CoE 164

Computing Platforms

02a: Data Ownership

DATA OWNERSHIP

Rust enables programmers to access the memory more directly than other high-level programming languages.

Due to this power, Rust is paranoid about data and memory safety. Most “errors” related to data misuse and improper cleanup are caught at compile time.

PROGRAM MEMORY

Programs are usually given a **stack** and a **heap** as memory locations where they can store data.

Depending on where data is stored, Rust variables act differently within the program.

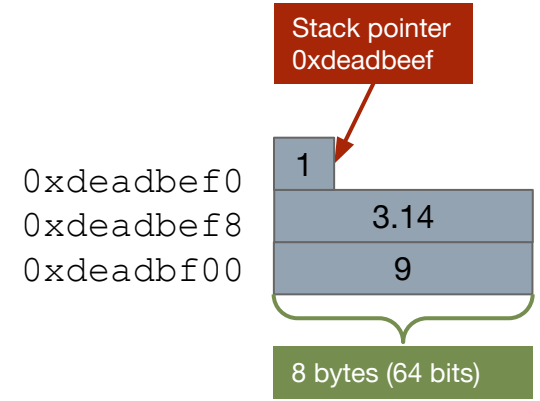
MEMORY: STACK

The **stack** is where data of a *fixed size* is stored. It is usually growing "downward" and is faster to access.

We can push data into the stack for safekeeping. However, we can only get the topmost data from the stack at any given time.

Example

```
let a = 9;
let b = 3.14;
let c = true;
```



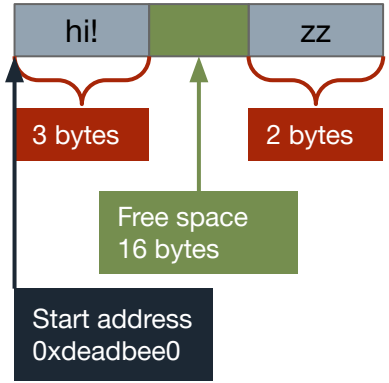
MEMORY: HEAP

The **heap*** is where data of a *variable size* is stored.

A space for the data should be *allocated* first before being able to save it into the heap. Conversely, data that will not be used anymore should be *deallocated* or *freed* from the heap.

Example

```
let dda = 3;
let a = "hi!".to_string();
let b = "zz".to_string();
```

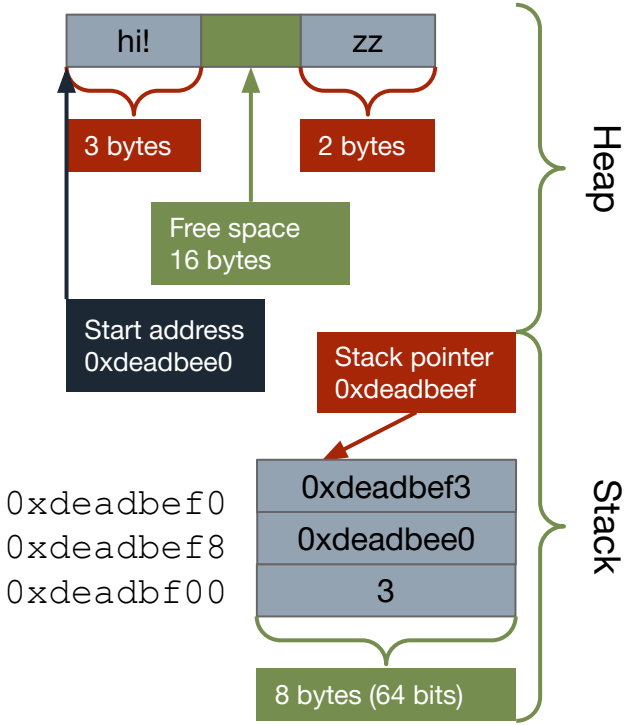


* Not to be confused with the heap data structure.

MEMORY: HEAP

Saving to the heap will also push a **pointer** to the stack. This pointer points to the address *in the heap* where the true value of the data can be found.

Heap data is accessed by first looking at the pointer from the stack, and then going to that address in the heap.



OWNERSHIP RULES

Rust follows some basic rules regarding ownership of a data.

1. Each value has an owner
2. There can only be one owner at any given time
3. When the owner is gone (*out of scope*), the value is deleted (*dropped*).

COPY AND MOVE

Rust enables data types to have a *Copy* or *Move trait*.

A **copy** implies that data of that type is "trivially" copied and can be stored on the stack.

A **move** implies that data of that type is stored in the heap.

COPY AND MOVE: MOVE

By default, data types have a Move trait only. Ownership rules (1) and (2) dictate that "passing" data from one variable to another scope or variable *moves* the data to the destination. Consequently, the origin variable becomes out of scope, and is therefore invalid.

```
let hello = String::from("hello!");
let hello2 = hello;

println!("{hello}"); // compile error
```

Example

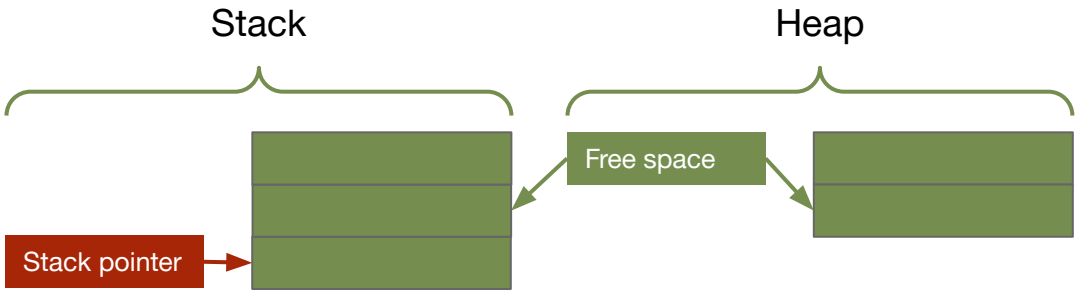
COPY AND MOVE: DROP

When a variable goes out of scope, Rust will call a special function named *drop* where memory cleanup and deallocation happens. In general, data types that have the Move trait should implement the Drop trait.

```
let hello = String::from("hello!");
let hello2 = hello;
// hello is dropped in the preceding line

println!("{hello2}");
```

Example

COPY AND MOVE: MOVE

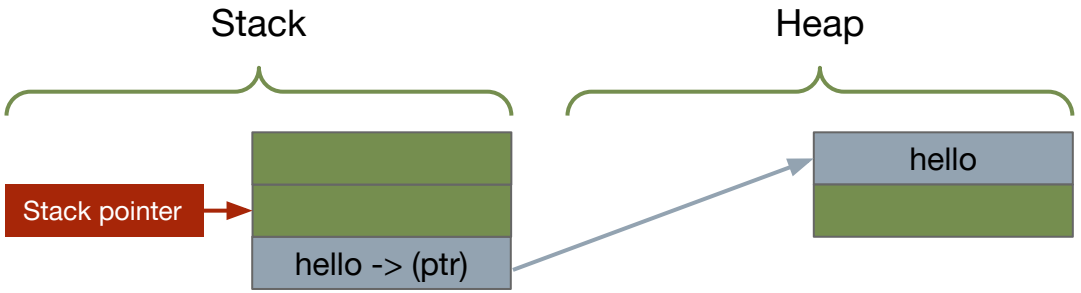


Example

Program counter

```
let hello = String::from("hello!");
let hello2 = hello;

println!("{hello}");
```

COPY AND MOVE: MOVE

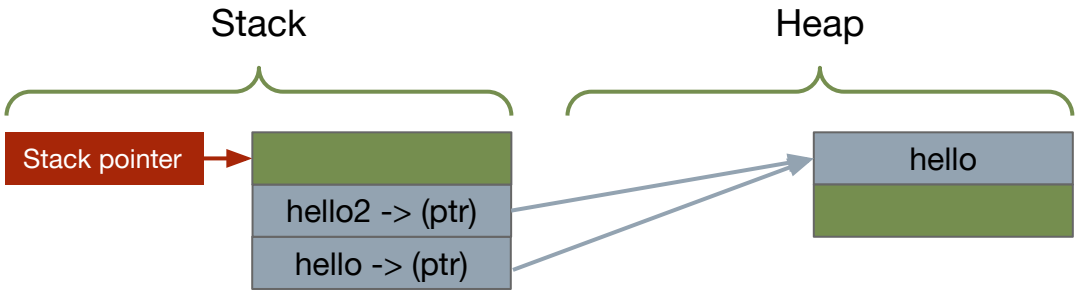


Example

Program counter

```
let hello = String::from("hello!");
let hello2 = hello;

println!("{hello}");
```

COPY AND MOVE: MOVE

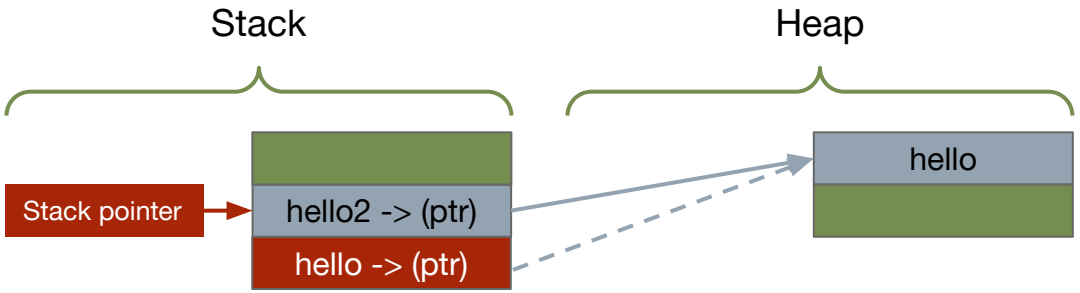


Example

Program counter

```
let hello = String::from("hello!");
let hello2 = hello;

println!("{hello}");
```

COPY AND MOVE: MOVE

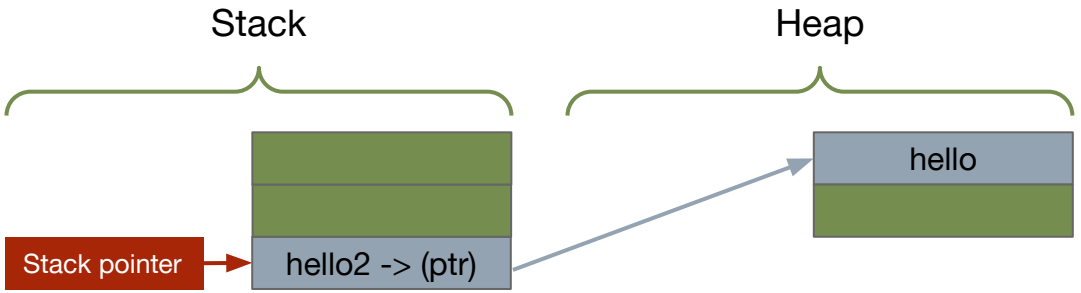


Example

Program counter

```
let hello = String::from("hello!");
let hello2 = hello;

println!("{hello}");
```

COPY AND MOVE: MOVE



Example

Program counter

```
let hello = String::from("hello!");
let hello2 = hello;

println!("{hello}");
```

COPY AND MOVE: COPY

Some data types have a Copy trait. "Passing" data from one scope or variable to another *copies* the data to the destination "trivially". The origin variable *does not* become out of scope.

```
let hello = 3;
let hello2 = hello;

println!("{hello}"); // NO compile error
println!("{hello2}"); // NO compile error
```

Example

COPY AND MOVE: CLONE

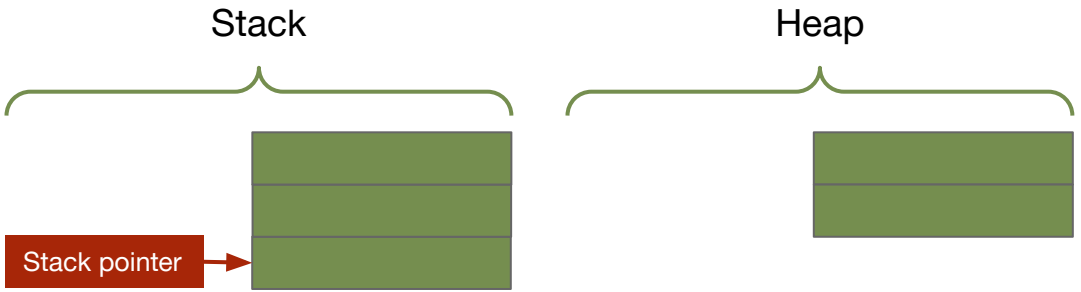
Optionally, data types that can only be moved may have the *Clone* trait, which copies *both* the stack and heap data to a destination. In this case, the origin variable *does not* become out of scope.

Cloning is usually computationally expensive, but may be useful in many cases.

```
let hello = String::from("hello");
let hello2 = hello.clone();

println!("{hello}"); // NO compile error
println!("{hello2}"); // NO compile error
```

Example

COPY AND MOVE: COPY

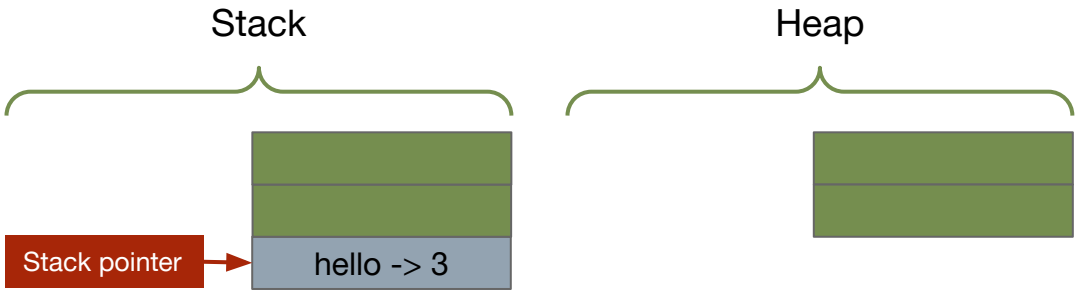


Example

Program counter

```
let hello = 3;
let hello2 = hello;

println!("{hello}");
println!("{hello2}");
```

COPY AND MOVE: COPY

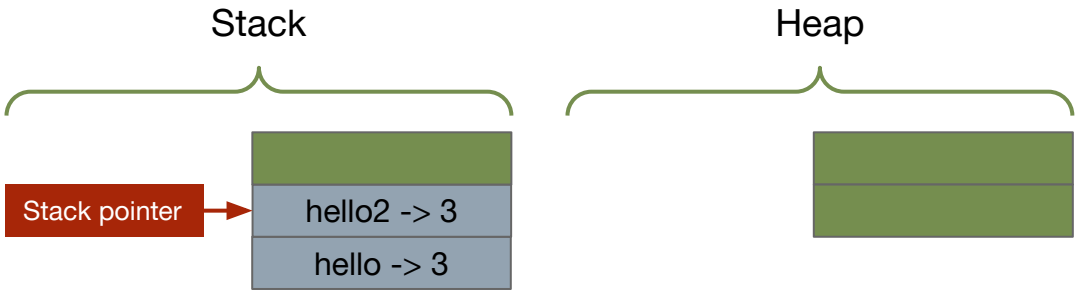


Example

Program counter

```
let hello = 3;
let hello2 = hello;

println!("{hello}");
println!("{hello2}");
```

COPY AND MOVE: COPY

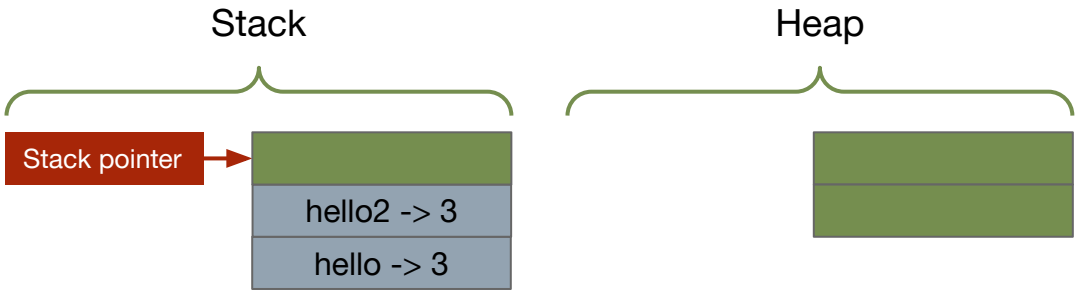


Example

Program counter

```
let hello = 3;
let hello2 = hello;

println!("{hello}");
println!("{hello2}");
```

COPY AND MOVE: COPY



Example

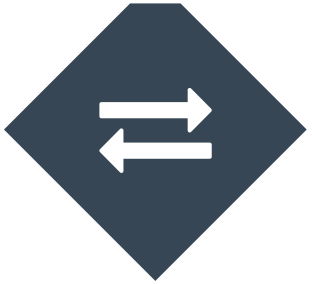
Program counter

```
let hello = 3;
let hello2 = hello;

println!("{hello}");
println!("{hello2}");
```

COPY AND MOVE: DATA TYPES

Copy

- Integers (signed and unsigned)
- Boolean
- Floating-point types
- Character
- Tuples whose every element has a Copy trait



Move

- Assume everything else

OWNERSHIP: FUNCTIONS

The Move and Copy traits are also applicable when passing data to functions.

For data types with the Move trait, the function *takes ownership* of the passed data, rendering the origin variable out of scope.

Example

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

fn take_copy(d: i64) {
    println!("{d}");
}

let s = String::from("z");
let d = 7;

take(s);
take_copy(d);

// compile error
println!("{s}");
println!("{d}"); // OK
```

OWNERSHIP: FUNCTIONS

Functions can also *give ownership* of data via return values. Although the function becomes out of scope, the data itself is *not dropped* due to the transfer.

Similarly, we can pass data to functions, which they can consequently return once it returns. This *borrowing* is a common procedure in Rust.

Example

```
fn gtake(s: String) {
    s
}

fn give() {
    120
}

let s = String::from("z");
let s2 = give();
let s3 = gtake(s);

println!("{s2}"); // OK
println!("{s3}"); // OK
```

REFERENCES

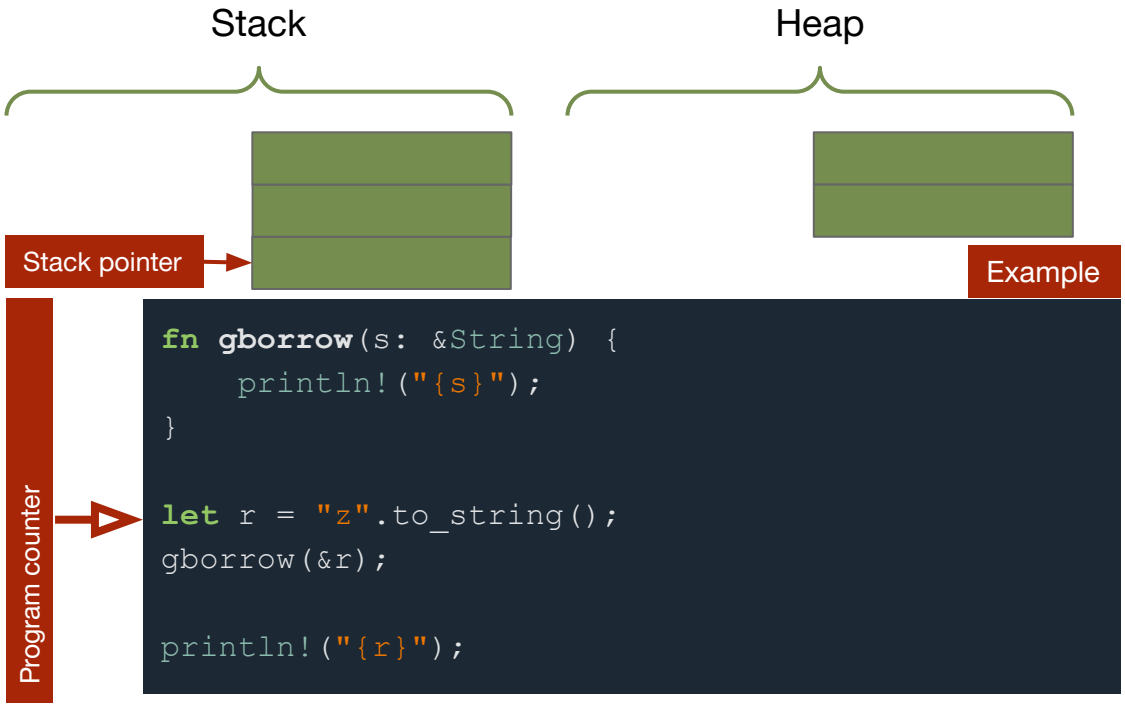
A **reference** to a data enables *lending* of such data to a function. When a reference is passed to a function, the function *does not* own the data.

A reference of a variable can be retrieved by writing an ampersand (&) before it. A reference data type is also prefixed as such.

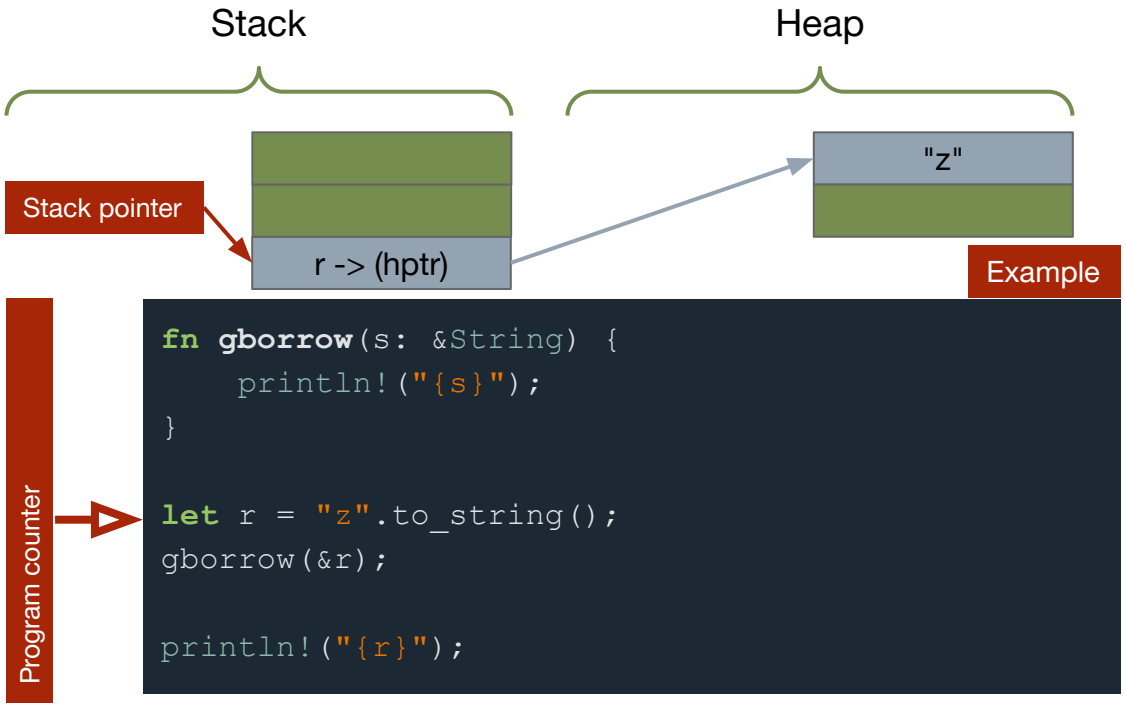
Example

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

let r = "z".to_string();
gborrow(&r);

// r is still valid!
println!("{r}");
```

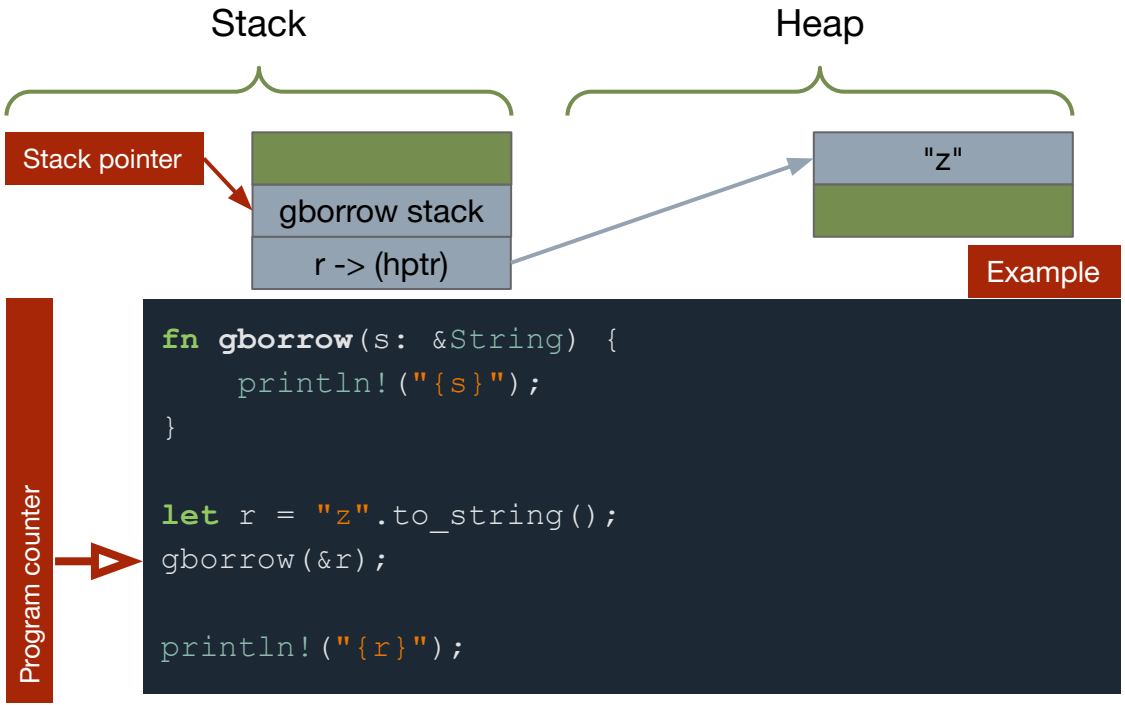
REFERENCES


Stack
Heap
Stack pointer
Example
Program counter
fn gborrow(s: &String) {
println!("{s}");
}
let r = "z".to_string();
gborrow(&r);
println!("{r}");

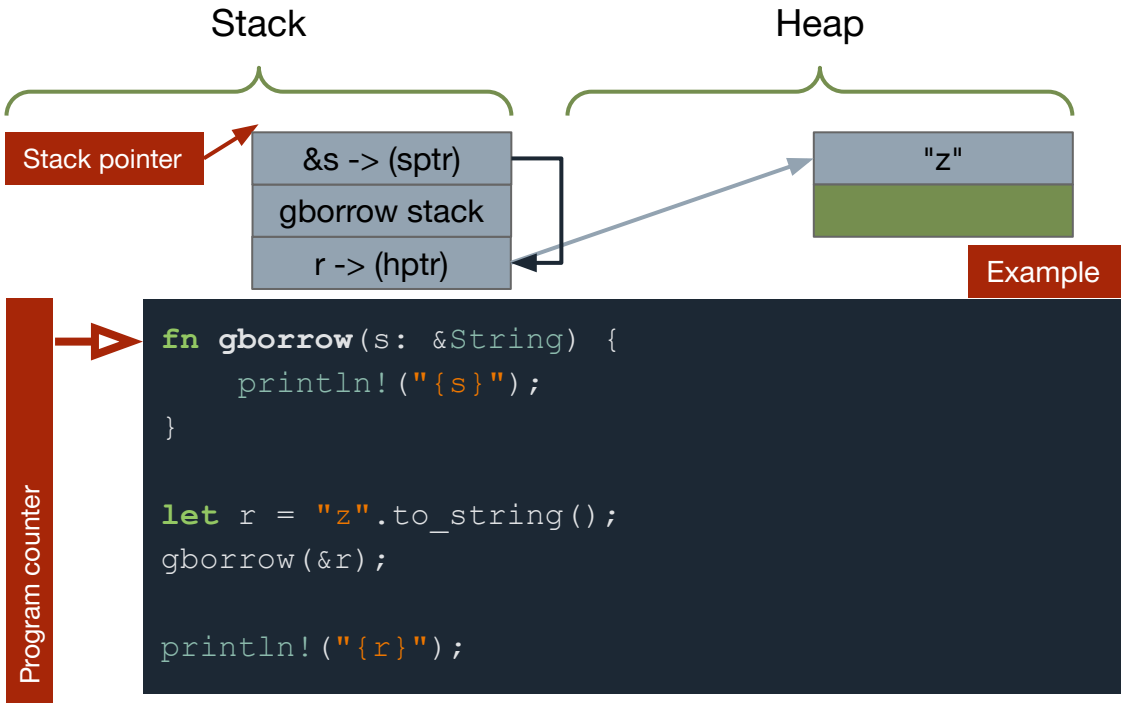
REFERENCES


Stack
Heap
Stack pointer
r -> (hptr)
"z"
Example
Program counter
fn gborrow(s: &String) {
println!("{s}");
}
let r = "z".to_string();
gborrow(&r);
println!("{r}");

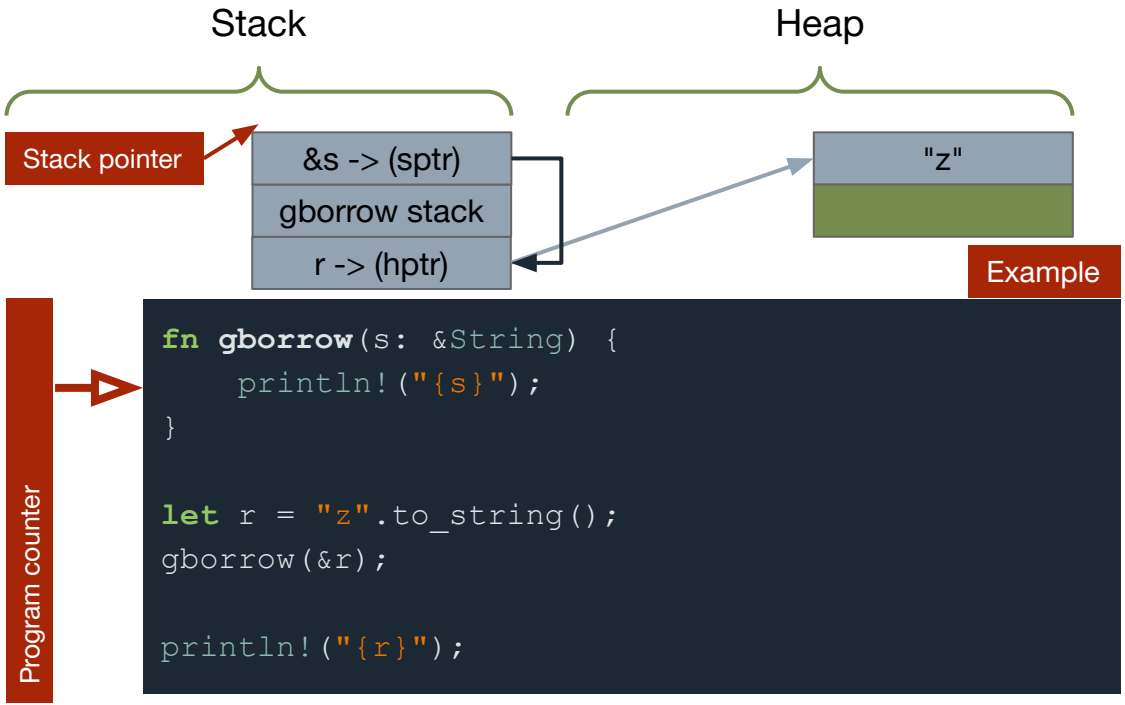
REFERENCES


Stack
Heap
Stack pointer
gborrow stack
r -> (hptr)
"z"
Example
Program counter
fn gborrow(s: &String) {
println!("{s}");
}
let r = "z".to_string();
gborrow(&r);
println!("{r}");

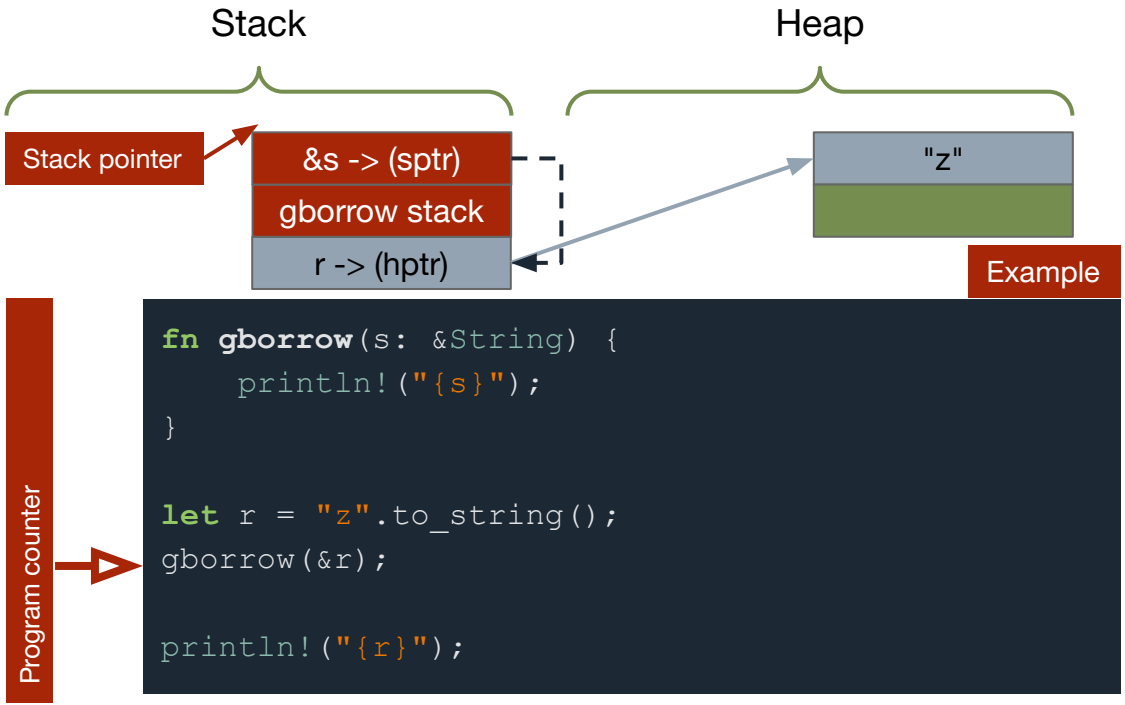
REFERENCES


Stack
Heap
Stack pointer
&s -> (sptr)
gborrow stack
r -> (hptr)
"z"
Example
Program counter
fn gborrow(s: &String) {
println!("{s}");
}
let r = "z".to_string();
gborrow(&r);
println!("{r}");

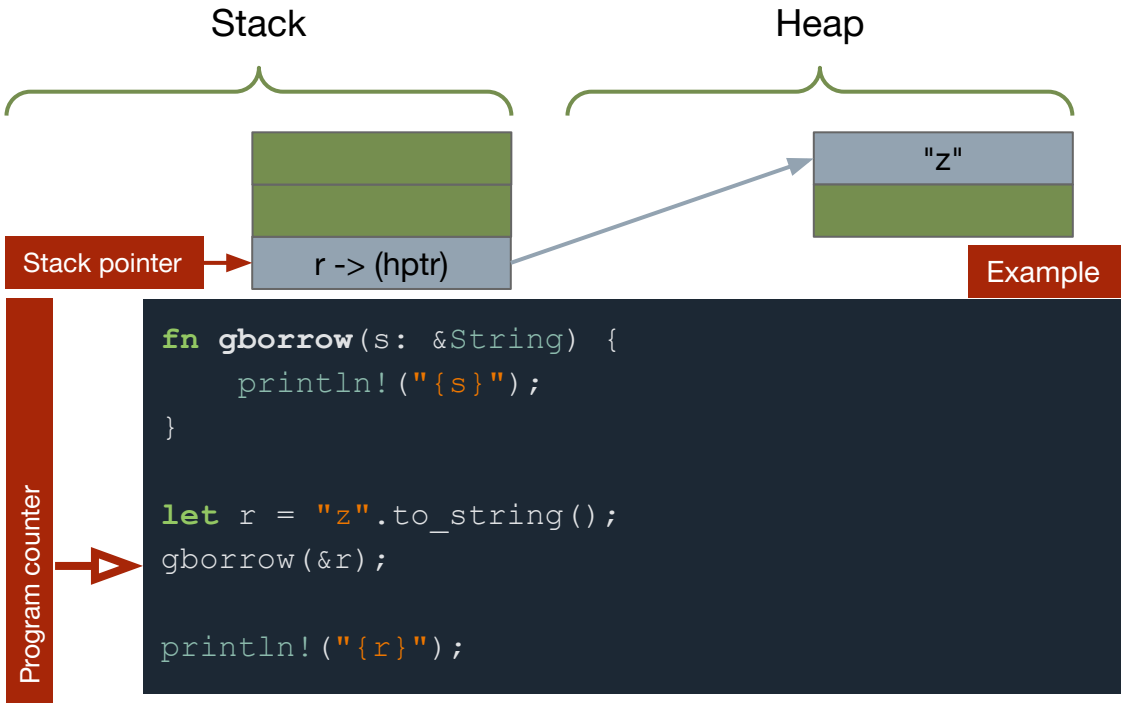
REFERENCES


Stack
Heap
Stack pointer
&s -> (sptr)
gborrow stack
r -> (hptr)
"z"
Example
Program counter
fn gborrow(s: &String) {
println!("{s}");
}
let r = "z".to_string();
gborrow(&r);
println!("{r}");

REFERENCES


Stack
Heap
Stack pointer
&s -> (sptr)
gborrow stack
r -> (hptr)
"z"
Example
Program counter
fn gborrow(s: &String) {
println!("{s}");
}
let r = "z".to_string();
gborrow(&r);
println!("{r}");

REFERENCES


Stack
Heap
"z"
Stack pointer
r -> (hptr)
Example
Program counter
fn gborrow(s: &String) {
println!("{s}");
}
let r = "z".to_string();
gborrow(&r);
println!("{r}");

REFERENCES: MUTABLES

Data pointed to by references are by default *immutable*. However, we can make a reference mutable by adding the `mut` keyword after the ampersand.

The variable should itself be mutable to be able to get a mutable reference.

Example

```
fn gmut(s: &mut String) {
    s.push_str("za");
}

let mut r = "z".to_string();
gmut(&mut r);

// r contains "zza"
println!("{r}");
```

REFERENCES: MUTABLES

We can only have one mutable reference at any given time.

Additionally, we can either have only one mutable reference or any number of immutable references at any given time.

Example

```
let mut h = String::from("hi");

// Cannot be!
let hptr = &mut h;
let hptr2 = &mut h;

hptr.push_str("!!!");

// Cannot be either!
let a = &h;
let b = &h;
let hptr3 = &mut h;

println!("{a} {b}");
```

REFERENCES: DANGLES

It is possible to create functions that return a reference. However, if the reference points to data that will become out of scope soon, it is called a **dangling reference**. The Rust compiler ensures that such cases will not happen.

```
fn hello_dangle() -> &String {
    let a = String::from("hello!");

    // after function exit, a will be invalid
    &a
}
```

Example

REFERENCES RULES

Rust follows some basic rules regarding ownership of a data.

1. There can only *either* be one mutable *or* any number of immutable references at any given time
2. References must always be valid.

RESOURCES

- The Rust Book

CoE 164

Computing Platforms

02a: Data Ownership