Introduction to Rust

```
Module Code: ELEE1119

Module Name: Advanced Computer Engineering

Credits: 30

Module Leader: Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA
```

Download as a PDF

Rust

Building tools, to writing web apps, working on servers, and creating embedded systems etc

```
fn fibonacci(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fibonacci(n - 1) + fibonacci(n - 2),
    }
}
```

Rust features

- **Type safe**: The compiler assures that no operation will be applied to a variable of a wrong type.
- **Memory safe**: Rust pointers (known as references) always refer to valid memory.
- **Data race free**: Rust's borrow checker guarantees thread-safety by ensuring that multiple parts of a program can't mutate the same value at the same time.
- **Zero-cost abstractions**: Rust allows the use of high-level concepts, like iteration, interfaces, and functional programming, with minimal to no performance costs. The abstractions perform as well, as if you wrote the underlying code by hand.
- **Minimal runtime**: Rust has a minimal and optional runtime. The language also has no garbage collector to manage memory efficiently. In this way, Rust is most similar to languages like C and C++.
- **Targets bare metal**: Rust can target embedded and "bare metal" programming, making it suitable to write an operating system kernel or device drivers.

Unique features of Rust

- Collection of Features called **Rust Module System**
  - Crates
  - Modules
  - Paths

```
my_library  // crate
├── src
│   ├── lib.rs
│   ├── models
│   │   ├── mod.rs // module
│   │   ├── user.rs // module
│   │   └── product.rs // module
│   └── utils
│       ├── mod.rs // module
│       └── math.rs // module
└── Cargo.toml // config
```

```
crate::utils::math::add // path
```

Crate

- **Crates:**
  - A crate is a compilation unit. It's the smallest piece of code the Rust compiler can run.
  - The code in a crate is compiled together to create a binary executable or a library.
  - Only crates are compiled as reusable units.
  - A crate contains a hierarchy of Rust modules with an implicit, unnamed top-level module.

```
my_library  // crate
├── src
│   ├── lib.rs
│   ├── models
...
```

Modules

- **Modules:**
  - Used to split code into logical units.
  - A module is a collection of items such as `functions`, `structs`, `traits`, `implementation blocks`, and even other modules.
  - Help manage visibility between different parts of the code, allowing you to specify which items are public (accessible outside the module) and which are private (accessible only within the module)

```
my_library  // crate
├── src
│   ├── lib.rs
│   ├── models
│   │   ├── mod.rs // module
...
```

```rust
// models/mod.rs
mod user;
mod product;
```

Paths

- Are used to refer to items in modules.
- They allow you to name items and bring them into scope with the use keyword.
- For example, if you have an `Asparagus` type in the `garden vegetables` module, you would refer to it as `crate::garden::vegetables::Asparagus`

```
my_library  // crate
├── src
│   ├── lib.rs
│   ├── models
│   │   ├── mod.rs // module
│   │   ├── user.rs // module
│   │   └── product.rs // module
│   └── utils
│       ├── mod.rs // module
│       └── math.rs // module
└── Cargo.toml // config
```

```rust
// models/user.rs
use crate::utils::math::add;
```

Rust Crates and Libraries

- Rust Standard Library `std` contains reusable code for fundamental definitions and operations in Rust programs.
- There are tens of thousands of libraries and third-party crates available to use in Rust programs most of which can be accessed through Rust's third-party crate repository crates.io

Some crates we will use:

- `std` - The Rust standard library. In the Rust exercises, you'll notice the following modules:
    - `std::collections` - Definitions for collection types, such as HashMap.
    - `std::env` - Functions for working with your environment.
    - `std::fmt` - Functionality to control output format.
    - `std::fs` - Functions for working with the file system.
    - `std::io` - Definitions and functionality for working with input/output.
    - `std::path` - Definitions and functions that support working with file system path data.
- `structopt` - A third-party crate for easily parsing command-line arguments.
- `chrono` - A third-party crate to handle date and time data.
- `regex` - A third-party crate to work with regular expressions.
- `serde` - A third-party crate of serialization and deserialization operations for Rust data structures.

Create and manage projects with Cargo

While it's possible to use the Rust compiler (`rustc`) directly to build crates, most projects use the Rust build tool and dependency manager called `Cargo`.

`Cargo` does lots of things for you, including:

- Create new project templates with the `cargo new` command.
- Build a project with the `cargo build` command.
- Build and run a project with the `cargo run` command.
- Test a project with the `cargo test` command.
- Check project types with the `cargo check` command.
- Build documentation for a project with the `cargo doc` command.
- Publish a library to crates.io with the `cargo publish` command.
- Add dependent crates to a project by adding the crate name to the `Cargo.toml` file.

Rust Naming Conventions 1

Basic Rust naming conventions are described in RFC 430.

Item	Convention
Crates	`snake_case` (but prefer single word)
Modules	`snake_case`
Types	`UpperCamelCase`
Traits	`UpperCamelCase`
Enum variants	`UpperCamelCase`
Functions	`snake_case`
Methods	`snake_case`
General constructors	`new` or `with_more_details`
Conversion constructors	`from_some_other_type`
Local variables	`snake_case`

Rust Naming Conventions 2

Convention	Example	General Meaning
`to_*()`	`str::to_string()`	A conversion from one type to another that may have an allocation or computation cost. Usually a *Borrowed* type to *Owned* type.
`as_*()`	`String::as_str()`	Convert an *Owned* type into a *Borrowed* type. It is usually cheap (maybe even zero-cost) to use this function.
`into_*()`	`String::into_bytes()`	Consume a type `T` and convert it into an *Owned* type `U`.
`from_*()`	`SocketAddr::from_str()`	Create an *Owned* type from an *Owned* or *Borrowed* type.
`*_mut()`	`str::split_at_mut()`	Denotes a mutable reference.
`try_*()`	`usize::try_from()`	Method will return a `Result` or `Option` type. Usually `Result`.
`with_*()`	`Vec::with_capacity()`	A constructor that has one or more parameters used to configure the type.

Rust Syntax

```rust
fn main(){
    let an_integer = 1u32;
    let a_boolean = true;
    let unit = ();

    // copy `an_integer` into `copied_integer`
    let copied_integer = an_integer;

    println!("An integer: {:?}", copied_integer);
    println!("A boolean: {:?}", a_boolean);
    println!("Meet the unit value: {:?}", unit);

    // The compiler warns about unused variable bindings; these warnings can
    // be silenced by prefixing the variable name with an underscore
    let _unused_variable = 3u32;

    let noisy_unused_variable = 2u32;
    // FIXME ^ Prefix with an underscore to suppress the warning
    // Please note that warnings may not be shown in a browser
}
```

```
An integer: 1
A boolean: true
Meet the unit value: ()
```

Mutability

Variable bindings are **immutable** by default, but this can be overridden using the `mut` modifier.

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    println!("Before mutation: {}", mutable_binding);

    // Ok
    mutable_binding += 1;

    println!("After mutation: {}", mutable_binding);

    // Error! Cannot assign a new value to an immutable variable
    _immutable_binding += 1;
}
```

Scope and Shadowing

Variable bindings have a scope, and are constrained to live in a block. A block is a collection of statements enclosed by braces `{}`.

```rust
fn main() {
    // This binding lives in the main function
    let long_lived_binding = 1;

    // This is a block, and has a smaller scope than the main function
    {
        // This binding only exists in this block
        let short_lived_binding = 2;

        println!("inner short: {}", short_lived_binding);
    }
    // End of the block

    // Error! `short_lived_binding` doesn't exist in this scope
    println!("outer short: {}", short_lived_binding);
    // FIXME ^ Comment out this line

    println!("outer long: {}", long_lived_binding);
}
```

Scope and Shadowing

```rust
fn main() {
    let shadowed_binding = 1;

    {
        println!("before being shadowed: {}", shadowed_binding);

        // This binding *shadows* the outer one
        let shadowed_binding = "abc";

        println!("shadowed in inner block: {}", shadowed_binding);
    }
    println!("outside inner block: {}", shadowed_binding);

    // This binding *shadows* the previous binding
    let shadowed_binding = 2;
    println!("shadowed in outer block: {}", shadowed_binding);
}
```

Freezing

When data is bound by the same name immutably, it also *freezes*. Frozen data can't be modified until the immutable binding goes out of scope:

```rust
fn main() {
    let mut _mutable_integer = 7i32;
    {
        // Shadowing by immutable `_mutable_integer`
        let _mutable_integer = _mutable_integer;

        // Error! `_mutable_integer` is frozen in this scope
        _mutable_integer = 50;
        // FIXME ^ Comment out this line

        // `_mutable_integer` goes out of scope
    }
    // Ok! `_mutable_integer` is not frozen in this scope
    _mutable_integer = 3;
}
```

Freezing with `const`

```
fn main() {
    const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
    {
        println!("before being shadowed: {}", THREE_HOURS_IN_SECONDS);
        const THREE_HOURS_IN_SECONDS: u32 = 60u32 * 60u32 * 4u32;
        println!("shadowed in inner block: {}", THREE_HOURS_IN_SECONDS);
    }
}
```

▶ What happens and why?

> [!NOTE]
>
> This alert uses [!NOTE]