Module 01: Indirections

Introduction

Rust is basically operating on the same hardware abstraction level as C. As such, it does have a way to create pointers to any existing value. In Rust, however, it is *impossible* to create invalid pointers. When using that language, the compiler *ensures* statically that every pointer you create won't ever be invalidated while you are using it. To provide this guarentee, Rust uses a system known as the *Borrow Checker*.

Rust's Borrow Checker can be a bit hard to get used to, but remember that 99% of the programs it rules out are actually invalid and could potentially lead to memory unsafety and undefined behavior. This module will introduce you to how it works, and what information it uses to determine whether a program is valid or not.

General Rules

- Any exercise you turn in must compile using the `cargo` package manager, either with `cargo run` if the subject requires a *program*, or with `cargo test` otherwise. Only dependencies specified in the `allowed dependencies` section are allowed. Only symbols specified in the `allowed symbols` section are allowed.
- Every exercise must be part of a virtual Cargo workspace, a single `workspace.members` table must be declared for the whole module.
- Everything must compile *without warnings* with the `rustc` compiler available on the school's machines without additional options. You are *not* allowed to use `unsafe` code anywere in your code.
- 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.
- You are *strongly* encouraged to write extensive tests for the functions and systems you turn in. Correcting an already well-tested exercise is easier and faster than having to write them during defense. Tests (when not specifically required by the subject) can use the symbols you want, even if they are not specified in the `allowed symbols` section.

Exercise 00: Creating References

```
turn-in directory:
    ex00/
```

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

Create two **functions**. Both must add two integers together.

```rust
fn add(a: &i32, b: i32) -> i32;
fn add_assign(a: &mut i32, b: i32);
```

- `add` must return the result of the operation.
- `add_assign` must store the result of the operation in `a`.

Exercise 01: Point Of No Return (v2)

```
turn-in directory:
    ex01/

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

Write a **function** that returns the smallest value among two numbers.

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

- Note that you may have to add some *lifetime annotations* to the function in order to make it compile.
- The `return` keyword is still disallowed.

Exercise 02: The Name Of Colors

```
turn-in directory:
    ex02/

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

Create a **function** that maps three color components to a name.

The name of a color is determined using the following rules, applied in order. The first rule that `match`es the input color must be selected.

- The color `[0, 0, 0]` is "pure black".
- The color `[255, 255, 255]` is "pure white".
- The color `[255, 0, 0]` is "pure red".
- The color `[0, 255, 0]` is "pure green".
- The color `[0, 0, 255]` is "pure blue".
- The color `[128, 128, 128]` is "perfect grey".
- Any color whose components are all bellow 31 is "almost black".
- Any color whose red component is above 128, whose green and blue components are between 0 and 127 is "redish".

- Any color whose green component is above 128, whose red and blue components are between 0 and 127 is "greenish".
- Any color whose blue component is above 128, whose red and green components are between 0 and 127 is "blueish".
- Any other color is named "unknown".

```
const fn color_name(color: &[u8; 3]) -> &str;
```

You might need to add *lifetime* annotations to the function to make it compile. Specifically, the following test must compile and run:

```
#[cfg(test)]
#[test]
fn test_lifetimes() {
    let name_of_the_best_color;

    {
        let the_best_color = [42, 42, 42];
        name_of_the_best_color = color_name(&the_best_color);
    }

    assert_eq!(name_of_the_best_color, "unknown");
}
```

Exercise 03: Largest Group

```
turn-in directory:
    ex03/

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

allowed symbols:
    <[u32]>::{len, is_empty, contains}
```

Write a **function** that returns the largest subslice of `haystack` that contains *all* numbers in `needle`.

```
fn largest_group(haystack: &[u32], needle: &[u32]) -> &[u32];
```

- When multiple groups match the `needle`, the largest one is returned.
- When multiple largest groups are found, the first one is returned.

Example:

```
assert_eq!(largest_group(&[1, 3, 4, 3, 5, 5, 4], &[5, 3]), &[3, 5, 5]);
assert_eq!(largest_group(&[1, 3, 4, 3, 5, 5, 4], &[5], &[5, 5]));
assert_eq!(largest_group(&[1, 3, 4, 3, 5, 5, 4], &[], &[]));
assert_eq!(largest_group(&[1, 3, 4, 3, 5, 5, 4], &[4, 1], &[]));
```

Once again, you may need to specify some *lifetime annotations* for the function. To check whether your annotations are correct for that case, you can use this pre-defined `test_lifetimes` test. It must compile and run.

```
#[test]
#[cfg(test)]
fn test_lifetimes() {
    let haystack = [1, 2, 3, 2, 1];
    let result;

    {
        let needle = [2, 3];
        result = largest_group(&haystack, &needle);
    }

    assert_eq!(result, &[2, 3, 2]);
}
```

Exercise 04: Boxes Into Boxes

```
turn-in directory:
    ex04/

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

allowed symbols:
    <[i32]>::{len, is_empty, swap}  std::{assert, assert_eq, panic}
```

You are given a list of boxes (`[width, height]`). Sort that list of boxes in a way for every box to be *contained* in the previous one. If the operation is not possible, the function must panic.

```
fn sort_boxes(boxes: &mut [[u32; 2]]);
```

Example:

```
let mut boxes = [[3, 3], [4, 3], [1, 0], [5, 7], [3, 3]];
sort_boxes(&mut boxes);
assert_eq!(boxes, [[5, 7], [4, 3], [3, 3], [3, 3], [1, 0]]);
```

Exercise 05: Deduplication

```
turn-in directory:
    ex05/

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

```
allowed symbols:
    std::vec::Vec::{remove, len}
```

Write a **function** that removes all repeated elements of a list, preserving its initial ordering.

```
fn deduplicate(list: &mut Vec<i32>);
```

Example:

```
let mut v = vec![1, 2, 2, 3, 2, 4, 3];
deduplicate(&mut v);
assert_eq!(v, [1, 2, 3, 4]);
```

Exercise 06: Big Add

```
turn-in directory:
    ex06/

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

allowed symbols:
    <[i32]>::{is_empty, len}
    std::vec::Vec::{push, len, is_empty, new, reverse}
    u8::is_ascii_digit
    std::assert
```

Write a **function** that multiplies two numbers together. The numbers are given as a list of decimal digits and may be arbitrarly large.

```
fn big_add(a: &[u8], &[u8]) -> Vec<u8>;
```

- `a` and `b` must only contain digits (`b'0'` to `b'9'` included). If anything else is found, the function must panic.
- If either `a` or `b` is empty, the function panics.
- Input numbers may contain leading zeros, but the result must not have any.

Example:

```
assert_eq!(big_add(b"2", b"4"), b"6");
assert_eq!(big_add(b"0010", b"0200"), b"210");
```

Exercise 07: Justify Yourself!

```
turn-in directory:
    ex07/
```

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

allowed dependencies:
    ftkit  unicode-width(v0.1.10)

allowed symbols:
    ftkit::ARGS  ftkit::read_line
    unicode-width::UnicodeWidthStr
    std::vec::Vec::{new, push, clear}
    <[T]>::{len, is_empty}
    std::string::String::{new, as_str}
    str::{parse, trim, is_empty, len, split_whitespace, to_string}
    std::{eprintln, print, println}
    std::result::Result::unwrap
    std::{assert, assert_eq, panic}
```

Create a **program** that takes a number of columns as an input, and tries to justify the text it is given in the standard input as best as it can to that number of columns.

- The input is separated into "paragraphs". Each "paragraph" is separated by at least two line feeds `'\n'`. The last line of each paragraph is *not* justified.
- In the final output, multiple spaces are replaced by a single one.
- In the final output, paragraphs are always separated by a single empty line.
- If a word do not fit on a single line, it gets its own line and ignores the column requirement.
- If the user provides no arguments, or too many, or if the argument is invalid, the program is allowed to panic.

Example:

```
>_ << EOF cargo run -- 20 | cat -e
Hey,         how   are
you?      Can
you hear me     screaming in your ears?



I        don't!
EOF
Hey,  how  are  you?$
Can   you   hear  me$
screaming   in  your$
ears?$
$
```

```
I don't!$
>_ << EOF cargo run -- 10
a
b
0000000000000000000000000000000000000000
c
d
EOF
a        b
0000000000000000000000000000000000000000
c d
```