Distributed Systems Lab 01

Introduction to Rust and Organization of the Labs

Welcome to the Distributed Systems laboratory classes! Before proceeding, please, read additional information about the organization of the labs.

This lab introduces the Rust programming language. We will use files contained in this package, which you should download and extract locally.

Learning Section

Why Rust?

Although the theory of distributed systems is not related to any specific programming language, during the labs we will use **Rust** to implement discussed distributed algorithms. It is a compiled language which aims at **efficiency** by providing zero (or low) cost abstractions, and **reliability** by assuring memory-safety and thread-safety at compile-time, making it an ideal choice for the labs. You are not expected to already know Rust, as the first few labs are dedicated to learning the language.

Installing Rust

Follow the instructions on the Install Rust website. Verify that your installation works by running the following commands:

```
rustc --version
cargo --version
```

You can compile a single Rust file with:

```
rustc main.rs
```

However, Rust has a tool for building and managing projects, called ***cargo***. Usually, there is no need to call *rustc* directly and, in fact, it is not recommended, as its options can be changed without a notice.

Cargo

Essentially, *Cargo* is a **package manager** for Rust. It takes care of downloading and preparing dependencies (libraries, called **crates**) of a package. Each package includes a manifest file, `Cargo.toml`, which, in addition to the dependencies, also lists the basic metadata and configuration of the package. For instance:

```
# General information about the package:
[package]
```

```
name = "lab-01"
version = "0.1.0"
authors = ["Filip Plata <fp371335@students.mimuw.edu.pl>"]
edition = "2021"

# Crates required by this package.
# You can browse the Rust crate registry at https://crates.io:
[dependencies]

# Dependencies required only by examples and tests:
[dev-dependencies]
ntest = "0.7"

# Binary target settings.
# E.g., path to a file which defines the main function of the binary:
[[bin]]
path = "main.rs"


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

Cargo is also a **command line tool** for performing common tasks on a package. The basic commands that we will use are:

```
# Compile the package:
cargo build

# Run the main function of the package
# (when run in the directory with Cargo.toml file):
cargo run

# Run tests for the package:
cargo test

# Format source code in the package:
cargo fmt

# Perform analysis of the source code to get hints on potential
# problems and possible simplifications. Makes it easier to write
# idiomatic code:
cargo clippy

# Generate HTML documentation from doc comments:
cargo doc
```

The default result of running *cargo* is a *debug* build. To change the target to *release*, add the `--release` flag to *cargo* commands.

Editors for programming in Rust

Visit the Rust website for a list of editors/IDEs that are known to have a Rust integration available. Visual Studio Code seems to be the most popular one among the Rust community. Another option, which is not mentioned on the website and is available for free for students (at the time of writing this text), is CLion with a Rust plugin (with a native debugger built in).

For Rust support in editors, you should use rust-analyzer. Previously Rust Language Server (RLS) was used in many editors, but it has been deprecated and is no longer supported.

During the labs you can use whichever editor you prefer. The examples and assignments will be built and tested using *cargo*. However, it is advisable to have a comfortable setup to code in Rust.

Learning Rust

Assuming that you already know a few programming languages, we will just shortly highlight the most characteristic features of Rust.

Let us start with variables. They are declared using a `let` keyword. By default, variables are **immutable**. If you want to be allowed to modify their values, use also a `mut` keyword to declare them as **mutable**. For example:

```rust
let i1: i32 = 42;
// Won't compile:
// i1 = 43;
println!("Immutable i32: {}", i1);

let mut i2: i32 = 42;
i2 = 43;
println!("Mutable i32: {}", i2);

let a1: [u32; 2] = [0, 1];
// Won't compile:
// a1[0] = 42;
println!("Immutable array with two u32 values: {:?}", a1);
// Above `{:?}` marker displays arrays for debugging purposes.
// Arrays don't implement user-friendly formatting (the `{}` marker).

let mut a2: [u32; 2] = [0, 1];
a2[0] = 42;
println!("Mutable array with two u32 values: {:?}", a2);

let v1: Vec<i64> = vec![0, 1];
// Above `vec!` is a macro which simplifies creation of vectors.
// Won't compile:
// v1.truncate(1);
println!(
    "Immutable vector (heap-allocated) with two i64 values: {:?}",
    v1
);

let mut v2: Vec<i64> = vec![0, 1];
v2.truncate(1);
println!(
    "Mutable vector (heap-allocated) with two i64 values: {:?}",
    v2
);

let s1: String = String::from("Hello");
// Won't compile:
// s1.push('!');
println!("Immutable string (with heap-allocated buffer): {}", s1);

```

```
let mut s2: String = String::from("Hello");
s2.push('!');
println!("Mutable string (with heap-allocated buffer): {}", s2);
```

Arguably the most important feature of Rust is a concept of **ownership**: for each value there is exactly one variable that owns it. This approach facilitates automatic resource management (when a variable goes out of scope, its value is dropped, for example, removed from the heap) and safety (eliminates problems like double free, resource leaking, data races, etc.). When a value is assigned to another variable (or passed to a function), the ownership is transferred to the new variable—the value is **moved**—and the old variable cannot be used again (you can imagine that it becomes "empty"). Only when assigning (or passing to a function) values of simple types (more precisely: types that implement the `Copy` trait, but more on traits later) are the values **copied**, and thus both variables can be still used as they own different values. To obtain a copy of a value of a complex type (more precisely, of a type that implements the `Clone` trait) you can use the `clone()` method. An example follows:

```
{
    let i1 = 42u32;
    // Above type of the variable is inferred from the type
    // of the value: u32.
    let v1 = vec![0, 1];
    // Above type of the vector's elements defaults to i32.

    let i2 = i1;
    // Above value is copied so both variables own their own values.
    let v2 = v1;
    // Above value is moved so `v1` no longer owns the vector.
    let v3 = v2.clone();
    // Above value is cloned so `v3` owns a copy of the vector.

    println!("Old integer: {}", i1);
    // Won't compile:
    // println!("Old vector: {:?}", v1);
    println!("New integer: {}", i2);
    println!("Moved vector: {:?}", v2);
    println!("Cloned vector: {:?}", v3);
} // `v2` and `v3` go out of scope, so their values are dropped
  // (the vectors are deallocated). `v1` also goes out of scope
  // but since it doesn't own any value here, nothing is dropped
  // (thus the vector isn't dropped twice).
```

Although the rule of the single owner is really handy from the perspectives of resource management and safety, without additional concepts, it would introduce significant burden on even simple tasks. For instance, consider a helper function that prints a vector:

```
fn main() {
    let mut v: Vec<u8> = vec![0, 1];

    v = print_by_value_and_return(v);
    // Above vector is moved out of the variable to the function.
    // Then it is returned by the function and reassigned (moved)
    // to the variable.

    print_by_value(v);
    // Above vector is moved out of the variable to the function.

    // Won't compile, `v` no longer owns the vector:
```

```rust
    // println!("Vector: {:?}", v);
}

fn print_by_value_and_return(v: Vec<u8>) -> Vec<u8> {
    println!("The vector: {:?}", v);
    v
    // The last expression of a function is its return value.
    // There is no need to write `return v;` (but one can do so).
}

fn print_by_value(v: Vec<u8>) {
    println!("The vector: {:?}", v);
} // `v` goes out of scope so the vector is dropped when the
  // function ends.
```

Fortunately, Rust features also **references**. & is a referencing operator, * is a dereferencing operator. & is also used to denote types that are references to some type. When a reference to a value is created, it does not change the owner of the value. Instead, it **borrows** the value from the owner. The "lease" ends when the variable that holds the reference goes out of scope. References are immutable unless they are declared with the `mut` keyword. The compiler verifies that at any given time there is either any number of immutable references to an object, or there is at most one mutable reference to the object. The compiler analyzes also **lifetimes** of all values to detect problems like dangling references. Rust provides also so-called **slices**, which are references to contiguous sequences of elements (e.g., to a part of an array, of a vector, of a string, and the like). An example follows:

```rust
fn main() {
    let mut v: Vec<u8> = vec![0, 1];

    print_by_reference(&v);
    // Above function call performs immutable borrowing.

    print_by_slice(&v[0..2]);
    // Above function call performs immutable borrowing
    // of the range [0th element, 2nd element) of the vector.

    // Multiple immutable references are safe:
    let v_imm_ref_1 = &v;
    let v_imm_ref_2 = &v;

    // Won't compile, there are already immutable references:
    // let v_mut_ref = &mut v;

    print_by_reference(v_imm_ref_2);
    print_by_reference(v_imm_ref_1);

    // The above immutable references are not used below so
    // it is now safe to have a mutable reference:
    let v_mut_ref = &mut v;
    add_by_mut_reference(v_mut_ref);
    print_by_reference(v_mut_ref);
}

fn print_by_reference(v: &Vec<u8>) {
    println!("The vector: {:?}", v);
}

fn add_by_mut_reference(v: &mut Vec<u8>) {
```

```
    v.push(42);
}

fn print_by_slice(v: &[u8]) {
    println!("The vector: {:?}", v);
}

// Won't compile, the lifetime of the returned reference is longer
// than the lifetime of the value:
// fn create_vector() -> &Vec<u8> {
//     let v: Vec<u8> = vec![0, 1];
//     &v
// }
```

The concepts discussed above are crucial features of Rust. You can read more about them in Rust Book and in Rust by Example.

Rust provides also many other useful features, of course. However, since the following concepts are present also in other programming languages (although they may be named there differently), and you probably already know them quite well, we do not discuss them here. Instead, we provide links to the Rust Book pointing at chapters that you should read yourself:

- Structs with methods (Chapter 5),
- Enums and pattern matching (Chapter 6),
- Error handling (Chapter 9),
- Traits and generics (Chapter 10),
- Using simple macros (Chapter 19.5),
- Writing tests (Chapter 11),
- Creating, building and managing a Rust project (Chapter 1.3).

In the package with additional files for this lab there is an `examples/rust_overview.rs` file which presents of most of the features. You can run the examples using an `--example` option to *cargo*:

```
# In the main directory (i.e., where Cargo.toml is):

# Build and run the examples:
cargo run --example rust_overview

# Build and run tests of the examples:
cargo test --example rust_overview
```

Small Assignment

This is only a trial assignment: it is *not* worth any points. However, you can get feedback that will be useful in assessing how subsequent assignments should be implemented, tested, and packaged. Therefore, we strongly recommend submitting your solution to this assignment.

Your task is to implement in Rust calculations of Fibonacci numbers. The first numbers in the sequence are 0 and 1. We start indexing from zero, so your implementation shall return:

- 0 -> 0
- 1 -> 1
- 2 -> 1
- 3 -> 2
- …

More specifically, you are to implement a `Fibonacci` struct. It shall provide:

1. An associated function (see Chapter 5.3 of the Rust Book) `fibonacci`, which returns the n-th Fibonacci number. The calculations shall be performed using `u8` integers. To implement wrapping addition you can use the `wrapping_add()` method of `u8`.
2. An iterator (i.e., the `Iterator` trait), which iterates over the Fibonacci sequence (see Implementing Iterator to learn how to implement a simple iterator). The calculations shall be performed using `u128` integers. To implement checked addition you can use the `checked_add()` method of `u128`.

In the package with additional files for this lab there is a `solution.rs` file. It is a template for your solution. You are to implement the missing parts (marked there with the `unimplemented!` macro). Do not change the interface defined in the template, follow the requirements specified in the doc comments.

The package includes also a `main.rs` file, which demonstrates an exemplary usage of the `Fibonacci` struct, and some simple tests.

Additional Homework

Get comfortable with Rust. Browse the Rust Book and the Rust by Example. Play with the `examples/rust_overview.rs` file. Practice coding in Rust, because the next lab we will assume that you can write single-threaded programs in Rust.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski.