can use the alternative sort_by() method that uses a function:

</> code/closures-3/src/main.rs </>

```
1  use std::cmp::Ordering;
2
3  fn descending(a: &i32, b: &i32) -> Ordering {
4      if a < b { Ordering::Greater }
5      else { Ordering::Less }
6  }
7
8  fn main() {
9      let mut a = [21, 4, 28, 45, 99, 5, 9];
10
11     println!("{:?}", a);
12     a.sort_by(descending);
13     println!("{:?}", a);
14 }
```

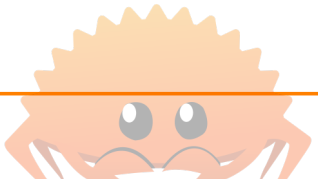


Sorting using a closure as order function

Using closures is more elegant:

code/closures-5/src/main.rs

```
use std::cmp::Ordering;

fn main() {
    let mut a = [21, 4, 28, 45, 99, 5, 9];

    println!("{:?}", a);
    a.sort_by(|a, b| {
        if a < b {
            Ordering::Greater
        } else if a > b {
            Ordering::Less
        } else {
            Ordering::Equal
        }
    });
    println!("{:?}", a);
```



Sorting using a std ordering function in a closure

Even more elegant:

</> *code/closures-6/src/main.rs* </>

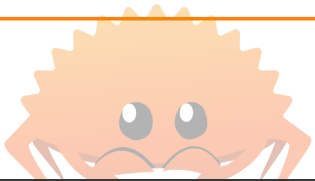
```
1  fn main() {
2      let mut a = [21, 4, 28, 45, 99, 5, 9];
3
4      println!("{:?}", a);
5      a.sort_by(|a, b| b.cmp(a));
6      println!("{:?}", a);
7  }
```

Functions cannot capture the environment

Functions cannot use `let` variables of the outside blocks:

code/closures-4/src/main.rs

```
fn main() {
    let five: f64 = 5.0;
    fn print_f64(x: f64) {
        print!("{}", x * five);
    }

    print_f64(3.1);
}
```

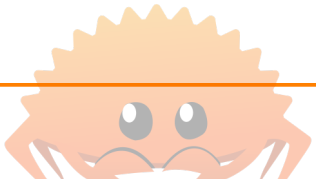


Several ways to invoke closures

There are several ways closures can be invoked:

</> code/closures-7/src/main.rs </>

```
1  fn main() {
2      let factor = 2;
3      let mul = |a| a * factor;
4      print!("{}", mul(4));
5      let mul_ref = &mul;
6
7      println!(
8          " {} {} {} {} {}",
9          (*mul_ref)(4),
10         mul_ref(4),
11         (|a| a * factor)(4),
12         (|a: i32| a * factor)(4),
13         |a| -> i32 { a * factor }(4)
14     );
15 }
```

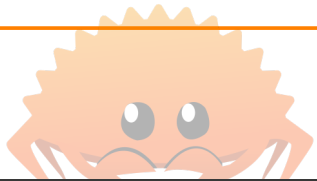


Type inference

Type inference on closures only happens once:

</> *code/closures-9/src/main.rs* </>

```
1  fn main() {
2      let closure = |num| num;
3
4      let var_1 = closure(5);
5      println!("{}", var_1);
6
7      let var_2 = closure(2.5);
8      println!("{}", var_2);
9  }
```



Moving closures

Moving closures take ownership of used ext variables

</> *code/closures-8/src/main.rs* </>

```
1  struct Circle {
2      radius: f64,
3      txt: String,
4  }
5
6  fn main() {
7      let circle = Circle {
8          radius: 1.0,
9          txt: "test".to_string(),
10     };
11     // Closure:
12     let closure = || {
13         println!("radius: {:?}", circle.radius);
14     };
15     closure();
16     println!("radius: {:?}", circle.radius);
17
18     let circle = Circle {
19         radius: 1.0,
20         txt: "test".to_string(),
21     };
```

Rust Generics

The problem we want to solve:

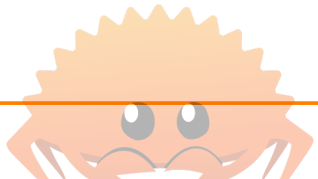
code/generics-3/src/main.rs

```rust
fn get_smallest(numbers: Vec<i32>) -> i32 {

    let mut smallest = numbers[0];
    for n in numbers {
        if n < smallest {
            smallest = n;
        }
    }
    smallest
}
fn get_smallest_f64(numbers: Vec<f64>) -> f64 {

    let mut smallest = numbers[0];
    for n in numbers {
        if n < smallest {
            smallest = n;
        }
    }
    smallest
}

fn main() {
    let a = vec![4,52,1,2,3,8,-8,22,42,81];
```

With generics:

</> code/generics-4/src/main.rs </>

```rust
fn get_smallest<T>(numbers: Vec<T>) -> T {

    let mut smallest = numbers[0];
    for n in numbers {
        if n < smallest {
            smallest = n;
        }
    }
    smallest
}

fn main() {
    let a = vec![4,52,1,2,3,8,-8,22,42,81];
    let b = vec![3.1,-2.3,2.8,37.3,21.1];
    println!("{}",get_smallest(a));
    println!("{}",get_smallest(b));
}
```

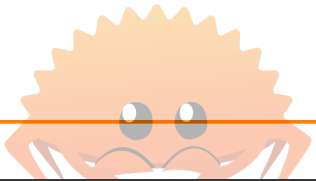


Trait Bounds

We try to cover to many types, we need to set bounds

</> *code/generics-5/src/main.rs* </>

```rust
fn get_smallest<T>(numbers: Vec<T>) -> T
where
    T: PartialOrd + Copy,
{
    let mut smallest = numbers[0];
    for n in numbers {
        if n < smallest {
            smallest = n;
        }
    }
    smallest
}

fn main() {
    let a = vec![4, 52, 1, 2, 3, 8, -8, 22, 42, 81];
    let b = vec![3.1, -2.3, 2.8, 37.3, 21.1];
    println!("{}", get_smallest(a));
    println!("{}", get_smallest(b));
}
```



Generic Structs

Generics can also be applied to Structs

</> *code/generics-6/src/main.rs* </>

```
1  struct Pixel<T,U>{ x: T, y: U}
2
3  fn main() {
4      let p1 = Pixel{x: 1,y: 2};
5      let p2 = Pixel{x: 0.5,y: 10.5};
6      let p3 = Pixel{x: 0.5,y: 10};
7  }
```

Generic Enums

And enums:

</> *code/generics-7/testing/test1.rs* </>

```
1  struct Pixel<T,U>{ x: T, y: U}
2
3  fn main() {
4      let p1 = Pixel{x: 1,y: 2};
5      let p2 = Pixel{x: 0.5,y: 10.5};
6      let p3 = Pixel{x: 0.5,y: 10};
7  }
```

Combination of Generic and specific impl

</> *code/generics-8/testing/test1.rs* </>

```
1  struct Pixel<T,U>{ x: T, y: U}
2
3  fn main() {
4      let p1 = Pixel{x: 1,y: 2};
5      let p2 = Pixel{x: 0.5,y: 10.5};
6      let p3 = Pixel{x: 0.5,y: 10};
7  }
```

Rust Dynamic Dispatch or Trait Objects

The problem we want to solve:

</> *code/dynamic-1/src/main.rs* </>

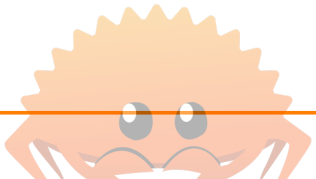
```
19  fn main() {
20      let mut animals: Vec<Box<dyn Speak>> = Vec::new();
21
22      animals.push(Box::new(Dog));
23      animals.push(Box::new(Cat));
24
25      for animal in animals.iter() {
26          animal.speak();
27      }
28  }
```

Rust Trait Objects

With Rust Trait Objects / Dynamic Dispatch:

</> *code/dynamic-1/src/main.rs* </>

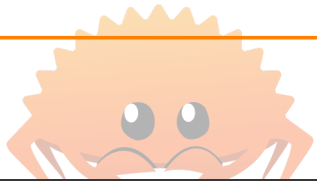
```
1  trait Speak {
2      fn speak(&self);
3  }
4
5  struct Dog;
6  impl Speak for Dog {
7      fn speak(&self) {
8          println!("Woof!");
9      }
10 }
11
12 struct Cat;
13 impl Speak for Cat {
14     fn speak(&self) {
15         println!("Meow!");
16     }
17 }
18
```



With Rust Trait Objects / Dynamic Dispatch:

</> *code/dynamic-1/src/main.rs* </>

```
19  fn main() {
20      let mut animals: Vec<Box<dyn Speak>> = Vec::new();
21
22      animals.push(Box::new(Dog));
23      animals.push(Box::new(Cat));
24
25      for animal in animals.iter() {
26          animal.speak();
27      }
28  }
```



Trait Objects

- Performance Implications - runtime
- Object Safety
- Lifetime Specifiers
- Downcasting - reflection

Alternatives to Trait Objects

- Enums
- Generics and Static Dispatch

Rust Iterators

Rust iterators

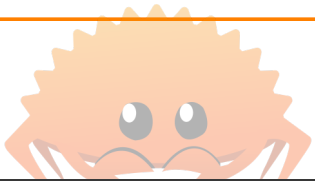
Iterator types:

- Adapters -> iterator in -> iterator out
- Consumers -> iterator in -> other type out

Simple iterator

code/iterators-1/src/main.rs

```
fn main() {
    let v1 = vec![1,2,3,5,7,11,13,17,19];

    let v1_iter = v1.iter();


    for i in v1_iter {
        println!("{}",i);
    }
}
```



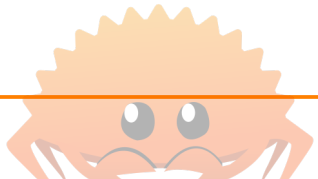
Mutable iterator

Use iter_mut() for a mutable iterator

- Mind the `mut` on the vector

</> *code/iterators-2/src/main.rs* </>

```
1  fn main() {
2      let mut v1 = vec![1,2,3,5,7,11,13,17,19];
3
4      let v1_iter = v1.iter_mut();
5
6      for i in v1_iter {
7          *i +=1 ;
8      }
9
10     let v1_iter2 = v1.iter();
11     for i in v1_iter2 {
12         println!("{}",i);
13     }
14 }
```



Iterator consumer: sum

</> code/iterators-3/src/main.rs </>

```
1  fn main() {
2      let v1 = vec![1,2,3,5,7,11,13,17,19];
3      let v1_iter = v1.iter();
4
5      let sum: i64 = v1.iter().sum();
6
7      println!("Sum is: {}",sum);
8  }
```

Iterator adapter map

The iterator adapter map applies changes to the set

code/iterators-4/src/main.rs

```
fn main() {
    let v1 = vec![1,2,3,5,7,11,13,17,19];
    let v1_iter = v1.iter();

    v1.iter().map(|x| x*2);
    for n in v1.iter() {
        println!("{}",n);
    }
}
```

Iterator adapter map

The iterator adapter map applies changes to the set

</> *code/iterators-5/src/main.rs* </>

```
1  use std::any::type_name;
2
3  fn print_type_of<T>(_: &T) {
4      println!("{}", std::any::type_name::<T>())
5  }
6
7  fn main() {
8      let v1 = vec![1,2,3,5,7,11,13,17,19];
9
10     let v2: Vec<_> = v1.iter().map(|x| x*2).collect();
```

Iterator adapter filter

The iterator adapter filter selects elements from the set

- Mind the use of the double dereference in the selection part

</> *code/iterators-6/src/main.rs* </>

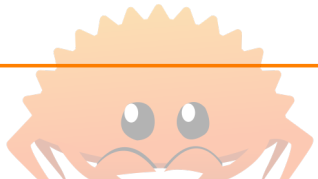
```
1  fn main() {
2      let v1 = vec![1,2,3,-4,5,7,-8,11,-1,13,17,19];
3
4      let mut v2_iter = v1.iter().filter(|x| **x < 0 );
5
6      while let Some(v) = v2_iter.next() {
7          println!("{}",v);
8      }
9  }
```

Iterator adapter zip

The zip adapter combines the elements in two sets into a tuple

</> *code/iterators-7/src/main.rs* </>

```
1  fn main() {
2      let emp_no = vec![1, 2, 3];
3      let age = vec![20, 30, 35];
4
5      let iter_emp = emp_no.iter();
6      let iter_age = age.iter();
7
8      let zip_iter = iter_emp.zip(iter_age);
9
10     for item in zip_iter{
11         println!("({}, {})", item.0,item.1);
12     }
13 }
```

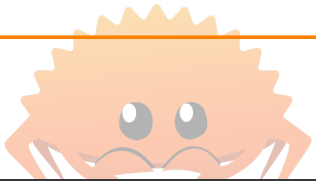


Implementing an iterator on a type

Step 1: define the structs and the constructor methods

</> *code/iterators-8/src/main.rs* </>

```
1  struct Counter {
2      count: u64,
3  }
4
5  impl Counter {
6      fn new() -> Counter {
7          Counter { count: 0 }
8      }
9  }
10
```



Implementing an iterator on a type

Step 2: implement the next() method

</> *code/iterators-8/src/main.rs* </>

```
10
11  impl Iterator for Counter {
12      type Item = u64;
13
14      fn next(&mut self) -> Option<Self::Item> {
15          if self.count < 9 {
16              self.count += 1;
17              Some(self.count)
18          } else {
19              None
20          }
21      }
22  }
23
```

Implementing an iterator on a type

Step 3: Testing/using the iterator in main()

</> *code/iterators-8/src/main.rs* </>

```
24  fn main() {
25      let mut c = Counter::new();
26      while let Some(v) = c.next() {
27          println!("{}", v);
28      }
29  }
```

Rust testing

Rust testing

Rust provides means to do:

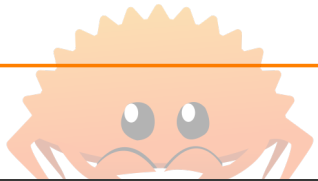
- Unit testing
- Integration testing

Unit testing

Unit testing in Rust:

</> *code/testing-1/src/main.rs* </>

```rust
pub fn mul(a: i32, b: i32) -> i32 {
    a * b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_mul() {
        assert_eq!(mul(5, 2), 10);
    }
}
```



Unit testing

Unit testing is executed with `cargo test`

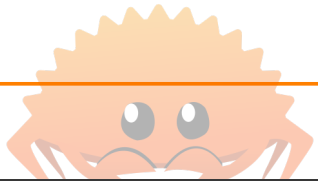
- Run tests in sequence or concurrent (default)
- Run only a subset of tests

Table driven testing

Rust does not have a standard way to do table driven testing, but we can implement it like this:

</> *code/testing-3/src/main.rs* </>

```
11  #[cfg(test)]
12  mod tests {
13      use super::*;
14
15      #[test]
16      fn test_fac() {
17          let tbl: &[(i64, i64)] = &[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24)];
18
19          for (n, expected) in tbl {
20              assert_eq!(fac(*n), *expected);
21          }
22      }
23  }
```

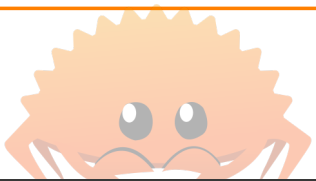


Integration testing

Integration testing is implemented by placing test code in the tests directory in the root of the package:

</> *code/testing-4/tests/integration1.rs* </>

```rust
use testing_4::add;
#[test]
fn test_add() {
    assert_eq!(add(3, 2), 5);
}
```



Rust benchmarking

Rust benchmarking

Rust provides means to benchmark code:

- Rust native (not stable, nightlies)
- Criterion package

Benchmarking with Criterion

Benchmarking with Criterion in rust:

- External crate
- Derived from Haskell criterion tool
- Creates useful reports

Setting up Criterion

To setup Criterion for benchmarking we need to:

- Configure `Cargo.toml`
- Import the Criterion crate
- Create a `benches` directory with Benchmark tests

Cargo.toml for Criterion

We need to configure Cargo.toml for Criterion:

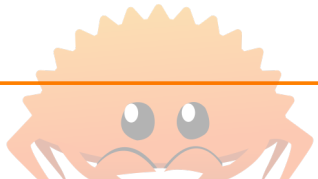
- Add the criterion package
- Configure criterion package (html-reports)
- Disable internal benchmarking (harnass)
- Name our bechmarking set

Configuring Cargo.toml for Criterion

The `Cargo.toml` file for benchmarking with Criterion

</> *code/simple-bench/Cargo.toml* </>

```
[package]
name = "simple-bench"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]

[dev-dependencies]
criterion = { version="0.4.0", features=["html_reports"] }

[[bench]]
name = "simple-bench"
harness = false
```

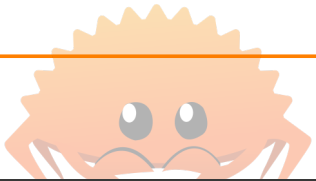


Example lib.rs

The example code to benchmark:

</> *code/simple-bench/src/lib.rs* </>

```rust
pub fn minus_one(number: i32) -> i32 {
    number - 1
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_minus_one() {
        assert_eq!(minus_one(1), 0);
    }
}
```

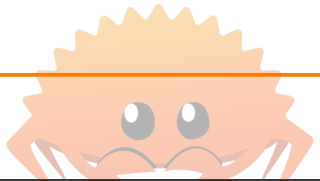


The benchmark code in benches

The Benchmark code is written in rust files in the benches directory in the crate's root:

code/simple-bench/benches/simple-bench.rs

```rust
use simple_bench::minus_one;
use criterion::BenchmarkId;
use criterion::Criterion;
use criterion::{criterion_group, criterion_main};

fn minus_one_benchmark(c: &mut Criterion) {
    let size: usize = 1024;

    c.bench_with_input(BenchmarkId::new("Simple bench", size), &size, |b, &s| {
        b.iter(|| minus_one(1));
    });
}
criterion_group!(benches, minus_one_benchmark);
criterion_main!(benches);
```



Running the benchmark

- Run the benchmark using: `cargo bench`
- Reports are in `target/criterion/report`

Rust Module System

Rust Module System

To make projects/code manageable, Rust provides:

- Crates
- Packages
- Modules
- Paths

Crates

Crates in Rust:

- are the smallest amount of code for Rust
- are a single source code file
- can contain modules
- come in two types
  - Binary crate
  - Library crate
-

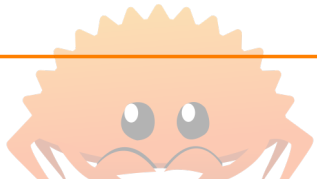
Binary crates

Rust binary crates:

- are programs you can compile
- have a main() function
- are create using `cargo create <crate-name>`

</> *code/crates-1/src/main.rs* </>

```rust
// Example of a binary crate

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



Library crates

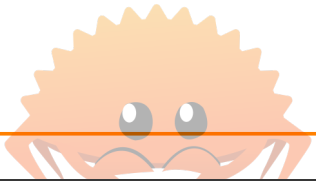
Rust library crates:

- don't have a `main()` function
- do not compile into an executable
- contain functionaly to be shared with multiple projects
- can be created using `cargo new <name> --lib`

When Rustaceans mention crate, they mean library crates.

</> *code/crates-2/src/lib.rs* </>

```rust
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
```



Packages

A Rust package:

- is bundle of one or more crates
- contains a `Cargo.toml` file
- contains 1 or many binary crates
- contains only one library crate

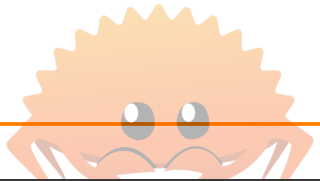
Modules

A rust module

- give your code structure
- control visibility of the items in them (pub/priv)
- contains `items` like
  - Functions, Types, Traits, Impl blocks
  - Macros, Constants, Statics
  - Imports, modules, Ext blocks and crates etc.

</> *code/shaper/src/shapes.rs* </>

```rust
use std::f64::consts::PI;
pub struct Circle { pub radius: f64 }

impl Circle {
    pub fn perimeter(&self) -> f64 {
        self.radius*2.0*PI
    }
}
```



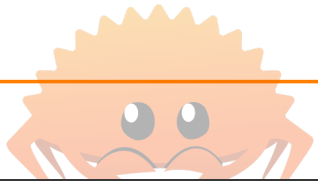
Paths

Paths in rust:

- Absolute path to an item
- Relative path to an item

</> *code/crates-4/src/lib.rs* </>

```
1  mod front_of_house {
2      pub mod hosting {
3          pub fn add_to_waitlist() {}
4      }
5  }
6
7  pub fn eat_at_restaurant() {
8      // Absolute path
9      crate::front_of_house::hosting::add_to_waitlist();
10 }
```



Rust Threads

Rust threads

Error types:

- Std only supports 1:1 model
- Green threads (N:M) available in external crates

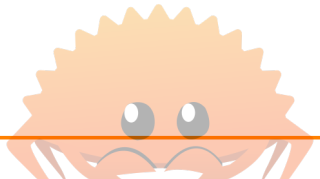
Creating threads

Threads are created using the `thread::spawn()` function

- Functions
- Closures

</> *code/concurrency-1/src/main.rs* </>

```
1  use std::thread;
2  use std::time::Duration;
3
4
5  fn main() {
6      thread::spawn(|| {
7          for n in 1..10 {
8              println!("Spawned thead nr: {}",n);
9              thread::sleep(Duration::from_millis(1));
10         }
11     });
12
13     for n in 1..5 {
14             println!("Main thead nr: {}",n);
15             thread::sleep(Duration::from_millis(1));
16     }
```

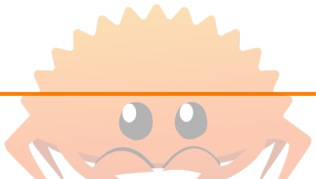


Syncing thread execution

To make sure all threads are executed before the main thread exits use `join`

</> *code/concurrency-2/src/main.rs* </>

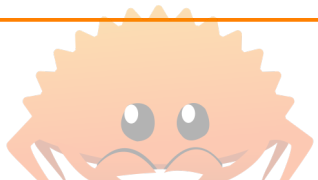
```
1  use std::thread;
2  use std::time::Duration;
3
4
5  fn main() {
6      let joinhandle = thread::spawn(|| {
7          for n in 1..10 {
8              println!("Spawned thead nr: {}",n);
9              thread::sleep(Duration::from_millis(1));
10         }
11     });
12
13     for n in 1..5 {
14             println!("Main thead nr: {}",n);
15             thread::sleep(Duration::from_millis(1));
16     }
```



Threads and memory safety challenges 1

</> code/concurrency-3/src/main.rs </>

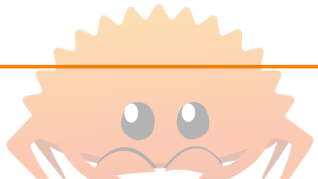
```
1  use std::thread;
2  use std::time::Duration;
3
4
5  fn main() {
6
7      let v1 = vec![1,2,3,4];
8      // let joinhandle = thread::spawn(move || {
9      let joinhandle = thread::spawn(move || {
10         for n in 1..10 {
```



Sharing memory by communicating - channels

code/concurrency-4/src/main.rs

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn( move || {
        let m = String::from("Message in a bottle");
        tx.send(m).unwrap();
    });

    let recv = rx.recv().unwrap();
    // let recv = rx.try_recv().unwrap();
    println!("Fetching bottle with content: {}",recv);
}
```



Sharing memory by communicating - channels

code/concurrency-5/src/main.rs

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn( move || {
        let msgs = vec!["I", "sent","an","SOS","to","the","world"];

        for m in msgs {
            tx.send(m).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for recv in rx {
        println!("Fetching bottle with content: {}",recv);
    }
}
```

Rust Async

Rust Async/Await

Async/Await in Rust provides Cooperative multitasking

- Async like sync programming
- Similar to async/await & promises in JavaScript
- Difference: futures are lazy in Rust

Async components

- Async
- Await
- Futures (zero-cost)

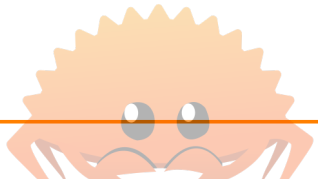
Async

Async transforms a Rust codeblock into a kind of statemachine that implement the Future trait:

- Async Fn
- Async Block
- Both return the value implementing a Future trait

</> *code/async-3/src/main.rs* </>

```rust
async fn add(a: u32, b: u32) -> u32 {
    a + b
}

fn main() {
    let result: impl Future<Output: u32> = add(2, 3);
    async {
        // Code
    };
}
```



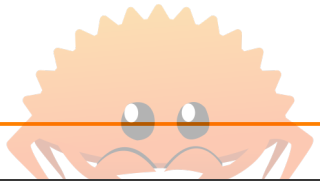
Await

`Await` is the mechanism to run a `Future`.

- Asynchrononously (a)waits for the future to complete
- Not ready -> Yields the current thread
- Can only be used in `async fn` or `async block`

</> *code/async-4/src/main.rs* </>

```rust
async fn add(a: u32, b: u32) -> u32 {
    a + b
}

fn main() {
    let result: impl Furute<Output=u32> = add(2, 3);
    async {

        // Waiting for the Future to complete (async)
        let data: u32 = result.await;
        // Rest of the code
    };
}
```



Futures

Futures in Rust

- Zero cost (Polling)
- Created by invoking `async fn`
- Invoking does not schedule the `fn` -> lazy
- An `await` on the `future`
  - `Executor` takes the `future`
  - Drive/run's it to completion
  - Calling `poll|` when progress can be made

Bare demo of Async

code/async-5/src/main.rs

```
1  use futures::executor::block_on;
2  use async_std::task;
3  use std::time::Duration;
4
5
6  async fn func_1() {
7      for i in 1..10 {
8          print!("f1 ");
9          if i == 5 {
10             task::sleep(Duration::from_secs(2)).await;
11         }
12     }
13 }
14
15 async fn func_2() {
16     for i in 1..10 {
```

With tokio runtime

</> code/async-6/src/main.rs </>

```rust
use std::{thread, time};
use tokio::time::{sleep, Duration};
use rand::prelude::*;
#[tokio::main]

async fn main() {
  my_afunc().await;
}

async fn my_afunc() {
    println!("Async func");
    let s1 = read_from_db().await;
    println!("{}",s1);
    let s2 = read_from_db().await;
    println!("{}",s2);
}

async fn read_from_db() -> String {
    let mut rng = rand::thread_rng();
    let delay = rng.gen_range(0..5);
    println!("Sleeping for {}s",delay);
    sleep(Duration::from_millis(delay*1000)).await;
    "DB data".to_owned()
}
```

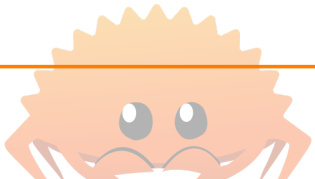
Rust async Runtimes

- Rust does not provide execution context
- No runtime, no async
- Popular choices:
  - `tokio`
  - `async-std`
  - `smol`

Spawning async tasks with Tokio

code/async-7/src/main.rs

```
5  async fn main() {
6
7      let mut handles = vec![];
8
9      for i in 0..2 {
10         let handle = tokio::spawn(async move {
11             my_afunc(i).await;
12         });
13         handles.push(handle)
14     }
15
16     for handle in handles {
17         handle.await.unwrap();
18     }
19 }
```



Containerized Rust

Demo

- Non-recoverable errors
- Recoverable errors