Borrowing variables in Rust

Mutability, references and lifetimes

Sander Karsten and Niek Janssen

23 November 2020

Outline

Memory in program execution
- The stack
- The heap

Heap memory management
- Heap memory management
- Ownership
- Moving and copying

Referencing and borrowing
- Immutable References
- Mutable references
- Dangling references

Lifetimes
- Introduction
- Lifetimes in Functions

Memory in program execution

Memory in program execution- The stack

```
1  fn main() {
2      let x: u8 = 8;
3      f();
4  }
5
6  fn f() {
7      let x = "Hello world!";
8      let y: u8 = 53;
9      let z: u64 = 100000000;
10 }
```

Memory in program execution- The stack

```
1  fn main() {
2      let x: u8 = 8;
3      f();
4  }
5
6  fn f() {
7      let x = "Hello world!";
8      let y: u8 = 53;
9      let z: u64 = 100000000;
10 }
```

- Variables on stack
- Size is known at compile time

Memory in program execution- The stack

```
1  fn main() {
2      f();
3  }
4
5  fn f() {
6      let x = String::from("Hello");
7      x.push_str(" world!");
8  }
```

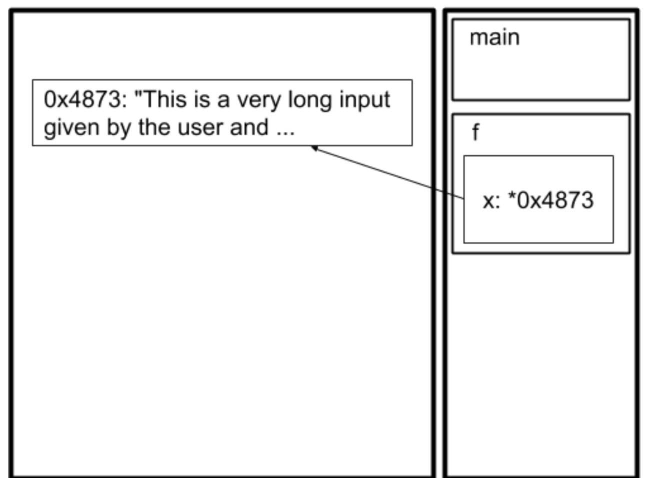
main

f

?

- Problem: size is **not** known at compile time

Memory in program execution- The heap



Memory in program execution- The heap

Allocating memory on the heap is very easy, but...

Who cleans it up?

Heap memory management- Heap memory management

Traditional memory management

Heap memory management- Heap memory management

Traditional memory management

Explicit

- Lot's of control
- Lot's of bugs
- Fast

Traditional memory management

Explicit
- Lot's of control
- Lot's of bugs
- Fast

Garbage collection
- Little control
- No bugs
- Lag spikes?

Heap memory management- Heap memory management

Traditional memory management

Explicit
- Lot's of control
- Lot's of bugs
- Fast

Garbage collection
- Little control
- No bugs
- Lag spikes?

Can we have a fast, safe way of managing memory?

2020-11-23

Borrowing variables in Rust
└─Heap memory management
  └─Heap memory management
    └─Heap memory management- Heap memory management

- Bugs:
  - Dangling pointers (use after free)
  - Memory leaks
- Bugs:
  - No control over garbage collection
  - Lag spikes when garbage collection kicks in
  - Unsuitable for responsive programs

Yes we can...

Ownership!

Ownership rules:

1. Each *value* has a *variable* which is it's *owner*
2. A value can only have one owner at a time
3. When the owner *goes out of scope*, the value will be dropped

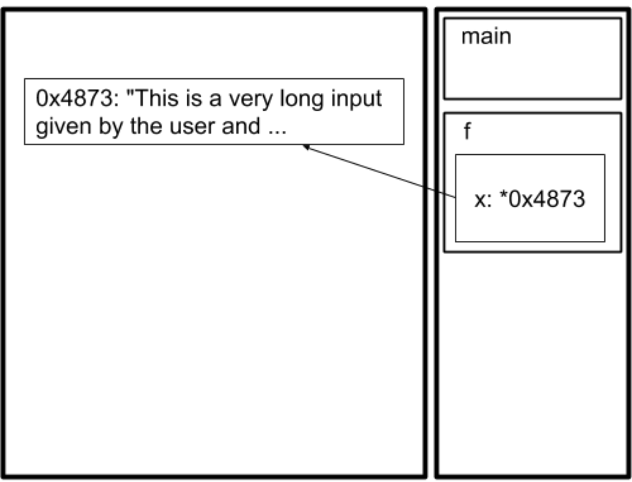
Ownership rules:

- Rule 1: Each *value* has a *variable* which is it's *owner*

Heap memory management- Ownership

Ownership rules:

- Rule 1: Each *value* has a *variable* which is it's *owner*



Heap memory management- Ownership

Ownership rules:

- Rule 1: Each *value* has a *variable* which is it's *owner*

```
1 let s = String::from("Hello world!");
```

- Not every value is stored in a variable, sometimes a heap
- Value on heap still has variable that is owner

Heap memory management- Ownership

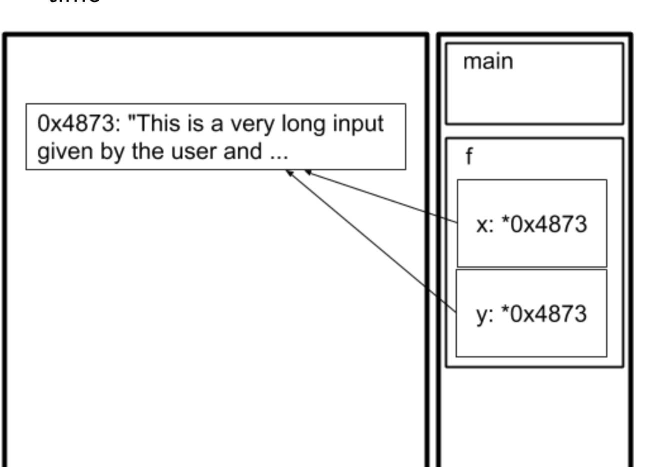
Ownership rules:

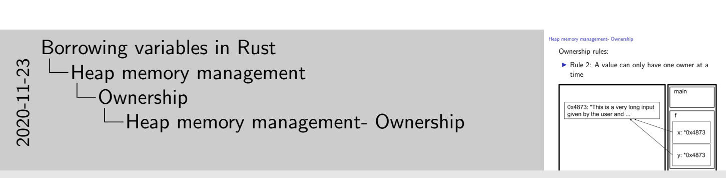
- Rule 2: A value can only have one owner at a time

Heap memory management- Ownership

Ownership rules:

- Rule 2: A value can only have one owner at a time





- Two pointers can exist, but only one of them is the owner
- Sander will talk about this more

Heap memory management- Ownership

Ownership rules:

- Rule 3: When the owner *goes out of scope*, the value will be dropped

Heap memory management- Ownership

Ownership rules:

- Rule 3: When the owner *goes out of scope*, the value will be dropped

```
{
    let s = "hello";
    // do stuff with s
}
```


Scope is everything between {}

Heap memory management- Ownership

The owner decides on mutability

```
let s1 = String::from("hello");
let mut s2 = s1;

s1.push_str(" world!"); // Not ok
s2.push_str(" world!"); // Ok
```

Heap memory management- Moving and copying

Moving values

Reading from s2 is allowed

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s2);
```

Heap memory management- Moving and copying

Moving values

Reading from s1 is not allowed

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Heap memory management- Moving and copying

Moving values

Reading from s1 is not allowed

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

- Value moved from s1 to s2
- First succeeds
- Second crashes: value not available after it has moved

Heap memory management- Moving and copying

Copying values

```
1 let s1 = String::from("hello");
2 let s2 = s1.clone();
3
4 println!("s1 = {}, s2 = {}", s1, s2);
```

2020-11-23
Borrowing variables in Rust
└Heap memory management
└Moving and copying
└Heap memory management- Moving and copying

Heap memory management- Moving and copying

Copying values

let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);

- Clone makes *deep copy* of data
- Both variables are available
- Both variables have their own data

Heap memory management- Moving and copying

Implicit copying:

- Any type implementing `Copy`
- Only if all data is on the stack

Already implemented for atomics, and some more

2020-11-23
Borrowing variables in Rust
└Heap memory management
└Moving and copying
└Heap memory management- Moving and copying

Heap memory management- Moving and copying

Implicit copying:
Any type implementing Copy
Only if all data is on the stack
Already implemented for atomics, and some more

- Usually used for atomics
- No implicit *deep copies*

Heap memory management- Moving and copying

Moves / implicit copies also occur when:

- Passing an argument
- Returning a value

Moves / implicit copies also occur when:

- Passing an argument
- Returning a value

This means we have to return each value we pass to a function to be able to use it again...

We can do better:

Referencing and borrowing

Referencing

- No ownership
- References/points to value
- Indicated by &
- Discarded after last use

Referencing

- No ownership
- References/points to value
- Indicated by &
- Discarded after last use

Borrowing

- Reference as function parameter

Borrowing a value

```rust
fn main() {
    let hello = String::from("hello");
    let length = calculate_length(&hello);

    println!("The length of '{}' is {}.", hello, length);
}

fn calculate_length(ref_string: &String) -> usize {
    return ref_string.len();
}
```


Borrowing a value

```rust
fn main() {
    let hello = String::from("hello");
    let length = calculate_length(&hello);

    println!("The length of '{}' is {}.", hello, length);
}

fn calculate_length(ref_string: &String) -> usize {
    return ref_string.len();
}
```

ref_string

name	value
ptr	→ hello

hello

name	value
ptr	→
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

Point out:

- s is a reference to string
- s goes out of scope after calculate_length function, does not own reference so nothing happens
- a reference to s1 is created by &s1 as argument

Changing a borrowed value (does not work!)

```
1 fn main() {
2     let s1 = String::from("this is");
3     change(&s1);
4 }
5
6 fn change(s: &String) {
7     s.push_str(" not possible!");
8 }
```


Changing a borrowed value (does not work!)

```
1 fn main() {
2     let s1 = String::from("this is");
3     change(&s1);
4 }
5
6 fn change(s: &String) {
7     s.push_str(" not possible!");
8 }
```


But how can this value be changed?

Point out:

- references are by default immutable just like variables =¿ solved by using mutable references

This can be changed with . . .

Mutable References!

Mutable references (does work)

```
1 fn main() {
2     let mut s1 = String::from("this is");
3     change(&mut s1);
4 }
5
6 fn change(s: &mut String) {
7     s.push_str(" possible!");
8 }
```

Point out:

- reference is now mutable and can be modified
- mut must be used in all 3 locations!

Only one mutable reference, prevents data races

Data races

A data race occurs when there are multiple references to the same data and one of these references writes. This causes the other references to need to synchronize which can cause problems.

Point out:

- Data races can cause undefined behaviour in programs
- Undefined behaviour can be hard to diagnose
- Rust won't compile code that contains data races

Possible (multiple scopes):

```
1 let mut a = String::from("Simultaneous");
2 {
3     let b = &mut a;
4 }
5 let c = &mut a;
```


Possible (multiple scopes):

```
1 let mut a = String::from("Simultaneous");
2 {
3     let b = &mut a;
4 }
5 let c = &mut a;
```


Not possible (combining mutable and immutable):

```
1 let mut a = String::from("Multiple");
2 let good_var_1 = &a;
3 let good_var_2 = &a;
4 let very_bad_var = &mut a;
5 println!("{}, {}, and {}", good_var_1, good_var_2, very_bad_var);
```


Point out:

- b goes out of scope so a new reference can be created to a with no problem
- you can't combine mutable and immutable references, will not compile
- If you have an immutable reference you don't expect it to change by a later defined mutable reference
- Multiple immutable is allowed because data is only read so no synchronization occurs

Referencing and borrowing- Mutable references

Possible:

```
1 let mut a = String::from("Scopes");
2
3 let x = &a;
4 println!("{}", x);
5
6 let y = &mut a;
7 println!("{}", y);
```

Referencing and borrowing- Mutable references

Possible:

```
1 let mut a = String::from("Scopes");
2
3 let x = &a;
4 println!("{}", x);
5
6 let y = &mut a;
7 println!("{}", y);
```

What if we reference a location that is out of scope?

Point out:

- scope of x ends after println, now y can be introduced as mutable

Referencing and borrowing- Mutable references

We meet our old friend the . . .

Dangling References

Referencing and borrowing- Dangling references

Dangling references

- Create a reference to some memory
- The owner goes out of scope, so the memory will be dropped
- The reference still exists

Point out:

- Enforced by compiler

Referencing and borrowing- Dangling references

Not possible (dangling reference)

```rust
1  fn main() {
2      let invalid_reference = dangling_function();
3  }
4
5  fn dangling_function() ->&String {
6      let my_new_variable = String::from("Hello!!!");
7      return &my_new_variable;
8  }
```

2020-11-23

Borrowing variables in Rust
└Referencing and borrowing
  └Dangling references
    └Referencing and borrowing- Dangling references

Point out:

- this function will return a borrowed value while there is no value to be borrowed from
- in this case invalidReference is a dangling reference
- dangleFunction() returns reference to String, we return reference to string, string goes out of scope and memory is deallocated reference would now be pointing to invalid string
- fix this by making danglingfunction not return reference
- when fixed variable is returned to caller the ownership is moved to caller

Referencing and borrowing- Dangling references

Possible (Move ownership to caller)

```rust
1  fn main() {
2      let valid_reference = non_dangling_function();
3  }
4
5  fn non_dangling_function() -> String {
6      let my_new_variable = String::from("Hello!!!");
7      return my_new_variable;
8  }
```

Referencing and borrowing- Dangling references

Possible (Move ownership to caller)

```rust
1  fn main() {
2      let valid_reference = non_dangling_function();
3  }
4
5  fn non_dangling_function() -> String {
6      let my_new_variable = String::from("Hello!!!");
7      return my_new_variable;
8  }
```

How can we prevent dangling references?

Lifetimes- Introduction

How can we prevent dangling references?

Compare lifetimes of value and reference

Lifetimes- Introduction

A lifetime is the scope for which a value or reference is valid

```
{
    let r;                          // ---------+-- 'a
    {                               //          |
        let x = 5;                  // -+-- 'b  |
        r = & x;                    //  |       |
    }                               // -+       |
    println ! ("r: {}", r);         //          |
}                                   // ---------+
```


Lifetime of s ends before ref_s

```
fn main() {
    let ref_s = create_str();
    println!("{}", ref_s)
}

fn create_str() -> &String {
    let s = String::from("Hi!");
    return &s;
}
```

Lets call:

- Lifetime of ref_s = 'a
- Lifetime of s = 'b

Lets call:

- Lifetime of ref_s = 'a
- Lifetime of s = 'b

```
fn main() {
    let ref_s = create_str() {          // Lifetime 'a starts here
        let s = String::from("Hi!");    // Lifetime 'b starts here
        return &s;
    }                                   // Lifetime 'b ends here
    println!("{}", ref_s);
}                                       // Lifetime 'a ends here
```


We want to create a function that returns the longest string (does not compile)

```
fn main() {
    let string1 = String::from("short");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}
```


We want to create a function that returns the longest string (does not compile)

```
fn main() {
    let string1 = String::from("short");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}
```

Rust's borrow checker will not let you compile this!

For main:

- There is a main with 2 strings where each variable has a different lifetime, e.g. 'x for 1 and 'y for 2
- After this a function longest is called on 2 immutable string references
- The question is which lifetime does this function now return, the one of 1 or 2?

For longest:

- In the function the problem is "reversed"
- Which of these 2 variables now has a valid lifetime? We don't know!
- Because of the if statement both lifetimes can now be returned.
- How do we fix this?

Borrow checker:

- Scopes are compared by the borrow checker, to determine if all borrows are valid
- It sees that the function has two possible lifetimes

Lifetimes- Lifetimes in Functions

We add a new lifetime annotation `'a` to the `longest` function (compiles)

```rust
fn main() {
    let string1 = String::from("long");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() {
        return a;
    } else {
        return b;
    }
}
```

2020-11-23
Borrowing variables in Rust
└Lifetimes
└Lifetimes in Functions
└Lifetimes- Lifetimes in Functions

Lifetime annotations:

- Won't change how long these references live
- Describes relations between lifetimes of multiple references

How is the function fixed?:

- the parameters will live at least as long as lifetime 'a
- the slice returned will now live at least as long as 'a
- This means that the lifetime returned by longest is now the smallest of the lifetimes passed in
- Meaning in this case it is the part of scope x that collides with scope y
- NOTE! We are not changing lifetimes of parameters passed with this function