Basic Programming Concepts

Performant Software Systems with Rust — Lecture 3

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

Teaching style in this course — examples and demos

Mutable and immutable variables

Variables are immutable by default — you are allowed to **bind** a value to an immutable variable only once

```rust
fn main() {
    let _immutable = 1;
    let mut mutable = 1;

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

    mutable += 1; // Okay to modify

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

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

```
error[E0384]: cannot assign twice to immutable variable `_immutable`
  --> main.rs:13:5
   |
2  |     let _immutable = 1;
   |         ---------- first assignment to `_immutable`
...
13 |     _immutable += 1;
   |     ^^^^^^^^^^^^^^^ cannot assign twice to immutable variable
   |
help: consider making this binding mutable
   |
2  |     let mut _immutable = 1;
   |         +++

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0384`.
```

Constants

- Naming convention: all upper case with underscores
- `const`: a constant value that can be completely computed at compile time
  - any code that refers to them is replaced with the constant's computed value at compile time
  - Just a convenient name for a particular value
- `static`: global variable (may only be modified with `unsafe`)
- Both constants and globals need explicit type annotation

Contants

```rust
// Globals are declared outside all other scopes
static LANGUAGE: &str = "Rust";
const THRESHOLD: i32 = 10;

fn main() {
    println!("This is {}", LANGUAGE);
    println!("The threshold is {}", THRESHOLD);

    THRESHOLD = 5; // Error! Cannot modify a `const`
    LANGUAGE = "Go"; // or a `static`
}
```

Demo: Constants and Globals

Scope and Shadowing

- Scope
  - Variable bindings are constrained to live in a **block**
  - A **block** is a collection of statements enclosed by braces `{ }`
- Shadowing
  - Okay to declare a new variable with the same name as a previous variable

Shadowing in the Guessing Game

```rust
let mut guess = String::new();

io::stdin()
    .read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse().expect("Enter a number: ");
```

Rust is a **statically typed** language — the compiler must know the types of all variables at **compile-time**

Why is Rust designed as a **statically typed** language?

Before we talk about the benefits of static types, let’s take a look at why Javascript and Python use dynamic types

Rust vs. Javascript

Rust

```rust
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Javascript

```javascript
function add(a, b) {
  return a + b;
}
```

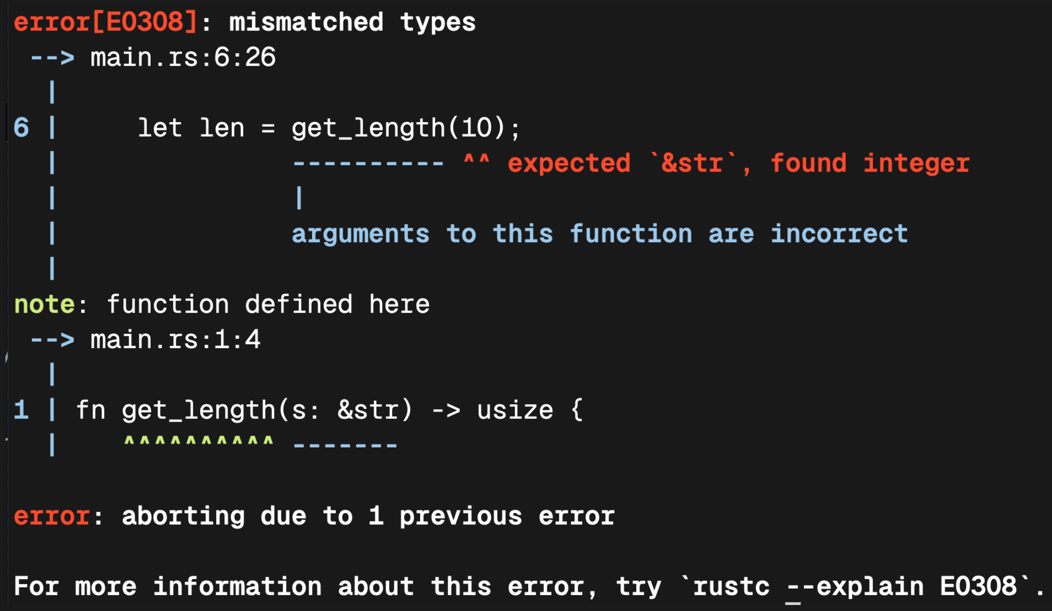
But what if we wish to add two **floating-point** numbers?

But what are the benefits of **static types**, then?

Rust vs. Python — Rust

```rust
fn get_length(s: &str) -> usize {
    s.len()
}

fn main() {
    let len = get_length(10);
}
```



```
error[E0308]: mismatched types
 --> main.rs:6:26
  |
6 |     let len = get_length(10);
  |               ---------- ^^ expected `&str`, found integer
  |               |
  |               arguments to this function are incorrect
  |
note: function defined here
 --> main.rs:1:4
  |
1 | fn get_length(s: &str) -> usize {
  |    ^^^^^^^^^^ -------

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0308`.
```

Rust vs. Python — Python

```python
def get_length(s) {
  return len(s)
}

print(get_length(10))
```

```
# Runtime error!
TypeError: object of type 'int' has no len()
```

Run-time errors $\rightarrow$ compile-time errors

But can’t run-time errors be easily caught and fixed in Python?

Rust vs. Javascript — Javascript

```javascript
function add(a, b) {
    return a + b;
}

let result = add(5, "10");
console.log("Result: " + result);
```

```
# Logical error!
510
```

Logical errors → **compile-time** errors

Static types and **Rust's strict compiler** make it much easier to catch all kinds of errors!

Scalar Data Types

Integer Types

Length	Signed	Unsigned
32-bit	`i32`	`u32`
arch-dep	`isize`	`usize`

Floating-Point Types

Length	Type
32-bit	`f32`
64-bit	`f64`

Numeric Operations

```rust
fn main() {
    let sum = 5 + 10;
    let difference = 95.5 - 4.3;
    let product = 4 * 30;
    let quotient = 56.7 / 32.2;

    // integer division truncates toward zero to the nearest integer
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

The Boolean Type

```rust
fn main() {
    let t = true; // with type inference
    let f: bool = false; // with explicit type annotation
}
```

The Character Type

```rust
fn main() {
    let c = 'z'; // with type inference
    let z: char = 'ℤ'; // with explicit type annotation
    let hugging_face = '🤗'; // emojis and CJK characters
}
```

Compound Data Types

The Tuple Type

- Groups together some values with a variety of types
- Once declared, cannot grow or shrink in size
- Useful when a function needs to return multiple values

```rust
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}

fn calculate_area_perimeter(x: i32, y: i32) -> (i32, i32) {
    // calculate the area and perimeter of rectangle
    let area = x * y;
    let perimeter = 2 * (x + y);
    (area, perimeter)
}
```

Using Pattern Matching to Destructure Tuples

```rust
fn main() {
    let tup = (500, 6.4, 1); // with type inference
    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

Accessing Elements of a Tuple

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;
    let six_point_four = x.1;
    let one = x.2;
}
```

The Unit Type: The Tuple Without Any Values

- The value and its type are both `()`
- Empty value and empty type
- Returned by expressions and functions if they do not return any other value

The Array Type

- Arrays have a fixed length
- Space for data in arrays are allocated on the stack
- Use **vectors** if you wish to grow or shrink in size

```rust
fn main() {
    let a = [1, 2, 3, 4]; // with type inference
    let a: [i32; 4] = [1, 2, 3, 4]; // with explicit type annotation
    let a = [3; 5]; // [initial value; length]
    let first_element = a[0]; // accessing an element in the array
}
```

What if you try to access an element outside the bounds of an array?

Rust will panic, but only at run-time, because the compiler can’t possibly know the value used to index the array!

Functions

- We have seen them before already
- No restrictions on the order of function definitions
- The return type is declared after `->` (the unit type `()` is the default)
- The last expression in the function is the return value

Functions

```rust
// Function that returns a boolean value
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    // Corner case, early return
    if rhs == 0 {
        return false;
    }

    // Expression as the return value
    // The `return` keyword is not necessary here
    lhs % rhs == 0
}
```

Control Flow — `if` Expressions

- Same as C but no need for parentheses
- Just like any expression, it evaluates to a value

Control Flow — `if` Expressions

```rust
fn main() {
    let n = 5;

    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }
    let big_n =
        if n < 10 && n > -10 {
            println!(", and is a small number, increase ten-fold");

            // This expression returns an `i32`
            10 * n
        } else {
            println!(", and is a big number, halve the number");

            // This expression must return an `i32` as well
            n / 2 // Try suppressing this expression with a semicolon
        }; // Don't forget to put a semicolon here

    println!("{} -> {}", n, big_n);
}
```

Repetition with Loops — `loop`

A `loop` loop can return a value with the `break` keyword

```rust
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

Repetition with Loops — `while`

A `while` loop is just like C, minus the parentheses

```rust
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number} ");

        number -= 1;
    }

    println!("Liftoff!");
}
```

Repetition with Loops — `for`

- **Concise** — typically used to iterate through a collection
- **Safer** than iterating using an index — most often used

```rust
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }

    for number in (1..4).rev() {
        println!("{number}");
    }

    println!("Liftoff!");
}
```

Required Additional Reading

The Rust Programming Language, Chapter 3