Rust Basics

Mustakimur R. Khandaker

let Statements

By default, Rust variables are immutable.

- Usage checked by the compiler.

mut is used to declare a resource as mutable.

```
fn main() {
    let a: i32 = 0;
    a = a + 1;
    println!("{}", a);
}
```

Shadowing:

- declare a new variable with the same name as a previous variable, and the new variable shadows the previous variable.

```
fn main() {
    let mut a: i32 = 0;
    a = a + 1;
    println!("{}", a);
}
```

```
fn main() {
    let x = 5;
    let x: i32 = 5; //type annotation
    let mut x = 5; //mutable x: i32
    x = 10;
}
```

Functions

Function definitions in Rust start with fn and have a set of parentheses after the function name.

- The curly brackets tell the compiler where the function body begins and ends.
- Rust doesn't care where you define your functions, only that they're defined somewhere.
- Functions can return values to the code that calls them.
    - We do declare their type after an arrow (->).

```
fn main() {
    another_function(5, 6);
}

fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

```
fn main() {
    let x = plus_one(5);
    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Statements

Statements do not return values. Therefore, you can't assign a let statement to another variable, as the following code tries to do; you'll get an error.

```
fn main() {
    let x = (let y = 6);
}
```

```
error: expected expression, found statement (`let`)
  --> src/main.rs:40:14
   |
40 |     let x = (let y = 6);
   |              ^^^^^^^^^
   |
   = note: variable declaration using `let` is a statement
```

The let y = 6 statement does not return a value, so there isn't anything for x to bind to.

- In C, we can write x = y = 6 and have both x and y have the value 6.

Expression

Expressions evaluate to something and make up most of the rest of the code that you'll write in Rust.

- Calling a function is an expression.
- Calling a macro is an expression.
- The block that we use to create new scopes, {}, is an expression.

```
fn main() {
    let x = 5;
    let y = {
        let x = 3;
        x + 1
    };
    println!("The value of y is: {}", y);
}
```

Data Types

Rust is a statically typed language, which means that it must know the types of all variables at compile time.

- The compiler can usually infer what type we want to use based on the value and how we use it.
- two data type subsets: scalar and compound.

A scalar type represents a single value. Rust has four primary scalar types: integers, floating-point numbers, Booleans, and characters.

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Integer Types:

Signed numbers are stored using two's complement representation.

- Each signed variant can store numbers from $-(2^{n-1})$ to $2^{n-1} - 1$ inclusive, where n is the number of bits that variant uses.
- Unsigned variants can store numbers from 0 to 2^{n-1}.
- `isize` and `usize` types depend on the kind of computer your program is running on.

Continue ....

All number literals except the byte literal allow a type suffix, such as 57u8, and _ as a visual separator, such as 1_000.

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

Floating-Point Types:
Rust also has two primitive types for floating-point numbers.

- Rust's floating-point types are f32 and f64.
- Floating-point numbers are represented according to the IEEE-754 standard.

Boolean Type:
A Boolean type in Rust has two possible values: true and false.

Character Type:
Rust's char type is the language's most primitive alphabetic type (*four bytes* in size and represents *Unicode Scalar Value*).

- Note that char literals are specified with single quotes, as opposed to string literals, which use double quotes.

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let heart_eyed_cat = '😻';
}
```

Compound Types

Compound types can group multiple values into one type.

- Rust has two primitive compound types: tuples and arrays.

The Tuple Type:
A tuple is a general way of grouping together a number of values with a variety of types into one compound type.

- Tuples have a fixed length: once declared, they cannot grow or shrink in size.

```
fn main() {
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
    println!("The value of y is: {}", y);
}
```

```
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;
}
```

Continue ....

The Array Type:
Unlike a tuple, every element of an array must have the same type.

- Arrays in Rust have a fixed length.
- Arrays are allocated on the *stack* rather than the *heap*.

```
let months = ["January", "February", "March", "April", "May", "June", "July", "August", "September",
"October", "November", "December"];
let a: [i32; 5] = [1, 2, 3, 4, 5];
let a = [3; 5]; // let a = [3, 3, 3, 3, 3];

fn main() {
    let a = [1, 2, 3, 4, 5];
    let first = a[0];
    let second = a[1];
}
```

Alternative:
A vector is a similar collection type provided by the standard library that is allowed to grow or shrink in size.

Invalid Array Element Access

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;

    let element = a[index];

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

```
error: this operation will panic at runtime
   --> src/main.rs:101:19
    |
101 |     let element = a[index];
    |                   ^^^^^^^^ index out of bounds: the length is 5 but the index is 10
    |
    = note: `#[deny(unconditional_panic)]` on by default
```

Control Flow

```rust
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

Continue ....

The error indicates that Rust expected a bool but got an integer. Unlike languages such as Ruby and JavaScript, Rust will not automatically try to convert non-Boolean types to a Boolean.

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

    if number {
        println!("number was three");
    }
}
```

```rust
fn main() {
    let number = 3;
    if number != 0 {
        println!("number was something other than zero");
    }
}
```

```
error[E0308]: mismatched types
  --> src/main.rs:123:8
    |
123 |     if number {
    |        ^^^^^^ expected `bool`, found integer
```

Continue ....

```rust
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };
    println!("The value of number is: {}", number);
}
```

```rust
fn main() {
    let condition = true;
    let number = if condition { 5 } else { "six" };
    println!("The value of number is: {}", number);
}
```

```
error[E0308]: `if` and `else` have incompatible types
   --> src/main.rs:146:44
    |
146 |     let number = if condition { 5 } else { "six" };
    |                                 -            ^^^^^ expected integer, found `&str`
    |                                 |
    |                                 expected because of this
```

Casting

Rust provides no implicit type conversion (coercion) between primitive types.

- Explicit type conversion (casting) can be performed using the as keyword.

Rules for converting between integral types follow C conventions.

- Except in cases where C has undefined behavior.

```
fn main() {
    let decimal: f32 = 65.4321;
    //let integer: u32 = decimal;

    let integer: u32 = decimal as u32;
    println!("{} => {}", decimal, integer);

    //let character: char = integer as char;
    let short_integer: u8 = integer as u8;
    let character: char = short_integer as char;
    println!("{} => {} => {}", integer,
short_integer, character);

    let s_integer: i32 = -10;
    let integer: u32 = s_integer as u32;
    println!("{} => {}", s_integer, integer);
}
```

Aliasing

The type statement can be used to give a new name to an existing type.

- Types must have **UpperCamelCase** names, or the compiler will raise a warning.

```rust
fn main() {
    type NanoSecond = u64;
    type Inch = u64;

    let nanoseconds: NanoSecond = 5;
    let inches: Inch = 2;

    println!(
        "{} nanoseconds + {} inches = {} unit?",
        nanoseconds,
        inches,
        nanoseconds + inches
    );
}
```

Factorial

```
use std::io;

fn fact(n: i32) -> i32 {
    if n == 0 {
        1
    } else {
        let x = fact(n - 1);
        n * x
    }
}
```

```
fn main() {
    let mut input = String::new();

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

    let u_int: i32 =
input.trim().parse().expect("Please type a number!");

    let res = fact(u_int);
    println!("fact({}) = {}", u_int, res);
}
```

Loops

Rust has three kinds of loops: loop, while, and for.

The loop keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop.

```
fn main() {
    loop {
        println!("again!");
    }
}
```

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s)
in 0.29s
     Running `target/debug/loops`
again!
again!
again!
again!
^Cagain!
```

When we run this program, we'll see again! printed over and over continuously until we stop the program manually. Most terminals support a keyboard shortcut, ctrl-c, to interrupt a program that is stuck in a continual loop.

Returning Values from Loops

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

    let result = loop {
        counter += 1;

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

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

We can add the value we want returned after the break expression we use to stop the loop; that value will be returned out of the loop so we can use it.

Conditional Loops with while

This construct eliminates a lot of nesting that would be necessary if we used loop, if, else, and break, and it's clearer. While a condition holds true, the code runs; otherwise, it exits the loop.

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

```
$ cargo run
   Compiling loops v0.1.0
(file:///projects/loops)
    Finished dev [unoptimized +
debuginfo] target(s) in 0.32s
     Running `target/debug/loops`
the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50
```

Looping Through a Collection with for

The while approach in last code is error prone.

- we could cause the program to panic if the index length is incorrect.
- It's also slow, because the compiler adds runtime code to perform the conditional check on every element on every iteration through the loop.

A more concise alternative, we can use a for loop and execute some code for each item in a collection.

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

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

```
fn main() {
    for x in (1..10).step_by(2) {
        println!("{}", x);
    }
}
```

< Rust Basics />