A Brief Introduction to Rust

Rossano Venturini

University of Pisa

rossano.venturini@gmail.com

Rust: Why?

- **Modern tools**
  - `cargo add`, `cargo fmt`, `cargo clippy`, `cargo test`, `cargo doc`, ...
- **Modern language**
  - Iterators, combinators, pattern matching, ...
- **Efficiency**
  - Performance comparable to C/C++
  - Zero-cost abstractions
- **Safety**
  - Memory safety without a garbage collector
  - Most common pitfalls (null pointers, buffer overflows) are impossible
  - (Usually) if it compiles, it works!
- **Concurrency**
  - Fearless concurrency with message passing, ownership, and data race prevention

Cargo new

```
$ cargo new my_project
```

```
Creating binary (application) `my_project` package
note: see more `Cargo.toml` keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
```

A folder is created with a template project.

```
$ cd my_project
$ cargo run
```

```
Compiling m v0.1.0 (/Users/rossanoventurini/Library/CloudStorage/Dropbox/myPaper/Teaching/RustIntro/m)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.59s
  Running `target/debug/m`
Hello, world!
```

Use an external library

```
cargo add rand
```

You can now immediately use the external library in your code.

```
use rand::Rng;

fn main() {
  let mut rng = rand::thread_rng();
  let n: u32 = rng.gen_range(1..101);
  println!("Random number: {}", n);
}
```

```
cargo run
```

```
...
  Compiling rand_chacha v0.3.1
  Compiling rand v0.8.5
  Compiling m v0.1.0 (/Users/rossanoventurini/Library/CloudStorage/Dropbox/myPaper/Teaching/RustIntro/m)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 3.49s
     Running `target/debug/m`
Random number: 49
```

Testing

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

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

    #[test]
    fn test_sum() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
```

```
$ cargo test
```

```
running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Testing and Documentation

```rust
/// Adds two `usize` numbers and returns the result.
///
/// # Examples
///
/// ```
/// let result = my_crate::add(2, 3);
/// assert_eq!(result, 5);
/// ```
pub fn add(left: usize, right: usize) -> usize {
    left + right
}

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

    #[test]
    fn test_sum() {
        let result = add(2, 2);
        assert_eq!(result, 4);
    }
```

`cargo test` checks also the code in the documentation. This forces to keep the documentation updated.

Automatic Code Formatting with `cargo fmt`

- **Unformatted Code:**

```
fn main() {
let x= 42;println!("Value: {}", x);}
```

- **Run `cargo fmt`:**

```
$ cargo fmt
```

- **Formatted Code:**

```
fn main() {
    let x = 42;
    println!("Value: {}", x);
}
```

Improving Code Quality with `cargo clippy`

- **Code with a naked loop:**

```
fn main() {
  let vec = vec![1, 2, 3, 4, 5];
  let mut sum = 0;

  for i in 0..vec.len() {
      sum += vec[i];
  }
  println!("Sum: {}", sum);
}
```

- **Run `cargo clippy` for suggestions:**

```
warning: the loop variable `i` is only used to index `vec`
--> src/main.rs:5:11
  |
5 |     for i in 0..vec.len() {
  |
help: consider using an iterator
  |
5 |     for <item> in &vec {
```

Why Rust is Safe

- **Type System**
  - Strong, static typing catches many errors at compile time.
  - Lifetimes manage the scope and duration of references, ensuring memory safety.
- **No Null or Undefined Behavior**
  - Rust avoids common pitfalls like null pointer dereferencing with its `Option` and `Result` types.
- **Ownership System**
  - Rust's unique ownership model ensures memory safety without a garbage collector.
- **Borrowing and References**
  - Borrowing ensures controlled access to memory.
  - Rust enforces strict rules on borrowing, preventing dangling pointers and data races.
    - Only one mutable reference at a time or multiple immutable references.

Uninitialized Variables

```
#include <iostream>

using namespace std;

int main() {
  int x;
  cout << x << end;
}
```

```
$ g++ uv.cpp -o uv
```

```
-
```

```
$ ./uv
```

```
0%
```

Uninitialized Variables

```
fn main() {
    let x: i32;
    println!("{x}");
}
```

```
$ cargo build
```

```
 --> src/main.rs:3:15
  |
2 |     let x: i32;
  |         - binding declared here but left uninitialized
3 |     println!("{x}");
  |               ^^^ `x` used here but it isn't initialized
```

Dangling Pointers 1/2

```
#include <iostream>

using namespace std;

int* f() {
  int x = 10;
  return &x;
}

int main() {
  int* x = f();
  cout << *x << endl;;

}
```

```
$ g++ dangling.cpp -o dangling
```

```
dangling.cpp:5:10: warning: address of stack memory associated with local variable 'x' returned [-Wreturn-stack-address]
        return &x;
```

```
$ ./dangling
```

```
10
```

Dangling Pointers 1/2

```
fn f() -> &i32 {
  let x = 10;
  &x

}

fn main() {
  let x = f();
  println!("{*x}");
}
```

```
$ cargo build
```

```
error[E0515]: cannot return reference to local variable `x`
 --> src/main.rs:4:2
  |
4 |     &x
  |     ^^ returns a reference to data owned by the current function
```

Dangling Pointers 2/2

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> v;
  v.push_back(42);
  int* x = v.data();
  v.push_back(20);
  cout << *x << endl;
}
```

```
$ g++ free.cpp -o free
```

```
-
```

```
$ ./dangling
```

```
-1298153424
```

Dangling Pointers 2/2

```
fn main() {
    let mut v = Vec::new();
    v.push(42);
    let x = &v[0];
    v.push(20);
    println!("{}", x);
}
```

```
$ cargo build
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:5:5
  |
4 |     let x = &v[0];
  |             - immutable borrow occurs here
5 |     v.push(20);
  |     ^^^^^^^^^^ mutable borrow occurs here
6 |     println!("{}", x);
  |                    - immutable borrow later used here
```

Use-After-Free

```
#include <iostream>

using namespace std;

int main() {
  int* x = new int(10);
  delete x;
  std::cout << *x;
}
```

```
$ g++ free.cpp -o free
```

```
-
```

```
$ ./free
```

```
0%
```

Use-After-Free

```
fn main() {
  let x = Box::new(10);
  drop(x);
  println!("{}", *x);
}
```

```
$ cargo build
```

```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:4:20
  |
2 |     let x = Box::new(10);
  |         - move occurs because `x` has type `Box<i32>`, which does not implement the `Copy` trait
3 |     drop(x);
  |          - value moved here
4 |     println!("{}", *x);
  |                    ^^ value borrowed here after move
```

Out-of-Bounds Access

```
#include <iostream>

using namespace std;

int main() {
  vector<int> v = {1, 2, 3};
  cout << v[5];
}
```

```
$ g++ -std=c++11 oob.cpp -o oob
```

```
-
```

```
$ ./oob
```

```
0%
```

Out-of-Bounds Access

```
fn main() {
  let v = vec![1, 2, 3];
  println!("{}", v[5]);
}
```

```
$ cargo build
```

```
-
```

```
$ ./target/debug/oop
```

```
thread 'main' panicked at src/main.rs:3:21:
index out of bounds: the len is 3 but the index is 5
```

Data Races in Multithreading

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

using namespace std;

int main() {
    int x = 0;
    std::thread t1([&]() { x++; });
    std::thread t2([&]() { x++; });

    t1.join();
    t2.join(); // Wait for both threads to complete

    cout << x << endl;  // Potential undefined behavior
}
```

```
$ g++ -std=c++11 mt.cpp -o mt
```

```
-
```

```
$ ./mt
```

```
2
```

Data Races in Multithreading

```
fn main() {
  let mut x = 0;
  let t1 = thread::spawn(|| { x += 1; });  // Error: `x` cannot be borrowed mutably in multiple threads
  let t2 = thread::spawn(|| { x += 1; });
  t1.join().unwrap();
  t2.join().unwrap();
}
```

```
$ cargo build
```

```
error[E0499]: cannot borrow `x` as mutable more than once at a time
 --> src/main.rs:8:28
5 |       let t1 = thread::spawn(|| {
  |                -             -- first mutable borrow occurs here
  |  _____________|
6 | |         x += 1;
  | |         - first borrow occurs due to use of `x` in closure
7 | |     }); // Error: `x` cannot be borrowed mutably in mult...
  | |______- argument requires that `x` is borrowed for `'static`
8 |       let t2 = thread::spawn(|| {
  |                              ^^ second mutable borrow occurs here
9 |           x += 1;
  |           - second borrow occurs due to use of `x` in closure
```

Learn More

- The Rust Programming Language: The Book!
- Let's Get Rusty: Video Lectures based on the book
- Rustlings: Exercises to familiarize with syntax
- List of Resources to Learn Rust

Slides

- Basic Syntax
- Ownership and References
- Advanced Syntax
- Traits and Generics