Rust programming

Module 2: Foundations of Rust

Unit 1

Basic Syntax

Learning objectives

- Understand basic Rust syntax

Meeting Rust

A new project

```
$ cargo new hello-world
```

```
$ cd hello-world
$ cargo run
```

```
Compiling hello-world v0.1.0 (/home/teach-rs/Projects/hello-world)
Finished dev [unoptimized + debuginfo] target(s) in 0.74s
Running `target/debug/hello-world`
Hello, world!
```

Hello, world!

```rust
fn main() {
    println!("Hello, world! fib(6) = {}", fib(6));
}

fn fib(n: u64) -> u64 {
    if n <= 1 {
        n
    } else {
        fib(n - 1) + fib(n - 2)
    }
}
```

```
Compiling hello-world v0.1.0 (/home/teach-rs/Projects/hello-world)
Finished dev [unoptimized + debuginfo] target(s) in 0.28s
Running `target/debug/hello-world`
Hello, world! fib(6) = 8
```

Basic Syntax

Variables

```rust
fn main() {
    let some_x = 5;
    println!("some_x = {}", some_x);
    some_x = 6;
    println!("some_x = {}", some_x);
}
```

```
Compiling hello-world v0.1.0 (/home/teach-rs/Projects/hello-world)
error[E0384]: cannot assign twice to immutable variable `some_x`
--> src/main.rs:4:5
  |
2 |     let some_x = 5;
  |         ------
  |         |
  |         first assignment to `some_x`
  |         help: consider making this binding mutable: `mut some_x`
3 |     println!("some_x = {}", some_x);
4 |     some_x = 6;
  |     ^^^^^^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `hello-world` due to previous error
```

Variables

```rust
fn main() {
    let mut some_x = 5;
    println!("some_x = {}", some_x);
    some_x = 6;
    println!("some_x = {}", some_x);
}
```

```
Compiling hello-world v0.1.0 (/home/teach-rs/Projects/hello-world)
Finished dev [unoptimized + debuginfo] target(s) in 0.26s
Running `target/debug/hello-world`
some_x = 5
some_x = 6
```

Assigning a type to a variable

```rust
fn main() {
    let x: i32 = 20;
    //     ^^^^^  Type annotation
}
```

- Rust is strongly and strictly typed
- Variables use type inference, so no need to specify a type
- We can be explicit in our types (and sometimes have to be)

Integers

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

- Rust prefers explicit integer sizes
- Use `isize` and `usize` sparingly

Literals

```rust
fn main() {
    let x = 42; // decimal as i32
    let y = 42u64; // decimal as u64
    let z = 42_000; // underscore separator

    let u = 0xff; // hexadecimal
    let v = 0o77; // octal
    let w = 0b0100_1101; // binary
    let q = b'A'; // byte syntax (stored as u8)
}
```

Floating points and floating point literals

```
fn main() {
    let x = 2.0; // f64
    let y = 1.0f32; // f32
}
```

- `f32`: single precision (32-bit) floating point number
- `f64`: double precision (64-bit) floating point number
- `f128`: 128-bit floating point number

Numerical operations

```rust
fn main() {
    let sum = 5 + 10;
    let difference = 10 - 3;
    let mult = 2 * 8;
    let div = 2.4 / 3.5;
    let int_div = 10 / 3; // 3
    let remainder = 20 % 3;
}
```

- These expressions do overflow/underflow checking in debug
- In release builds these expressions are wrapping, for efficiency
- You cannot mix and match types here, not even between different integer types

```rust
fn main() {
    let invalid_div = 2.4 / 5;          // Error!
    let invalid_add = 20u32 + 40u64;    // Error!
}
```

Booleans and boolean operations

```rust
fn main() {
    let yes: bool = true;
    let no: bool = false;
    let not = !no;
    let and = yes && no;
    let or = yes || no;
    let xor = yes ^ no;
}
```

Comparison operators

```rust
fn main() {
    let x = 10;
    let y = 20;
    x < y; // true
    x > y; // false
    x <= y; // true
    x >= y; // false
    x == y; // false
    x != y; // true
}
```

Note: as with numerical operators, you cannot compare different integer and float types with each other

```rust
fn main() {
    3.0 < 20; // invalid
    30u64 > 20i32; // invalid
}
```

Characters

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

- A `char` is a 32-bit unicode scalar value
- Very much unlike C/C++ where `char is 8 bits

`String`s

```rust

        let s1 = String::from("Hello, 🌍!");
        //       ^^^^^^ Owned, heap-allocated string
```

- Rust `String`s are UTF-8-encoded
- Unlike C/C++: *Not null-terminated*
- Cannot be indexed like C strings
- `String` is heap-allocated
- Actually many types of strings in Rust
  - `CString`
  - `PathBuf`
  - `OsString`
  - ...

Tuples

```rust
fn main() {
    let tup: (i32, f32, char) = (1, 2.0, 'a');
}
```

- Group multiple values into a single compound type
- Fixed size
- Different types per element
- Create by writing a comma-separated list of values inside parentheses

```rust
fn main() {
    let tup = (1, 2.0, 'Z');
    let (a, b, c) = tup;
    println!("({}, {}, {})", a, b, c);

    let another_tuple = (true, 42);
    println!("{}", another_tuple.1);
}
```

- Tuples can be destructured to get to their individual values
- You can also access individual elements using the period operator followed by a zero based index

Arrays

```rust
fn main() {
    let arr: [i32; 3] = [1, 2, 3];
    println!("{}", arr[0]);
    let [a, b, c] = arr;
    println!("[{}, {}, {}]", a, b, c);
}
```

- Also a collection of multiple values, but this time all of the same type
- Always a fixed length at compile time (similar to tuples)
- Use square brackets to access an individual value
- Destructuring as with tuples
- Rust always checks array bounds when accessing a value in an array

Control flow

```rust
fn main() {
    let mut x = 0;
    loop {
        if x < 5 {
            println!("x: {}", x);
            x += 1;
        } else {
            break;
        }
    }

    let mut y = 5;
    while y > 0 {
        y -= 1;
        println!("y: {}", x);
    }

    for i in [1, 2, 3, 4, 5] {
        println!("i: {}", i);
    }
}
```

Functions

```rust
fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn returns_nothing() -> () {
    println!("Nothing to report");
}

fn also_returns_nothing() {
    println!("Nothing to report");
}
```

- The function boundary must always be explicitly annotated with types
- Type inference may be used in function body
- A function that returns nothing has the return type *unit* (`()`)
- Function body contains a series of statements optionally ending with an expression

Statements

- Statements are instructions that perform some action and do not return a value
- A definition of any kind (function definition etc.)
- The `` `let var = expr;` `` statement
- Almost everything else is an expression

Example statements

```rust
fn my_fun() {
    println!("{}", 5);
}
```

```rust
let x = 10;
```

```rust
return 42;
```

```rust
let x = (let y = 10); // invalid
```

Expressions

- Expressions evaluate to a resulting value
- Expressions make up most of the Rust code you write
- Includes all control flow such as `if` and `loop`
- Includes scoping braces (`{` and `}`)
- Semicolon (`;`) turns expression into statement

```rust
fn main() {
    let y = {
        let x = 3;
        x + 1
    };
    println!("{}", y); // 4
}
```

Expressions - control flow

- Control flow expressions as a statement do not need to end with a semicolon if they return *unit* (`()`)
- Remember: A block/function can end with an expression, but it needs to have the correct type

```
fn main() {
    let y = 11;
    // if as an expression
    let x = if y < 10 {
        42
    } else {
        24
    };

    // if as a statement
    if x == 42 {
        println!("Foo");
    } else {
        println!("Bar");
    }
}
```

Scope

- We just mentioned the scope braces (`{` and `}`)
- Variable scopes are actually very important for how Rust works

```rust
fn main() {
    println!("Hello, {}", name);  // invalid: name is not yet defined
    let name = "world";  // from this point name is in scope
    println!("Hello, {}", name);
} // name goes out of scope
```

Scope

As soon as a scope ends, all variables for that scope can be removed from the stack

```
fn main() { // nothing in scope here
    let i = 10; // i is now in scope
    if i > 5 {
        let j = 20; // j is now also in scope
        println!("i = {}, j = {}", i, j);
    } // j is no longer in scope, i still remains
    println!("i = {}", i);
} // i is no longer in scope
```

Summary