Rust programming

Course – 2

Gavrilut Dragos

rev 11

Agenda for today

1. Prerequisite: String type
2. Ownership management
3. Borrowing & References
4. Reborrowing
5. Optimizations

Prerequisite: String type

Prerequisite: String type

- For the purpose of this course, we need to quickly understand some things about strings in Rust
- So….
  - Rust type: **String**
  - Format: **Dynamic** (can increase its size)
  - Encoding: **UTF-8**
  - Operations: addition, substring, find, …

We will cover strings in more detail on another course, for the moment we will learn a couple of things about object String that will be useful for the next chapters.

Prerequisite: String type

- Let’s see some examples:

1. How to create a string (keep in mind that there are several ways to create a string that we will cover in a different course).

Rust

```rust
fn main() {
    let mut s: String = String::from("a string");
    println!("s = {s}");
}
```

Output

```
s = a string
```

2. How to get the length of a string (via method .len())

Rust

```rust
fn main() {
    let mut s: String = String::from("a string");
    println!("len = {}", s.len());
}
```

Output

```
len = 8
```

Prerequisite: String type

- Let’s see some examples:

3. Concatenate strings (via operator += or method .push_str(…))

Rust

```rust
fn main() {
    let mut s: String = String::from("123");
    s += "456";
    s.push_str("789");
    println!("{s}");
}
```

Output

```
123456789
```

4. How to obtain a substring (a slice) of a string via range op [..])

Rust

```rust
fn main() {
    let s: String = String::from("ABCDEFG");
    println!("{}", &s[1..3]);
}
```

Output

```
BC
```

Prerequisite: String type

- Finally, keep in mind that strings in Rust are far more complex and require a more in-depth analysis.
- However, for the current being, this explanation should be enough.

Ownership management

Ownership

- In Rust, every memory zone has **ONE** and **ONLY ONE** owner at a time.
- Every owner has a lifetime (it exists within a scope).
- When an owner goes out of scope (its lifetime is over) the memory zone is freed ("*freed*" in this context has a different meaning – based on where that memory zone lies on : stack , heap or global).

Ownership

- As a rule, every variable (local / global) or a parameter can be considered the owner for the memory zone it represent.
- Let's analyze the following case:

Rust

```
fn main() {

}
```

Stack of function main				
Offset	*Content (4 bytes alignment)*			
100.000	?	?	?	?
99.996	?	?	?	?
99.992	?	?	?	?
99.988	?	?	?	?

Ownership

- As a general consent, every variable (local / global) or a parameter can be considered the owner for the memory zone they represent.
- Let’s analyze the following case:

Rust

```rust
fn main() {
    let mut sum: u32 = 10;
}
```

Stack of function main

Offset	*Content (4 bytes alignment)*			
100.000	?	?	?	?
99.996	**10**	**0**	**0**	**0**
99.992	?	?	?	?
99.988	?	?	?	?

Ownership

- As a general consent, every variable (local / global) or a parameter can be considered the owner for the memory zone they represent.
- Let’s analyze the following case:

Rust

```rust
fn main() {
    let mut sum: u32 = 10;
}
```

Stack of function main

Offset	*Content (4 bytes alignment)*			
100.000	?	?	?	?
99.996	**10**	**0**	**0**	**0**
99.992	?	?	?	?
99.988	?	?	?	?

- We can say that:
  - The memory from offset ***99.996*** to ***100.000*** is owned by variable ***sum***

Ownership

- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

Owner	Memory Type

Ownership

- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

Owner	Memory Type
sz	Stack (4 bytes)

Ownership

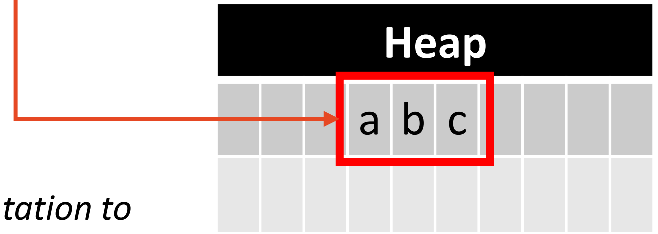
- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

Owner	**Memory Type**
sz	Stack (4 bytes)
s	Stack (12 or 24 bytes) and Heap (3 bytes)

String type (*)	
Field	*Type*
chars	Ptr to heap where the the UTF-8 text lies
len	usize (4 or 8 bytes)
capacity	usize (4 or 8 bytes)



** String internal structure is subject to change (this is an academic representation to better explain how ownership works).*

Ownership

- Let’s analyze another case:

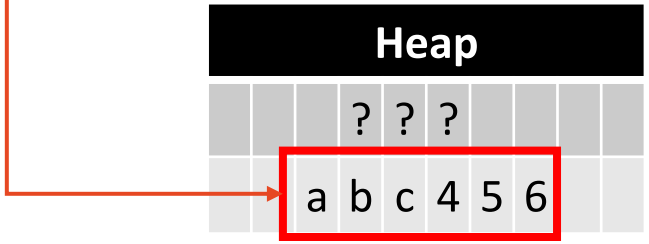
Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

- A new space of 6 characters is allocated on the heap
- The original text (“abc”) is copied in the new location
- The new string (“456”) is added after “abc”
- The old (3 bytes) space from the heap is freed

Owner	**Memory Type**
sz	Stack (4 bytes)
s	Stack (12 or 24 bytes) and Heap (6 bytes)

String type (*)	
Field	*Type*
chars	Ptr to heap where the the UTF-8 text lies
len	usize (4 or 8 bytes)
capacity	usize (4 or 8 bytes)



Heap									
			?	?	?				
		a	b	c	4	5	6		

Ownership

- Let’s analyze another case:

Rust

```rust
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

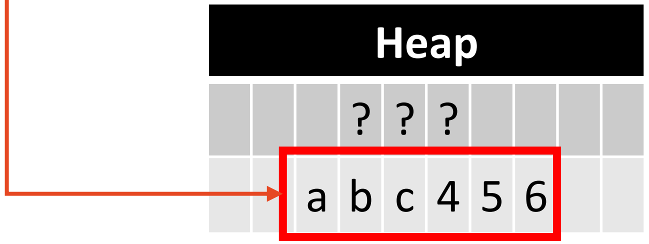
Lifetime of “s”

At this point the scope of “**s**” has ended. As such, the compiler decides to free all memory owned by “**s**”

Owner	**Memory Type**
sz	Stack (4 bytes)
s	Stack (12 or 24 bytes) and Heap (6 bytes)

String type (*)

Field	*Type*
chars	Ptr to heap where the the UTF-8 text lies
len	usize (4 or 8 bytes)
capacity	usize (4 or 8 bytes)



Heap

? ? ?

a b c 4 5 6

Ownership

- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

Owner	Memory Type
sz	Stack (4 bytes)
s	Stack (12 or 24 bytes) and Heap (0 bytes)

String type (*)	
Field	*Type*
chars	Null-pointer (data is deallocated)
len	usize (4 or 8 bytes)
capacity	usize (4 or 8 bytes)

Heap

			?	?	?				
		?	?	?	?	?	?		

- Freeing “s” means:
  - Free all heap memory associated if it
  - Clear stack memory where len, capacity and chars pointer were stored.

Ownership

- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

Owner	Memory Type
sz	Stack (4 bytes)
~~s~~	~~Stack (12 or 24 bytes) and Heap (0 bytes)~~

- At this point “s” is no longer valid. This means that the memory it owns is no longer available, nor is the access of variable “s”. In fact, any usage of “s” variable in this point will be considered a compiler error.

Ownership

- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

Lifetime of “sz”

Owner	**Memory Type**
~~sz~~	~~Stack (4 bytes)~~
~~s~~	~~Stack (12 or 24 bytes) and Heap (0 bytes)~~

- At this point, the scope of “sz” has ended and as such it is freed (all stack space is cleared).

Ownership

- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

C representation

```
void main() {
    unsigned int sz = 0;
    {
        char* s = new char[4] {"abc"} );
        …
        delete []s;
        s = NULL;
    }
    …
}
```

- A similar “C” code implies freeing the heap manually. If this action would have not been performed, the memory allocated by “s” remains allocated (a bug often called **<<memory leak>>**).

Ownership

- Let’s analyze another case:

Rust

```
fn main() {
    let mut sz: u32 = 0;
    {
        let mut s = String::from("abc");
        s.push_str("456");
        sz += s.len() as u32;
        println!("{s}");
    }
    println!("{sz}");
}
```

C++ representation (from C++11)

```

void main() {
    unsigned int sz = 0;
    {
        unique_ptr<char[]> s (new
                           char[4] {"abc"});
    }
    …
}
```

- Modern C++ has a type of allocation similar to what Rust has (called unique_ptr) that behaves in a similar manner.

Ownership

- What is the problem with the following C/C++ code ?

C

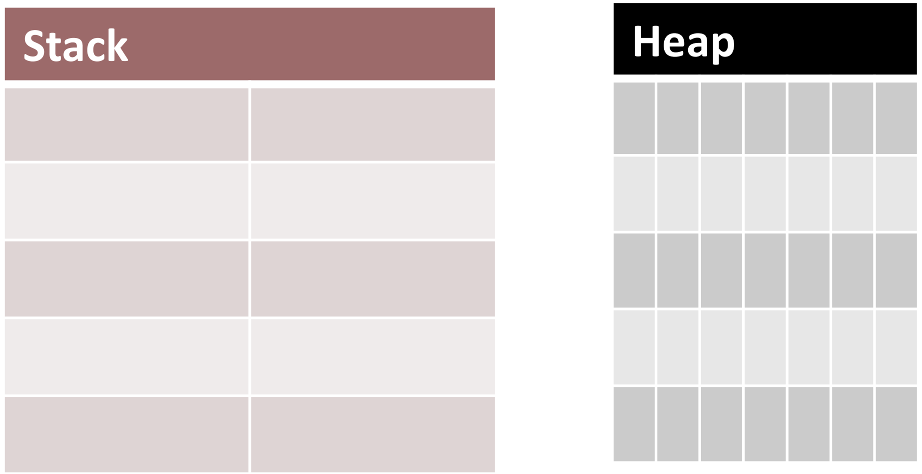
```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```

Ownership

- What is the problem with the following C/C++ code ?

C

```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```



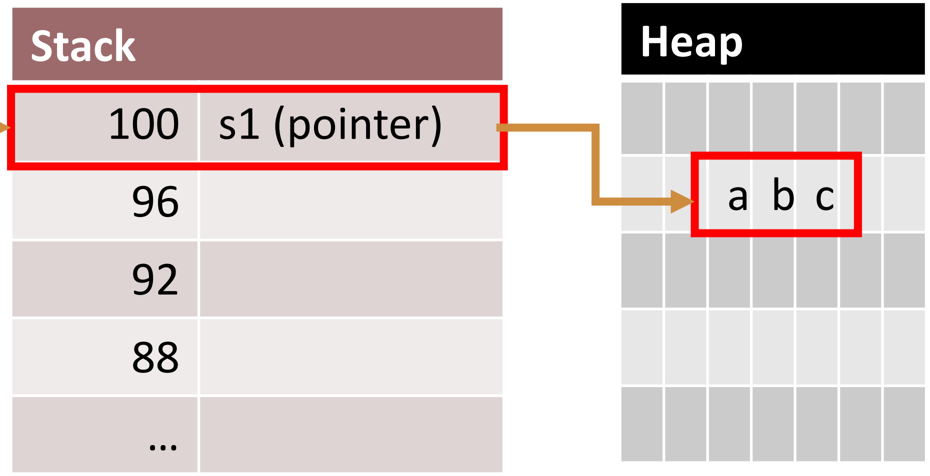
- Let’s run this code step by step …

Ownership

- What is the problem with the following C/C++ code ?

C

```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```



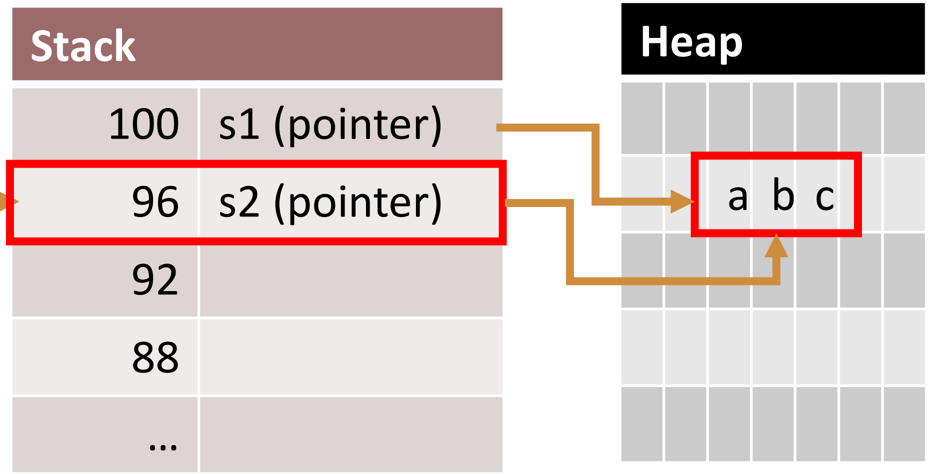
Stack	
100	s1 (pointer)
96	
92	
88	
…	

Ownership

- What is the problem with the following C/C++ code ?

C

```c
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```



Stack

100	s1 (pointer)
96	s2 (pointer)
92	
88	
…	

Heap

a b c

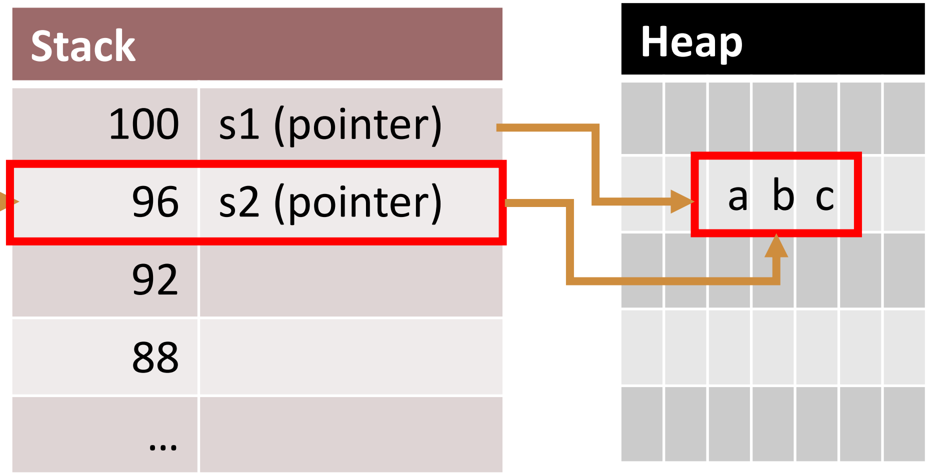
- What is the problem at this point ?

Ownership

- What is the problem with the following C/C++ code ?

C

```c
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```



Stack

100	s1 (pointer)
96	s2 (pointer)
92	
88	
…	

- What is the problem at this point ?

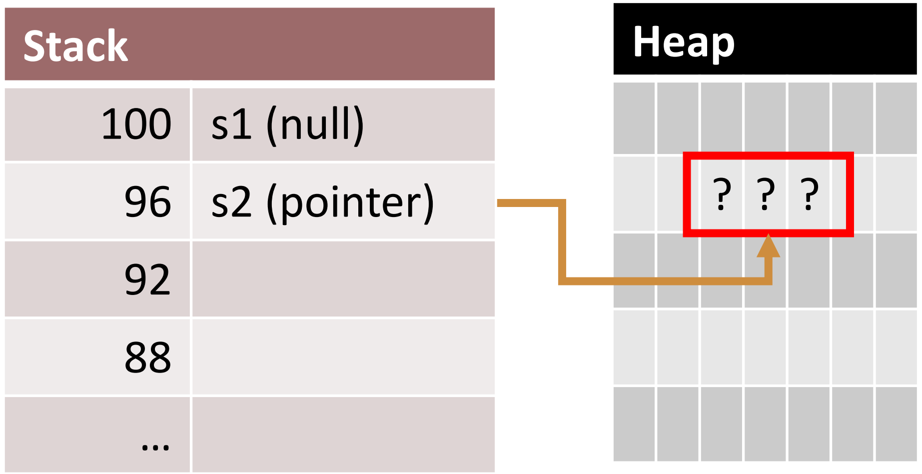
Both “s1” and “s2” point to the same memory location. Or in other words, for a specific memory address we have **TWO OWNERS**.

Ownership

- What is the problem with the following C/C++ code ?

C

```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```



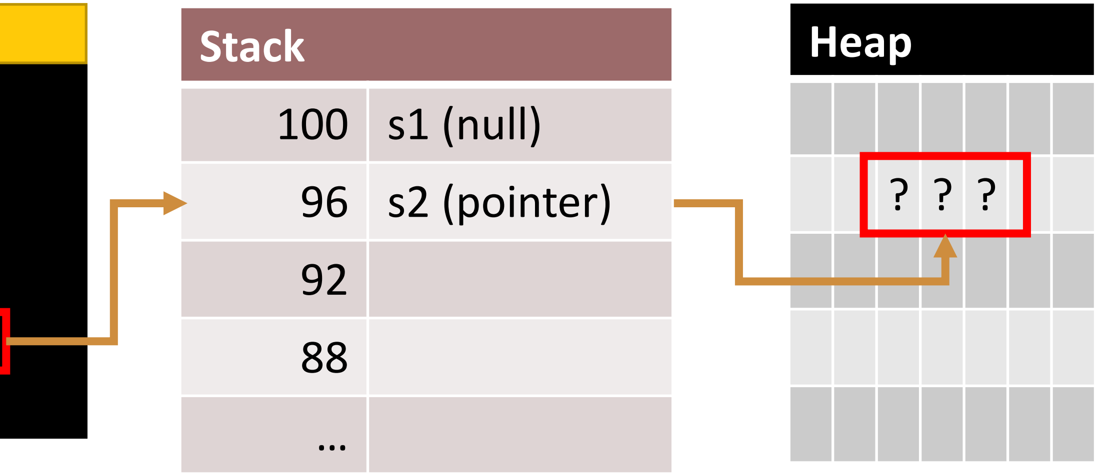
- "s1" pointer is deleted → and this translates that the memory zone that was allocated in the Heap to store the content for "s1" is freed as well.

Ownership

- What is the problem with the following C/C++ code ?

C

```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```



Stack

100	s1 (null)
96	s2 (pointer)
92	
88	
...	

Heap

? ? ?

- At this point, ***printf*** will try to access the content of “s2” that now points to a memory zone that was already freed from the previous line.
- **The behavior is undefined, and it is likely to produce a crash !!!**

Ownership

- What is the problem with the following C/C++ code ?

C

```c
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```

- The main issue from this code is that the assignment “char* s2 = s1” creates two owners (both s1 and s2 point to the same memory address)

Ownership

- So what can we do to make this code safe ?

C

```c
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```

- The main issue is how we understand the assignment (“char* s2 = s1”). The worst thing we can do is to duplicate the pointer (make two owners).
- Rust has a concept (called ***trait***) that for the moment can be considered as a property list for each type that explain how certain operations can be performed.
- For this particular example , the ***traits*** that are important are Copy and Move

Ownership

Disclaimer:

- Move trait does not exist in Rust (it is considered by default as something to be used if the trait Copy is not present).
- However, for the purpose of the next slides, we will consider that this trait (**Move**) exists (this will allow us to easily explain how some decisions in Rust are being made by the compiler).

Copy vs Move operations

Let’s consider that we have a type (called Student) and we write a statement like in the following way:

Rust

```
fn main() {
    let mut s1:Student = …;
    let mut s2:Student;
    s2 = s1;
}
```

Student
MathGrade → type u8
EnglishGrade → type u8
Name → heap buffer

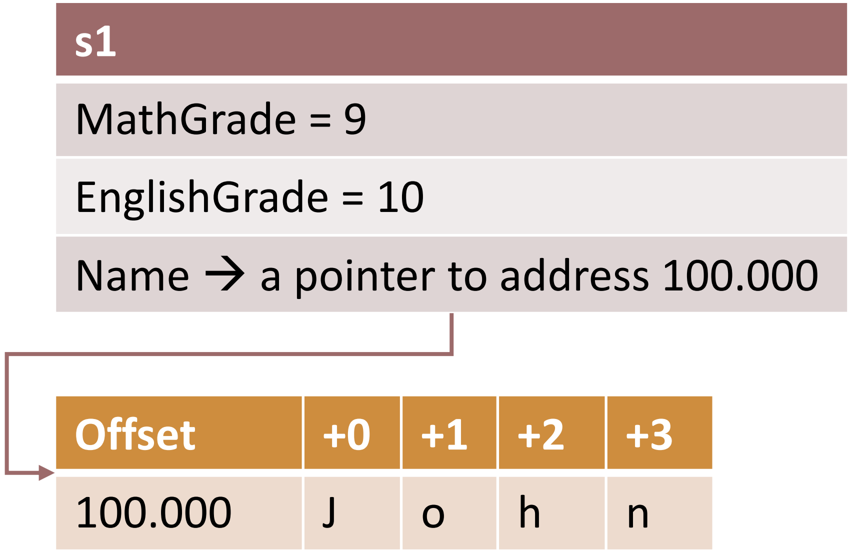
What happens when s2 is assigned with the value s1 ?
It depends on some traits that the object of type Student has. A trait (at this point) can be considered a property defined as a function with a specific purpose (in reality a trait is more similar to an interface).

If type Student has the trait Copy then Rust will compile the statement `s2 = s1` in a specific way, while if the Student has the trait Move, Rust will compile things differently.

What does Copy operation means

Let’s see what Copy trait implies for s2 = s1 :

s2
MathGrade = ?
EnglishGrade = ?
Name → <none> or <null>

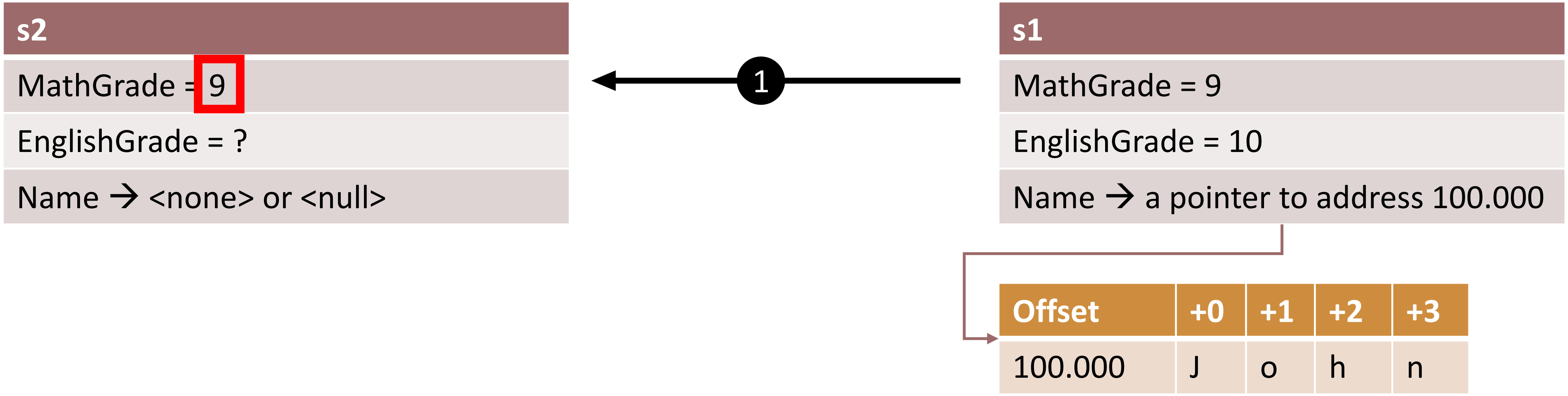


s1
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 100.000

Offset	**+0**	**+1**	**+2**	**+3**
100.000	J	o	h	n

What does Copy operation means

Let’s see what Copy trait implies for s2 = s1 :



s2
MathGrade = 9
EnglishGrade = ?
Name → <none> or <null>

s1
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 100.000

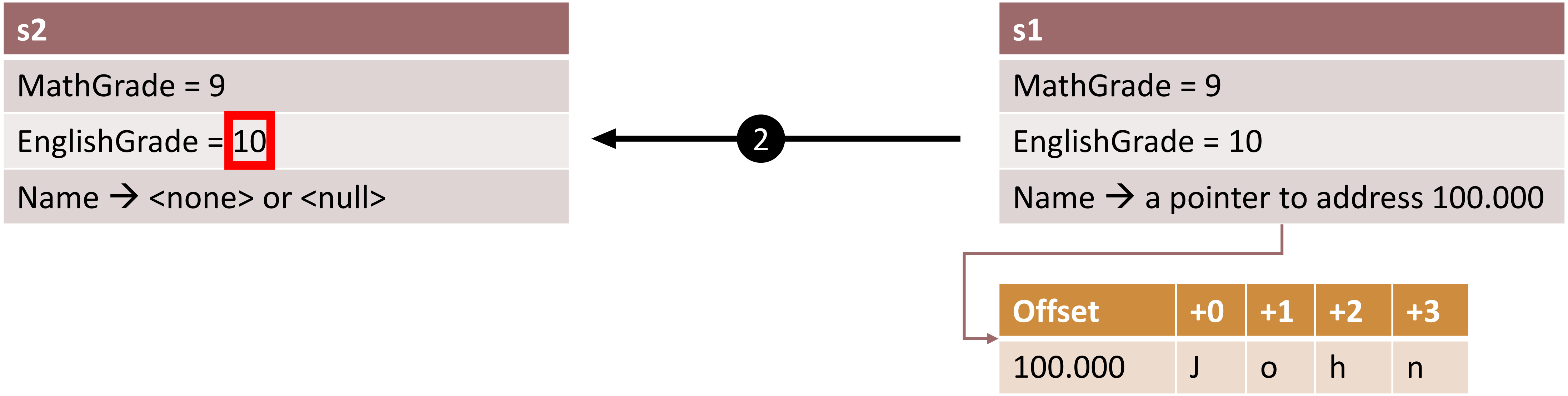
Offset	+0	+1	+2	+3
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade***

What does Copy operation means

Let's see what Copy trait implies for s2 = s1 :



s2
MathGrade = 9
EnglishGrade = 10
Name → <none> or <null>

s1
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 100.000

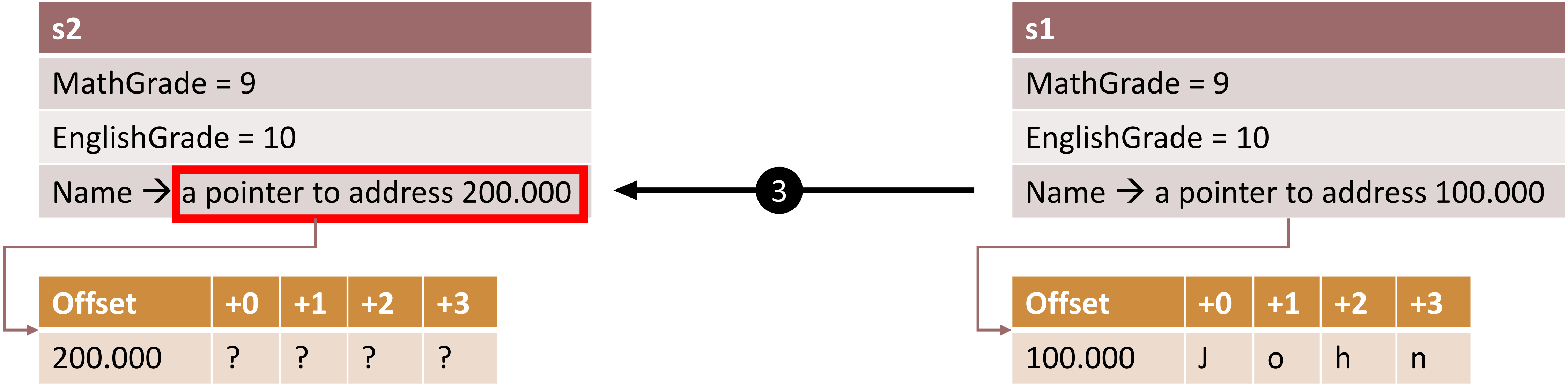
Offset	+0	+1	+2	+3
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade***
2. Bitwise copy the value of ***s1.EnglishGrade*** into ***s2.EnglishGrade***

What does Copy operation means

Let's see what Copy trait implies for s2 = s1 :



s2
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 200.000

Offset	**+0**	**+1**	**+2**	**+3**
200.000	?	?	?	?

s1
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 100.000

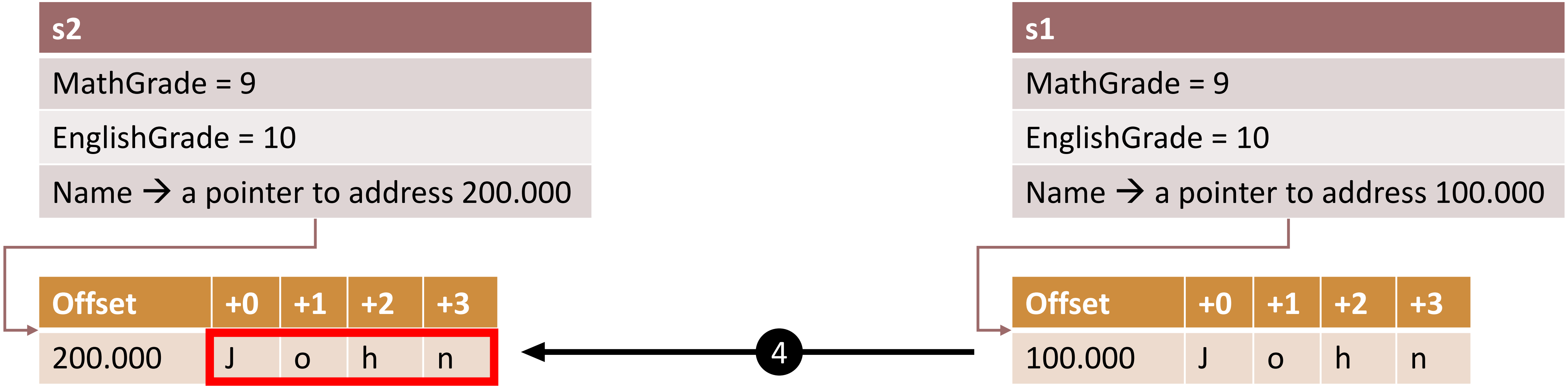
Offset	**+0**	**+1**	**+2**	**+3**
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade***
2. Bitwise copy the value of ***s1.EnglishGrade*** into ***s2.EnglishGrade***
3. Allocated 4 bytes to a new location on the heap and assign **s2.Name** pointer to that location

What does Copy operation means

Let’s see what Copy trait implies for s2 = s1 :



s2

MathGrade = 9

EnglishGrade = 10

Name → a pointer to address 200.000

Offset	+0	+1	+2	+3
200.000	J	o	h	n

s1

MathGrade = 9

EnglishGrade = 10

Name → a pointer to address 100.000

Offset	+0	+1	+2	+3
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade***
2. Bitwise copy the value of ***s1.EnglishGrade*** into ***s2.EnglishGrade***
3. Allocated 4 bytes to a new location on the heap and assign **s2.Name** pointer to that location
4. Bitwise copy 4 bytes from address <u>100.000</u> (***s1.Name***) to address <u>200.000</u> (***s2.Name***)

What does Move operation means

Let’s see what Move trait implies for s2 = s1 :

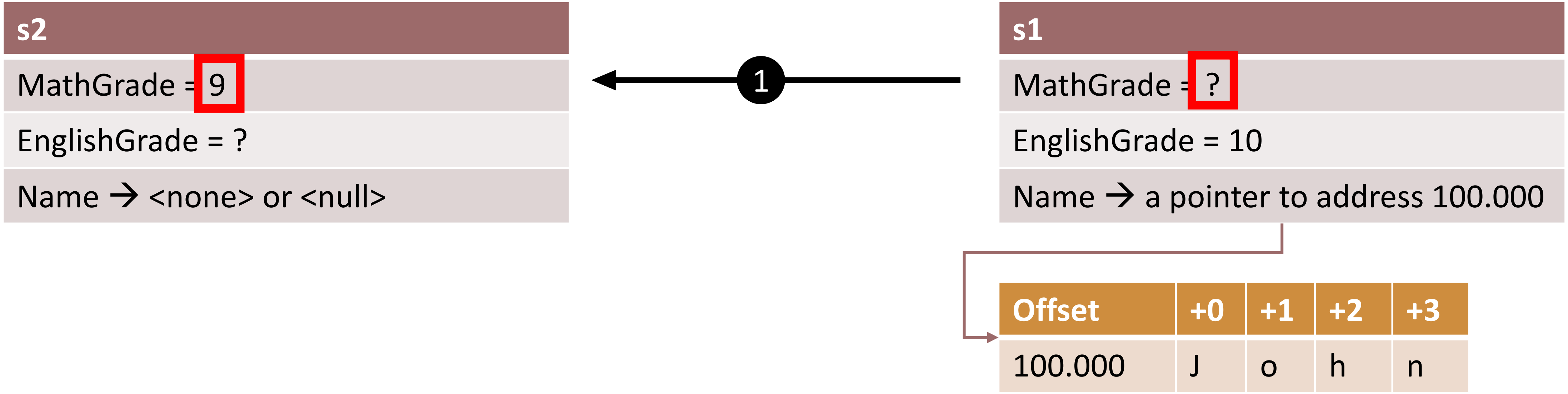
s2
MathGrade = ?
EnglishGrade = ?
Name → <none> or <null>

s1
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 100.000

Offset	**+0**	**+1**	**+2**	**+3**
100.000	J	o	h	n

What does Move operation means

Let's see what Move trait implies for s2 = s1 :



s2
MathGrade = 9
EnglishGrade = ?
Name → <none> or <null>

s1
MathGrade = ?
EnglishGrade = 10
Name → a pointer to address 100.000

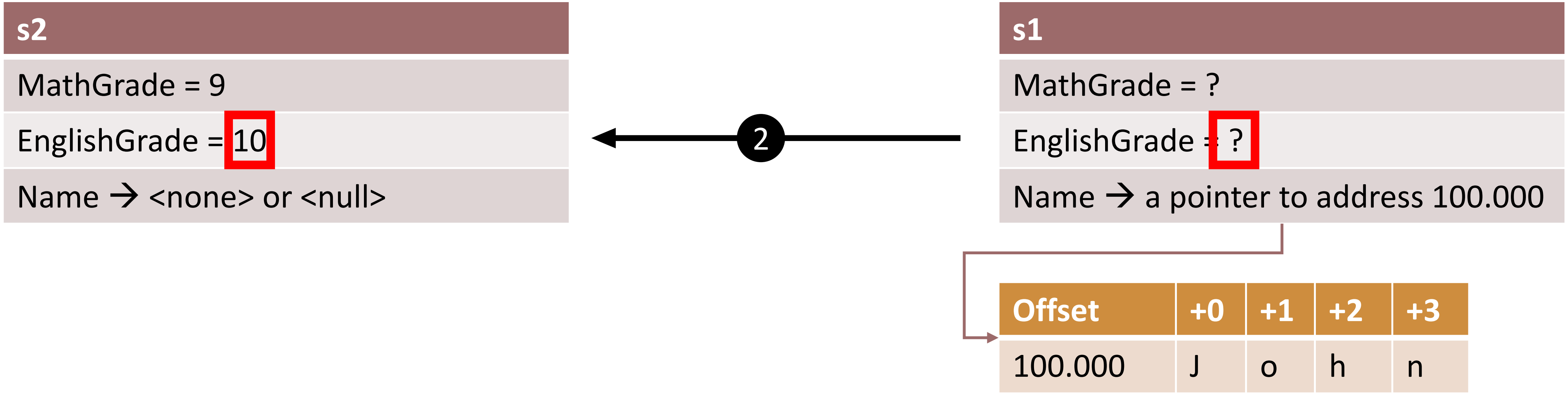
Offset	+0	+1	+2	+3
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade,*** and clear the value of ***s1.MathGrade***

What does Move operation means

Let’s see what Move trait implies for s2 = s1 :



s2

MathGrade = 9

EnglishGrade = 10

Name → <none> or <null>

s1

MathGrade = ?

EnglishGrade = ?

Name → a pointer to address 100.000

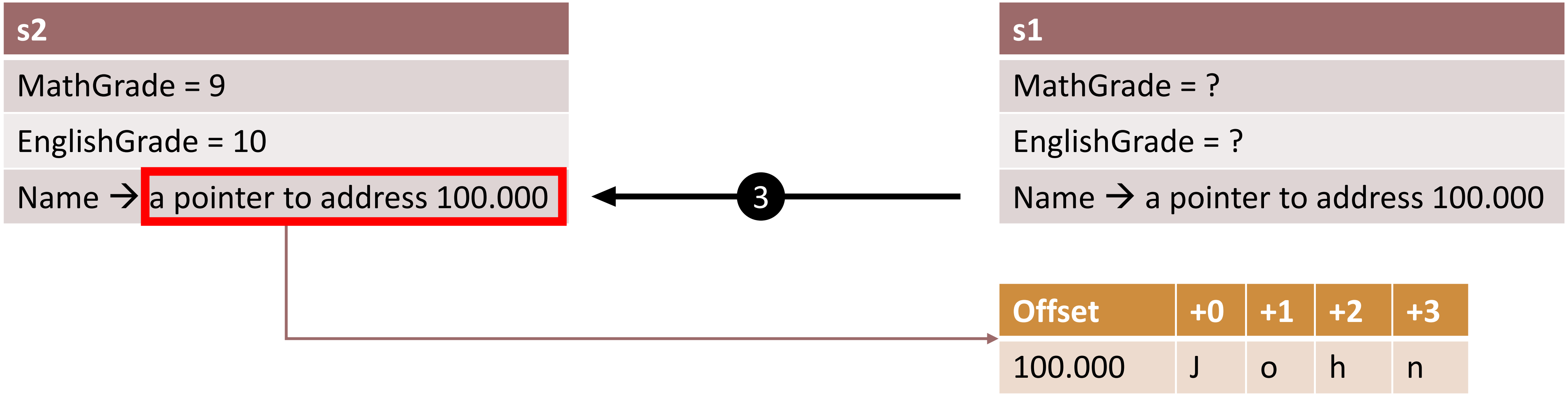
Offset	**+0**	**+1**	**+2**	**+3**
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade***, and clear the value of ***s1.MathGrade***
2. Bitwise copy the value of ***s1.EnglishGrade*** into ***s2.EnglishGrade*** , and clear the value of ***s1.EnglishGrade***

What does Move operation means

Let’s see what Move trait implies for `s2 = s1` :



s2
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 100.000

s1
MathGrade = ?
EnglishGrade = ?
Name → a pointer to address 100.000

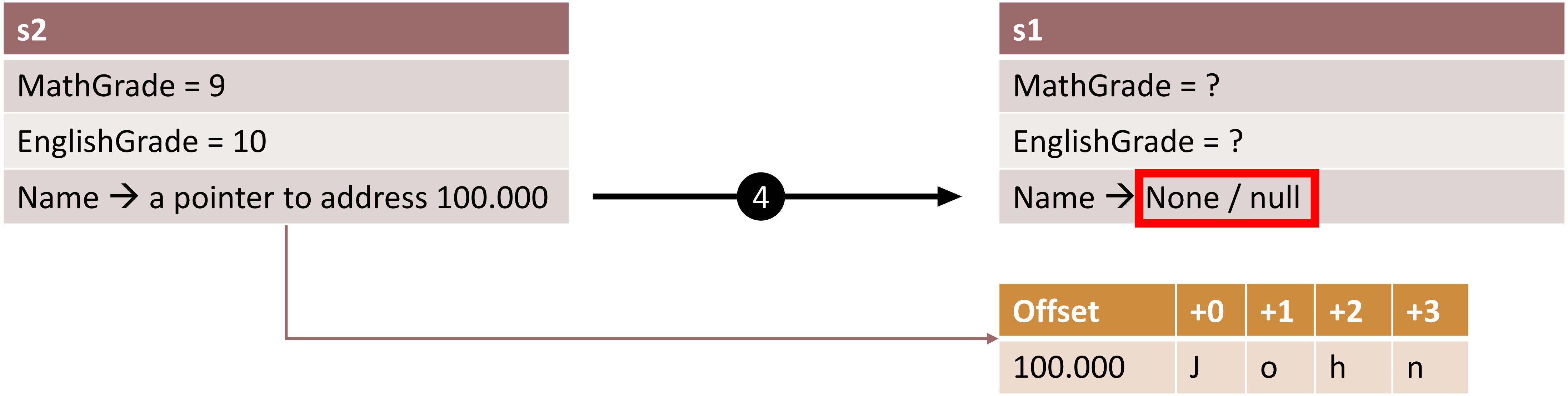
Offset	+0	+1	+2	+3
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade***, and clear the value of ***s1.MathGrade***
2. Bitwise copy the value of ***s1.EnglishGrade*** into ***s2.EnglishGrade*** , and clear the value of ***s1.EnglishGrade***
3. Assign ***s2.Name*** pointer to the offset 100.000

What does Move operation means

Let’s see what Move trait implies for s2 = s1 :



s2
MathGrade = 9
EnglishGrade = 10
Name → a pointer to address 100.000

s1
MathGrade = ?
EnglishGrade = ?
Name → None / null

Offset	**+0**	**+1**	**+2**	**+3**
100.000	J	o	h	n

Steps:

1. Bitwise copy the value of ***s1.MathGrade*** into ***s2.MathGrade**,* and clear the value of ***s1.MathGrade***
2. Bitwise copy the value of ***s1.EnglishGrade*** into ***s2.EnglishGrade*** , and clear the value of ***s1.EnglishGrade***
3. Assign ***s2.Name*** pointer to the offset 100.000
4. Clear the value of pointer ***s1.Name*** so that only one object points to the offset 100.000

Ownership

- So, what can we do to make this code safe ?

C (undefined behabior)

```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```

C (MOVE)

```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = s1; s1 = nullptr;
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```

C (COPY)

```
void main() {
    char* s1 = new char[4]{"abc"};
    char* s2 = strdup(s1);
    delete []s1; s1 = nullptr;
    printf("%s\n",s2);
}
```

Ownership

- By default, Rust uses **MOVE** operation for all of its object (except for the case where **COPY** trait is set up for on object)
- Basic types (u8..u128, i8..i128, bool, isize, usize, char) have the **COPY** trait.

Advantages:

1. No dangling pointers
2. No data races

Ownership

When ownership rules applies:

1. Whenever there is an assignment
   `x = y`
2. Whenever a parameter is passed to a function
   `my_function(x)`
3. Whenever a value is returned from a function
   `y = my_function(x)`

Ownership

Let’s see some examples:

Rust

```rust
fn main() {
    let s: String = String::from("AAA");
    let s2 = s;
    println!("{s2}");
}
```

Compiles ok

Output

```
AAA
```

Rust

```rust
fn main() {
    let s: String = String::from("AAA");
    let s2 = s;
    println!("{s}");
}
```

Compile error

```
error[E0382]: borrow of moved value: `s`
```

Ownership

Let’s see some examples:

Rust

```rust
fn main() {
    let s: String = String::from("AAA");
    let s2 = s;
    println!("{s2}");
}
```

Rust

```rust
fn main() {
    let s: String = String::from("AAA");
    let s2 = s;
    println!("{s}");
}
```

Error

```
error[E0382]: borrow of moved value: `s`
 --> src\main.rs:5:16
  |
3 |     let s:String = String::from("AAA");
  |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
4 |     let s2 = s;
  |              - value moved here
5 |     println!("{s}");
  |                ^ value borrowed here after move
```

Ownership

Let’s see some examples:

Rust

```rust
fn print_s(s: String) {
    println!("{s}");
}
fn main() {
    let s: String = String::from("AAA");
    print_s(s);
    println!("{s}");
}
```

In this particular case, calling print_s will transfer the ownership from the variable “s” from function main, to parameter “s” from function print_s. Once function print_s is over, parameter “s” lifetime is over as well, and its content is destroyed.

Error

```
error[E0382]: borrow of moved value: `s`
 --> src\main.rs:9:16
  |
7 |     let s:String = String::from("AAA");
  |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
8 |     print_s(s);
  |             - value moved here
9 |     println!("{s}");
  |                ^ value borrowed here after move
```

Ownership

Let’s see some examples:

Rust

```rust
fn print_s(s: String) -> String {
    println!("{s}");
    return s;
}
fn main() {
    let mut s: String = String::from("AAA");
    s = print_s(s);
    println!("{s}");
}
```

Output

```
AAA
AAA
```

- One solution to the above problem is to return the value of parameter “s” from function print_s and assigned it back to the variable “s” from function main (its original owner).

Borrowing

Borrowing

- Even if ownership rules are clear, there are cases where coding under this rules is difficult. Let’s look at the following case:

Rust

```rust
fn compute_len(s: String) -> usize {
    return s.len();
}
fn main() {
    let s = String::from("123");
    let l = compute_len(s);
    println!("The length of `{s}` is {l}");
}
```

- Is this code correct ?

Borrowing

- Even if ownership rules are clear, there are cases where coding under this rules is difficult. Let's look at the following case:

Rust

```rust
fn compute_len(s: String) -> usize {
    return s.len();
}
fn main() {
    let s = String::from("123");
    let l = compute_len(s);
    println!("T
}
```

The answer is NO !

Error

```
error[E0382]: borrow of moved value: `s`
 --> src\main.rs:8:31
  |
6 |     let s = String::from("123");
  |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
7 |     let l = compute_len(s);
  |                         - value moved here
8 |     println!("The length of `{s}` is {l}");
  |                               ^ value borrowed here after move
```

Borrowing

- So how can we solve this kind of cases ?
- Most programming languages have a concept (called *reference*) that represent a valid pointer to an object of a specific type.
- In Rust, we call this form **borrowing** (the reason is that the reference does not imply change of ownership → thus we can consider that an object has been borrowed, and it will be returned to its owner).
- Just like in C or C++, a reference in Rust is denoted by the symbol & . Similarly, a dereference process can be performed with the symbol * .

Borrowing

- Let’s see how the previous code changes if we are to use references:

Rust

```rust
fn compute_len(s: String) -> usize {
    return s.len();
}
fn main() {
    let s = String::from("123");
    let l = compute_len(s);
    println!("The length of `{s}` is {l}");
}
```

Compile error

```
error[E0382]: borrow of moved value: `s`
```

Rust

```rust
fn compute_len(s: &String) -> usize {
    return s.len();
}
fn main() {
    let s = String::from("123");
    let l = compute_len(&s);
    println!("The length of `{s}` is {l}");
}
```

Compiles ok

Output

```
The length of `123` is 3
```

Borrowing

- By default, a reference in Rust is immutable (meaning you can read its value, but you can not modify it).

Rust

```rust
fn compute_len(s: &String) -> usize {
    return s.len();
}
fn main() {
    let s = String::from("123");
    let l = compute_len(&s);
    println!("The length of `{s}` is {l}");
}
```

C++

```cpp
#include <string>
size_t compute_len(const std::string& s) {
    return s.length();
}
void main() {
    const std::string s = "abc";
    auto l = compute_len(s);
    printf("The length of `%s` is %d",
            s.c_str(),(int)l);
}
```

Borrowing

- A reference in Rust can be:
  - Immutable → denoted by the usage of & (default)

```
fn compute_len(s: &String) -> usize {…}
```

  - Mutable → denoted by the usage of &mut

```
fn compute_len(s: &mut String) -> usize {…}
```

However, the question that comes into everyone's mind is:

What is the purpose of ownership if we have references ?

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref_to_s: &String = &s;
    if isEmpty(ref_to_s) {
        println!("Empty string");
    } else {
        println!("`{s}` is not empty");
    }
}
```

➡

Rust

```
           push        rbp
           sub         rsp,0E0h
           lea         rbp,[rsp+80h]
let s:String = String::from("123");
           mov         qword ptr [rbp+58h],0FFFFFFFFFFFFFFFEh
           lea         rdx,[<address of "123" string>]
           lea         rcx,[s]
           mov         qword ptr [temp_ptr_to_s],rcx
           mov         r8d,3
           call        String::from
           mov         rcx,qword ptr [temp_ptr_to_s]
let ref_to_s:&String = &s;
           mov         qword ptr [ref_to_s],rcx
```

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn ma
    le
    let ref_to_s: &String = &s;
    if isEmpty(ref_to_s) {
        println!("Empty string");
    } else {
        println!("`{s}` is not empty");
    }
}
```

Put in “rdx” register the offset where the constant string “123” lies in memory

➡

Rust

```
          push      rbp
          sub       rsp,0E0h
          lea       rbp,[rsp+80h]
let s:String = String::from("123");
          mov       qword ptr [rbp+58h],0FFFFFFFFFFFFFFFEh
          lea       rdx,[<address of "123" string>]
          lea       rcx,[s]
          mov       qword ptr [temp_ptr_to_s],rcx
          mov       r8d,3
          call      String::from
          mov       rcx,qword ptr [temp_ptr_to_s]
let ref_to_s:&String = &s;
          mov       qword ptr [ref_to_s],rcx
```

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    le
    le
    if isEmpty(ref_to_s) {
        println!("Empty string");
    } else {
        println!("`{s}` is not empty");
    }
}
```

Put in “rcx” the stack offset where variable “s” should be created.

Rust

```
          push      rbp
          sub       rsp,0E0h
          lea       rbp,[rsp+80h]
let s:String = String::from("123");
          mov       qword ptr [rbp+58h],0FFFFFFFFFFFFFFFEh
          lea       rdx,[<address of "123" string>]
          lea       rcx,[s]
          mov       qword ptr [temp_ptr_to_s],rcx
          mov       r8d,3
          call      String::from
          mov       rcx,qword ptr [temp_ptr_to_s]
let ref_to_s:&String = &s;
          mov       qword ptr [ref_to_s],rcx
```

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    le
    le
    i

    } else {
        println!("`{s}` is not empty");
    }
}
```

Make of copy (also in stack) for the “s” offset. We need to do this because there is no guarantee that RCX will not be modified when the call to **Strig::from** occurs.

Rust

```
            push        rbp
            sub         rsp,0E0h
            lea         rbp,[rsp+80h]
let s:String = String::from("123");
            mov         qword ptr [rbp+58h],0FFFFFFFFFFFFFFEh
            lea         rdx,[<address of "123" string>]
            lea         rcx,[s]
            mov         qword ptr [temp_ptr_to_s],rcx
            mov         r8d,3
            call        String::from
            mov         rcx,qword ptr [temp_ptr_to_s]
let ref_to_s:&String = &s;
            mov         qword ptr [ref_to_s],rcx
```

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref_to_s: &String = &s;
    if i
        println!("Empty string");
    } else {
        println!("`{s}` is not empty");
    }
}
```

Rust

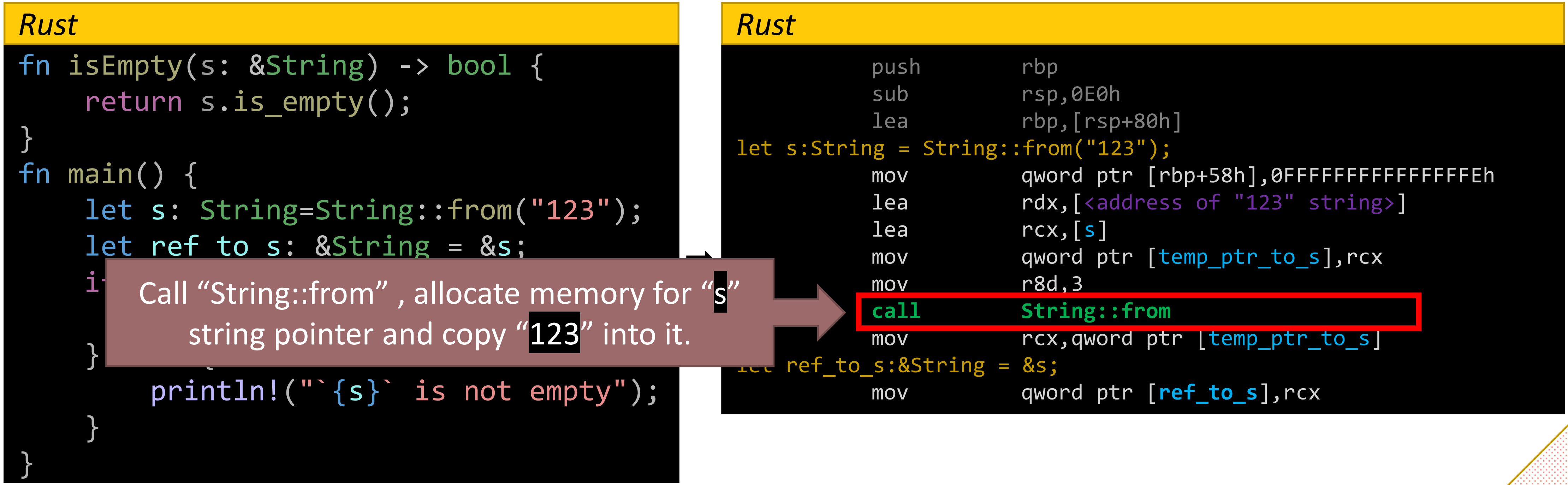
```
            push        rbp
            sub         rsp,0E0h
            lea         rbp,[rsp+80h]
let s:String = String::from("123");
            mov         qword ptr [rbp+58h],0FFFFFFFFFFFFFFFEh
            lea         rdx,[<address of "123" string>]
            lea         rcx,[s]
            mov         qword ptr [temp_ptr_to_s],rcx
            mov         r8d,3
            call        String::from
            mov         rcx,qword ptr [temp_ptr_to_s]
let ref_to_s:&String = &s;
            mov         qword ptr [ref_to_s],rcx
```

Put in r8d the size of string “123” (3 bytes)

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:



Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref to s: &String = &s;
    i

    }
        println!("`{s}` is not empty");
    }
}
```

Rust

```
            push        rbp
            sub         rsp,0E0h
            lea         rbp,[rsp+80h]
let s:String = String::from("123");
            mov         qword ptr [rbp+58h],0FFFFFFFFFFFFFFFEh
            lea         rdx,[<address of "123" string>]
            lea         rcx,[s]
            mov         qword ptr [temp_ptr_to_s],rcx
            mov         r8d,3
            call        String::from
            mov         rcx,qword ptr [temp_ptr_to_s]
let ref_to_s:&String = &s;
            mov         qword ptr [ref_to_s],rcx
```

Call “String::from” , allocate memory for “s” string pointer and copy “123” into it.

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref_to_s: &String = &s;
    if isEmpty(ref_to_s) {

    }
        println!("{s} is not empty");
    }
}
```

Rust

```
            push        rbp
            sub         rsp,0E0h
            lea         rbp,[rsp+80h]
let s:String = String::from("123");
            mov         qword ptr [rbp+58h],0FFFFFFFFFFFFFFFEh
            lea         rdx,[<address of "123" string>]
            lea         rcx,[s]
            mov         qword ptr [temp_ptr_to_s],rcx
            mov         r8d,3
            call        String::from
            mov         rcx,qword ptr [temp_ptr_to_s]
    ref_to_s:&String = &s;
            mov         qword ptr [ref_to_s],rcx
```

Restore the value of “RCX” to point to the stack offset of variable “s”

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref_to_s: &String = &s;
    if isEmpty(ref_to_s) {
        println!("Empty string");
}
```

➡

Rust

```
            push        rbp
            sub         rsp,0E0h
            lea         rbp,[rsp+80h]
let s:String = String::from("123");
            mov         qword ptr [rbp+58h],0FFFFFFFFFFFFFFFEh
            lea         rdx,[<address of "123" string>]
            lea         rcx,[s]
            mov         qword ptr [temp_ptr_to_s],rcx
            mov         r8d,3
            call        String::from
            mov         rcx,qword ptr [temp_ptr_to_s]
  ref_to_s:&String = &s;
            mov         qword ptr [ref_to_s],rcx
```

Copy “rcx” value to “ref_to_s” variable. Since RCX is the offset in stack for variable “s”, this actually makes “ref_to_s” to be a pointer to “s”.

- Let’s analyze the assembly listing …

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref_to_s: &String = &s;
    if isEmpty(ref_to_s) {
        println!("Empty string");
    } else {
        println!("`{s}` is not empty");
    }
}
```

➡

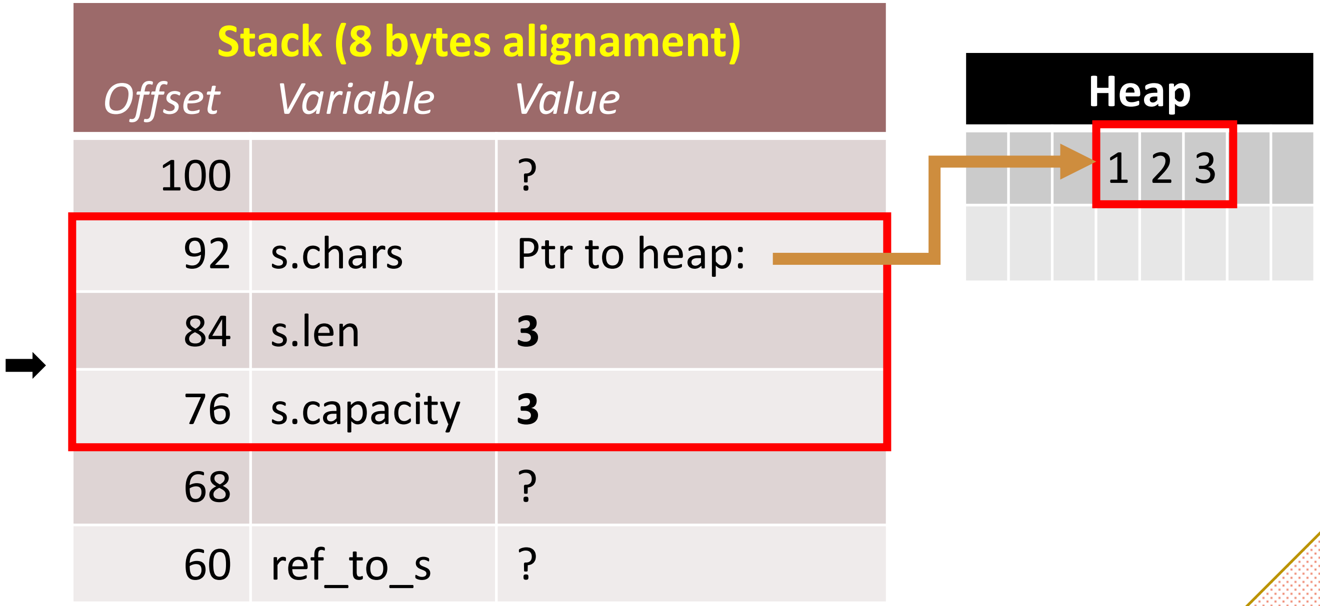
Stack (8 bytes alignment)		
Offset	*Variable*	*Value*
100		?
92	s.chars	?
84	s.len	?
76	s.capacity	?
68		?
60	ref_to_s	?

Borrowing

- Let’s discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref_to_s: &String = &s;
    if isEmpty(ref_to_s) {
        println!("Empty string");
    } else {
        println!("`{s}` is not empty");
    }
}
```



Stack (8 bytes alignament)

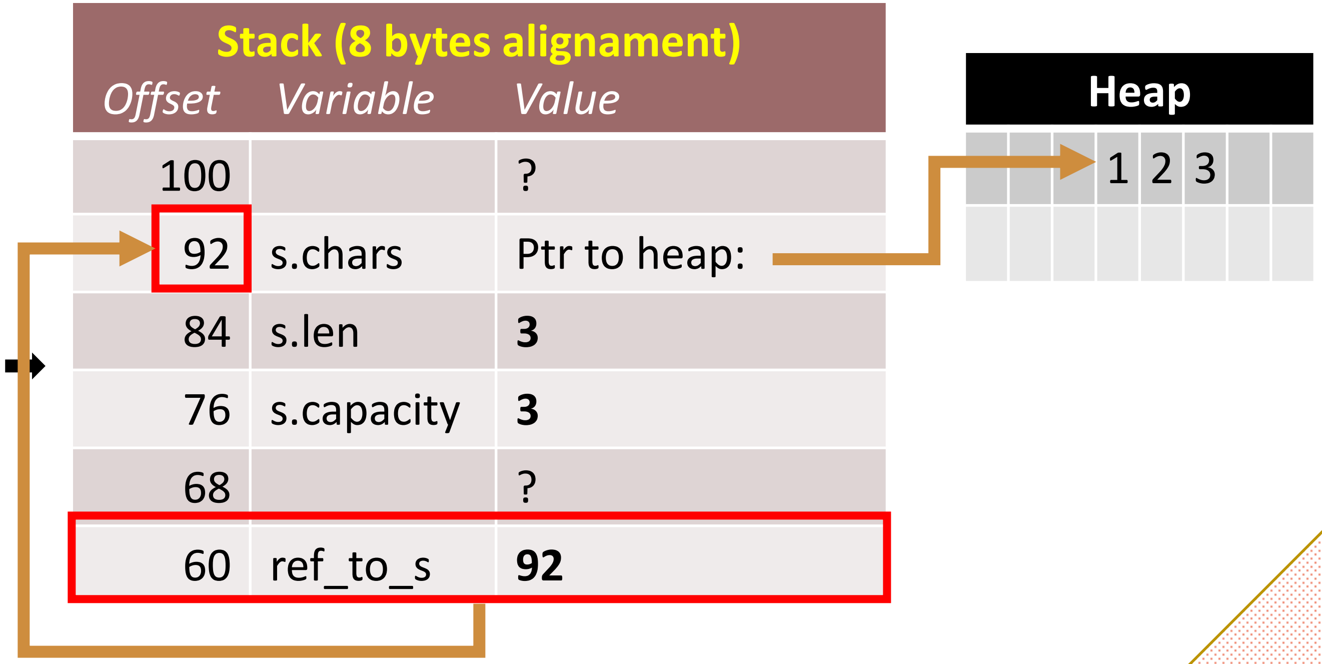
Offset	*Variable*	*Value*
100		?
92	s.chars	Ptr to heap:
84	s.len	**3**
76	s.capacity	**3**
68		?
60	ref_to_s	?

Borrowing

- Let's discuss a couple of scenarios to better understand the relationship between references and ownership:

Rust

```
fn isEmpty(s: &String) -> bool {
    return s.is_empty();
}
fn main() {
    let s: String=String::from("123");
    let ref_to_s: &String = &s;
    if isEmpty(ref_to_s) {
        println!("Empty string");
    } else {
        println!("`{s}` is not empty");
    }
}
```



Stack (8 bytes alignament)		
Offset	*Variable*	*Value*
100		?
92	s.chars	Ptr to heap:
84	s.len	**3**
76	s.capacity	**3**
68		?
60	ref_to_s	**92**

Heap

			1	2	3		

Borrowing

- In Rust, a reference (a borrow value) is a pointer to the original object (similar to how C/C++ treat references)
- This means that as long as the original object is valid, a reference will be valid as well

Because of this Rust has several rules related to references:

1. **At one given moment of time, there can be only one mutable reference to an object**
2. **At any given moment of time, there can be multiple immutable references to an object**
3. **Case 1 and 2 are exclusive meaning that if you have a mutable reference, you can not have another immutable one and vice-versa.**

Borrowing

Let’s see some cases that reflect how these references work.

1. Multiple immutable references & immutable object

Rust

```rust
fn main() {
    let s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
}
```

Output

```
123,123,123,123
```

Compiles ok

Borrowing

Let’s see some cases that reflect how these references work.

2. One mutable reference & immutable object

Rust

```
fn main() {
    let s: String = String::from("123");
    let mut_ref_to_s: &mut String = &mut s;
    println!("{s},{mut_ref_to_s}");
}
```

Error

```
error[E0596]: cannot borrow `s` as mutable, as it is not declared as mutable
 --> src\main.rs:4:36
  |
3 |     let s:String = String::from("123");
  |         - help: consider changing this to be mutable: `mut s`
4 |     let mut_ref_to_s:&mut String = &mut s;
  |                                    ^^^^^^ cannot borrow as mutable
```

Borrowing

Let’s see some cases that reflect how these references work.

3. One mutable reference & mutable object

Rust

```
fn main() {
    let mut s: String = String::from("123");
    let mut_ref_to_s: &mut String = &mut s;
    println!("{s},{mut_ref_to_s}");
}
```

Error

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed as mutable
 --> src\main.rs:5:16
  |
4 |    let mut_ref_to_s:&mut String = &mut s;
  |                                   ------ mutable borrow occurs here
5 |    println!("{s},{mut_ref_to_s}");
  |               ^   ------------ mutable borrow later used here
  |               |
  |               immutable borrow occurs here
```

Borrowing

Let’s see some cases that reflect how these references work.

3. One mutable reference & mutable object

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let mut_ref_to_s:&mut String = &mut s;
    println!("{s},{mut_ref_to_s}");
}
```

Error

```
error[E0502]: cannot borrow `s` as immutable because it is als
 --> src\main.rs:5:16
  |
4 |     let mut_ref_to_s:&mut String = &mut s;
  |                                    ------ mutable borrow o
5 |     println!("{s},{mut_ref_to_s}");
  |               ^   ------------ mutable borrow later used here
  |               |
  |               immutable borrow occurs here
```

Let’s explain what happens in this case:

1. When **println!** macro is trying to use the object ***“s”***, it tries to create an immutable reference.
2. However, at this moment of the execution there already is an mutable reference available (namely **“mut_ref_to_s”**)
3. As such, another one can not exists, and an error will be thrown.

Borrowing

Let’s see some cases that reflect how these references work.

3. One mutable reference & mutable object

Rust

```
fn main() {
    let mut s: String = String::from("123");
    let mut_ref_to_s: &mut String = &mut s;
    println!("{mut_ref_to_s}");
    println!("{s}");
}
```

Compiles ok

Why is this example working and using both variables in a single println doesn’t ?

Output

```
123
123
```

Borrowing

Let’s see some cases that reflect how these references work.

3. One mutable reference & mutable object

Rust

```
1  fn main() {
2      let mut s: String = String::from("123");
3      let mut_ref_to_s: &mut String = &mut s;
4      println!("{mut_ref_to_s}");
5      println!("{s}");
6  }
```

Lifetime of **mut_ref_to_s**

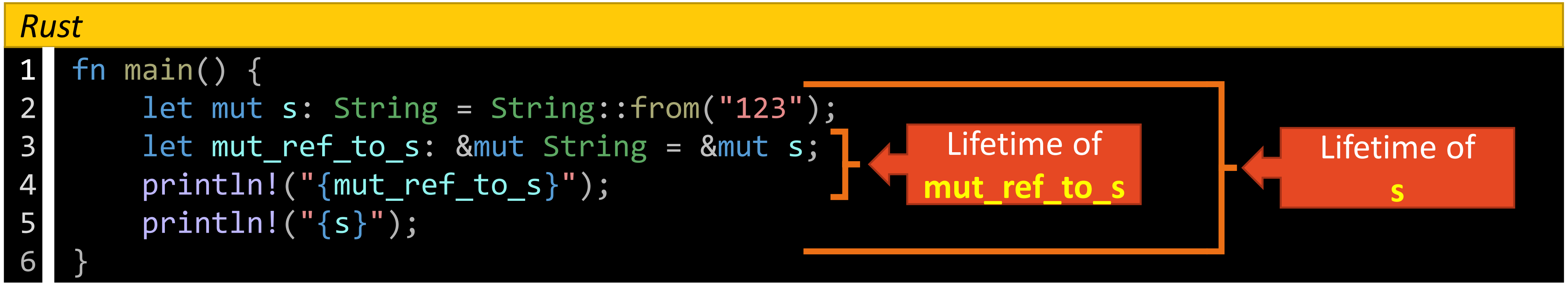
Let’s analyze this example:

- Variable “mut_ref_to_s” is created at line 3 and after line 4 it is no longer needed (used). As such it is destroy after line 4. We can say that its lifetime consists in 2 lines (3 and 4).
- When line 4 gets executed, there is only one mutable reference to object “s” – thus the rules are not broken, and no error is thrown

Borrowing

Let’s see some cases that reflect how these references work.

3. One mutable reference & mutable object



Rust

```
1  fn main() {
2      let mut s: String = String::from("123");
3      let mut_ref_to_s: &mut String = &mut s;
4      println!("{mut_ref_to_s}");
5      println!("{s}");
6  }
```

Lifetime of **mut_ref_to_s**

Lifetime of **s**

Let’s analyze this example:

- Variable “mut_ref_to_s” is created at line 3 and after line 4 it is no longer needed (used). As such it is destroy after line 4. We can say that its lifetime consists in 2 lines (3 and 4).
- When line 4 gets executed, there is only one mutable reference to object “s” – thus the rules are not broken, and no error is thrown
- Variable “s” has a lifetime that starts on line 2 and ends on line 5. However, when line 5 gets executed, and a new immutable reference is created, variable “mut_ref_to_s” has already been discarded and as such we would only have one reference and program compiles.

Borrowing

Let’s see some cases that reflect how these references work.

3. One mutable reference & mutable object

Rust

```
fn main() {
    let mut s: String = String::from("123");
    let mut_ref_to_s: &mut String = &mut s;
    println!("{s}");
    println!("{mut_ref_to_s}");
}
```

Following the previous logic, if we reverse the order of the println! macro, we change the lifetime of variable “s” and “mut_ref_to_s” and an error will be triggered !

Error

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed as mutable
 --> src\main.rs:5:16
  |
4 |     let mut_ref_to_s:&mut String = &mut s;
  |                                    ------ mutable borrow occurs here
5 |     println!("{s}");
  |                ^ immutable borrow occurs here
6 |     println!("{mut_ref_to_s}");
  |                ------------ mutable borrow later used here
  |
```

Borrowing

Let’s see some cases that reflect how these references work.

4. Multiple immutable reference & mutable object

Rust

```rust
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
}
```

Output

```
123,123,123,123
```

Compiles ok

Borrowing

Let’s see some cases that reflect how these references work.

4. Multiple immutable reference & mutable object

Rust

```
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
    s += "A larger string";
    println!("{s}");
}
```

Output

```
123,123,123,123
123A larger string
```

Compiles ok

Borrowing

Let’s see some cases that reflect how these references work.

4. Multiple immutable reference & mutable object

Rust

```rust
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
}
```

What will happen in this case ?

Borrowing

Let’s see some cases that reflect how these references work.

4. Multiple immutable reference & mutable object

Rust

```
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
    s += "A larger string";
    println!("{s},{ref_to_s_1},{r
}
```

Error

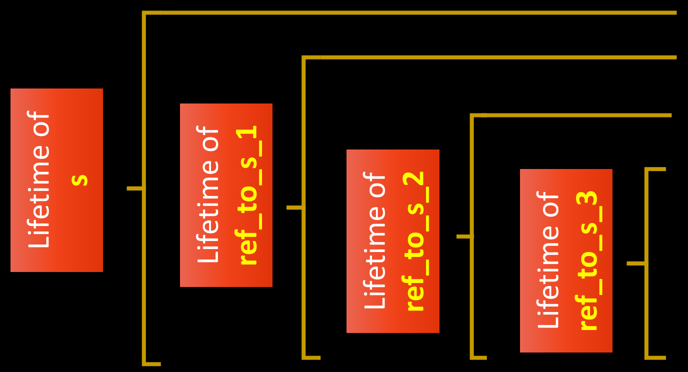
```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src\main.rs:8:5
  |
4 |     let ref_to_s_1:&String = &s;
  |                              -- immutable borrow occurs here
...
8 |     s += "A larger string";
  |     ^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
9 |     println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
  |                    ---------- immutable borrow later used here
```

Borrowing

Let’s see some cases that reflect how these references work.

4. Multiple immutable reference & mutable object

Rust



```
fn main() {
    1 let mut s:String = String::from("123");
    2 let ref_to_s_1:&String = &s;
    3 let ref_to_s_2:&String = &s;
    4 let ref_to_s_3:&String = &s;
    5 println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
    6 s += "A larger string";
    7 println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
}
```

When reaching line 6, a mutable reference is needed to perform that assignment (that changes the content of object “s”). Since at line 6 there already are 3 immutable references, the compiler will fail.

Borrowing

When analyzing the previous examples, there are some questions that need to be answered:

```
s += "A larger string";
```

1. Why the previous assignment implies creating a mutable reference ?
2. Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

To answer this question, let’s look on how “+=“ operator is defined !

Rust (source String.rs)

```
/// Implements the `+=` operator for appending to a `String`.
/// This has the same behavior as the [`push_str`][String::push_str] method.
#[cfg(not(no_global_oom_handling))]
#[stable(feature = "stringaddassign", since = "1.12.0")]
impl AddAssign<&str> for String {
    #[inline]
    fn add_assign(&mut self, other: &str) {
        self.push_str(other);
    }
}
```

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

To answer this question, let’s look on how “+=“ operator is defined !

Rust (source String.rs)*

```
/// Implements the `+=` operator for appending to a `String`.
/// This has the same behavior as the [`push_str`][String::push_str] method.
#[cfg(not(no_global_oom_handling))]
#[stable(feature = "stringaddassign", since = "1.12.0")]
impl AddAssign<&str> for String {
    #[inline]
    fn add_assign(&mut self, other: &str) {
        self.push_str(other);
    }
}
```

For the moment let’s ignore the attributes (lines that start with `#` (pound sign) character) and the whole `impl` structure as we will discuss this later.

Let’s focus on the `add_assign` method instead !

** Implementation of operator += for String class may vary in time (from version to version)*

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

To answer this question, let’s look on how “+=“ operator is defined !

Rust (source String.rs)*

```
/// Implements the `+=` operator for appending to a `String`.
/// This has the same behavior as the [`push_str`][String::push_str] method.
#[cfg(not(no_global_oom_handling))]
#[stable(feature = "stringaddassi
impl AddAssign<&str> for String {
    #[inline]
    fn add_assign(&mut self,
        self.push_str(other);
    }
}
```

Notice that the first parameter of this function is a mutable reference (denoted with the name self). To simply put it, **self** is the equivalent of C++ **this** and must be provided to every non-static method calls. And **self** is a reference (either mutable or immutable) !

** Implementation of operator += for String class may vary in time (from version to version)*

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

This means that the following line:

```
s += "A larger string";
```

is equivalent to the next one:

```
add_assign(&mut s , "A larger string");
```

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

Consider that any class non-static method implies creating a reference

- **A mutable reference** if that method changes something in the class
- **An immutable reference** if that method only reads information from the class

These references will respect references rules (either only one mutable or multiple immutable).

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

Let’s consider the following example:

Rust

```rust
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let l = s.len();
    println!("{s},{ref_to_s_1},{ref_to_s_2}, {l}");
}
```

Compiles ok

Output

```
123,123,123, 3
```

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

Let’s consider the following example:

Rust

```
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let l = s.len();
    println!("{s},{ref_to_s_1},{ref_to_s_2}, {l}");
}
```

What happens when we call `s.len()` ?

Compiles ok

Output

```
123,123,123, 3
```

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

Let's consider the following example:

Rust

```rust
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let l = s.len();
    println!("{s},{ref_to_s_1},{ref_to_s_2}, {l}");
}
```

What happens when we call `s.len()` ?

Rust (String.rs)

```rust
pub fn len(&self) -> usize {
    self.vec.len()
}
```

Compiles ok

Output

```
123,123,123, 3
```

`len` method requires an **immutable reference**. Since at that point we only have two other immutable reference, the code is safe and will be allowed to compile.

Borrowing

1. Why `s += "A larger string";` implies creating a mutable reference ?

Let's consider the following example:

Rust

```
fn main() {
    let mut s: String = String::from("123");
    let mut ref to s: &mut String = &mut s;
    let l = s.len();
    println!("{s},{mut_ref_to_s},{l}");
}
```

Error

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed as mutable
 --> src\main.rs:5:13
  |
4 |     let mut_ref_to_s:&mut String = &mut s;
  |                                    ------ mutable borrow occurs here
5 |     let l = s.len();
  |             ^^^^^^^ immutable borrow occurs here
6 |     println!("{s},{mut_ref_to_s},{l}");
  |               ------------ mutable borrow later used here
```

len method requires an **immutable reference**. However, when s.len() is compiled, an immutable reference is required, but … there already exists a mutable reference thru the variable "***mut_ref_to_s***". Since there can not be a mutable and immutable references at the same time, the compiler will throw an error.

Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```rust
fn main() {
    let mut s: String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2},{ref_to_s_3}");
}
```

Compile error

```
error[E0502]: cannot borrow
`s` as mutable because it is
also borrowed as immutable
```

Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2…
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2…
}
```

Let’s analyze the stack of the program, if compilation would have been successful.

Stack (8 bytes alignament)

Offset	*Variable*	*Value*
100		?
92		?
84		?
76		?
68		?
60		?
52		?
44		?

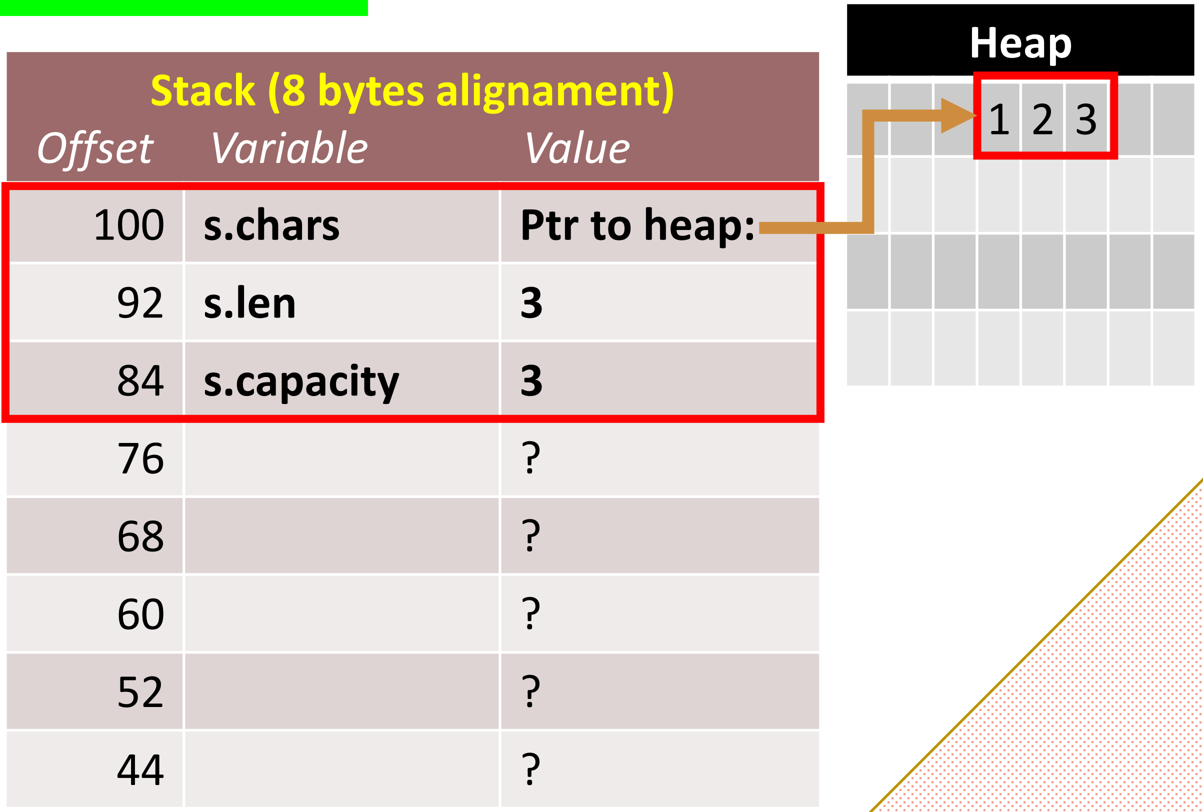
Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2…
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2…
}
```

Let’s analyze the stack of the program, if compilation would have been successful.



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	**s.chars**	**Ptr to heap:**
92	**s.len**	**3**
84	**s.capacity**	**3**
76		?
68		?
60		?
52		?
44		?

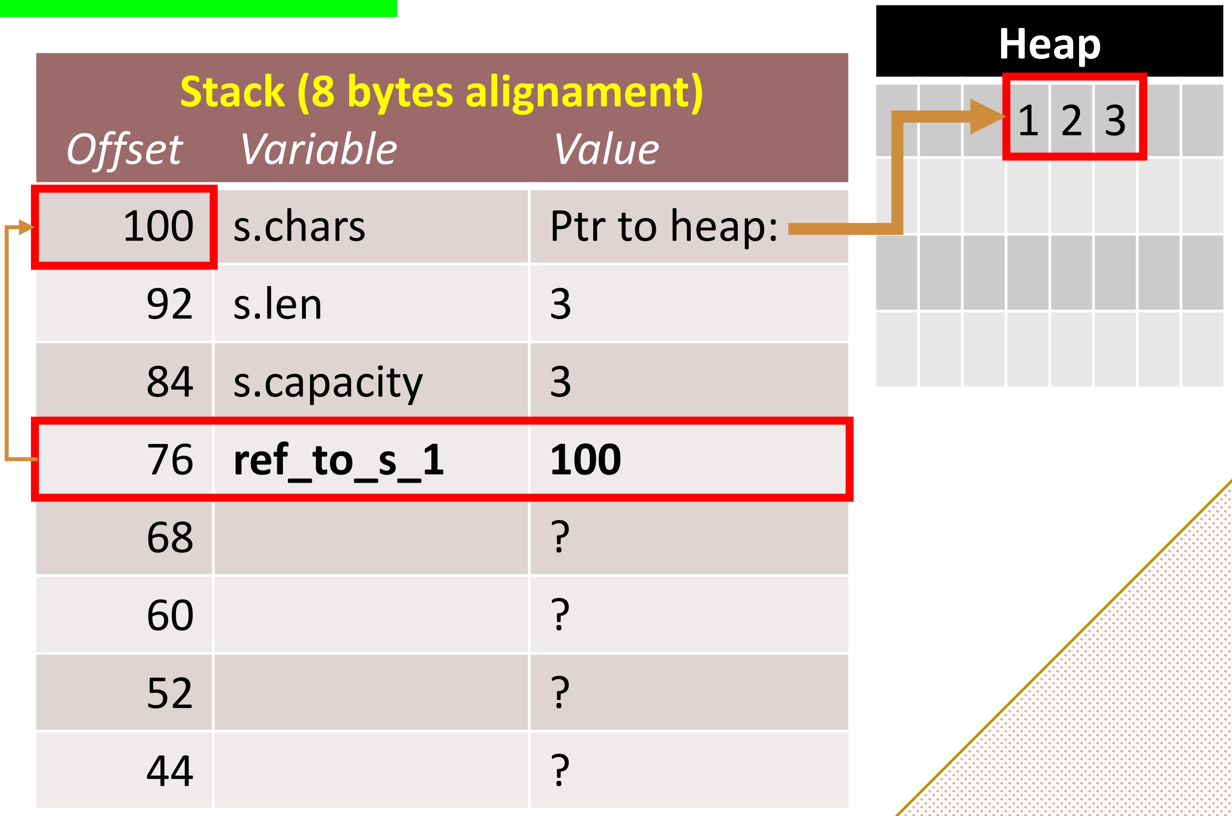
Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2…
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2…
}
```

Let’s analyze the stack of the program, if compilation would have been successful.



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	s.chars	Ptr to heap:
92	s.len	3
84	s.capacity	3
76	**ref_to_s_1**	**100**
68		?
60		?
52		?
44		?

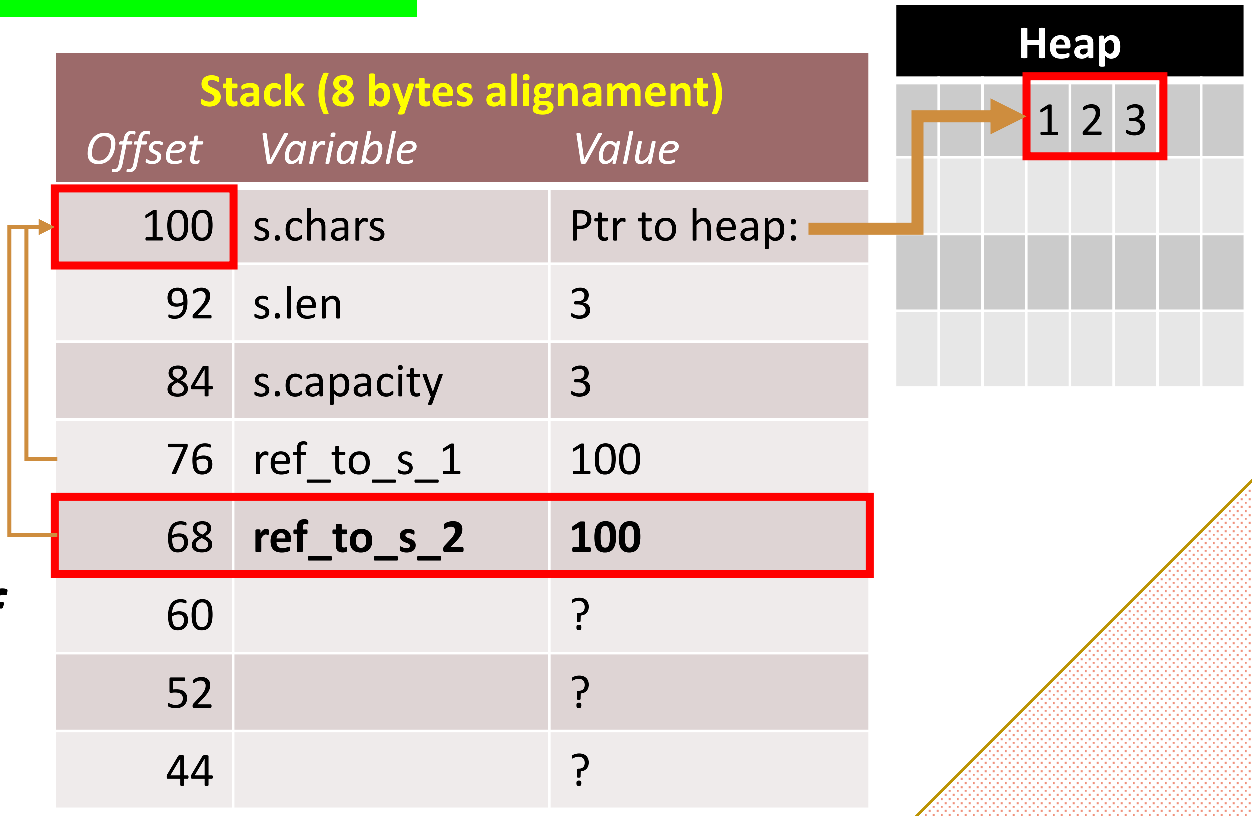
Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2…
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2…
}
```

Let’s analyze the stack of the program, if compilation would have been successful.



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	s.chars	Ptr to heap:
92	s.len	3
84	s.capacity	3
76	ref_to_s_1	100
68	**ref_to_s_2**	**100**
60		?
52		?
44		?

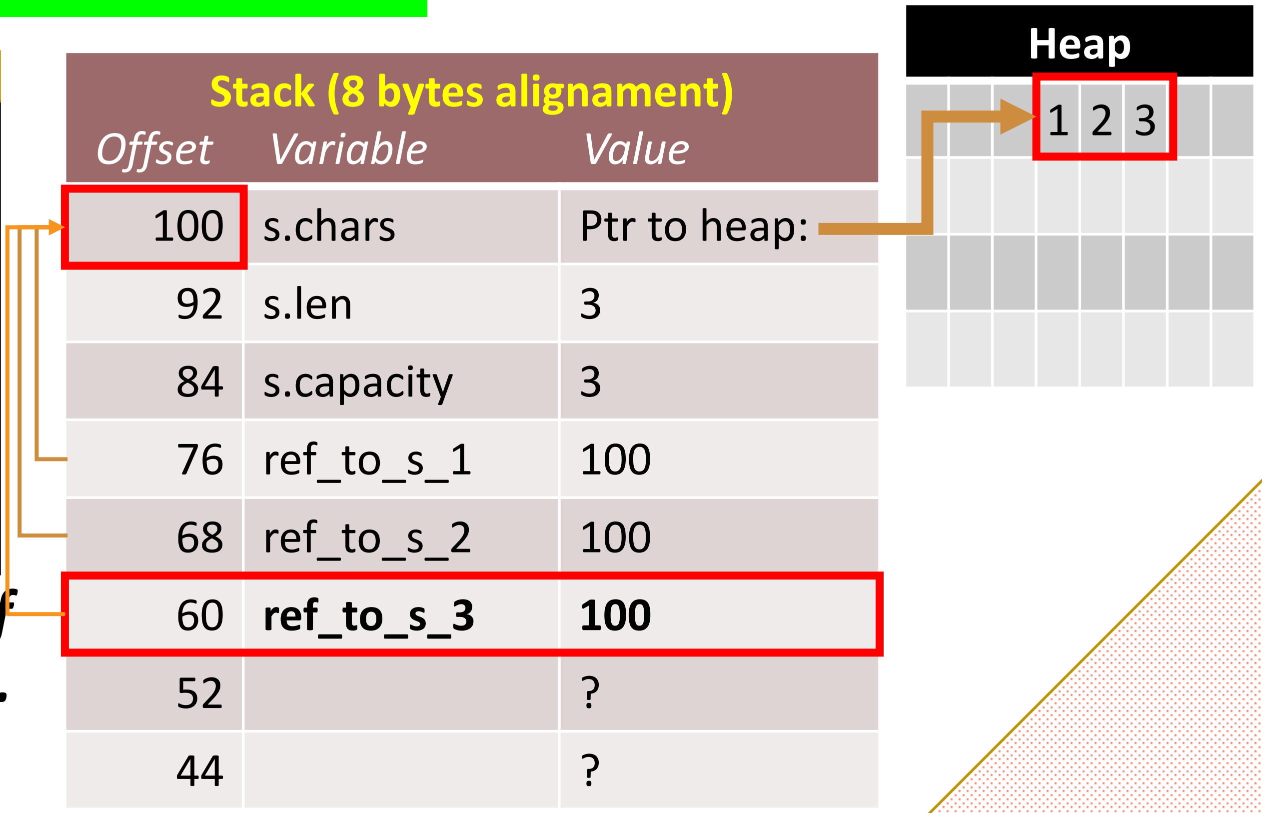
Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2…
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2…
}
```

Let’s analyze the stack of the program, if compilation would have been successful.



Stack (8 bytes alignament)

Offset	*Variable*	*Value*
100	s.chars	Ptr to heap:
92	s.len	3
84	s.capacity	3
76	ref_to_s_1	100
68	ref_to_s_2	100
60	**ref_to_s_3**	**100**
52		?
44		?

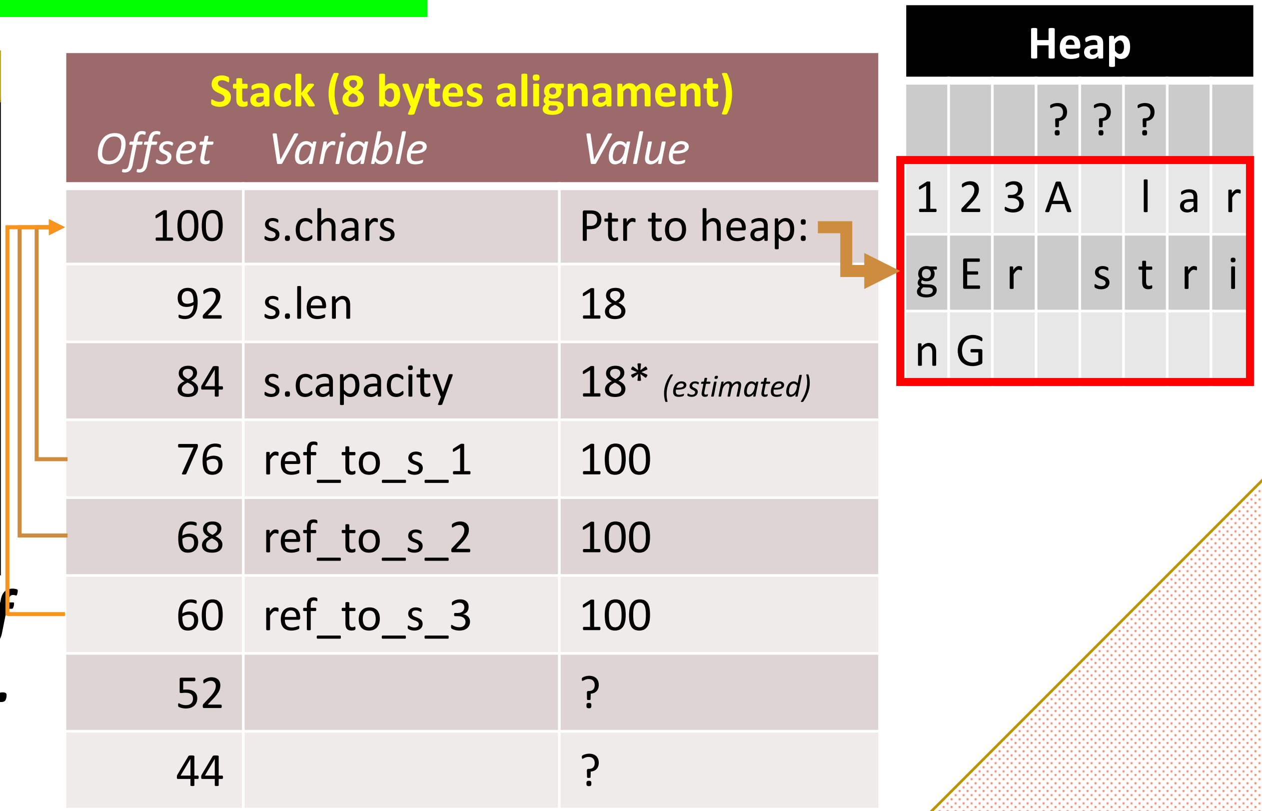
Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```rust
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2…
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2…
}
```

Let’s analyze the stack of the program, if compilation would have been successful.



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	s.chars	Ptr to heap:
92	s.len	18
84	s.capacity	18* *(estimated)*
76	ref_to_s_1	100
68	ref_to_s_2	100
60	ref_to_s_3	100
52		?
44		?

Heap

			?	?	?		
1	2	3	A		l	a	r
g	E	r		s	t	r	i
n	G						

Borrowing

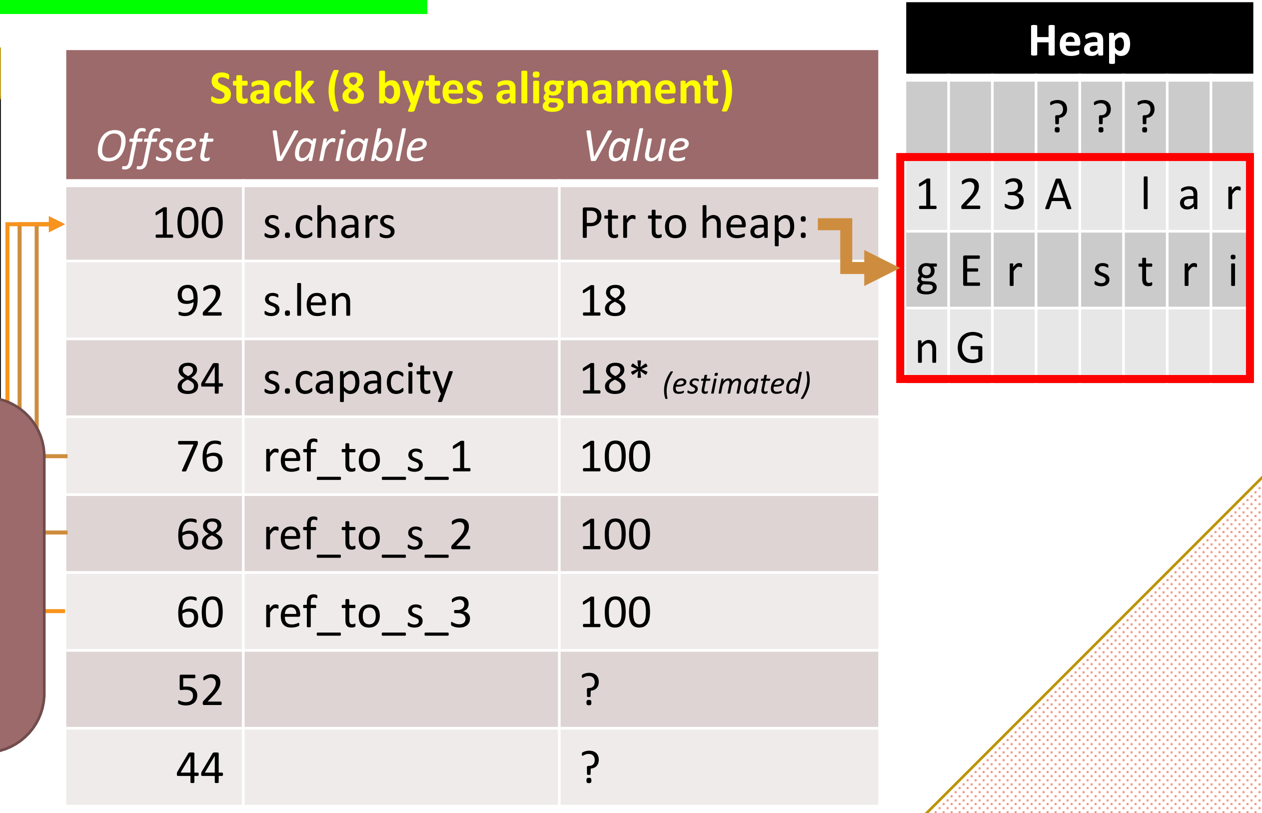
2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2
}
```

Has anything changed for one of the immutable references ?

The answer is NO. All 3 **immutable** references point to the same object (and could access that object just like they could before s.chars data member was changed).



Stack (8 bytes alignament)

Offset	*Variable*	*Value*
100	s.chars	Ptr to heap:
92	s.len	18
84	s.capacity	18* *(estimated)*
76	ref_to_s_1	100
68	ref_to_s_2	100
60	ref_to_s_3	100
52		?
44		?

Heap

			?	?	?		
1	2	3	A		l	a	r
g	E	r		s	t	r	i
n	G						

Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Rust

```
fn main() {
    let mut s:String = String::from("123");
    let ref_to_s_1: &String = &s;
    let ref_to_s_2: &String = &s;
    let ref_to_s_3: &String = &s;
    println!("{s},{ref_to_s_1},{ref_to_s_2…
    s += "A larger string";
    println!("{s},{ref_to_s_1},{ref_to_s_2…
}
```

So … if this code is safe, why does not Rust allow it ?

Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

To answer the last question, we need to look into another feature of Rust, called ***slices*** !

Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

Let’s analyze the following code:

Rust

```rust
fn main() {
    let s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    println!("{s},{slice_of_s}");
}
```

Output

```
AABBBCC,BBB
```

Slices are very similar to ***std::string_view*** / ***std::u8string_view*** from C++ (in the sense that they hold a pointer and a size).

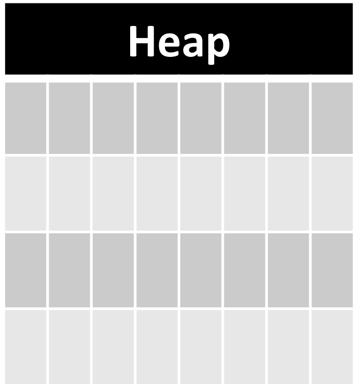
Borrowing

Let’s see how the stack looks like in this case:

Rust

```
fn main() {
    let s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    println!("{s},{slice_of_s}");
}
```

Stack (8 bytes alignment)		
Offset	*Variable*	*Value*
100		?
92		?
84		?
76		?
68		?
60		?
52		?
44		?

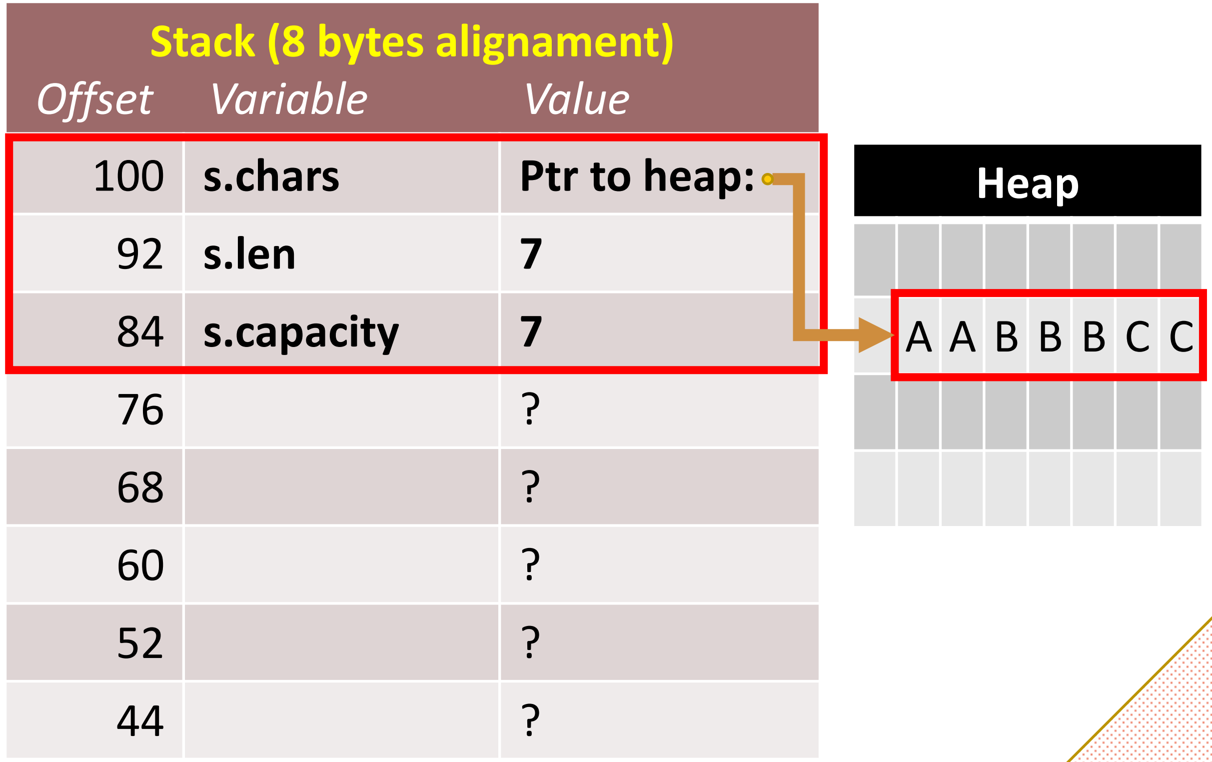


Borrowing

Let’s see how the stack looks like in this case:

Rust

```
fn main() {
    let s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    println!("{s},{slice_of_s}");
}
```



Stack (8 bytes alignament)

Offset	*Variable*	*Value*
100	**s.chars**	**Ptr to heap:**
92	**s.len**	**7**
84	**s.capacity**	**7**
76		?
68		?
60		?
52		?
44		?

Heap

A	A	B	B	B	C	C

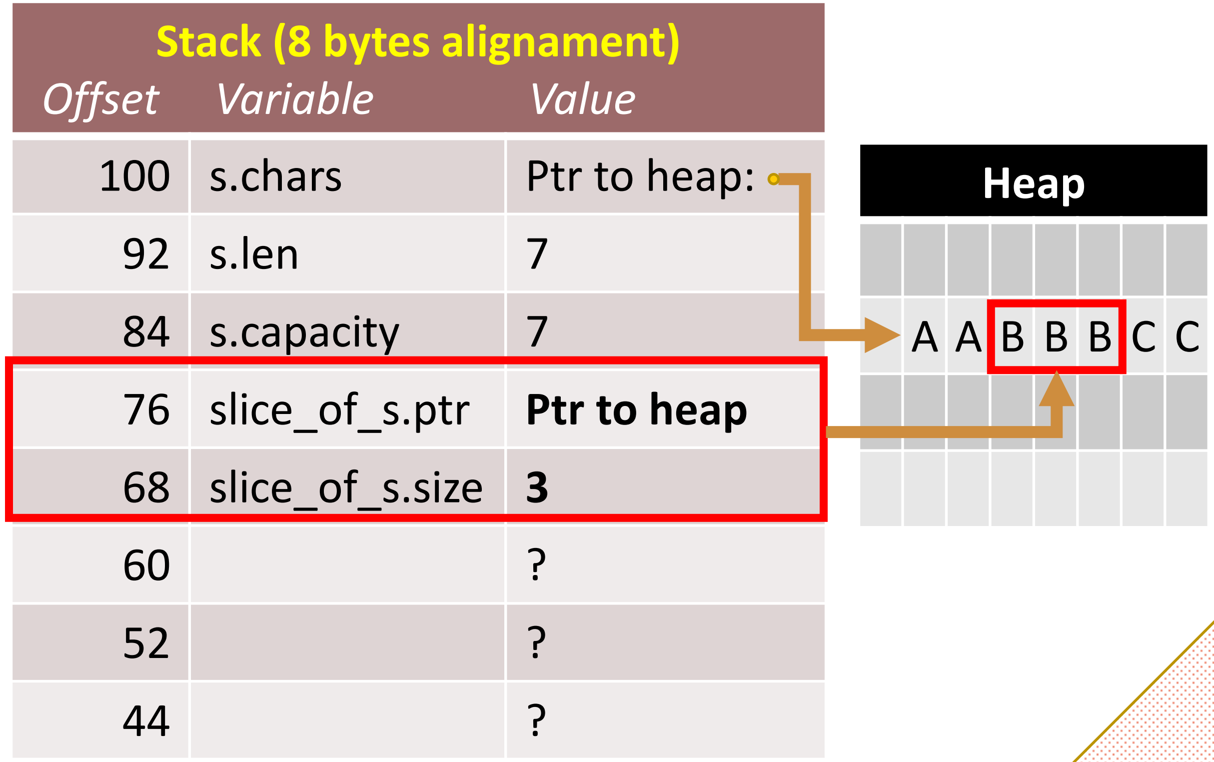
Borrowing

Let’s see how the stack looks like in this case:

Rust

```rust
fn main() {
    let s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    println!("{s},{slice_of_s}");
}
```

This part is in particular important: Notice that “**slice_of_s**” maintains a pointer within a memory that “s” owns.



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	s.chars	Ptr to heap:
92	s.len	7
84	s.capacity	7
76	slice_of_s.ptr	**Ptr to heap**
68	slice_of_s.size	**3**
60		?
52		?
44		?

Heap

	A	A	B	B	B	C	C

Borrowing

So … what will happen if we change the previous code in the following way (**let's assume that this code will compile under Rust** and evaluate what execution will do to the stack):

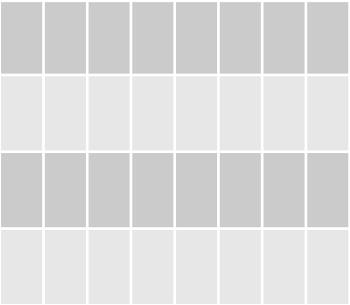
Rust

```
fn main() {
     let mut s: String = String::from("AABBBCC");
     let slice_of_s = &s[2..5];
     s.push_str("DDDDD");
     println!("{s},{slice_of_s}");
}
```

Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100		?
92		?
84		?
76		?
68		?
60		?
52		?
44		?

Heap

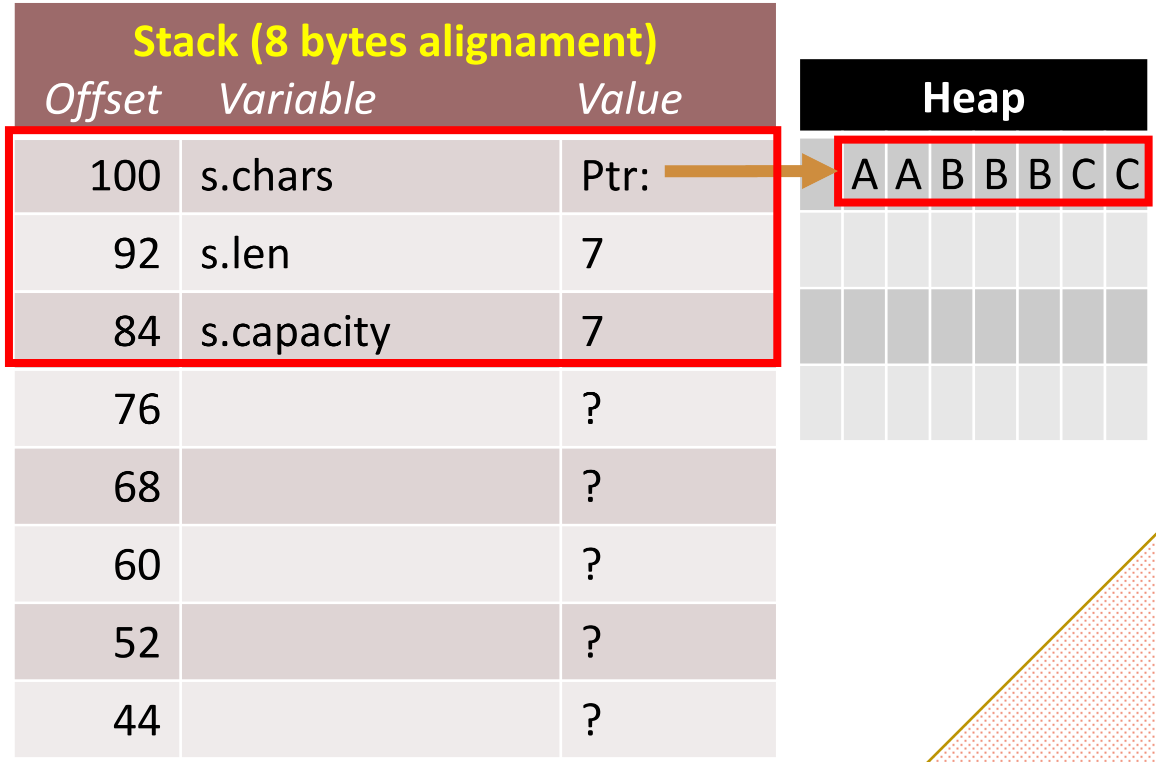


Borrowing

So … what will happen if we change the previous code in the following way (**let's assume that this code will compile under Rust** and evaluate what execution will do to the stack):

Rust

```
fn main() {
    let mut s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    s.push_str("DDDDD");
    println!("{s},{slice_of_s}");
}
```



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	s.chars	Ptr:
92	s.len	7
84	s.capacity	7
76		?
68		?
60		?
52		?
44		?

Heap

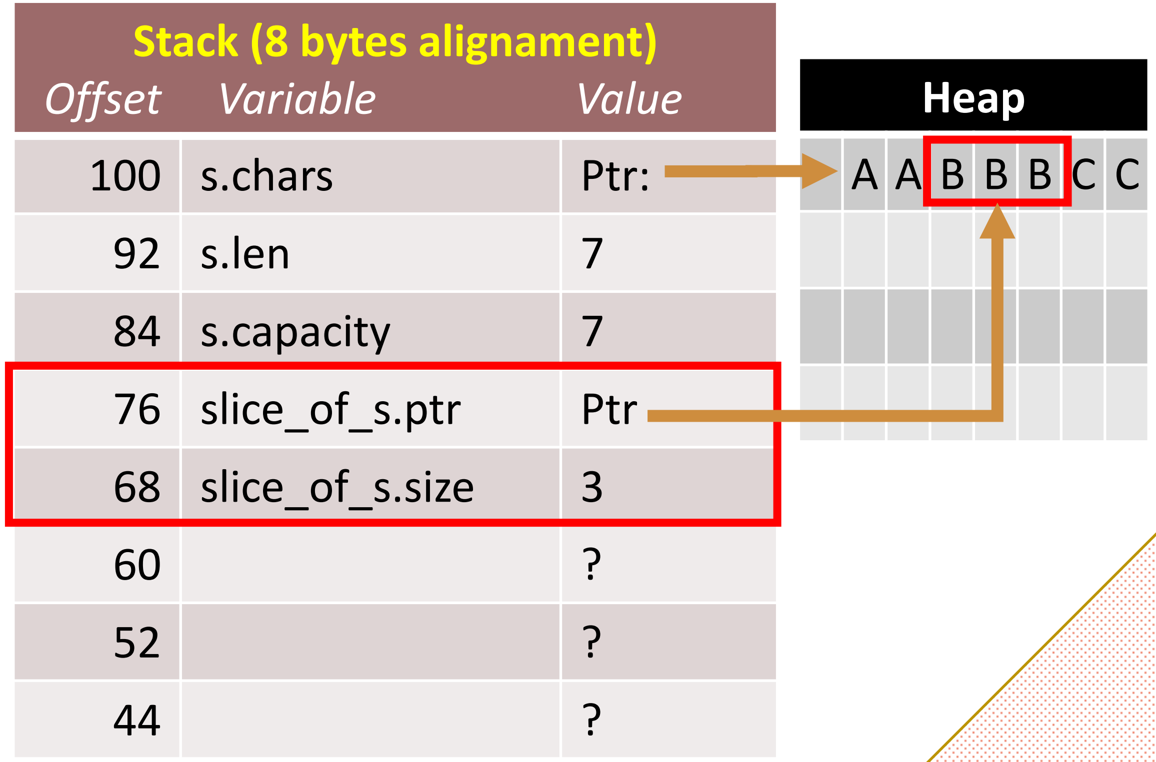
	A	A	B	B	B	C	C

Borrowing

So … what will happen if we change the previous code in the following way (**let's assume that this code will compile under Rust** and evaluate what execution will do to the stack):

Rust

```
fn main() {
    let mut s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    s.push_str("DDDDD");
    println!("{s},{slice_of_s}");
}
```



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	s.chars	Ptr:
92	s.len	7
84	s.capacity	7
76	slice_of_s.ptr	Ptr
68	slice_of_s.size	3
60		?
52		?
44		?

Heap

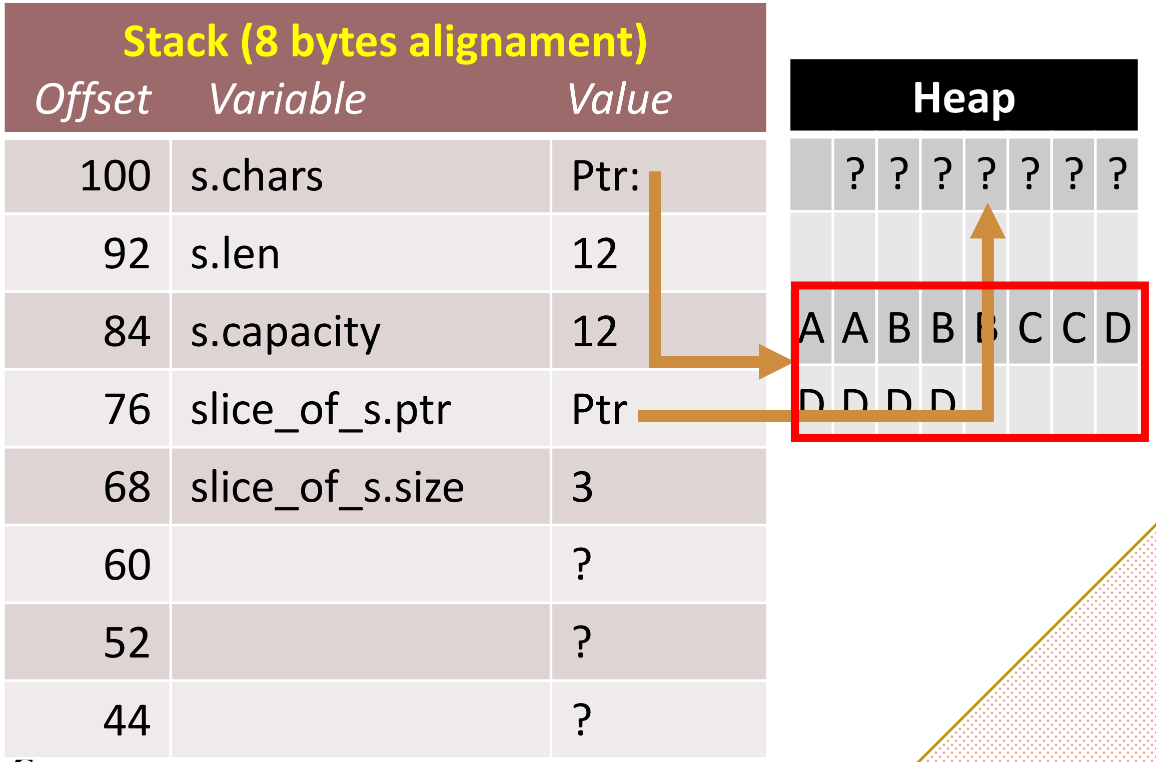
A	A	B	B	B	C	C

Borrowing

So … what will happen if we change the previous code in the following way (**let’s assume that this code will compile under Rust** and evaluate what execution will do to the stack):

Rust

```rust
fn main() {
    let mut s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    s.push_str("DDDDD");
    println!("{s},{slice_of_s}");
}
```



Stack (8 bytes alignment)

Offset	*Variable*	*Value*
100	s.chars	Ptr:
92	s.len	12
84	s.capacity	12
76	slice_of_s.ptr	Ptr
68	slice_of_s.size	3
60		?
52		?
44		?

Heap

	?	?	?	?	?	?	?
A	A	B	B	B	C	C	D
D	D	D	D				

When the execution reaches this line, a new memory will be allocated for “s” and the previous one will be deallocated. The result is that we get a dangling pointer from “slice_of_s that points to the original memory allocated from “s” that is currently invalid.

Borrowing

In reality, Rust will not compile the next code and will throw the following error:

Rust

```
fn main() {
    let mut s: String = String::from("AABBBCC");
    let slice_of_s = &s[2..5];
    s.push_str("DDDDD");
    println!("{s},{slice_of_s}");
}
```

Error

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src\main.rs:6:5
  |
5 |     let slice_of_s = &s[2..5];
  |                       - immutable borrow occurs here
6 |     s.push_str("DDDDD");
  |     ^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
7 |     println!("{s},{slice_of_s}");
  |                    ---------- immutable borrow later used here
```

Borrowing

2. Now let's tackle the second question: Immutable references can not change the value of an object. If the value of the object changes, why can’t we have immutable references at that time ?

So … returning to the original question, we can not have immutable references and a mutable one at the same time, **because there are scenarios (like using a slice) that could lead to a dangling pointer** !

Remarks: Slices are considered references as well !

Reborrowing

Reborrowing

Let’s evaluate the following scenario:

Rust

```rust
fn goo(y: &mut String) {
    y.push('G');
}
fn foo(x: &mut String) {
    x.push('-');
    goo(x);
    x.push('-');
}
fn main() {
    let mut s = String::from("ABC");
    let mut_ref_to_s = &mut s;
    foo(mut_ref_to_s);
    mut_ref_to_s.push('!');
    println!("{s}");
}
```

Output

```
ABC-G-!
```

Reborrowing

Let’s evaluate the following scenario:

Rust

```rust
fn goo(y: &mut String) {
    y.push('G');
}
fn foo(x: &mut String) {
    x.push('-');
    goo(x);
    x.push('-');
}
fn main() {
    let mut s = String::from("ABC");
    let mut_ref_to_s = &mut s;
    foo(mut_ref_to_s);
    mut_ref_to_s.push('!');
    println!("{s}");
}
```

The ownership rules clearly state that if we pass an object to a function, we lose the ownership and as such we can not use that object anymore after that.

So … why is this code WORKING ?

Reborrowing

The previous code works because Rust uses a different mechanism when passing references to a function than when it passes an object.

While an immutable reference is **COPY** *and a mutable one is* **MOVE***, when passing a mutable reference to a method, Rust uses another concept called REBORROWING.*

Reborrowing allows you to temporarily transfer access to a mutable reference while keeping the original reference valid for later use.

Reborrowing

Let’s analyze the previous code from this new perspective:

Rust

```rust
fn goo(y: &mut String) {
    y.push('G');
}
fn foo(x: &mut String) {
    x.push('-');
    goo(x);
    x.push('-');
}
fn main() {
    let mut s = String::from("ABC");
    let mut_ref_to_s = &mut s;
    foo(mut_ref_to_s);
    mut_ref_to_s.push('!');
    println!("{s}");
}
```

Reborrowing

Let’s analyze the previous code from this new perspective:

Rust

```rust
fn goo(y: &mut String) {
    y.push('G');
}
fn foo(x: &mut String) {
    x.push('-');
    goo(x);
    x.push('-');
}
fn main() {
    let mut s = String::from("ABC");
    let mut_ref_to_s = &mut s;
    foo(mut_ref_to_s);
    mut_ref_to_s.push('!');
    println!("{s}");
}
```

We create **mut_ref_to_s** (only ONE mutable reference towards variable s)

Reborrowing

Let’s analyze the previous code from this new perspective:

Rust

```
fn goo(y: &mut String) {
    y.push('G');
}
fn foo(x: &mut String) {
    x.push('-');
    goo(x);
    x.push('-');
}
fn main() {
    let mut s = String::from("ABC");
    let mut_ref_to_s = &mut s;
    foo(mut_ref_to_s);
    mut_ref_to_s.push('!');
    println!("{s}");
}
```

We reborrow **mut_ref_to_s** to function ***foo*** where it will be used as “x” (we can look at this as we temporary create an alias for mut_ref_to_s that is called x, that can only be used within the function ***foo***).

The rules of ownership are still valid, as we know for sure that **mut_ref_to_s** can only be used after function foo ends

Reborrowing

Let’s analyze the previous code from this new perspective:

Rust

```
fn goo(y: &mut String) {
    y.push('G');
}
fn foo(x: &mut String) {
    x.push('-');
    goo(x);
    x.push('-');
}
fn main() {
    let mut s = String::from("ABC");
    let mut_ref_to_s = &mut s;
    foo(mut_ref_to_s);
    mut_ref_to_s.push('!');
    println!("{s}");
}
```

A similar logic applies in this case as well, we know that “x” from foo will only be available after function goo ends and as such the ownership rules are not broken.

What should be noticed in this example is that at any given time, there is only **ONE** available to use mutable reference towards object “s”

Reborrowing

However, placing a reference in a structure follows the ownership rules:

Rust

```rust
struct MutRef<'a> { value: &'a mut String }

fn goo(x: MutRef) {
    x.value.push('G');
}
fn foo(x: MutRef) {
    x.value.push('-');
    goo(x);
    x.value.push('-');
}
fn main() {
    let mut s = String::from("ABC");
    let mut_ref_to_s = MutRef { value: &mut s };
    foo(mut_ref_to_s);
    mut_ref_to_s.value.push('!');
    println!("{s}");
}
```

Error

```
error[E0382]: borrow of moved value: `x`
  --> src/main.rs:10:5
   |
7  | fn foo(x: MutRef) {
   |        - move occurs because `x` has type
   |          `MutRef<'_>`, which does not implement the
   |          `Copy` trait
...
  --> src/main.rs:4:11
   |
4  | fn goo(x: MutRef) {
   |    ---    ^^^^^^ this parameter takes ownership of
   |                  the value

error[E0382]: borrow of moved value: `mut_ref_to_s`
  --> src/main.rs:16:5
   |
16 |     mut_ref_to_s.value.push('!');
   |     ^^^^^^^^^^^^^^^^^^ value borrowed here after
   |                         move
   |
```

Reborrowing

As previously explained, we can not create **more than one mutable reference** towards am object (in this example, **mut_ref_2** can not exist as there is already a mutable reference (**mut_ref_1**) that was created).

Rust

```rust
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let mut_ref_2 = &mut x;
    *mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```

Error

```
error[E0499]: cannot borrow `x` as mutable more than once at a time
 --> src/main.rs:6:21
  |
5 |     let mut_ref_1 = &mut x;
  |                     ------ first mutable borrow occurs here
6 |     let mut_ref_2 = &mut x;
  |                     ^^^^^^ second mutable borrow occurs here
7 |     *mut_ref_2 += 1;
8 |     *mut_ref_1 += 1;
  |     --------------- first borrow later used here
```

Reborrowing

As previously explained, we can not create **more than one mutable reference** towards am object (in this example, mut_ref_2 can not exist as there is already a mutable reference (mut_ref_1) that was created).

Rust

```rust
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let mut_ref_2 = &mut x;
    *mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```

In this example, Rust will notice that we want to create another mutable reference towards "x" (besides the first one – mut_ref_1) and will trigger a compiler error.

Reborrowing

However, we can reborrow the initial mutable reference and then use it. To do this, we have to start from the original mutable reference (and NOT from the object itself) and dereference it to reborrow it.

Rust

```rust
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref = &mut *mut_ref_1;
    *reborrowed_mut_ref += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```

Output

```
2
```

Reborrowing

However, we can borrow the initial mutable reference and then use it. To do this, we have to start from the original mutable reference (and NOT from the object itself) and dereference it to reborrow it.

Rust

```rust
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref = &mut *mut_ref_1;
    *reborrowed_mut_ref += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```

Output

```
2
```

Notice that while reborrowed_mut_ref is being use we don't use mut_ref_1 as well (this way we don't break any ownership rules).

Reborrowing

However, changing the order of how we use **mut_ref_1** and **reborrowed_mut_ref** will produce a compiler error if ownership rules are being broken:

Rust

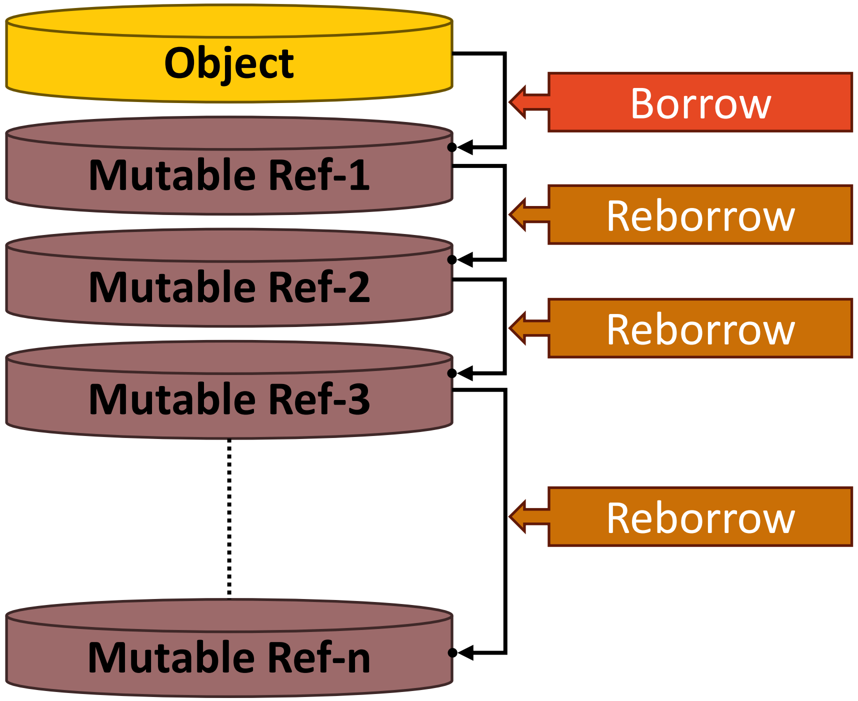
```
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref = &mut *mut_ref_1;
    *mut_ref_1 += 1;
    *reborrowed_mut_ref += 1;
    println!("{}", x);
}
```

Error

```
error[E0503]: cannot use `*mut_ref_1` because it was mutably
borrowed
 --> src/main.rs:7:5
  |
6 |     let reborrowed_mut_ref = &mut *mut_ref_1;
  |                              ---------------
  |                              `*mut_ref_1` is borrowed
  |                               here
7 |     *mut_ref_1 += 1;
  |     ^^^^^^^^^^^^^^^ use of borrowed `*mut_ref_1`
8 |     *reborrowed_mut_ref += 1;
  |     ------------------------ borrow later used here
```

Reborrowing

One way of looking into the logic related to reborrowing mutable reference is to image that Rust sees them as a stack:



1. We are allowed to use only the last mutable reference from the stack
2. When an ***assignment operation*** is performed (eg: `let x = <last mutable ref>`), the last mutable reference is pop out of the stack, and "x" is pushed in
3. Whenever the scope of the last mutable reference from the stack ends, it would be removed from the stack and the next reference can be used.
4. If the scope of the original object ends, the compiler will destroy it.

Reborrowing

Let’s evaluate the following case:

Rust

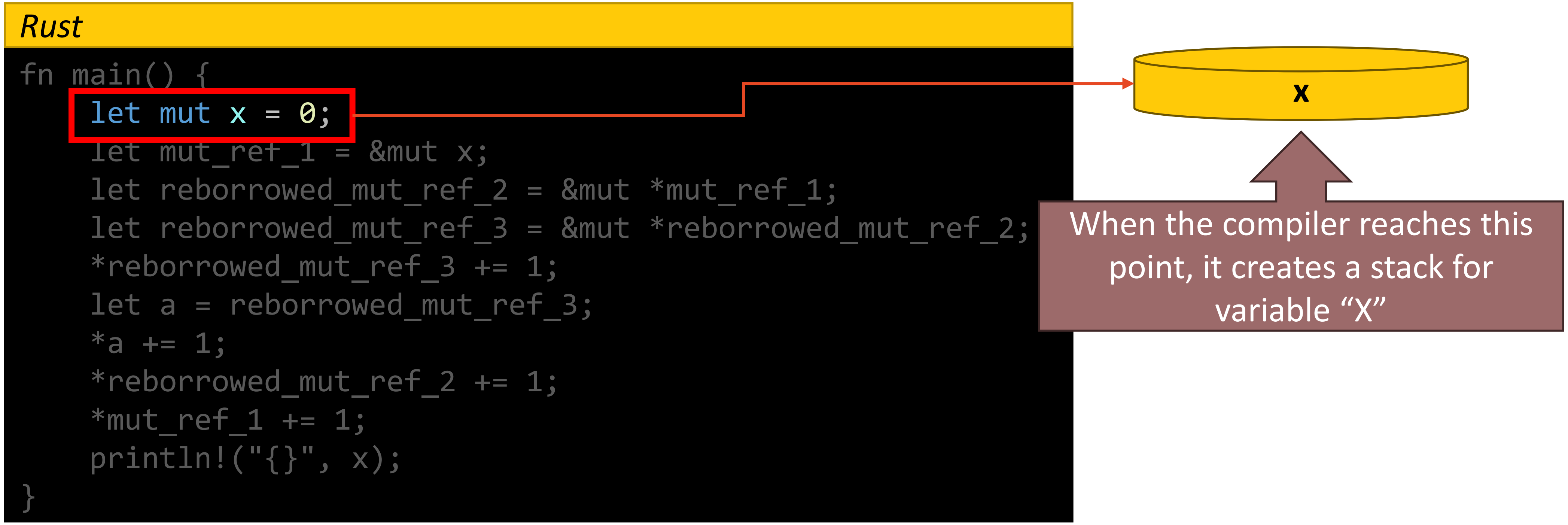
```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```

Output

```
4
```

Reborrowing

Let’s evaluate the following case:



Rust

```rust
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```

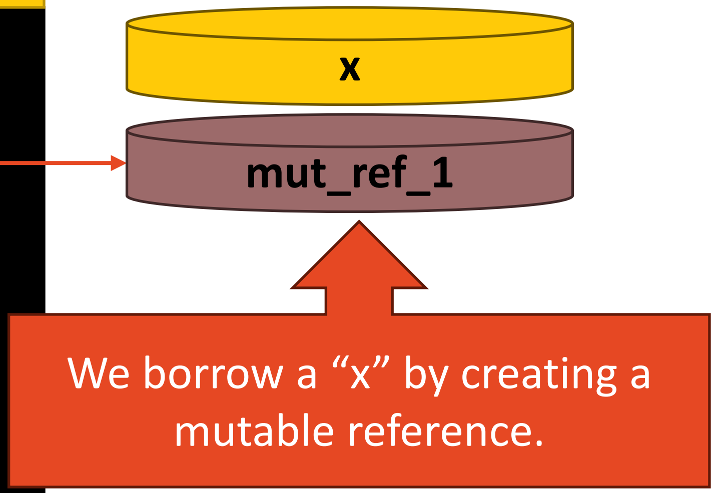
When the compiler reaches this point, it creates a stack for variable “X”

Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



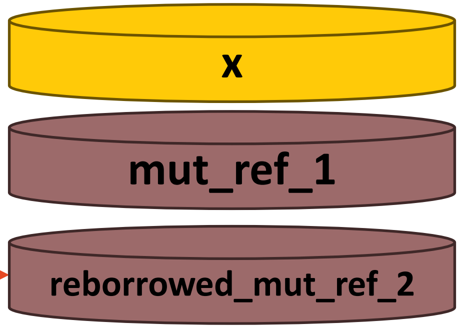
We borrow a “x” by creating a mutable reference.

Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



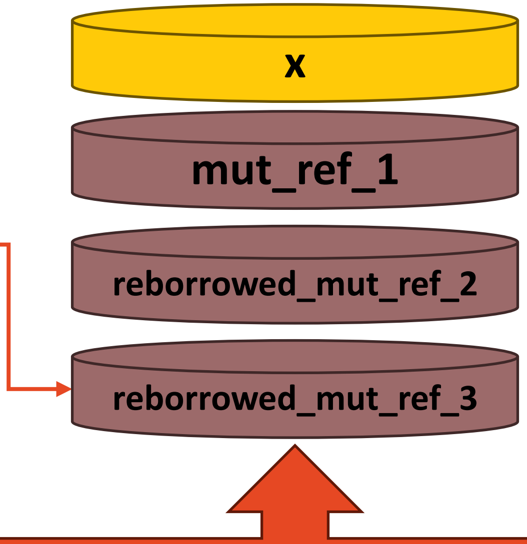
Now we reborrow mut_ref_1 as reborrowed_mut_ref_2

Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



Now we reborrow reborrowed_mut_ref_2 as reborrowed_mut_ref_3

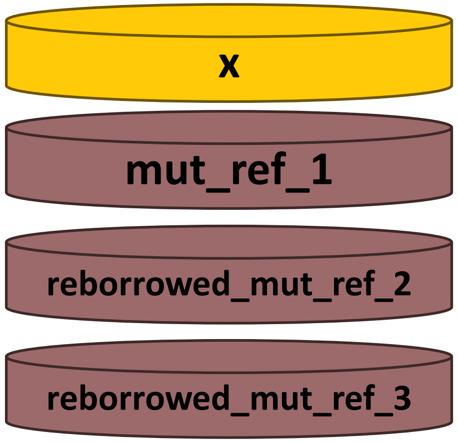
Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
}
```

It is OK to use “reborrowed_mut_ref_3” to access “x” because it is the last from the stack.

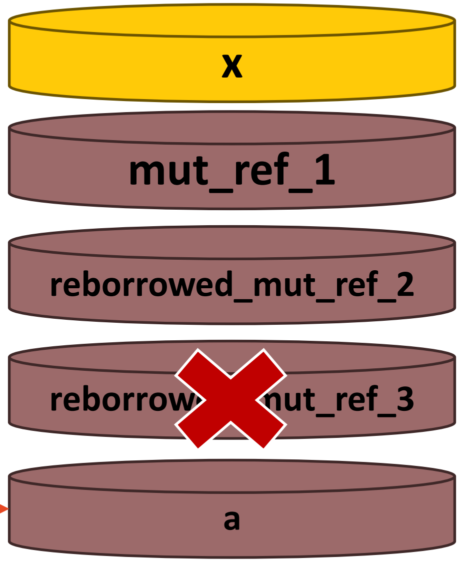


Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



1. Remove ”reborrowed_mut_ref_3“ from the stack.
2. Copy the reference of ”reborrowed_mut_ref_3“ into a
3. Push “a” into stack

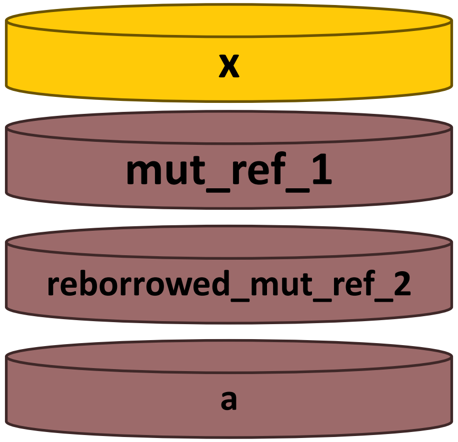
Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mu     f_1 += 1;
```

It is OK to use “a” to access “x” because it is the last from the stack.

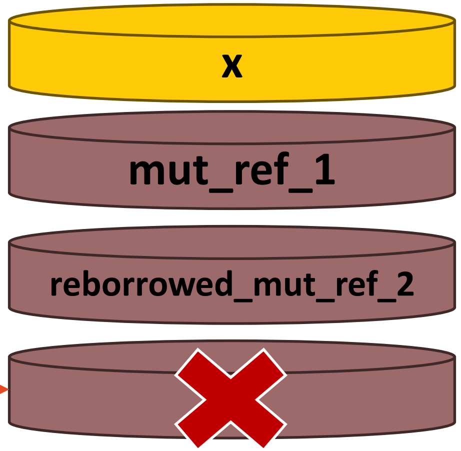


Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    pr
}
```



Notice that after this step, “a” is no longer used in the program. This means that its scope has ended, and we can remove it from the stack.

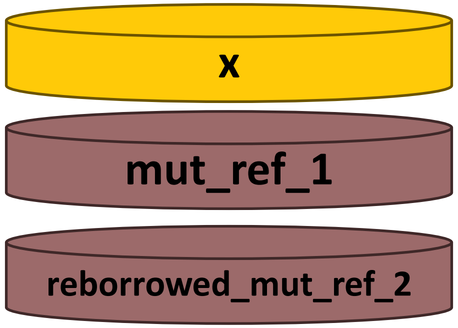
Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
```

It is OK to use “reborrowed_mut_ref_2” to access “x” because it is the last from the stack.

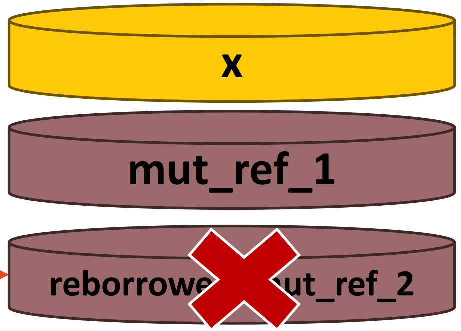


Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



Notice that after this step, “reborrowed_mut_ref_2” is no longer used in the program. This means that its scope has ended, and we can remove it from the stack.

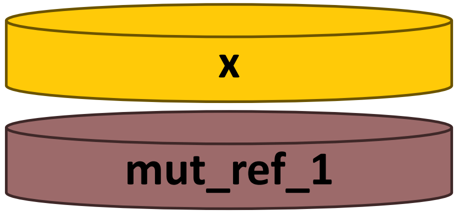
Reborrowing

Let’s evaluate the following case:

Rust

```
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```

It is OK to use “mut_ref_1” to access “x” because it is the last from the stack.

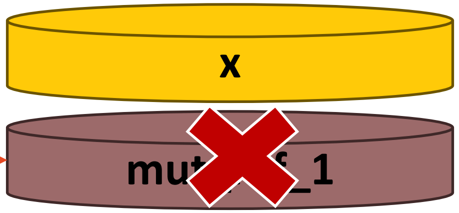


Reborrowing

Let’s evaluate the following case:

Rust

```rust
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



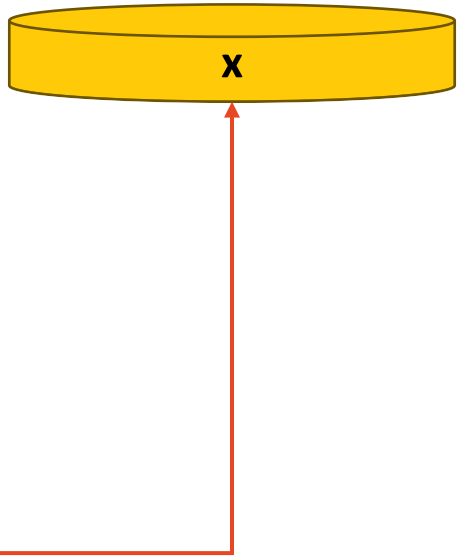
Notice that after this step, “mut_ref_1” is no longer used in the program. This means that its scope has ended, and we can remove it from the stack.

Reborrowing

Let’s evaluate the following case:

Rust

```rust
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



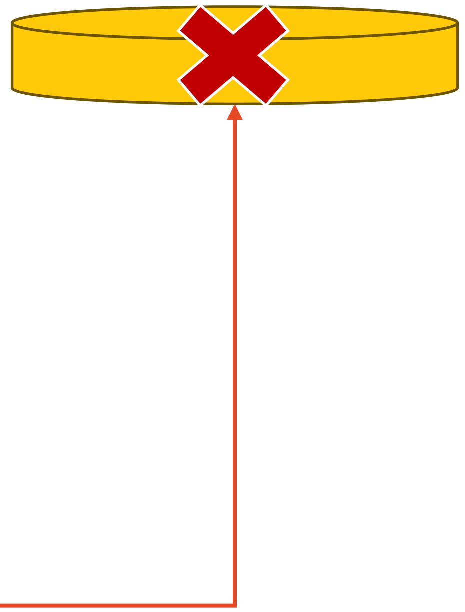
Since there are no mutable references towards variable “x” it is safe now to create an immutable one that can be used in the println! Macro.

Reborrowing

Let’s evaluate the following case:

Rust

```rust
fn main() {
    let mut x = 0;
    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref_2 = &mut *mut_ref_1;
    let reborrowed_mut_ref_3 = &mut *reborrowed_mut_ref_2;
    *reborrowed_mut_ref_3 += 1;
    let a = reborrowed_mut_ref_3;
    *a += 1;
    *reborrowed_mut_ref_2 += 1;
    *mut_ref_1 += 1;
    println!("{}", x);
}
```



Now we can destroy varianle “x” as its scope has ended.

Reborrowing

On the other hand, lets re-analyze the previous example:

Rust

```rust
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref = &mut *mut_ref_1;
    *mut_ref_1 += 1;
    *reborrowed_mut_ref += 1;
    println!("{}", x);
}
```

Error

```
error[E0503]: cannot use `*mut_ref_1` because it was mutably
borrowed
 --> src/main.rs:7:5
  |
6 |     let reborrowed_mut_ref = &mut *mut_ref_1;
  |                              ---------------
  |                              `*mut_ref_1` is borrowed
  |                               here
7 |     *mut_ref_1 += 1;
  |     ^^^^^^^^^^^^^^^ use of borrowed `*mut_ref_1`
8 |     *reborrowed_mut_ref += 1;
  |     ------------------------ borrow later used here
```

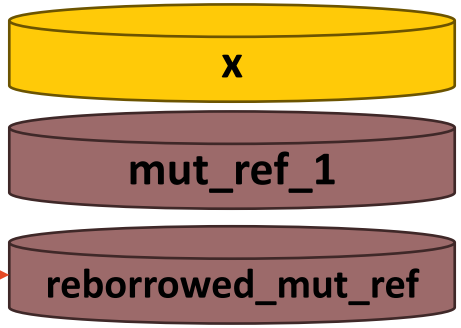
Reborrowing

On the other hand, lets re-analyze the previous example:

Rust

```
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref = &mut *mut_ref_1;
    *mut_ref_1 += 1;
    *reborrowed_mut_ref    1;

}
```

Once the compiler reaches this line, the stack with mutable references for variable “x” looks like in the right image.



Reborrowing

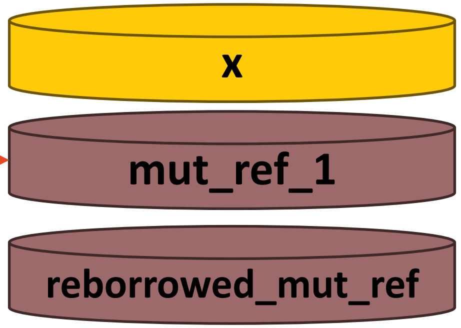
On the other hand, lets re-analyze the previous example:

Rust

```
fn main() {
    let mut x = 0;

    let mut_ref_1 = &mut x;
    let reborrowed_mut_ref = &mut *mut_ref_1
    *mut_ref_1 += 1;
    *reborrowed_mut_ref += 1;
    println!("{}", x);
}
```

However, this code will produce an error since “mut_ref_1” is not the last one from the stack.

Notice that the scope of “reborrowed_mut_ref” ends after the next line (and as such can not be removed from the stack)



Reborrowing

Reborrow mechanism is critical in several Rust programing cases:

1. Passing a reference to a function
2. Calling a method (associated with a trait or a structure/enum)

In particular, it is useful for **mutable references** where it can allow us to reuse the original reference after it was borrowed.

Key takeaways

The following table presents how Rust applies the ownership and borrowing rules over different kind of objects and scenarios:

Object Type	Assignment (x = y)	Function call f(x)
Object with **COPY** trait	COPY	COPY
Object without **COPY** trait	MOVE	MOVE
Immutable references (&T)	COPY	COPY
Mutable references (&mut T)	MOVE	REBORROW

Optimizations

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Let’s consider the following C/C++ code and its Rust equivalent:

C/C++

```
void foo(const unsigned int * input, unsigned int * output) {
    *output += *input;
    *output += *input;
}
```

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

Rust

```
pub fn foo(input: &u32, output: &mut u32) {
    *output += *input;
    *output += *input;
}
```

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

Where:
- rdi = input
- rsi = output

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Why do we have this difference ?

Let's consider the following C/C++ code and its Rust equivalent:

C/C++

```
void foo(const unsig
    *output += *inpu
    *output += *inpu
}
```

Rust

```
u32) {
```

Notice that we have used the `+=` operator. This means that the compiler first needs to read the value from the pointer ***output***, then add to that value the value from pointer ***input***, and finally write the new value into the ***output*** pointer.

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Why do we have this difference ?

Let’s consider the following C/C++ code and its Rust equivalent:

C/C++

```
void foo(const unsigned int *
    *output += *input;
    *output += *input;
}
```

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

Rust

However, *there is no guarantee* that the **output** pointer can’t be access from a different thread. As such, the compiler has to write the new value to **output** pointer so that if another thread is trying to read it, it will read a correct value. ***This also means that it has to perform a similar write for the second operation !***

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

Optimizations

Rust has several optimizations for mutable references since at one moment of time there could be only one mutable reference.

Let's consider the following C/C++ code and its Rust equivalent:

C/(

```
voi

}
```

```
, output: &mut u32) {
p ;
nput;
```

On the other hand, Rust knows that since output is a mutable reference, there is **only one such reference** and no other thread can access it. Because of this, it does not have to write the value after the first call to `+=` operator. Furthermore, since there **can not be mutable reference towards the input** variable (as there is already an immutable one), it can reuse its value for the second operation.

```
mov     eax, dword ptr [rsi]
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
add     eax, dword ptr [rdi]
mov     dword ptr [rsi], eax
```

```
mov     eax, dword ptr [rdi]
add     eax, eax
add     dword ptr [rsi], eax
```

&
A
Q