Rust: Syntax and Semantics

Jeff Shen

Last revised May 2, 2020

Contents

Variable Bindings

Use `let` to introduce a **variable binding**. These are immutable by default. Use `mut` to make them mutable:

```
let mut x = 0;
```

Rust is **statically typed**: specify your types up front.

```
let x: i32 = 5;
```

Bindings cannot be accessed outside of the scope they are defined in.

Bindings can be **shadowed** (overwritten):

```
let mut y: i32 = 1;
y = 2; // mutate y
let y = y; // y now immutable and bound to 2
let y = "text"; // rebind y to different type
```

Functions

Define a function with `fn`:

```
fn foo() {
    // do stuff here
}
```

Every program has a `main` function.

Functions can take arguments. The type of the argument must be declared.

Functions can return arguments. Use `->` to indicate the return, and declare the type after the arrow. The last line in the function is what is returned. Do not insert a semicolon at the end of that line.

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

Expressions and Statements

Expressions return a value, and **statements**, indicated by a semicolon, do not. Semicolons are used to turn expressions into statements (ie. suppress output).

Assignments to already-bound variables are expressions, but the value returned is `()` rather than the "expected" value. This is because the assigned value can only have one owner:

```
let mut y = 5;
let x = (y = 6); // x has value '()' rather than 6
```

Variable bindings can point to functions:

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
let f: fn(i32) -> i32 = add; // or, let f = add;
let six = f(1, 5);
```

Primitive Types

Boolean

(`bool`): `true` or `false`

char

A single Unicode value. Created with `''`.

```
let x = 'x';
let x = '1';
```

Numerics

- **Signed vs unsigned**: Signed integers support both positive and negative values, whereare unsigned integers can only store positive values. For a fixed size, an unsigned integer can store larger positive values. Signed integers are denoted by `i` (eg. `i8` for a signed eight-bit number), and unsigned by `u` (eg. `u16`).
- **Fixed vs variable size**: Fixed size types have a specific number of bits they can store. Sizes can be `8`, `16`, `32` or `64` (eg. `i32`, `u16`). Variable size types are denoted by `isize` and `usize`.
- **Floating-point**: Denoted by `f32` (single precision) and `f64` (double precision).

Arrays

An array is a fixed-size list of elements of the same type. They are immutable by default.

```
let a = [1, 2, 3];
let b = [0; 20]; // 20 elements, each with a value of 0
let a_length = a.len();
let a_first = a[0]
```

Slicing

Slicing allows a "view" into a data structure without copying. Use `&` to indicate that slices are like references.

```
let a = [0, 1, 2, 3, 4];
let complete = &a[..] // slice with all elements
let middle = &a[1..4] // slice with 1, 2, 3
```

Tuples

Tuples are ordered lists of fixed sizes. They can contain multiple types. Fields of tuples can be **destructured** using `let`:

```
let x: (i32, &str) = (1, "hello");
let (a, b) = x; // a gets 1, b gets "hello"
let (c, d) = ("test", 5);
```

Elements of a tuple can be accessed using dot notation:

```
let tup = (1, 2, 3, 4);
let x = tup.0;
let y = tup.3;
```

if

Use an `if` expression (not statement!) to conditionally run code:

```
let x = 5;

if x == 5 {
    println!("x is five")
} else if x == 6 {
    println!("x is six")
} else {
    println!("asdf")
}
```

Since `if` is an expression, it can return a value:

```
let x = 5;
let y = if 5 { 10 } else { 15 };  // y is 10
```

If there is no `else`, then the return value is `()`.

Loops

Use for loops to loop over an iterable:

```
for i in 0..10 {
    println!("{}", x);
}
```

where `0..10` gives an iterable range.

Use `.enumerate()` to keep track of how many times you have looped:

```rust
for (i, j) in (2..5).enumerate() {
    println!("{} {}", i, j)
}

// Output:
// 0 2
// 1 3
// 2 4
```

Use `while` for while loops. Keep looping while some condition holds.

```rust
let mut x = 5;
let mut done = false;

while !done {
    x += 1;
    if x % 10 == 0 {
        done = true;
    }
}
```

Use `loop` for infinite loops (instead of writing `while true`)

```rust
loop {
    println!("loop forever")
}
```

Use `break` to break out of the loop (can combine with `loop` instead of explicitly defining a `done` condition).

Use `continue` to skip to the next iteration.