Rust programming

Module 2: Foundations of Rust

Unit 2

Ownership and References

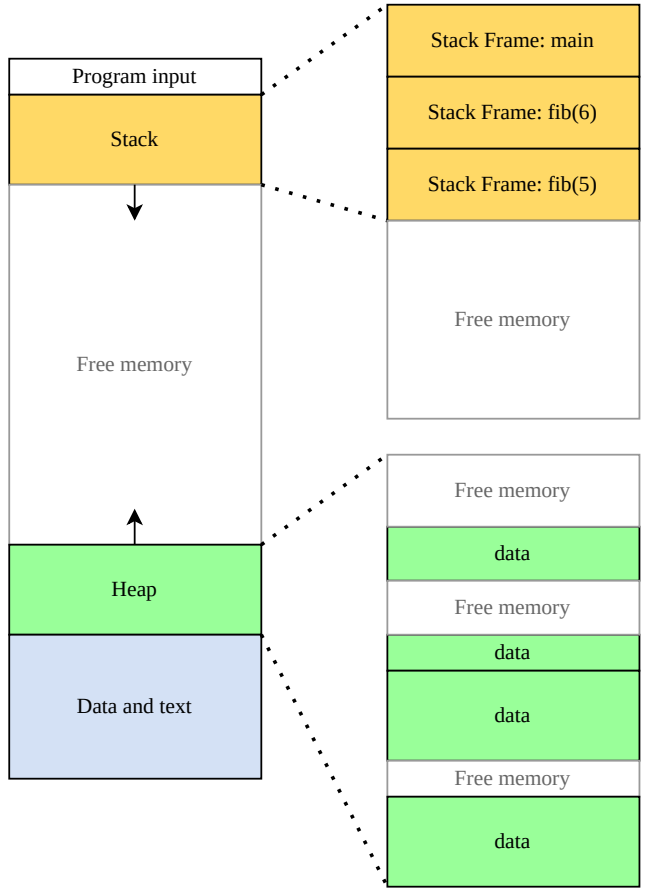
Learning objectives

- Understand Move Semantics
- Reason about Rust’s ownership and borrowing model

Move Semantics

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?

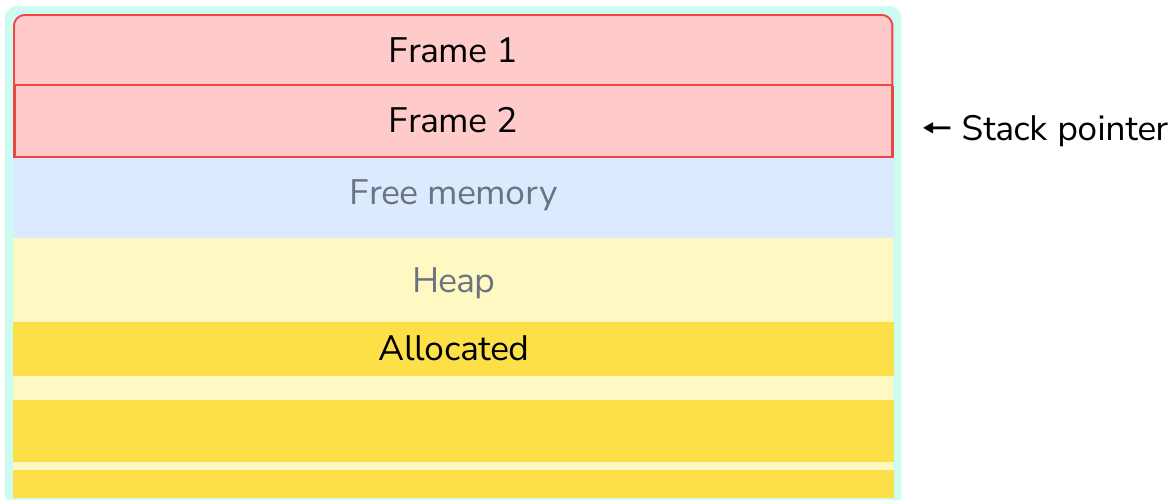


Memory

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

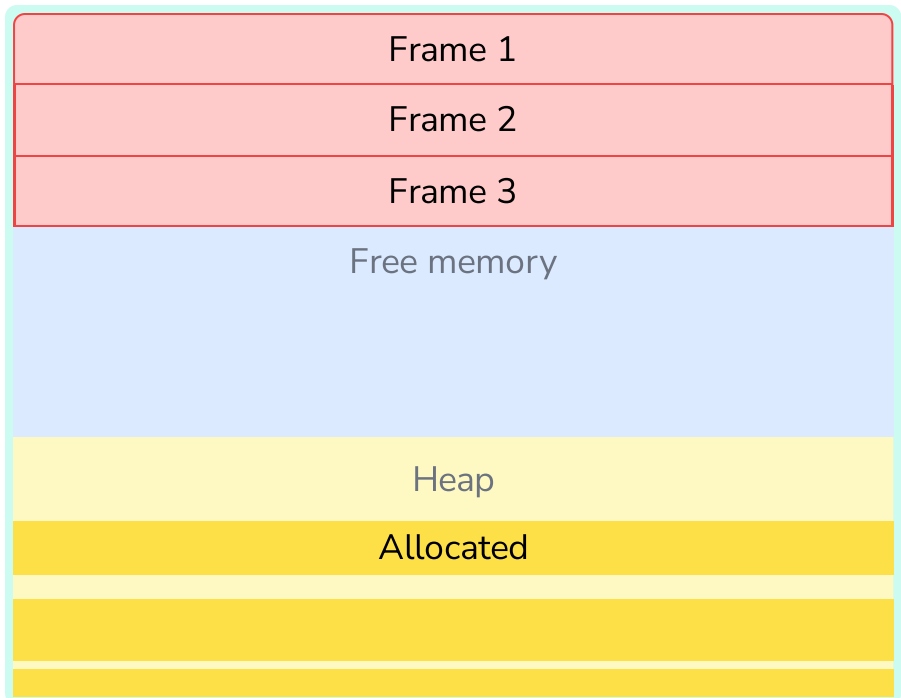
Fundamentals

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



Fundamentals

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

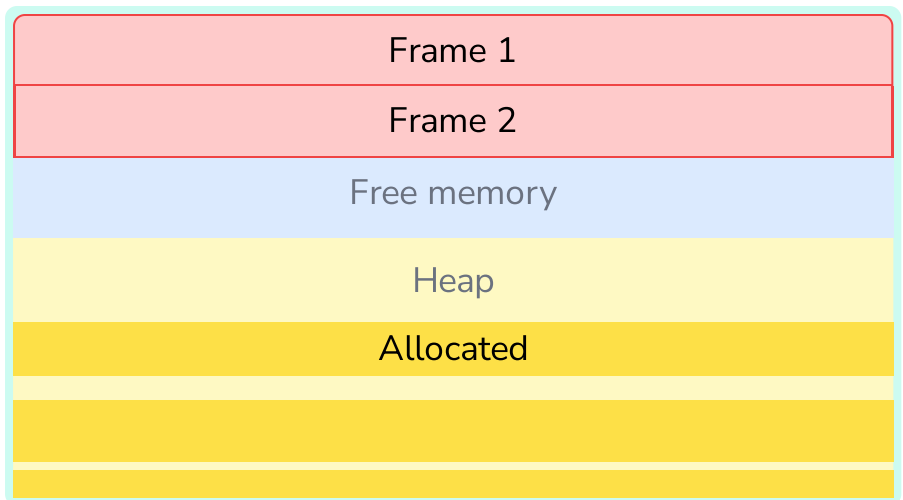


← Stack pointer

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?

Ownership

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

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 4
Running `target/debug/playground`
5
```

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

Strings store their data on the heap because they can grow

```
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 `Str
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here
```

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*

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(s1);
    println!("The length of '{}' is {}.", s1, 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 '{}' is {}.", s1, len);
  |                                       ^^ value borrowed here after move
```

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 '{}' is {}.", s1, 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 = get_length(x.clone());
    println!("{}: {}", x, len);
}

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

Ownership and borrowing

Ownership

We previously talked about ownership

- In Rust there is always a single owner for each stack value
- Once the owner goes out of scope any associated values should be cleaned up
- Copy types creates copies, all other types are *moved*

Moving out of a function

We have previously seen this example

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

- This does not compile because ownership of `s1` is moved into `calculate_length`, meaning it is no longer available in `main` afterwards
- We can use `Clone` to create an explicit copy
- We can give ownership back by returning the value
- What about other options?

Borrowing

- We can make an analogy with real life: if somebody owns something you can borrow it from them, but eventually you have to give it back
- If a value is borrowed, it is not moved and the ownership stays with the original owner
- To borrow in Rust, we create a *reference*

```
fn main() {
    let x = String::from("hello");
    let len = get_length(&x);
    println!("{}: {}", x, len);
}

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

References (immutable)

```rust
fn main() {
    let s = String::from("hello");
    change(&s);
    println!("{}", s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- help: consider changing this to be a mutable reference: `&mut String`
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be borro

For more information about this error, try `rustc --explain E0596`.
error: could not compile `playground` due to previous error
```

References (mutable)

```
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
    println!("{}", s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 2.55s
     Running `target/debug/playground`
hello, world
```

- A mutable reference can even fully replace the original value
- To do this, you can use the dereference operator (`*`) to modify the value:

```
*some_string = String::from("Goodbye");
```

Rules for borrowing and references

- You may only ever have **one mutable reference** at the same time
- You may have **any number of immutable references** at the same time **as long as there is no mutable reference**
- References cannot *live* longer than their owners
- A reference will always at all times *point to a valid value*

These rules are enforced by Rust's borrow checker.

Borrowing and memory safety

Combined with the ownership model we can be sure that whole classes of errors cannot occur.

- Rust is memory safe without having to use any runtime background process such as a garbage collector
- But we still get the performance of a language that would normally let you manage memory manually

Reference example

```rust
fn main() {
    let mut s = String::from("hello");
    let s1 = &s;
    let s2 = &s;
    let s3 = &mut s;
    println!("{} - {} - {}", s1, s2, s3);
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:5:14
  |
3 |     let s1 = &s;
  |              -- immutable borrow occurs here
4 |     let s2 = &s;
5 |     let s3 = &mut s;
  |              ^^^^^^ mutable borrow occurs here
6 |     println!("{} - {} - {}", s1, s2, s3);
  |                              -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `playground` due to previous error
```

Returning references

You can return references, but the value borrowed from must exist at least as long

```
fn give_me_a_ref() -> &String {
    let s = String::from("Hello, world!");
    &s
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:23
  |
1 | fn give_me_a_ref() -> &String {
  |                       ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
help: consider using the `'static` lifetime
  |
1 | fn give_me_a_ref() -> &'static String {
  |                       ~~~~~~~~

For more information about this error, try `rustc --explain E0106`.
error: could not compile `playground` due to previous error
```

Returning references

You can return references, but the value borrowed from must exist at least as long

```
fn give_me_a_ref(input: &(String, i32)) -> &String {
    &input.0
}
```

```
fn give_me_a_value() -> String {
    let s = String::from("Hello, world!");
    s
}
```

Summary