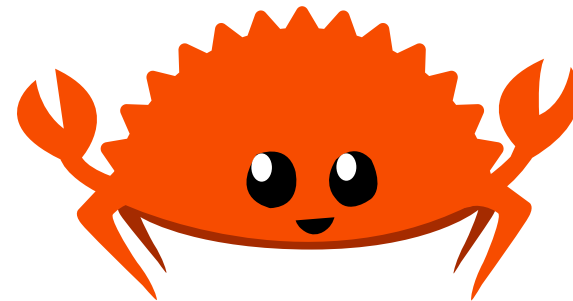


Safe Programming in Rust

Lecture 2 - Ownership & Borrowing

Frans Skarman



Where is Data Stored?

- Static memory
- On the *stack*
- On the *heap*

Static Memory

```
const MESSAGE: &str = "Hello, world";
```

Allocated on startup, ~ 0 cost, but global and inflexible

The Stack

- Stores local variables
- Shrinks and grows with function calls
- Cheap allocation
- Data cannot outlive the function

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

When main is called, its locals are allocated

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

```
- main_a: ????
```

When main is called, its locals are allocated

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

And initialized

Stack content:

```
- main_a: ????
```

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

And initialized

Stack content:

- main_a: 0

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0

Calling `outer` allocates
its locals

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Calling `outer` allocates
its locals

Stack content:

– main_a: 0

– return address: ...

– outer_a: ????

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

And they get initialized

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0
- return address: ...
- outer_a: ????

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

And they get initialized

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0
- return address: ...
- outer_a: 0

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0	
- return address: ...	
- outer_a: 0	
- return address: ...	
- inner1_a: 0	
- inner1_b: 0	

Repeat for inner 1

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;
```

```
}
```

At the end of the function,
pop the stack content

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0	
- return address: ...	
- outer_a: 0	
- return address: ...	
- inner1_a: 0	
- inner1_b: 0	

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;
```

```
}
```

At the end of the function,
pop the stack content

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0	
- return address: ...	
- outer_a: 0	
- return address: ...	
- inner1_a: ????	
- inner1_b: ????	

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

Next function call re-uses the stack space

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0	
- return address: ...	
- outer_a: 0	
- return address: ...	
- inner1_a: ????	
- inner1_b: ????	

The Stack

```
fn inner1() {  
    let inner1_a = 0;  
    let inner1_b = 0;  
}
```

```
fn inner2() {  
    let inner2_a = 0;  
}
```

```
fn outer() {  
    let outer_a = 0;  
    inner1();  
    inner2();  
}
```

Next function call re-uses the stack space

```
fn main() {  
    let main_a = 0;  
    outer();  
}
```

Stack content:

- main_a: 0	
- return address: ...	
- outer_a: 0	
- return address: ...	
- inner2_a: 0	
- inner1_b: ????	

Pointers to the stack

```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}  
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:

– ptr: ????

Pointers to the stack

```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}
```

```
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:

– ptr: ????
– return address: ...
– a: 5

Pointers to the stack

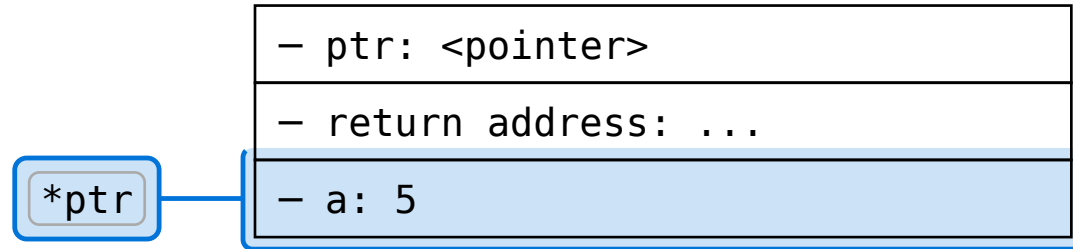
```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}
```

```
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:



Pointers to the stack

```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}
```

```
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:

– ptr: <pointer>

*ptr

– a: ????

Pointers to the stack

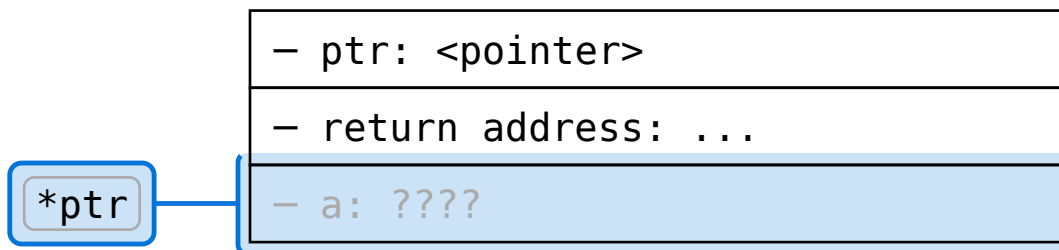
```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}
```

```
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:



Pointers to the stack

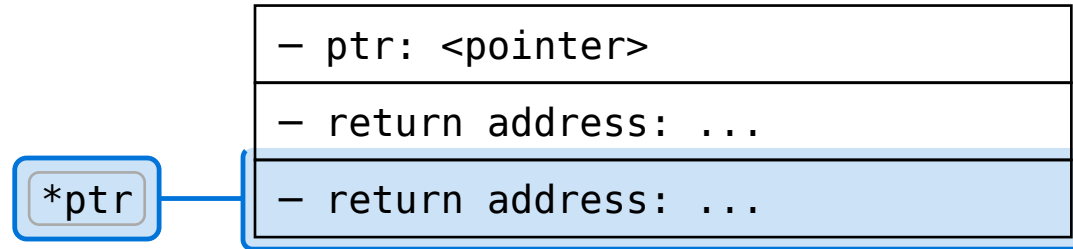
```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}
```

```
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:



Pointers to the stack

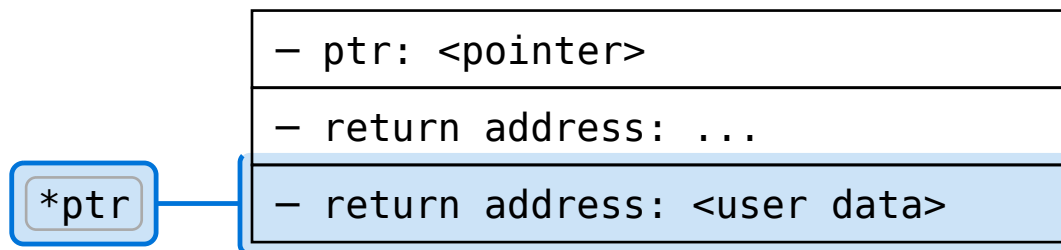
```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}
```

```
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:



Pointers to the stack

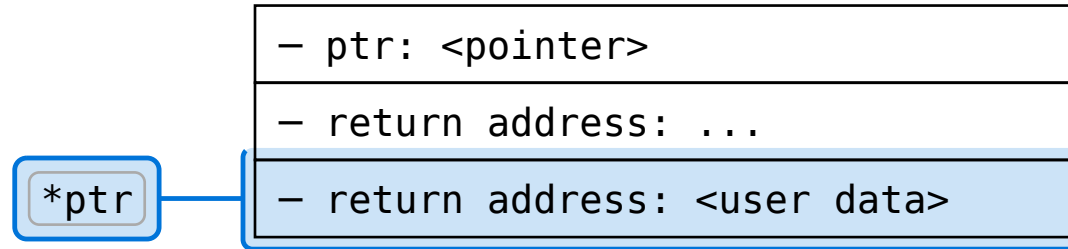
```
int* bad() {  
    int a = 5;  
    return *a;  
}
```

```
void other_function(int* ptr) {  
    *ptr = get_user_input();  
}
```

```
void intermediate(int* ptr) {  
    other_function(ptr);  
}
```

```
int main() {  
    int* ptr = bad();  
    other_function(ptr);  
}
```

Stack content:



Wohooo, **arbitrary code execution** for the user!

The Stack

- Free/cheap allocation

The Stack

- Free/cheap allocation
- Automatically managed allocation/deallocation

The Stack

- Free/cheap allocation
- Automatically managed allocation/deallocation
- Values cannot outlive the function

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```



^ s1

The Heap

Allocate 10 chars on the heap

```
char* s1 = malloc(10 * sizeof(char));
```

```
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```



^ s1

The Heap

Allocate 10 chars on the heap

```
char* s1 = malloc(10 * sizeof(char));
```

```
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```



^ s1

The Heap

```
char* s1 = malloc(10 * sizeof(char));
```

```
*s1 = *"password1";
```

Put some data there

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```



^ s1

The Heap

```
char* s1 = malloc(10 * sizeof(char));
```

```
*s1 = "password1";
```

Put some data there

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

x x x x p a s s w o r d 1 x x x x

^ s1

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

Create a new pointer to
the same addr

x x x x p a s s w o r d 1 x x x x

^ s1

^ s2

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

Create a new pointer to
the same addr

x x x x p a s s w o r d 1 x x x x

^ s1

^ s2

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

Deallocate the data via
the first pointer



^ s1

^ s2

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

Deallocate the data via
the first pointer

x x x x p a s s w o r d 1 x x x x

^ s1

^ s2

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

Deallocate the data via
the first pointer

x x x x p a s s w o r d 1 x x x x

^ s1

^ s2

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

Deallocate the data via
the first pointer

x x x x p a s s w o r d 1 x x x x

^ s1

^ s2

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

Print via the second pointer

x x x x p a s s w o r d 1 x x x x

^ s1

^ s2

The Heap

```
char* s1 = malloc(10 * sizeof(char));  
*s1 = *"password1";
```

```
char* s2 = s1;
```

```
free(s1);
```

```
printf("%s\n", s2);
```

x x x x p a s s w o r d 1 x x x x

^ s1

^ s2

☒ Oh no! ☒

Best case scenario: SIGSEGV

Or, you're now sending this email:

Oopsie woopsie, sorry for leaking
your password

☒☒☒☒

What Went Wrong?

If memory is allocated, *someone* has to free it. But who?

What Went Wrong?

If memory is allocated, *someone* has to free it. But who?

“The last user”, but who is that?

What Went Wrong?

If memory is allocated, *someone* has to free it. But who?

“The last user”, but who is that?

☒ **It depends!** ☒

What Went Wrong?

If memory is allocated, *someone* has to free it.
“The last user”, but who is that?

☒It depends!☒



Ownership

1. In Rust, **all data has an owner**

Ownership

1. In Rust, **all data has an owner**
2. There can only be 1 owner at a time

Ownership

1. In Rust, **all data has an owner**
2. There can only be 1 owner at a time
 - Transferring ownership is called **moving**

Ownership

1. In Rust, **all data has an owner**
2. There can only be 1 owner at a time
- Transferring ownership is called **moving**
3. If the owner goes out of **scope**, the owned data is **dropped**

```
struct String {  
    int length;  
    // Needs-freeing  
    char* data;  
}  
  
struct File {  
    // Needs closing  
    int file_descriptor;  
}
```

Strings

```
fn main() {  
    let s = String::from("password1");  
}
```

Stack:

```
s1: String  
├ length: 12  
├ capacity: 12  
└ data: <pointer>
```

Heap:

x
x
x
p
a
s
s
w
o
r
d
1

x
x
x

Examining Ownership with Strings

Stack:

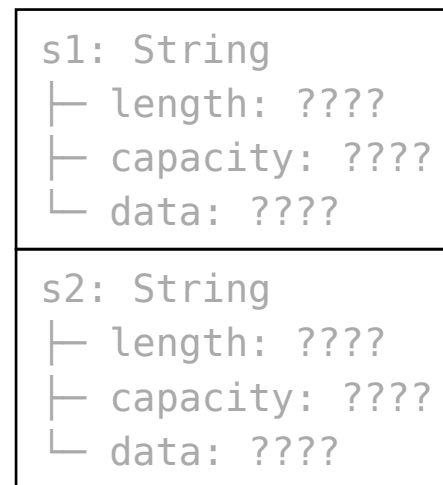
Heap:

```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Examining Ownership with Strings

```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Stack:



Heap:



Examining Ownership with Strings

```
fn main() {  
  let s1 = String::from("password1");  
  let s2 = s1;  
  drop(s1); // Explicitly call the "destructor"  
  println!("{}", s2);  
}
```

Initialize s1

Stack:

s1: String
├ length: ????
├ capacity: ????
└ data: ????
s2: String
├ length: ????
├ capacity: ????
└ data: ????

Heap:

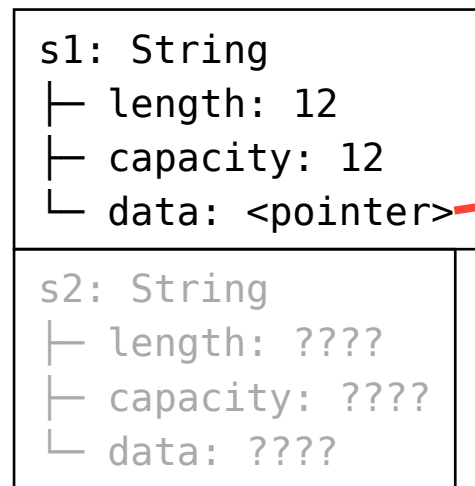
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x

Examining Ownership with Strings

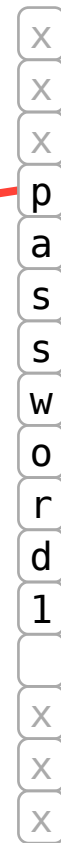
```
fn main() {  
  let s1 = String::from("password1");  
  let s2 = s1;  
  drop(s1); // Explicitly call the "destructor"  
  println!("{}", s2);  
}
```

Initialize s1

Stack:



Heap:



Examining Ownership with Strings

```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Move s1 to s2

Stack:

s1: String
├ length: 12
├ capacity: 12
└ data: <pointer>

s2: String
├ length: ????
├ capacity: ????
└ data: ????

Heap:

x
x
x
p
a
s
s
w
o
r
d
1

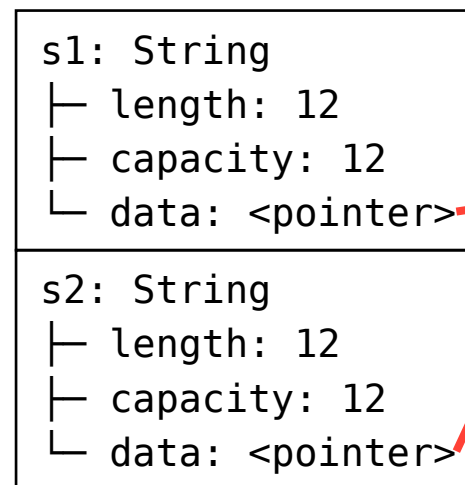
x
x
x

Examining Ownership with Strings

```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Move s1 to s2

Stack:



Heap:

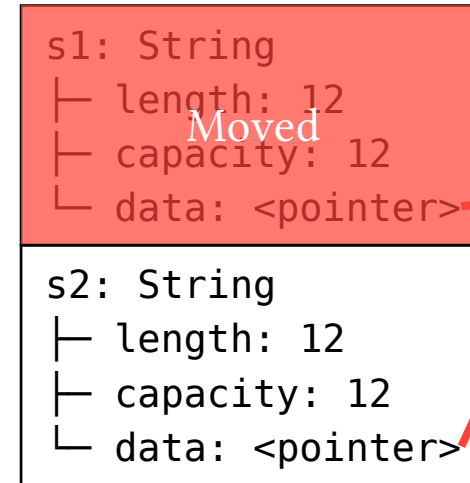


Examining Ownership with Strings

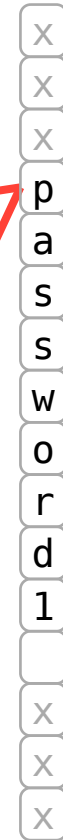
```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Move s1 to s2

Stack:



Heap:



Examining Ownership with Strings

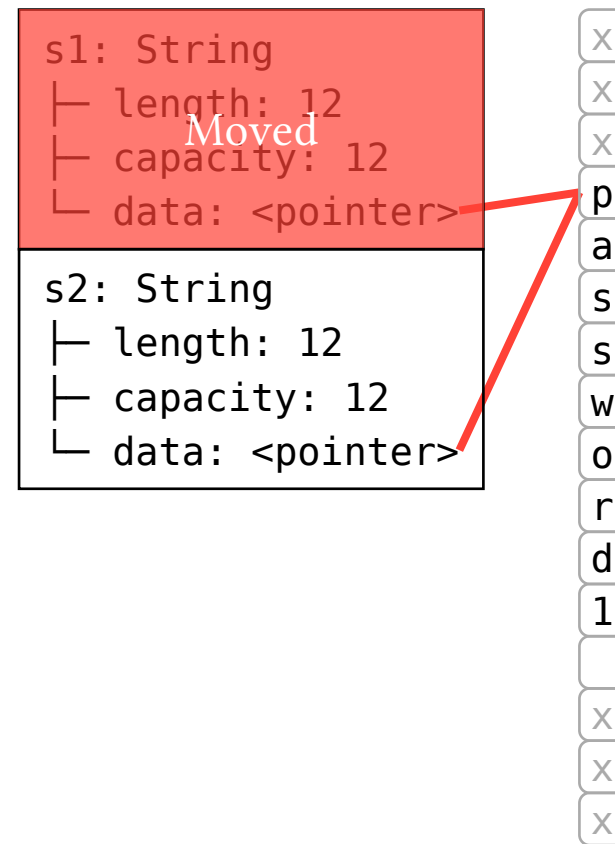
```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Drop s1

≈ Explicitly run the destructor

Stack:

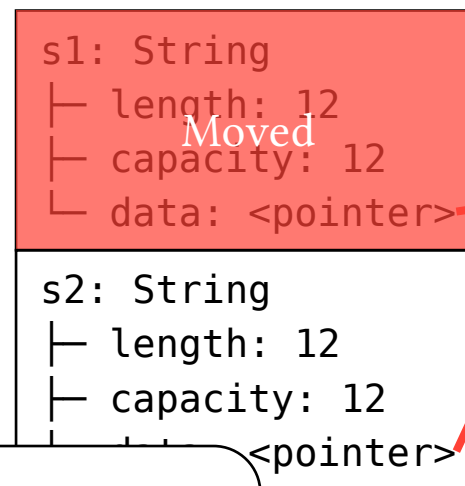
Heap:



Examining Ownership with Strings

```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Stack:



Heap:



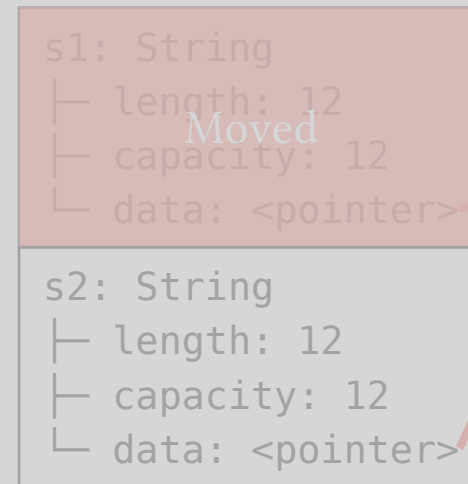
error[E0382]: use of moved value: `s1`

```
2 | let s1 = String::from("password1");  
  |      -- move occurs because `s1` has type `String`, which  
  |      does not implement the `Copy` trait  
3 | let s2 = s1;  
  |      -- value moved here  
4 | drop(s1);  
  |      ^^ value used here after move
```

Examining Ownership with Strings

```
fn main() {  
    let s1 = String::from("password1");  
    let s2 = s1;  
    drop(s1); // Explicitly call the "destructor"  
    println!("{}", s2);  
}
```

Stack:



Heap:



What does **Move** mean?

≈ Shallow copy + invalidate old owner

$O(1)$

(But what does invalidate mean?)

Transferring Ownership

- Assignment (seen in the previous example)
- Function Calls

Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}  
  
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```



Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

Main locals pushed to the stack



Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

main

```
s1: String  
├ length: ????  
├ capacity: ????  
└ data: ????
```

Main locals pushed to the stack



Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

main

s1: String
├ length: ????
├ capacity: ????
└ data: ????

The string is allocated

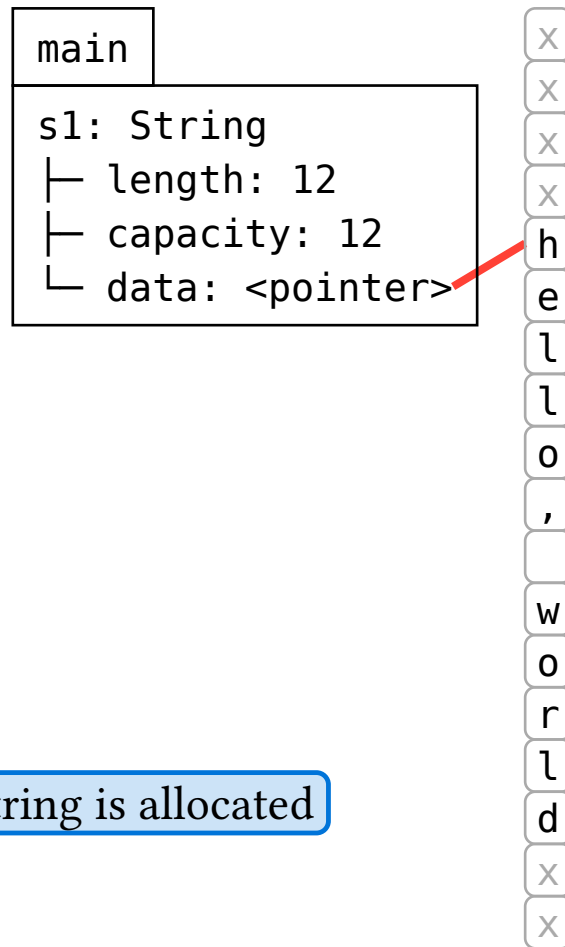
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x

Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

The string is allocated

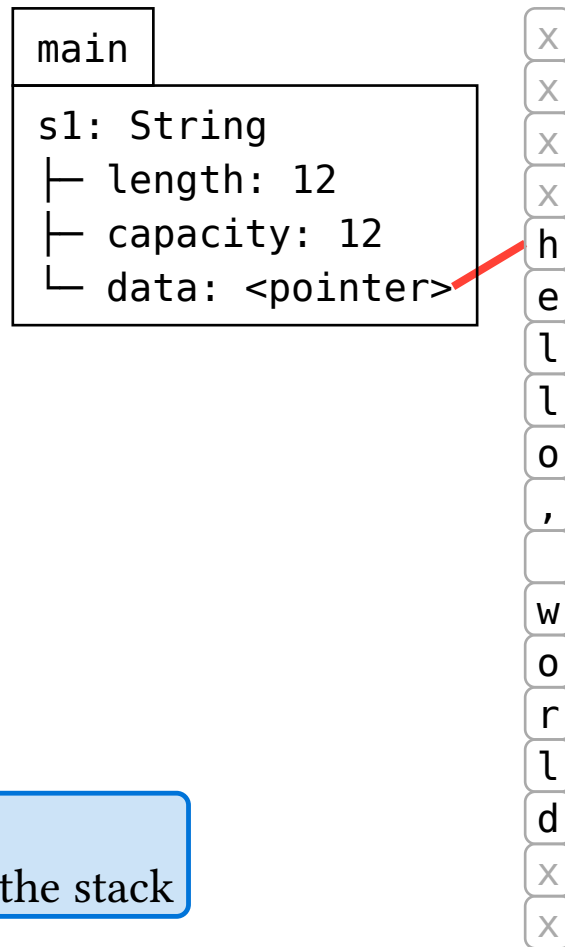


Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

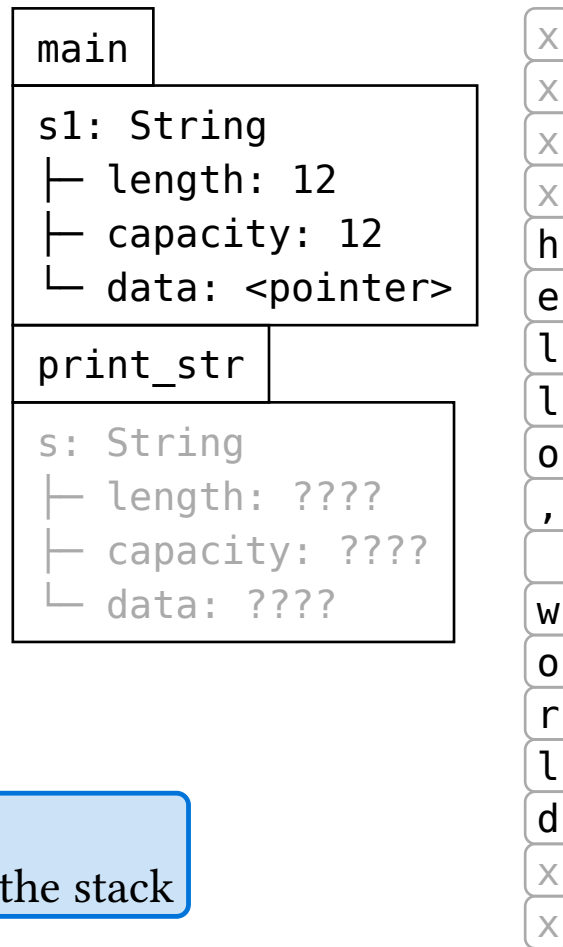
Call to print_str.
print_str locals pushed to the stack



Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}  
  
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

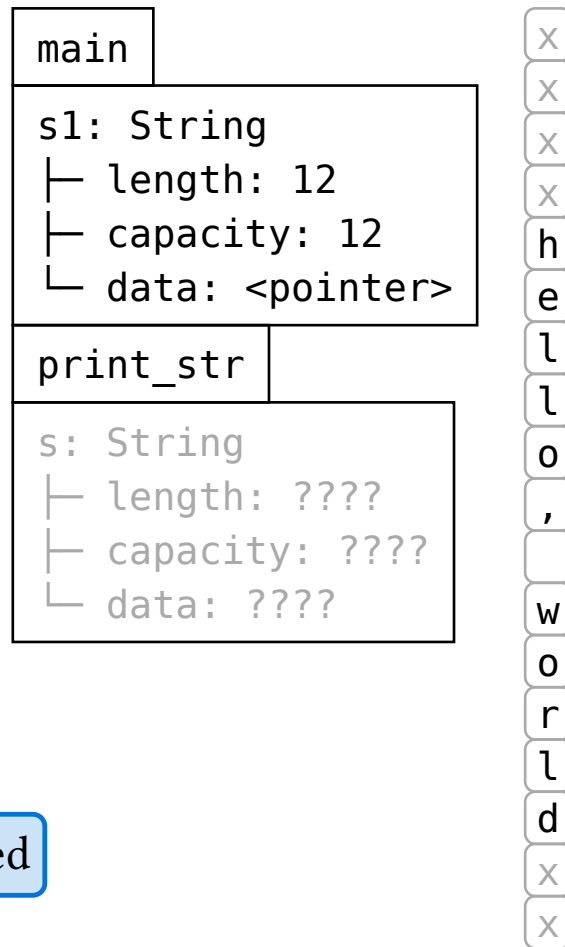
Call to print_str.
print_str locals pushed to the stack



Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}  
  
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

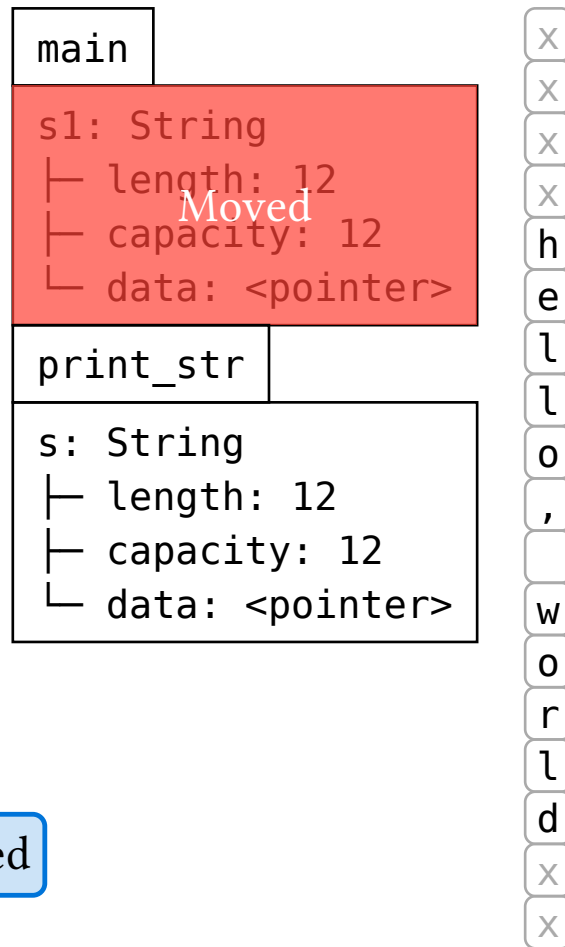
And `s1` gets moved



Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}  
  
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

And `s1` gets moved

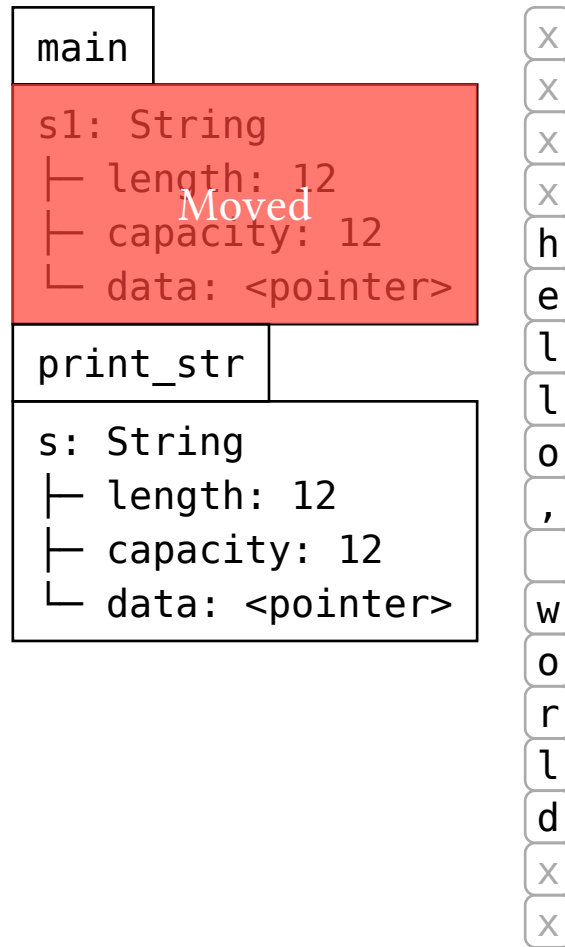


Ownership and Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

Print runs

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```



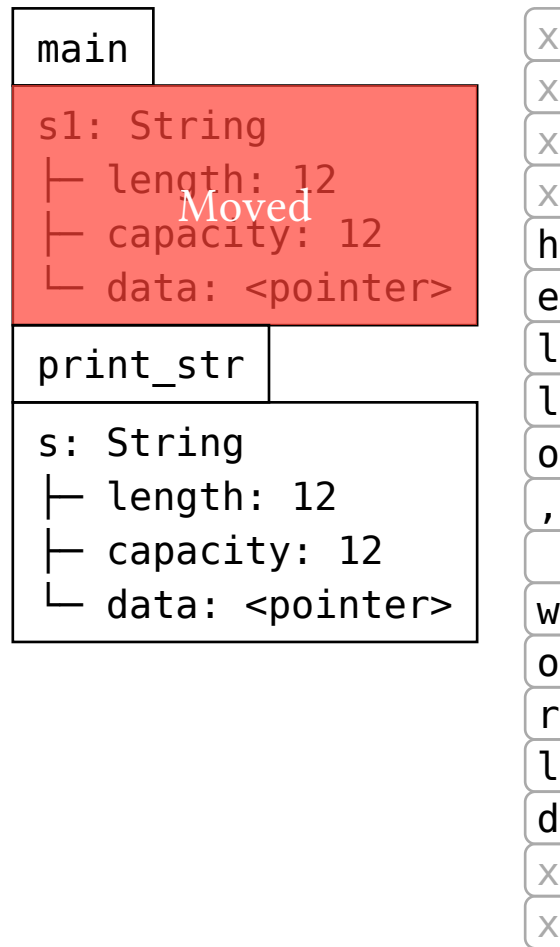
Ownership and Function Calls

End of scope where `s` is owned.

Drop gets called

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```



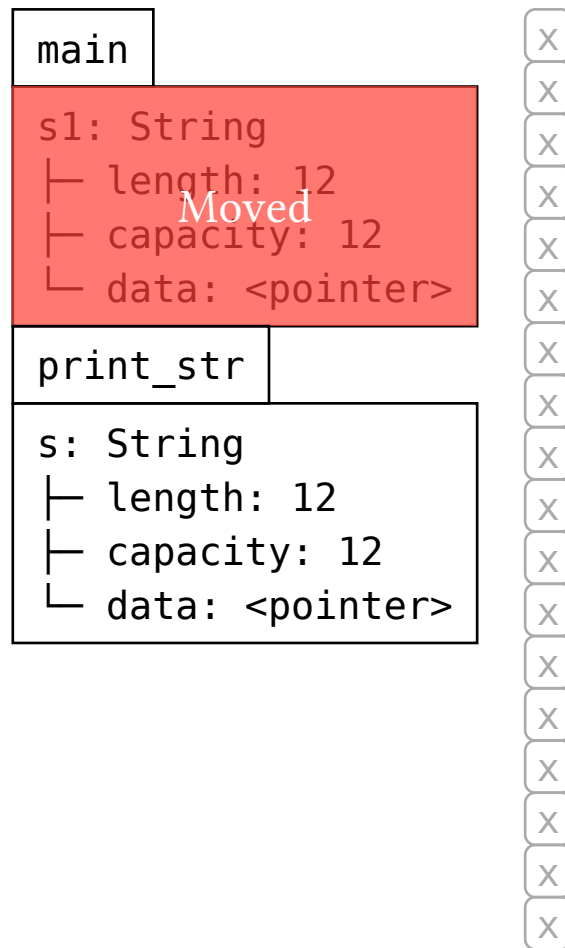
Ownership and Function Calls

End of scope where `s` is owned.

Drop gets called

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

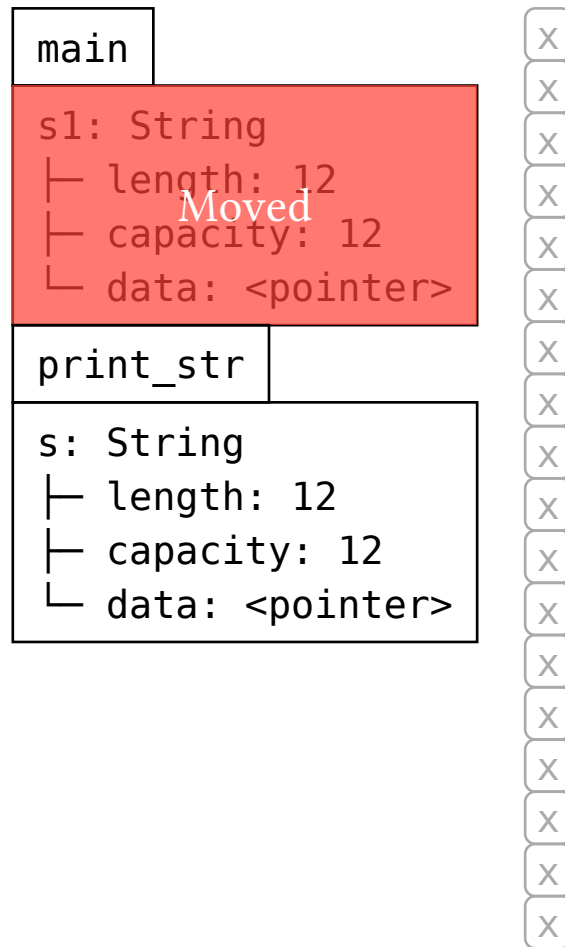


Ownership and Function Calls

And we return, popping the stack

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```



Ownership and Function Calls

And we return, popping the stack

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
}
```

main

Moved

```
s1: String  
├ length: 12  
├ capacity: 12  
└ data: <pointer>
```

x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x
x

More Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
    do_stuff(s1)  
}
```

More Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
    do_stuff(s1);  
}
```

Using `s1` after printing it

More Function Calls

```
fn print_str(s: String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
    do_stuff(s1)  
}
```

Using `s1` after printing it

What should happen?

More Fun

```
fn print_str  
    println!  
}
```

```
error[E0382]: use of moved value: `s1`
```

```
--> src/le2.rs:73:14
```

```
70 |         let s1 = String::from("Hello, world");  
    |             -- move occurs because `s1` has type `String`, which  
              does not implement the `Copy` trait  
71 |         print_str(s1);  
    |                   -- value moved here  
73 |         do_stuff(s1)  
    |                   ^^ value used here after move
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    print_str(s1);  
    do_stuff(s1)  
}
```

A Very Annoying Solution

```
fn print_str(s: String) -> String {  
    println!("{}", s);  
    s  
}  
  
fn main() {  
    let s1 = String::from("Hello, world");  
    let s1 = print_str(s1);  
    do_stuff(s1)  
}
```

A Very Annoying Solution

```
fn print_str(s: String) -> String {  
    println!("{}", s);  
    s  
}
```

Return the ownership of `s`
after printing

```
fn main() {  
    let s1 = String::from("Hello, world");  
    let s1 = print_str(s1);  
    do_stuff(s1)  
}
```

A Very Annoying Solution

```
fn print_str(s: String) -> String {  
    println!("{}", s);  
    s  
}  
  
fn main() {  
    let s1 = String::from("Hello, world");  
    let s1 = print_str(s1);  
    do_stuff(s1)  
}
```

And re-bind it here

A Very Annoying Solution

```
fn print_str(s: String) -> String {  
    println!("{}", s);  
    s  
}
```

```
fn main() {  
    let s1 = String::from("Hello, world");  
    let s1 = print_str(s1);  
    do_stuff(s1)  
}
```

Pros:

- ✓ It compiles

Cons:

- Yes
- ☒

Ownership and Borrowing

An object can be **borrowed** temporarily

- While borrowing an object you can read/modify it
 - But you cannot move it
- While data is borrowed, it cannot be moved
 - Alternatively, a borrow cannot outlive the object

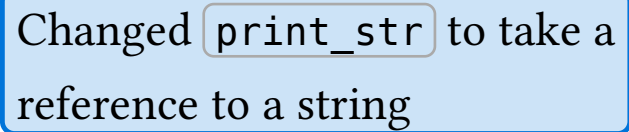
References

Borrowed data is denoted `&T`

To borrow a variable, use `&data`

```
fn print_str(s: &String) {  
    println!("{}", s);  
}  
  
fn main() {  
    let s = String::from("Hello, world");  
  
    print_str(&s);  
  
    do_stuff(s)  
}
```

Changed `print_str` to take a reference to a string



```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s = String::from("Hello, world");  
  
    print_str(&s);  
  
    do_stuff(s)  
}
```

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s = String::from("Hello, world");  
  
    print_str(&s);  
  
    do_stuff(s)  
}
```

Pass a reference instead of ownership

But What *Is* Borrowing and Moving?

Moving is a copy + compile time marker that the object is dead

Borrowing is just a pointer, but the compiler makes sure
it does not outlive its pointee

Borrowing Example

main

```
fn print_str(s: &String) {  
    println!("{}", s);  
}  
  
fn main() {  
    let s = String::from("Hello, world");  
  
    print_str(&s);  
  
    do_stuff(s)  
}
```



Borrowing Example

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s = String::from("Hello, world");  
  
    print_str(&s);  
  
    do_stuff(s)  
}
```

main

s1: String
├ length: ????
├ capacity: ????
└ data: ????



Borrowing Example

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s = String::from("Hello, world");  
    print_str(&s);  
    do_stuff(s)  
}
```

main

```
s1: String  
├ length: ????  
├ capacity: ????  
└ data: ????
```

The string is allocated



Borrowing Example

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s = String::from("Hello, world");  
    print_str(&s);  
    do_stuff(s)  
}
```

main

s1: String
├ length: 12
├ capacity: 12
└ data: <pointer>

The string is allocated



Borrowing Example

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s = String::from("Hello, world");  
    print_str(&s);  
    do_stuff(s)  
}
```

main

s1: String
├ length: 12
├ capacity: 12
└ data: <pointer>

The string is allocated

x
x
x
x
h
e
l
l
o
,

w
o
r
l
d
x
x

Borrowing Example

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

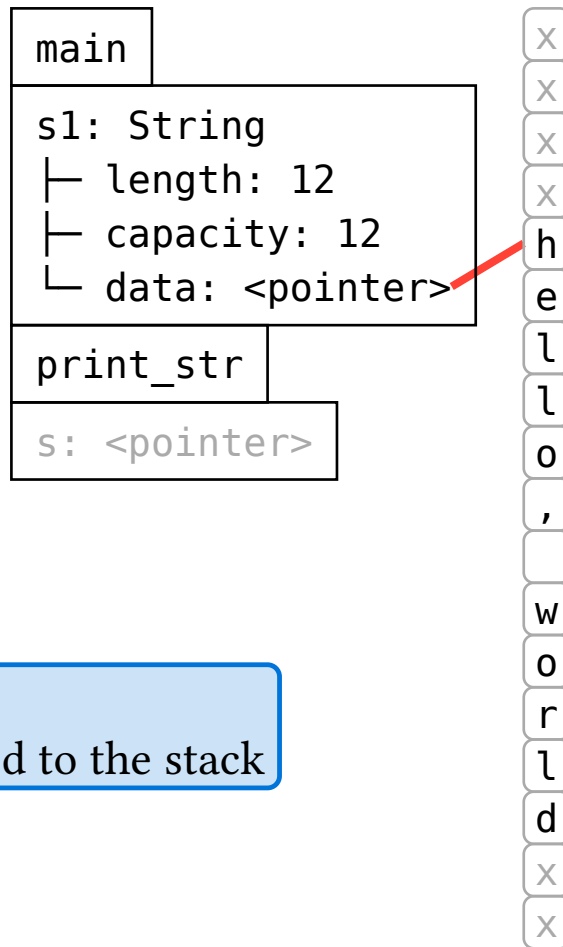
```
fn main() {  
    let s = String::from("Hello, world");
```

```
    print_str(&s);
```

```
    do_stuff(s)
```

```
}
```

Call to print_str.
print_str locals pushed to the stack



Borrowing Example

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

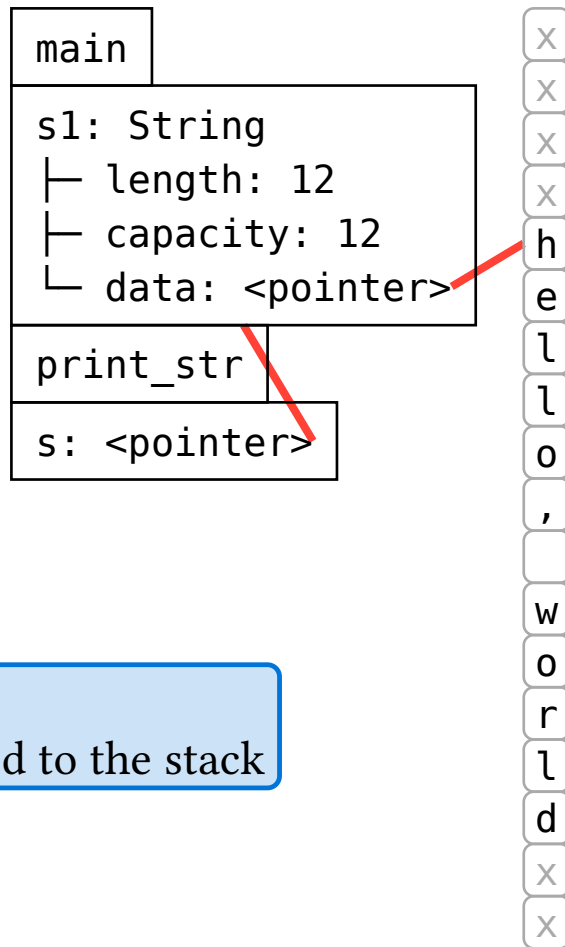
```
fn main() {  
    let s = String::from("Hello, world");
```

```
    print_str(&s);
```

```
    do_stuff(s)
```

```
}
```

Call to print_str.
print_str locals pushed to the stack

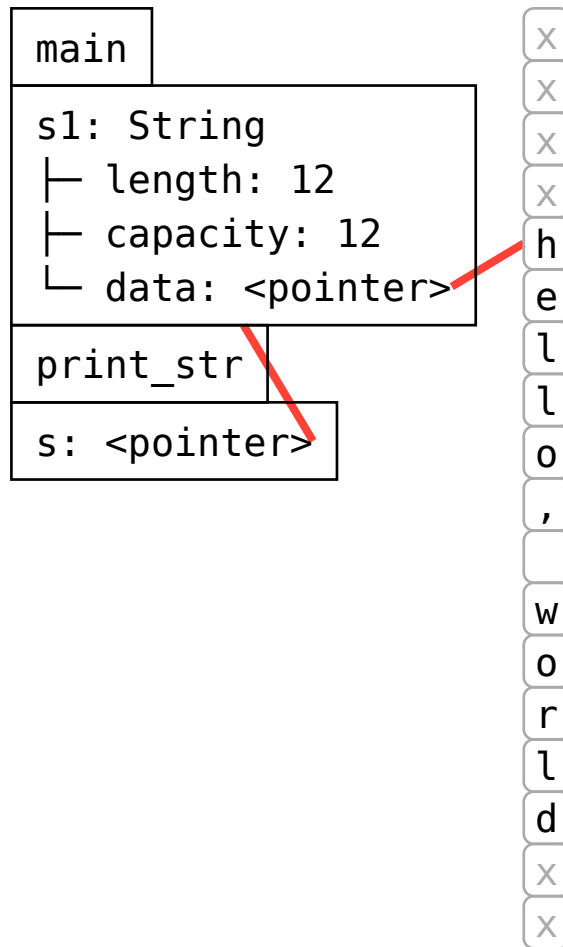


Borrowing Example

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

Print runs

```
fn main() {  
    let s = String::from("Hello, world");  
  
    print_str(&s);  
  
    do_stuff(s)  
}
```



Borrowing Example

End of scope where `s` is owned.

It is a reference, so no `drop`

```
fn print_str(s: &String) {  
    println!("{}", s);  
}
```

```
fn main() {  
    let s = String::from("Hello, world");  
  
    print_str(&s);  
  
    do_stuff(s)  
}
```

main

```
s1: String  
├ length: 12  
├ capacity: 12  
└ data: <pointer>
```

x
x
x
x
h
e
l
l
o
,

w
o
r
l
d
x
x

Some Examples

```
let x = String::new();  
let y = &x;  
let z = x;  
println!("{y}")
```

```
let x = String::new();  
let y = &x;  
println!("{y}");  
let z = x;
```

```
fn f1(a: String) {}  
  
let x = String::new();  
f1(x);  
let y = &x;  
println!("{y}");  
let z = x;
```

```
fn f1(a: &String) {}  
  
let x = String::new();  
f1(&x);  
println!("{x}");  
let z = x;
```

Varibale Lifetime

Variables live until the end of their scope

```
{
  let x = String::from("x");  -----+
                                |
  {
    let y = String::from("y");  --+
                                |
  } -----+
} -----+
```

Reference Lifetime

References live until their last use.

Reference Lifetime

References live until their last use.

```
{
  let x = String::from("x");
  let y = &x;  -----+
  let z = &x;   ----- | -+
  some_function(y); -----+ | // Last use of y here
  some_function(&x);         |
                          ++ // Temporary
  some_function(z);  -----+
  // ...
}
```

Reference Lifetime

References live until their last use.

```
{
  let x = String::from("x");
  let y = &x;  -----+
  let z = &x;   ----- | -+
  some_function(y); -----+ | // Last use of y here
  some_function(&x);         |
                          ++ // Temporary
  some_function(z);  -----+
  // ...
}
```

Sort of, we'll talk more about lifetimes later

Mutable Borrows

Recall: In Rust, everything is **immutable** by default

Mutable Borrows

Recall: In Rust, everything is **immutable** by default

`mut` makes a binding mutable

Mutable Borrows

Recall: In Rust, everything is **immutable** by default

`mut` makes a binding mutable

`&mut` is a mutable reference (the underlying data can be mutated)

mut is about variables, not objects

```
let s = String::from("Hello");  
let mut s1 = s;  
s1.append(", World");
```

Race Conditions

Happen when concurrent processes modify a value at the same time

```
let ptr = /*<pointer to some data>*/
```

```
let tmp = read_ptr(ptr);  
let result = tmp + 1;  
write_ptr(ptr, result)
```

```
let tmp = cread_ptr(ptr);  
let result = tmp * 2;  
write_ptr(ptr, result)
```

Race Conditions

Happen when concurrent processes modify a value at the same time

```
let ptr = /*<pointer to some data>*/  
  
let tmp = read_ptr(ptr);  
let result = tmp + 1;  
write_ptr(ptr, result)  
  
let tmp = cread_ptr(ptr);  
let result = tmp * 2;  
write_ptr(ptr, result)
```

We will discuss concurrency later

Mutable Borrows

Rules for **borrows**:

- Only `1` mutable borrow allowed at a time
- Or `n` immutable borrows

Slices

Write a function that

- takes a string of words
 - separated by space
- returns the end of the first word

Slices

Write a function that

- takes a string of words
 - separated by space
- returns the end of the first word

```
fn first_word(s: &String) -> // ?
```

Returning a String?

```
fn first_word(s: &String) -> String
```

Returning a String?

```
fn first_word(s: &String) -> String
```

This would allocate and require copying the data. Expensive!

Returning Indices

```
fn first_word(s: &String) -> usize
```

Returning Indices

```
fn first_word(s: &String) -> usize
```

```
let mut s = String::from("This has a few words");
```

```
let end = first_word(&s);
```

```
s.clear();
```

```
println!("{}", &s[..end]);
```

Returning Indices

```
fn first_word(s: &String) -> usize
```

```
let mut s = String::from("This has a few words");
```

```
let end = first_word(&s);
```

Find first word in the string
No link back to `s`

```
s.clear();
```

```
println!("{}", &s[..end]);
```

Returning Indices

```
fn first_word(s: &String) -> usize
```

```
let mut s = String::from("This has a few words");
```

```
let end = first_word(&s);
```

```
s.clear();
```

The string is modified

```
println!("{}", &s[..end]);
```

Returning Indices

```
fn first_word(s: &String) -> usize
```

```
let mut s = String::from("This has a few words");
```

```
let end = first_word(&s);
```

```
s.clear();
```

```
println!("{}", &s[..end]);
```

We use the now outdated index



Copy and **Clone**