

Rust programming

Module 2: Foundations of Rust

Unit 1

Basic Syntax

Learning objectives

- Understand basic Rust syntax

Meeting Rust

A new project

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

```
1  $ cd hello-world
```

```
2  $ cargo run
```

```
1  Compiling hello-world v0.1.0 (/home/teach-rs/Projects/hello-world)
```

```
2  Finished dev [unoptimized + debuginfo] target(s) in 0.74s
```

```
3  Running `target/debug/hello-world`
```

```
4  Hello, world!
```

Hello, world!

```
1  fn main() {  
2      println!("Hello, world! fib(6) = {}", fib(6));  
3  }  
4  
5  fn fib(n: u64) -> u64 {  
6      if n <= 1 {  
7          n  
8      } else {  
9          fib(n - 1) + fib(n - 2)  
10     }  
11 }
```

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

Basic Syntax

Variables

```
1 fn main() {  
2     let some_x = 5;  
3     println!("some_x = {}", some_x);  
4     some_x = 6;  
5     println!("some_x = {}", some_x);  
6 }
```

```
1 Compiling hello-world v0.1.0 (/home/teach-rs/Projects/hello-world)  
2 error[E0384]: cannot assign twice to immutable variable `some_x`  
3 --> src/main.rs:4:5  
4 |  
5 2 |     let some_x = 5;  
6 |         -----  
7 |         |  
8 |         first assignment to `some_x`  
9 |         help: consider making this binding mutable: `mut some_x`  
10 3 |     println!("some_x = {}", some_x);  
11 4 |     some_x = 6;  
12 |     ^^^^^^^^^^^ cannot assign twice to immutable variable  
13  
14 For more information about this error, try `rustc --explain E0384`.  
15 error: could not compile `hello-world` due to previous error
```

Variables

```
1 fn main() {  
2     let mut some_x = 5;  
3     println!("some_x = {}", some_x);  
4     some_x = 6;  
5     println!("some_x = {}", some_x);  
6 }
```

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


Assigning a type to a variable

```
1 fn main() {  
2     let x: i32 = 20;  
3     //    ^^^^^ Type annotation  
4 }
```

- 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	<code>`i8`</code>	<code>`u8`</code>
16 bits	<code>`i16`</code>	<code>`u16`</code>
32 bits	<code>`i32`</code>	<code>`u32`</code>
64 bits	<code>`i64`</code>	<code>`u64`</code>
128 bits	<code>`i128`</code>	<code>`u128`</code>
pointer-sized	<code>`isize`</code>	<code>`usize`</code>

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

Literals

```
1  fn main() {  
2      let x = 42; // decimal as i32  
3      let y = 42u64; // decimal as u64  
4      let z = 42_000; // underscore separator  
5  
6      let u = 0xff; // hexadecimal  
7      let v = 0o77; // octal  
8      let w = 0b0100_1101; // binary  
9      let q = b'A'; // byte syntax (stored as u8)  
10 }
```

Floating points and floating point literals

```
1  fn main() {  
2      let x = 2.0; // f64  
3      let y = 1.0f32; // f32  
4  }
```

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

Numerical operations

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

- 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

```
1  fn main() {  
2      let invalid_div = 2.4 / 5;           // Error!  
3      let invalid_add = 20u32 + 40u64;     // Error!  
4  }
```

Booleans and boolean operations

```
1  fn main() {  
2      let yes: bool = true;  
3      let no: bool = false;  
4      let not = !no;  
5      let and = yes && no;  
6      let or = yes || no;  
7      let xor = yes ^ no;  
8  }
```

Comparison operators

```
1  fn main() {  
2      let x = 10;  
3      let y = 20;  
4      x < y; // true  
5      x > y; // false  
6      x <= y; // true  
7      x >= y; // false  
8      x == y; // false  
9      x != y; // true  
10 }
```

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

```
1  fn main() {  
2      3.0 < 20; // invalid  
3      30u64 > 20i32; // invalid  
4  }
```

Characters

```
1  fn main() {  
2      let c: char = 'z';  
3      let z = 'Z';  
4      let heart_eyed_cat = '😻';  
5  }
```

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

`String`s

```
1
2     let s1 = String::from("Hello, 🌍!");
3     //      ^^^^^^ 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

```
1 fn main() {  
2     let tup: (i32, f32, char) = (1, 2.0, 'a');  
3 }
```

- 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

```
1 fn main() {  
2     let tup = (1, 2.0, 'Z');  
3     let (a, b, c) = tup;  
4     println!("{}", a, b, c);  
5  
6     let another_tuple = (true, 42);  
7     println!("{}", another_tuple.1);  
8 }
```

- 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

```
1  fn main() {  
2      let arr: [i32; 3] = [1, 2, 3];  
3      println!("{}", arr[0]);  
4      let [a, b, c] = arr;  
5      println!("[{}, {}, {}]", a, b, c);  
6  }
```

- 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

```
1  fn main() {  
2      let mut x = 0;  
3      loop {  
4          if x < 5 {  
5              println!("x: {}", x);  
6              x += 1;  
7          } else {  
8              break;  
9          }  
10     }  
11  
12     let mut y = 5;  
13     while y > 0 {  
14         y -= 1;  
15         println!("y: {}", x);  
16     }  
17  
18     for i in [1, 2, 3, 4, 5] {  
19         println!("i: {}", i);  
20     }  
21 }
```

Functions

```
1  fn add(a: i32, b: i32) -> i32 {  
2      a + b  
3  }  
4  
5  fn returns_nothing() -> () {  
6      println!("Nothing to report");  
7  }  
8  
9  fn also_returns_nothing() {  
10     println!("Nothing to report");  
11 }
```

- 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

```
1  fn my_fun() {  
2      println!("{}", 5);  
3  }
```

```
1  let x = 10;
```

```
1  return 42;
```

```
1  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

```
1 fn main() {  
2     let y = {  
3         let x = 3;  
4         x + 1  
5     };  
6     println!("{}", y); // 4  
7 }
```

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

```
1  fn main() {
2      let y = 11;
3      // if as an expression
4      let x = if y < 10 {
5          42
6      } else {
7          24
8      };
9
10     // if as a statement
11     if x == 42 {
12         println!("Foo");
13     } else {
14         println!("Bar");
15     }
16 }
```

Scope

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

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


Scope

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

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

Summary