

# Rust programming

Module A1: Language basics

# In this module

An introduction to the Rust language and basic concepts

# Learning objectives

- Get acquainted with Rust and its goals
- Introduction to ownership model
- Learn basic syntax and operators

Anyone have experience with  
Rust?

# Module A1

Language basics

# Content overview

- Why learn Rust?
- Basic Rust syntax (quickly)
- The ownership model

# Why learn Rust?

by Florian Gilcher

Founder of Ferrous Systems

- Rust training and evangelisation
- Company support
- Tooling
- Ferrocene: Rust in automotive

# Why learn Rust?

<https://www.youtube.com/embed/l8Qk5Nh6qsg>





# Basic Syntax

# A new project

```
1 $ cargo init hello-world
```

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

```
2 $ cargo run
```

```
Compiling hello-world v0.1.0 (/home/101-rs/Projects/hello-world)
Finished dev [unoptimized + debuginfo] target(s) in 0.74s
Running `target/debug/hello-world`
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 }
```

```
Compiling hello-world v0.1.0 (/home/101-rs/Projects/hello-world)
Finished dev [unoptimized + debuginfo] target(s) in 0.28s
Running `target/debug/hello-world`
Hello, world! fib(6) = 8
```

# 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 }
```

Compiling hello-world v0.1.0 (/home/101-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;
|     ^^^^^^^^^^^ c annot 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

```
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 }
```

Compiling hello-world v0.1.0 (/home/101-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

```
1 fn main() {  
2     let x: i32 = 20;  
3 }
```

- 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



# 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 }
```

# 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 = 'z';  
3     let z = 'z';  
4     let heart_eyed_cat = '🐱';  
5 }
```

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

# Strings

```
1 // Owned, heap-allocated string *slice*
2 let s1: String = String::from("Hello, \u0021");
```

- Rust strings are UTF-8-encoded
- Unlike C/C++: *Not null-terminated*
- Cannot be indexed like C Strings
- Actually many types of strings in Rust

# 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 a tuple 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
- Within the function body type inference may be used
- A function that returns nothing has the return type `unit`(`)```
- The function body contains a series of statements optionally ending with an expression

# Statements

- Expressions evaluate to a resulting value
- 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 let x = (let y = 10); // invalid
```

# Expressions

- Expressions make up most of the Rust code you write
- Includes all control flow such as `if` and `while`
- Includes scoping braces (`{` and `}`)
- An expression can be turned into a statement by adding a semicolon (`;`)

```
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 }
```

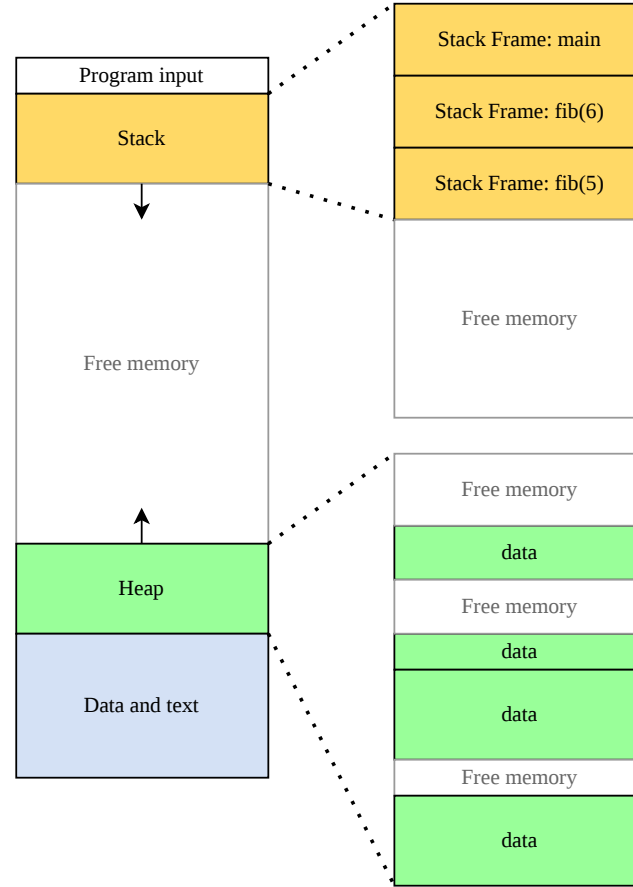
# Scope

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 }
```

# Memory management

- Most of what we have seen so far is stack-based and small in size
- All these primitive types are `Copy`: create a copy on the stack every time we need them somewhere else
- We don't want to pass a copy all the time
- Large data that we do not want to copy
- Modifying original data
- What about datastructures with a variable size?



# Rust's ownership model

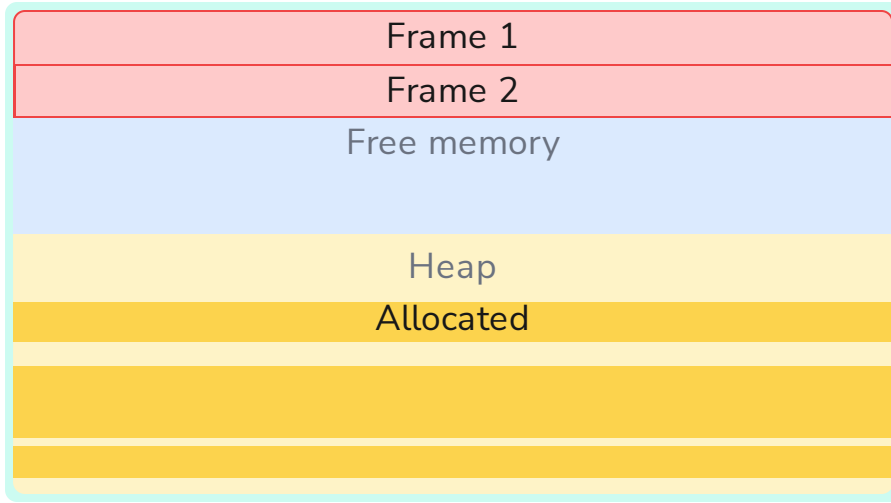


# Memory

- A computer program consists of a set of instructions
- Those instructions manipulate some memory
- How does a program know what memory can be used?

# Fundamentals

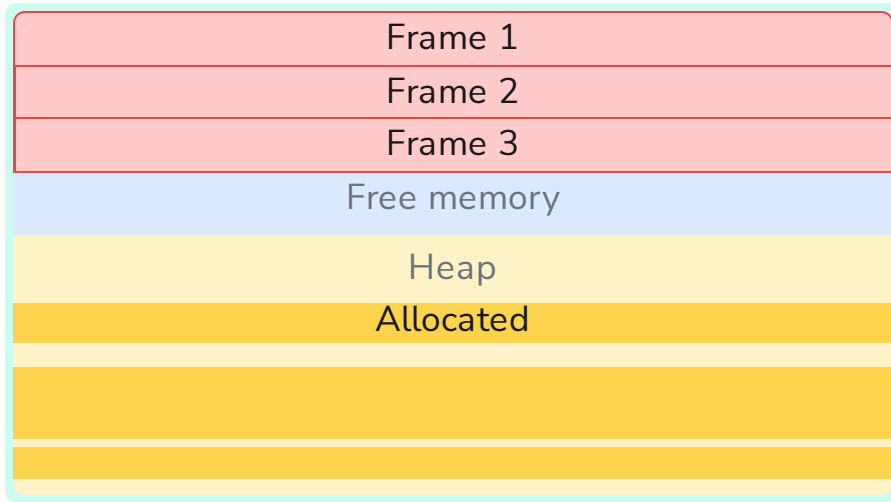
There are two mechanisms at play here, generally known as the stack and the heap



□ Stack pointer

# Fundamentals

There are two mechanisms at play here, generally known as the stack and the heap

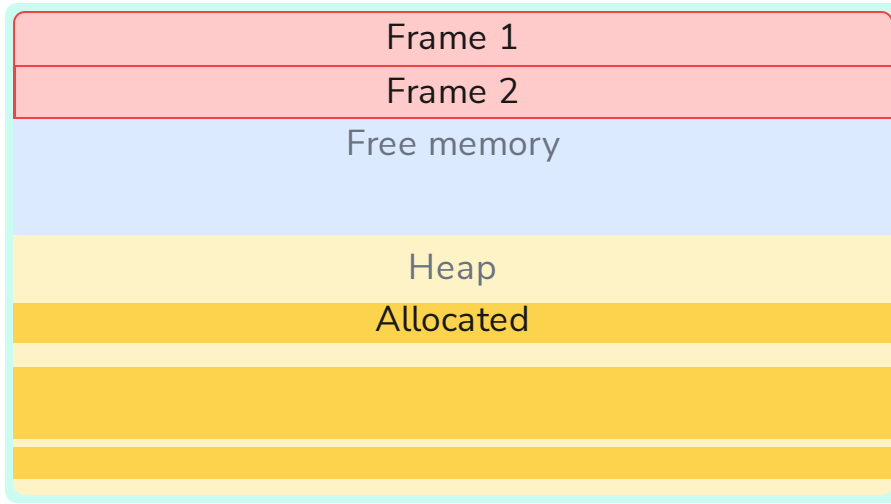


## □ Stack pointer

A stack frame is allocated for every function call. It contains exactly enough space for all local variables, arguments and stores where the previous stack frame starts.

# Fundamentals

There are two mechanisms at play here, generally known as the stack and the heap



□ Stack pointer

Once a function call ends we just move back up, and everything below is available as free memory once more.

# Stack limitations

The stack has limitations though, because it only grows as a result of a function call.

- Size of items on stack frame must be known at compile time
- If I don't know the size of a variable upfront: What size should my stack frame be?
- How can I handle arbitrary user input efficiently?

# The Heap

If the lifetime of some data needs to outlive a certain scope, it can not be placed on the stack. We need another construct: the heap.

It's all in the name, the heap is just one big pile of memory for you to store stuff in. But what part of the heap is in use? What part is available?

- Data comes in all shapes and sizes
- When a new piece of data comes in we need to find a place in the heap that still has a large enough chunk of data available
- When is a piece of heap memory no longer needed?
- Where does it start? Where does it end?
- When can we start using it?

# Variable scoping (recap)

```
1 fn main() { // nothing in scope here
2     let i = 10; // i is now in scope
3     if i > 5 {
4         let j = i; // 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` and `j` are examples containing a `Copy` type
- What if copying is too expensive?





# Ownership

- There is always ever only one owner of a stack value
- Once the owner goes out of scope (and is removed from the stack), any associated values on the heap will be cleaned up as well
- Rust transfers ownership for non-copy types: *move semantics*





# Moving out of a function

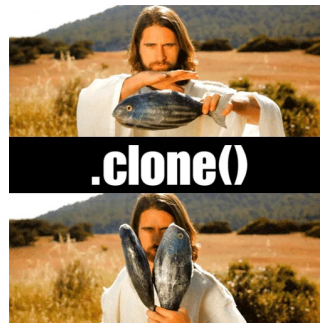
We can return a value to move it out of the function

```
1 fn main() {
2     let s1 = String::from("hello");
3     let (len, s1) = calculate_length(s1);
4     println!("The length of '{}' is {}.", s1, len);
5 }
6
7 fn calculate_length(s: String) -> (usize, String) {
8     (s.len(), s)
9 }
```

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 5.42s
Running `target/debug/playground`
The length of 'hello' is 5.
```

# Clone

- Many types in Rust are `Clone`-able
- Use can use `clone` to create an explicit clone (in contrast to `Copy` which creates an implicit copy).
- Creating a clone can be expensive and could take a long time, so be careful
- Not very efficient if a clone is short-lived like in this example



```
1 fn main() {
2     let x = String::from("hellothisisaverylongstring...");
3     let len = get_length(x.clone());
4     println!("{}", x, len);
5 }
6
7 fn get_length(arg: String) -> usize {
8     arg.len()
9 }
```

# Summary

- Loads of syntax
- Values are owned by variables
- Values may be moved to new owners or copied
- Some type may be explicitly `Clone`d

# Practicalities

- Follow installation instructions: <https://101-rs.tweede.golf>
- Exercises A1 during tutorial
- Help each other out!

