Rust’s syntax

Declaring variables: *let*

```
let x: i32 = 2;  // variables have a type
// types can be omitted if they can be deduced from the context
let x = 2;  // this is an integer, and by default they are signed and 32 bits
```

Example of Rust's standard types

Integers:

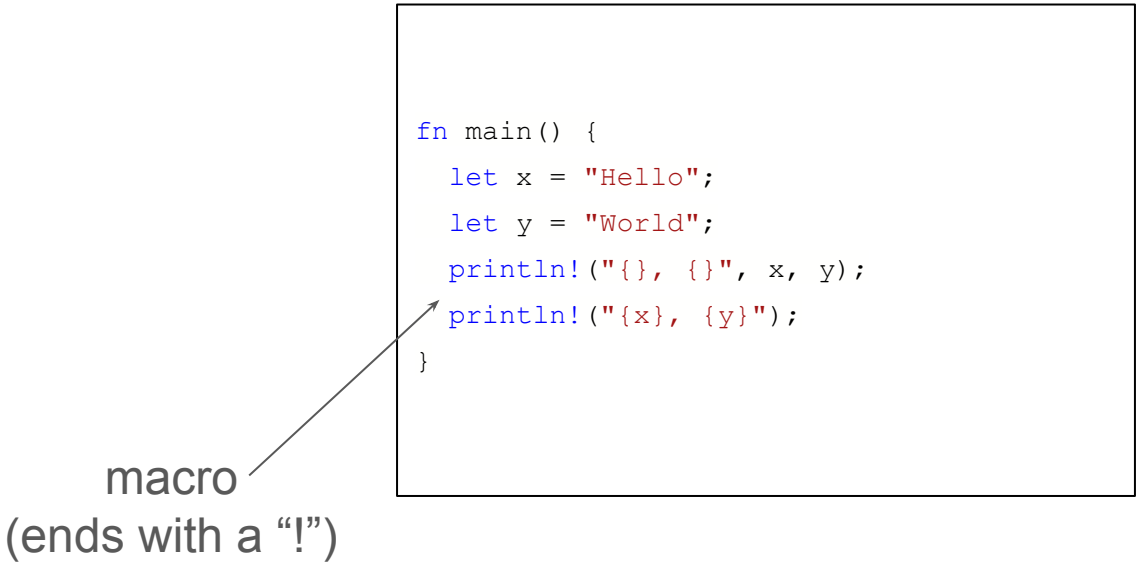
- u8: unsigned integer, 8 bits (i.e. a byte)
- i32: signed integer, 32 bits
- i64: signed integer, 64 bits
- u64: unsigned integer, 64 bits
- usize: e.g. 32 on 32 bits machine, 64 in 64 bits machine

String

Vec: a vector of elements that all have the same type

bool: true or false

Printing variables



Variables are immutable by default

```rust
fn main() {
  let x = 5;
  x += 1;
  println!("{x}");
}
```

```
└─ cargo run
   Compiling hello v0.1.0 (/home/lrobidou/hello)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:3:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
3 |     x += 1;
  |     ^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut x = 5;
  |         +++

For more information about this error, try `rustc --explain E0384`.
error: could not compile `hello` (bin "hello") due to 1 previous error
```

Expressions and statements

```
let x = 5;

let x = {
    let tmp = 4;
    tmp + 1 // no ";" at the end
};
```

Getting user input via standard input

```rust
// Create a string variable
let mut input: String = String::new();

std::io::stdin() // Get the standard input stream
    .read_line(&mut input) // Reads data until it reaches a '\n' character
    .expect("Unable to read Stdin"); // Panics on failure

println!("You entered: {}", input);
```

Rust’s syntax
-> control flow

Control flow - if

```
if /* test */ {
    //
} else if /* test */ {
    //
} else {
    //
}
```

```
let number_of_cat: u32 = 2;

if number_of_cat == 0 {
    println!("You have no cat :(");
} else if number_of_cat <= 3 {
    println!("You have some cats!");
} else {
    println!("Oo");
}
```

Control flow - if

```rust
fn main() {
  let y = 5;

  let mut x = 0;
  if y > 3 {
      x = 1;
  } else {
      x = 2;
  }

  println!("{x}");
}
```

equivalent

```rust
fn main() {
  let y = 5;

  let x = if y > 3 {
      1
  } else {
      2
  };

  println!("{x}");
}
```

equivalent

```rust
fn main() {
  let y = 5;

  let x = if y > 3 { 1 } else { 2 };

  println!("{x}");
}
```

Control flow - match

```
match /* expression */ {
    /* expression */ => {/* */}
    /* expression */ => {/* */}
    _ => {/* */} // if not exhaustive

}
```

```
let number_of_cat: u32 = 2;

match number_of_cat {
    0 => {
        println!("You have no cat :(");
    }
    ..=3 => {
        println!("You have some cats!");
    }
    _ => {
        println!("Oo");
    }
}
```

Control flow - match

```
let greeting = match age {
    0..2 => "",
    3..20 => "Suh",
    20..30 => "whatsup",
    30..40 => "what's up",
    _ => "I am most pleased to meet you",
};
```

(Note that it is possible to not use a block in the match)

loops

```rust
let mut count = 0;
// Infinite loop
loop {
    count += 1;

    if count == 5 {
        println!("OK, that's enough");

        // Exit this loop
        break;
    }
}
```

```rust
'outer: loop {
    println!("Entered the outer loop");

    'inner: loop {
        println!("Entered the inner loop");

        // This would break only the inner loop
        //break;

        // This breaks the outer loop
        break 'outer;
    }

    println!("This point will never be reached");
}
```

for loops

exclusive

```
fn main() {
  for i in 0..10 {
      println!("The value is: {i}");
  }
}
```

inclusive

```
fn main() {
   for i in 0..=10 {
       println!("The value is: {i}");
   }
}
```

reverse

```
fn main() {
  for i in (0..10).rev() {
      println!("The value is: {i}");
  }
}
```

custom step

```
fn main() {
  for i in (0..=10).step_by(2) {
      println!("The value is: {i}");
  }
}
```

while loops

```
// A counter variable
let mut n = 1;

// Loop while `n` is less than 101
while n < 101 {
    if n % 15 == 0 {
        println!("fizzbuzz");
    } else if n % 3 == 0 {
        println!("fizz");
    } else if n % 5 == 0 {
        println!("buzz");
    } else {
        println!("{}", n);
    }

    // Increment counter
    n += 1;
}
```

Declaring your own struct

```
struct /* name */ {
   /* field */ : /* type */,
   /* field */ : /* type */,
   /* field */ : /* type */,
}
```

```
struct Person {
   first_name: String,
   last_name: String,
   date_birth: u32,
}
```

Declaring your own enum

```
enum /* name */ {
   /* variant*/,
   /* variant*/,
   /* variant*/,
}
```

```
enum Animal {
   Cat,
   Dog,
   Duck
}
```

Matching on enum

```
enum Animal {
    Cat,
    Dog,
    Duck,
}

fn main() {
    let my_pet = Animal::Cat;
    match my_pet {
        Animal::Cat => println!("miaou"),
        Animal::Dog => println!("ouaf"),
        Animal::Duck => println!("coin coin"),
    }
}
```

Rust’s syntax
-> functions

Functions

(order of functions does not matter)

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


fn main() {
  let x = plus_one(5);

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

(return can be omitted)

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


fn main() {
  let x = plus_one(5);

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

Functions

```rust
fn plus_one(x: u8) -> u8 {
    x + 1
}

fn apply_function(x: u8, function: fn(u8) -> u8) -> u8 {
    function(x)
}

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

functions have a type…

… and can be used as e.g. arguments

Functions

```
fn print_string(s: String) {
  println!("{}", s);
}

fn main() {
  let my_string = String::from("Hello");
  print_string(my_string);
}
```

Functions

```
fn print_string(s: String) {
  println!("{}", s);
}

fn main() {
  let my_string = String::from("Hello");
  print_string(my_string);
}
```

OK, what if I want to call the function again?

Functions

```rust
fn print_string(s: String) {
  println!("{}", s);
}

fn main() {
  let my_string = String::from("Hello");
  print_string(my_string);
  print_string(my_string);
}
```

Wait… What!?

```
error[E0382]: use of moved value: `my_string`
 --> src/main.rs:8:18
  |
6 |     let my_string = String::from("Hello");
  |         --------- move occurs because `my_string` has type `String`, which does not implement the `Copy` trait
7 |     print_string(my_string);
  |                  --------- value moved here
8 |     print_string(my_string);
  |                  ^^^^^^^^^ value used here after move
```

Take home message

- Rust’s syntax looks a lot like C
- Blocks can evaluate to values
- But there seems to be something weird about functions…