Common Programming Concepts

CS392: Rust, in Theory and in Practice

September 4, 2025 (Lecture 2)

Outline

Basics of Rust

Workshop: Programming Practice

Variables and Constants

```
let x = 2;          // immutable variable
let x : i8 = 2;     // type annotated (immutable) variable
let mut x = 2.0     // mutable variable
x = 3.0             // assignment of mutable variable
                    // (new value must be same type)
const X : i32 = 2;  // constant
```

Variable are immutable by default, and can be shadowed

Variables are written in `snake_case` by convention and constants in `SCREAMING_SNAKE_CASE`

Constants are hard-coded by the compiler so their type, size, and value must be known at compile time

Primitive Types

Rust has all the usual primitive types with all the usual operators (see RPL for more details)

Integers	`i32` is the default	`1`, `2`, `-52`
Floats	`f64` is the default	`1.0`, `2.0`, `-5.2`
Characters	`char`	`'x'`
Booleans	`bool`	`true` or `false`
Tuples	`(t1, t2,..., t_k)`	`(1, 2.3, true)` (`p.i` is `i` component accessor)
Arrays	`[ty; usize]`	`[1, 2, 3]` (`l[i]` is `i` element accessor)

Note: Arrays are not the same as **vectors** which we'll see more of later. In particular, arrays are fixed length.

Functions

```rust
fn sum_of_squares(x : u32, y : u32) -> u32 {
  let x_squared = x * x;
  let y_squared = y * y;
  x_squared + y_squared // NO SEMICOLON
}
```

Function definitions are standard. Parameters types and output type are required

The body of a function is called a **block** which consists of a sequence of ;-separated statements

The last statement (if it is an expression) is the return value of function. If no last statement is given, then it's equivalent to writing `return ()`

Control Flow

```rust
fn is_prime(n: i32) -> bool {
    for i in 2..n {
        if n % i == 0 {
            return false
        }
    }
    true
}
```

Control flow is standard, we have `for`-loops, `loop`-loops, `while`-loops, and `if`-`else`-expressions

Blocks

```rust
fn main() {
    let mut x = 2;
    assert_eq!(x, 2);
    let y = 4;
    {
        let y = 3; // this `y` only exists within
        x = y      // the block
    }
    assert_eq!(x, 3);
    assert_eq!(y, 4);
}
```

Blocks aren't all that useful in everyday programming

We'll use them *extensively* to stress-test the type system and make sure that our own implementations of the type system behave correctly

Expressions vs. Statements

```rust
// THIS DOES NOT COMPILE
fn id(x: i32) {
    let _y = x // missing semicolon
}
fn main() {
    let _ = id(2);
}
```

Rust is an *expression-based* language. There are very few proper statements:

- function declarations
- variable and constant declarations
- loops

Statements do not have values, so they cannot be used at the end of "non-returning" functions (Actually, there are no non-returning functions! Every function has a return value, but it might be a unit, kinda like OCaml)

Fancy Tricks

There's a lot of fun stuff we're glossing over:

- `if-let`-expressions
- Inclusive ranges
- Labelled loops
- (How does looping over a collection actually work?)

We're gonna ignore all this for now. If you're interested, you should definitely start looking into it, but we can get away with a minimal subset of rust for a while.

Outline

Basics of Rust

Workshop: Programming Practice

Task

1. *Practice Problem:* Write a function `is_perfect_cube` which determines if an `i32` is a perfect cube. Write it both in terms of simple control flow and in terms of type casting (this will require lookup in, say, Rust by Example). **Please work in groups of 2-3.**
2. Look over Assignment 1 and begin working on it