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 has experience with Rust?

Module A1

Language basics

Content overview

- Basic Rust syntax (quickly)
- The ownership model

Why learn Rust?

How to choose a language

What characteristics do you want?

1. Efficiency
2. Safety
3. Elegance
4. Practical relevance

Most languages tick two of these boxes, if you are lucky you get three.

What Rust promises

1. Pedal to the metal
2. Comes with a warranty
3. Beautiful code
4. Rust is practical

Pedal to the metal

- Compiled language, not interpreted
- State-of-the-art code generation using LLVM
- No garbage collector getting in the way of execution
- Usable in embedded devices, operating systems and demanding websites

Rust comes with a warranty

- Strong type system helps prevent silly bugs
- Explicit errors instead of exceptions
- Type system tracks lifetime of objects
  - No more *"null pointer exception"*
- Programs don't trash your system accidentally
  - Warranty *can* be voided (`unsafe`)

"If it compiles, it is more often correct."

Rust code is elegant

- Data types can capture many problem domains
- Orthogonal, expression-oriented language
- Combine declarative and imperative paradigms
- Concise syntax instead of boilerplate
- Toolchain that suggests improvements to your code

Rust is practical

- Can interface with legacy C code
- Supported on many platforms
- Active user base maintains a healthy ecosystem
- Adoption by Microsoft, Amazon, Google, ...

Why should *you* learn Rust?

- Learning a new language teaches you new tricks
  - You will also write better C/C++ code
- Rust is a young, but quickly growing platform
  - You can help shape its future
  - Demand for Rust programmers will increase!

Basic Syntax

A new project

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

```
$ cd hello-world
$ 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!

```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/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

```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/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;
    |     ^^^^^^^^^^ 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/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

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

- 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

Note

Use `isize` and `usize` sparingly

Literals

```
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

Numerical operations

```
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

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

Booleans and boolean operations

```
fn main() {
    let a: bool = true;
    let b = false;
    let c = !b;          // Not
    let d = a && b;      // And
    let e = a || b;      // Or
}
```

Comparison operators

```
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

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

Characters

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

- 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::new("Hello, □!");
```

- 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

```
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 a tuple by writing a comma-separated list of values inside parentheses

```
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

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

```
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: {y}");
    }

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

Functions

```
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
- 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

- 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

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

```
let x = 10;
```

```
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 `while`
- Includes scoping braces (`{` and `}`)
- An expression can be turned into a statement by adding a semicolon (`;`)

```
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
- Hence, the `else` branch is required

```
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

```
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
```

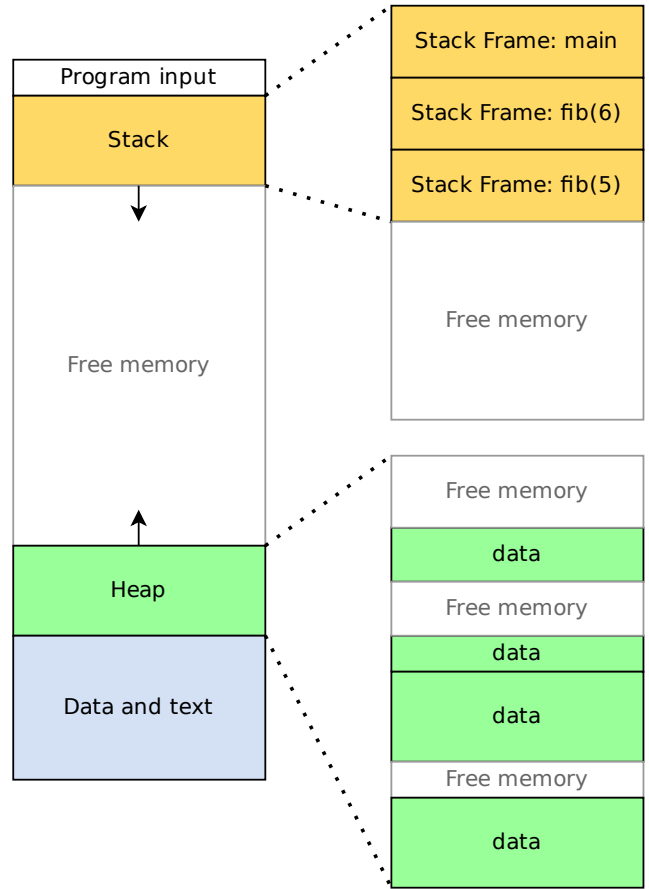
Scope

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 = {i}, j = {j}");
    } // j is no longer in scope, i still remains
    println!("i = {}", i);
} // i is no longer in scope
```

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 data structures with a variable size?



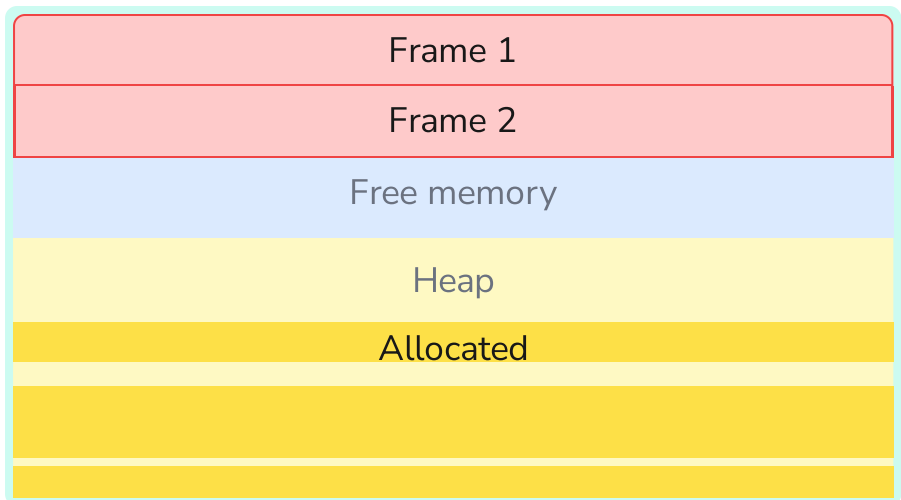
Rust’s ownership model

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

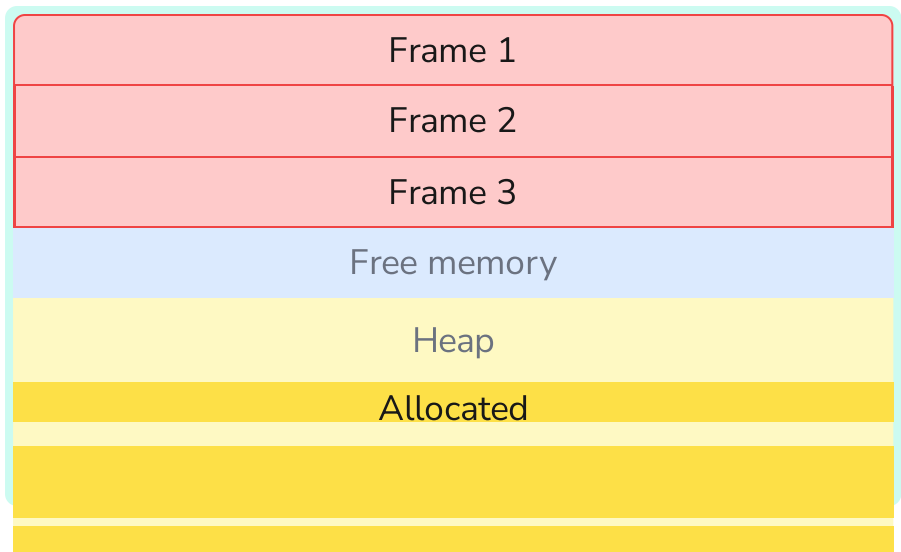
There are two mechanisms at play here, generally known as the stack and the heap



 Stack pointer

Fundamentals

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

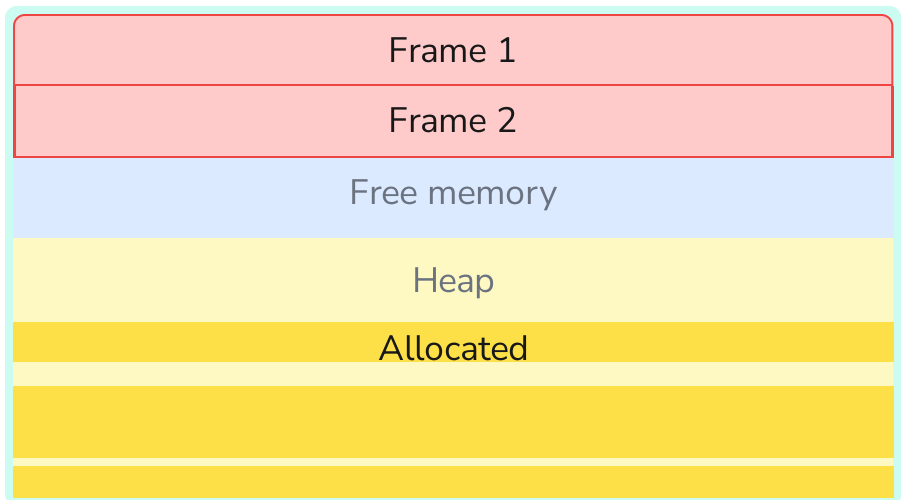


 Stack pointer

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 up front: 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)

```
fn main() { // nothing in scope here
    let i = 10; // i is now in scope
    if i > 5 {
        let j = i; // 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
```

- `i` and `j` are examples containing a `Copy` type
- What if copying is too expensive?

Rust’s ownership model

Ownership

```
let x = 5;
let y = x;
println!("{x}");
```

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in
4.00s
Running `target/debug/playground`
5
```

```
// Create an owned, allocated string
let s1 = String::from("hello");
let s2 = s1;
println!("{s1}, world!");
```

Copying large strings all over the place could become expensive! `String` does not implement `Copy`.

```
Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `s1`
--> src/main.rs:4:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type
`String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{s1}, world!");
  |                ^^ value borrowed here after move
```

Ownership

- There is always ever only one owner of a value
- Once the owner goes out of scope, any associated values will be cleaned up as well
- Rust transfers ownership for non-`Copy` types: *move semantics*

Ownership: moving into a function

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(s1);
    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: String) -> usize {
    s.len()
}
```

```
Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `s1`
--> src/main.rs:4:43
  |
2 | let s1 = String::from("hello");
  |     -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 | let len = calculate_length(s1);
  |                            -- value moved here
4 | println!("The length of '{s1}' is {len}.");
  |                           ^^ value borrowed here after move
```

Ownership: moving out of a function

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

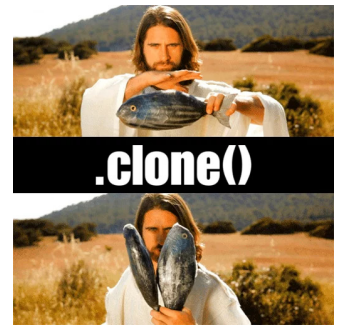
```
fn main() {
    let s1 = String::from("hello");
    let (len, s1) = calculate_length(s1);
    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: String) -> (usize, String) {
    (s.len(), s)
}
```

```
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




```
fn main() {
    let x = String::from("hellothisisaverylongstring...");
    let len = calculate_length(x.clone());
    println!("{x}: {len}");
}

fn calculate_length(arg: String) -> usize {
    arg.len()
}
```

Summary

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

Practicalities

- Follow instructions for A1 exercises: https://101.rustiec.be
  - Use `rust101` and `Rust101!` as username and password.
- Help each other out!

Cheat sheet for A1

https://101.rustiec.be

Functions:

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

Loops and control flow:

```
loop {}
for i in 2..15 { println!("{i}"); }
while foo < bar { foo += 3; }
if foo < bar { println!("hello"); } else { println!("world"); }
```

Ownership:

- To create a variable that can be changed later on, use the keyword `mut`
- There is always ever only one owner of a stack value
- Once the owner goes out of scope, any associated value will be cleaned up as well
- Rust transfers ownership for non-copy types: move semantics

END