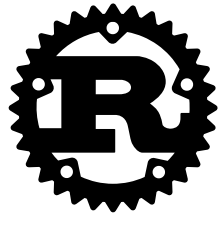
# Safe programming in Rust (TODO: fix title slide)

compscicenter.ru

training@ferrous-systems.com



# General coding workflow

# Overview

This chapter will introduce you to useful things you need to get started. It is not exhaustive.

# Structs

```
#[derive(Eq, PartialEq, Debug)]
(1)
pub struct Point { (2)
    x: i32,
    y: i32,
}
```

1. Derives allow to generate some standard functionality
2. Any type can carry a visibility modifier to export them

# Useful Derives: Debug

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 1, x: 2
},
    println!("{:?}", p); (1)
    println!("{:#?}", p); (2)
}
```

1. Debug makes the `Debug` formatting string work
2. There's also a more structured version, also enabled through it

# Useful Derives: Eq, PartialEq

```
#[derive(Eq,PartialEq,Debug)]
(1) (2)
struct Point {
    x: i32, (3)
    y: i32,
}

fn main() {
    let p1 = Point { x: 1, x: 2
},
    let p2 = Point { x: 1, x: 2
},
    if p1 == p2 { (4)
        println!("The same!");
    }
    assert_eq!(p1, p2); (5)
}
```

1. `Eq` describes total equality: for every pair of values, equality is defined
2. `PartialEq` is enough for getting `==`
3. Both can only be derived if all inner fields are both
4. Equality in action!
5. The `assert_eq!` compares to values and panics if they don't match!

# Enums

```
#[derive(Eq,PartialEq,Debug)]
(2)
enum Direction {
    North, (1)
    South, (1)
    West, (1)
    East, (1)
}

fn main() {
    let dir = Direction::North;
(3)
    println!("Direction: {:?}",
dir);
    let dir2 =
Direction::North;
    assert_eq!(dir, dir2);
}
```

1. Enums carry multiple *variants*.
2. Derives work as expected.
3. Variants are referred to through their main type.

# Enums with data

```
#[derive(Eq,PartialEq,Debug)]
enum Movement {
    North(u32), (1)
    South(u32),
    West(u32),
    East(u32),
    StickAround, (2)
}

fn main() {
    let mov =
Movement::North(1); (3)
    println!("Movement: {:?}",
dir);
    let mov2 =
Movement::StickAround;
    assert_eq!(mov, mov2);
}
```

1. Variants can carry data.
2. Variants carrying data and not can be mixed.
3. Construction works similar to structs.

# Usage: Option

```
fn main() {
    let vec = vec![1,2];
    let mut iter = vec.iter();
    check(iter.next()); ❶
    check(iter.next()); ❷
    check(iter.next()); ❸
}

fn check(item: Option<&u32>) {
    match item {
        Some(value) =>
            println!("{}",
value),
        None =>
            println!("Iterator
exhausted"),
    }
}
```

❶ prints "1"

❷ prints "2"

❸ prints "Iterator exhausted"

# Conclusion: Option

`Option` encodes the *potential, but expected* absence of a value.

TODO: Note-syntax from asciidoc

Note: Due to optimisations, `Option<&u32>` is as large as `&u32`.

# Useful enums: Result

```
enum Result<T,E> { (1)
    Ok(T), (2)
    Err(E), (3)
}
```

(1) Results are generic over *two* types.

(2) One is the value indicating success.

(3) The other is the value type indicating error.

# Result usage

```
fn main() -> Result<(),
io::Error> {
    let file_res: Result<File,
io::Error> = (2)
        File::open("test"); (1)
    match file_res { (3)
        Ok(file) => {
            //...
            Ok( () ) (4)
        },
        Err(e) => { (5)
            println!("Error
opening: {}", path);
            Err(e) (6)
        }
    }
}
```

1. `open` returns a `Result` indicating success or failure
2. Type annotation for clarity
3. `Result`s are also handled with match
4. If success holds no value, `Ok` with `()` is used. Usually written `Ok(())`.
5. Errors are handled the same
6. If errors are passed on, the must be wrapped again

# Special behaviour: must be used

```
fn main() -> Result<(), io::Error> {
    File::open("test"); 1
}
```

```
warning: unused `std::result::Result` that must be used
 --> scratch.rs:2:5
  |
2 |     std::fs::File::open("test");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: this `Result` may be an `Err` variant, which should be
handled
```

# Conclusion: Result

`Result` encodes the potential for error. It forces the user to inspect the `Result` and check if an error occured or denies access to the inner value otherwise.

TODO: Note-syntax from asciidoc

Note: Similar optimisations as to `Option` apply to `Result`

# Slices and Vectors

```
fn main() {
    let bytes: &[u8] = &
[1,2,3]; (1)
    let vec: Vec<u8> =
Vec::from(bytes); (2)
    vec.push(4);
    let vec_slice: &[u8] =
vec.as_slice(); (3)
}
```

1. The (reference) slice `&[..]` is a reference to a region of memory. It stores the length of the data and bounds checked.
2. The vector `Vec<..>` is an owned region of memory. It is growable and shrinkable.
3. Slices can be taken to the memory a vector owns, binding the slices to the vector.

# Vector reallocation

```
fn main {
    let mut v = vec![1,2]; ❶
    let slice = &v[..]; ❷

    vec.push(4); ❸
      //^^^^^^^^ Error here

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

❶ This is a shorthand vector initialization macro.

❷ Taking a slice borrows the memory region the vector owns. `&v[..]` is syntax for "borrow from beginning to end".

❸ Pushing on vector requires a mutable reference to it, violating borrow rules.

# Strings and their slices

Strings and string slices work much the same.

```
fn main() {
    let slice: &str = "Hello world!";
    let string: String = String::from(slice);
}
```

Strings and string slices are `Vec<u8>` and `&[u8]` internally, with the added invariant that they are UTF-8.

# Warning

TODO: fancy ascii-doc warning

Never bypass the UTF-8 invariant on `String` or `&str`, this might lead to memory unsafety.

# Owned vs. borrowed types

Borrowed types need you to make sure that the pointee is always alive. Owned types are easier. Liberally allocate `String` and `Vec` if you run into problems.

TODO: fancy note Note: Rust makes it easy to safely refactor towards more efficient code. Get something working first, before you avoid each and every copy.

# Testing

```
use my_library::my_function;
```
❶

```
#[test]
fn my_test() {
    assert_eq!(1, 1);
}

#[test]
#[should_fail]
fn failing_test() {
    assert_eq!(1, 2);
}
```

Rust and Cargo allows you to easily provide test for your code.

These can be put either directly in the source file or in any file in `tests`.

❶ Only needed when putting files in `tests`.