Module 00: First Steps

Introduction

The Rust programming language is still fairly new: its 1.0 release is younger than 42 itself! For this reason, please understand that the knowledge acquired here may become obsolete at some point. Though this is true for most programming languages, Rust is still updated quite regularly (almost on a monthly basis) and you should be aware of that. Never stop learning.

At times, Rust might feel a bit hard to get into, may it be its syntax, the borrow checker, or more generally, the number of details it requires the developer to think about when writing code. Trust the compiler. It is always (in 99% of cases) right - even when it complains for apparently nothing. Plus it is pretty well known to provide really good error messages. Read the errors, correct your code, and get over it! And trust me: it's worth it. The languages will pay you back for this.

Along you journey, do not fear the Internet. It can provide lots of resources to learn the Rust programming language. I will, however, advise you to to primarily focus on the official documentation. It contains the most up-to-date information about this rapidly evolving language.

General Rules

- Any code you turn in must compile *without warnings* using the `rustc` compiler available on the school's machines without additional options. You are *not* allowed to use `unsafe` code anywhere in your code (not until the last module :p).
- For exercises using the `cargo` package manager, the same rule applies. In that case, only the crates specified in the `allowed dependencies` section are allowed. Any other dependency is forbidden. More generally, only the symbols specified in `allowed symbols` are authorized within an exercise.
- You are generally *not* authorized to modify lint levels - either using `#\[attributes\]`, `#!\[global_attributes\]` or with command-line arguments. You may optionally allow the `dead_code` lint to silence warnings about unused variables, functions, etc.

```
// Either globally:
#![allow(dead_code)]

// Or locally, for a simple item:
#[allow(dead_code)]
fn my_unused_function() {}
```

Exercise 00: Hello, World!

```
turn-in directory:
    ex00/

files to turn in:
    hello.rs

allowed symbols:
    std::println
```

What's a program without side effects?

Create a **program** that prints the string `Hello, World!`, followed by a line feed.

```
>_ ./hello
Hello, World!
```

Exercise 01: Point Of No Return

```
turn-in directory:
    ex01/

files to turn in:
    min.rs

allowed symbols:
    std::println
```

Create a `min` **function** that takes two integers, and returns the smaller one. To make the file compile and for it to be testable, you must add a `main` function to showing that your function is indeed correct.

The function must be prototyped like this:

```
fn min(a: i32, b: i32) -> i32;
```

Oh, I almost forgot. The `return` keyword is forbidden! Good luck with that ~

Exercise 02: yyyyyyyyyyyyy

```
turn-in directory:
    ex02/

files to turn in:
    yes.rs  collatz.rs  print_bytes.rs

allowed symbols:
    std::println  str::bytes
```

Create three **functions**. Each function must use one kind of loop supported by Rust, and you cannot use the same loop kind twice.

The functions must be prototyped as follows:

```
fn yes() -> !;
fn collatz(start: u32);
fn print_bytes(s: &str);
```

The `yes` function must print the message `y`, followed by a line feed. It must do it indefinitely.

```
y
y
y
y
y
y
y
...
```

The `collatz` function must execute the following algorithm…

- Let n be any natural number.
- If n is even, then n becomes $n/2$
- If n is odd, then n becomes $3n + 1$

…until n equals 1. On each iteration, n must be displayed on the standard output, followed by a line feed.

```
Input:
3

Output:
3
10
5
16
8
4
2
1
```

The `print_bytes` function must print every byte of the provided string.

```
Input:
"Déjà Vu\n"

Output:
68
195
```

```
169
106
195
160
32
86
117
10
```

Once again, you must add `main` functions to prove that your functions are correct.

Exercise 03: FizzBuzz

```
turn-in directory:
    ex03/

files to turn in:
    fizzbuzz.rs

allowed symbols:
    std::println
```

Create a **program** that plays the popular (and loved!) game "fizz buzz" from 1 to 100.

The rules have changed a bit, however. They must be followed in order.

- When the number is both a multiple of 3 and 5, "fizzbuzz" must be displayed.
- When the number is a multiple of 3, "fizz" must be displayed.
- When the number is a multiple of 5, "buzz" must be displayed.
- When the number is congruent to 3 modulo 11 "FIZZ" is displayed.
- When the number is congruent to 5 modulo 11 "BUZZ" is displayed.
- Otherwise, the number itself is written.

Example:

```
>_ ./fizzbuzz
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
```

```
11
fizz
13
FIZZ
fizzbuzz
BUZZ
17
fizz
19
buzz
...
```

For this exercise, you can only use one `for` loop, and one `match` statement. Nothing more.

Exercise 04: Shipping With Cargo

```
turn-in directory:
    ex04/

files to turn in:
    src/default.rs  src/overflow.rs  src/other.rs  Cargo.toml

allowed symbols:
    std::println
```

Create a Cargo project.

- Its name must be "module00-ex04"
- It must use Rust edition 2021
- Its author must be you.
- Its description must be "my answer to the fifth exercise of the first module of 42's Rust Piscine"
- It must not be possible to publish the package, even when using `cargo publish`.
- The following commands must give this output:

```
>_ cargo run
Hello, Cargo!
>_ cargo run --bin other
Hey! I'm the other bin target!
>_ cargo run --release --bin other
Hey! I'm the other bin target!
I'm in release mode!
```

- In `release` mode, the final binary must not have any visible symbols in its binary.

```
>_ cargo build
>_ nm <target-dir>/debug/module00-ex04 | head
000000000004d008 V DW.ref.rust_eh_personality
0000000000049acc r GCC_except_table0
0000000000049ad8 r GCC_except_table1
00000000000493e0 r GCC_except_table1049
00000000000493f8 r GCC_except_table1051
0000000000049410 r GCC_except_table1060
0000000000049430 r GCC_except_table1072
0000000000049440 r GCC_except_table1073
000000000004945c r GCC_except_table1075
0000000000049470 r GCC_except_table1076
>_ cargo build --release
>_ nm <target-dir>/release/module00-ex04
nm: <target-dir>/release/module00-ex04: no symbols
```

- It must have a custom profile inheriting the "dev" profile. That profile must simply disable integer overflow checks. For this reason, you will name it `no-overflows`.

```
>_ cargo run --bin test-overflows
thread 'main' panicked at 'attempt to add with overflow', src/overflow.rs:3:5
>_ cargo run --profile no-overflows --bin test-overflows
255u8 + 1u8 == 0
```

Exercise 05: Friday The 13th

```
turn-in directory:
    ex05/

files to turn in:
    src/main.rs  Cargo.toml

allowed symbols:
    std::{assert, assert_eq, assert_ne}  std::panic  std::{print, println}
```

Write a **program** which prints every Friday that falls on the 13th of the month, since the first day of year 1 (it was a monday).

To complete this task, you must write and use the following function:

```rust
fn is_leap_year(year: u32) -> bool;
fn num_days_in_month(year: u32, month: u32) -> u32;
```

- `is_leap_year` must determine whether a given year is a leap year or not.

- `num_days_in_month` must compute how many days a given month of a given year has.

Example:

```
>_ cargo run
Friday, April 13, 1
Friday, July 13, 1
Friday, September 13, 2
Friday, December 13, 2
Friday, June 13, 3
Friday, February 13, 4
Friday, August 13, 4
Friday, May 13, 5
Friday, January 13, 6
Friday, October 13, 6
...
```

You must add tests to your Cargo project to verify that `is_leap_year` and `num_days_in_month` both work as expected. Specifically, you must show that:

- 1600 is a leap year.
- 1500 is a common year.
- 2004 is leap year.
- 2003 is a common year.
- February has 29 days on leap years, but 28 on common years.
- Other months have the correct number of days on both leap and common years.
- Passing an invalid month to `num_days_in_month` must make the function panic.
- Passing an year `0` to `is_leap_year` must make the function panic.

It must be possible to run those tests using `cargo test`.

Exercise 06: Guessing Game

```
turn-in directory:
    ex06/

files to turn in:
    src/main.rs  Cargo.toml

allowed dependencies;
    ftkit

allowed symbols:
    ftkit::read_number  ftkit::random_number
    i32::cmp  std::cmp::Ordering
```

Create a **program** that plays the guessing game.

```
>_ cargo run
Me and my infinite wisdom have found an appropriate secret you shall yearn for.
12
This student might not be as smart as I was told. This answer is obviously too weak.
25
Sometimes I wonder whether I should retire. I would have guessed higher.
19
That is right! The secret was indeed the number 19, which you have brilliantly discovered!
```

You can't use the `<`, `>`, `<=`, `>=` and `==` operators!

Exercise 07: String Pattern Compare

```
turn-in directory:
    ex07/

files to turn in:
    src/lib.rs  src/main.rs  Cargo.toml

allowed dependencies:
    ftkit

allowed symbols:
    std::{assert, assert_eq}  <[u8]>::{len, is_empty}
    str::as_bytes  ftkit::ARGS
```

Create a **library** that exposes the function `strpcmp`.

```rust
fn strpcmp(query: &[u8], pattern: &[u8]) -> bool;
```

- `strpcmp` determines whether `query` matches the given `pattern`.
- `pattern` may optionally contain `"*"` characters, which can match any number of any character in the query string.

Example:

```rust
assert!(strpcmp(b"abc", b"abc"));

assert!(strpcmp(b"abcd", b"ab*"));
assert!(!strpcmp(b"cab", b"ab*"));

assert!(strpcmp(b"dcab", b"*ab"));
assert!(!strpcmp(b"abc", b"*ab"));

assert!(strpcmp(b"ab000cd", b"ab*cd"));
assert!(strpcmp(b"abcd", b"ab*cd"));
```

```rust
assert!(strpcmp(b"", b"****"));
```

When in doupt, do what **Bash** does.

Your crate must also include a bin-target which may be used to test the library easily. Note that the `strpcmp` function must still be in the *library*!

```
>_ cargo run -- 'abcde' 'ab*'
yes
>_ cargo run -- 'abcde' 'ab*ef'
no
```