The Stack and Heap

CS392: Rust, in Theory and in Practice

September 9, 2025 (Lecture 3)

Outline

The Punchline: Ownership

The notion of ownership is based on two simple rules:

1. Every value has one **owner** at a given time
2. When the owner of a value **goes out of scope**, any memory associated with the value is **freed**

Areas of Memory

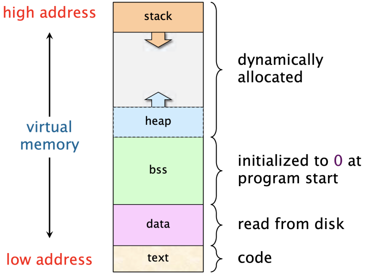
1. **Static Memory.** Where global variables are stored
2. **The Stack.** Where data local to a function call are stored
3. **The Heap.** Where persistent dynamically-sized data are stored

We will focus on the last two: the stack and the heap

Typical Memory Layout

The stack typically grows down and the heap grows up

The stack is often very small, something like 8MB



https://martinlwx.github.io/

Outline

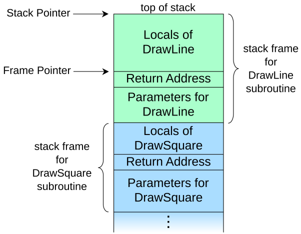
The Stack

The stack store local variables for function calls

It holds **activation records** or **call frames** which include extra data required by the function

It's fast to access, it's "right there"

It's well-organize, no wasted space



https://commons.wikimedia.org

What goes on the stack?

Anything whose size is fixed and known at compile time:

- primitives like numbers, string slices, arrays
- references

and which is not needed after control is returned to the function caller

Basic Example

```
fn bar() {
    let _z = 4;
    let _a = 5;
}

fn foo() {
    let _x = 2;
    let _y = 3;
    bar();
}

fn main() {
    let _w = 1;
    foo()
}
```

The "Problem" with the Stack

Not everything has fixed size known at compile time

We often want data that we can refer to *after* a function has returned control

These are things we do when we program

Growing Data

```
fn foo(n: i32, s: &mut String) {
    let _y = 2;
    for _ in 0..n {
        *s += "okay";
    }
}
fn main() {
    let mut x = String::default();
    foo(10, &mut x);
    println!("{x}");
}
```

Disappearing Data

```rust
fn fill(z : &mut &i32) {
    let w = 42;
    *z = &w;
}

fn main() {
    let x = 10;
    let mut y = &x;
    fill(&mut y);
    println!("{y}")
}
```

Outline

The Heap

The heap stores data that cannot be put on the stack (or in static memory)

It's slow to access, we have to follow *references*

It's less efficiently organized, it may become *fragmented* over time

But there's a lot of it and it's very flexible

What goes on the heap?

Dynamically-sized persistent data:

- Strings, Vectors, Hashmaps
- Pretty much everything other than references and primitive data.

We need the the heap to do "real" programming

Memory Allocation

In rough terms, a memory allocator figures out how to layout data in the heap. This means:

- finding an open spot of the right size
- returning the *address* of the beginning of the spot chosen

Memory Allocator (C)

```
int main(void) {
  int *x = (int*)malloc(sizeof(int));
  int *y = (int*)malloc(sizeof(int));
  int *z = (int*)malloc(sizeof(int));
  free(y);
  int *a =
    (int*)malloc(sizeof(int) * 10);
  int *b = (int*)malloc(sizeof(int));
  free(x);
  free(z);
  free(a);
  free(b);
  return 0;
}
```

Memory Bugs

Once we start *referring* to data on the heap, we're also able to create more problems:

- **Dangling Pointers.** references to invalid data
- **Memory Leaks.** Losing references to valid data
- **Data Races.** undefined behavior caused by changing the same data with multiple processes

Outline

Four Kinds of Memory Management

1. Explicit allocation/deallocation (C)
2. Ownership (Rust)
3. Automatic Reference Counting (Swift)
4. Garbage Collection (Python, Java, OCaml,. . .)

Explicit Allocation

```
int main(void) {
  int *x = (int*)malloc(sizeof(int)); // allocation
  printf("%d\n", *x);
  free(x); // deallocation
  return 0;
}
```

The approach of "traditional" systems languages like C: *the programmer is in charge of managing allocation/deallocation*

malloc allocates data on the heap and **free** deallocates it so it can be used again.

Benefits: It's simple and general

Downsides: It's highly prone to error

Dangling Pointer (C)

```c
int main(void) {
  int *x = (int*)malloc(sizeof(int));
  *x = 2;
  free(x);
  printf("%d\n", *x);
  return 0;
}
```

Memory Leak (C)

```
void leak(void) {
  int *x = (int*)malloc(sizeof(int));
  *x = 2;
  printf("%d\n", *x);
}

int main(void) {
  leak();
  return 0;
}
```

Garbage Collection

The approach of modern high-level languages: *periodically check the stack for what heap data is still valid and then clean up the heap*

Benefits: Easy on the programmer, works fine in most cases

Downsides: Very little programmer control, difficult to performance optimize

Rough Sketch

1. DFS from stack and mark "alive" data
2. Sweep the heap and clear unmarked data

Automatic Reference Counting

```
class Stuff {
    init() { print("allocating") }
    deinit() { print("deallocating") }
}
var r1 : Stuff? = Stuff()
var r2 : Stuff? = r1
r1 = nil
r2 = nil
```

The approach taken by Swift (and C++ and Rust via smart pointers): *Count the number of references to a piece of heap data, free when it's down to zero*

Benefits: Easy on the programmer like GC

Downsides: Reference cycles, overhead (?), still not much control

Ownership

The approach taken by Rust: *follow these two rules:*

1. *Every value has one* ***owner*** *at any given time*
2. *When the owner of a value goes* ***out of scope****, any memory associated with the value is freed*

Benefits: User-control without requiring explicit allocation

Downsides: Has a learning curve, often needs to be side-stepped

The Big Question

```
fn foo() {
    x = String::from("foo");
    println!("x: {x}");
    // the data associated with x is dropped here
}
```

If we're not explicitly allocating/deallocating memory, when should it happen?

Rust's answer: as soon as a variable/parameter referring to it goes out of scope

This allowes for a stupid-simple, cheap deallocation pattern, **at the expense of not being able to do "intuitive" things**

No References to the Same Data

```
fn main() {
    let x = String::from("hello world");
    let y = x;
    println!("{x}");
    println!("{y}");
}
```

It's not possible to have two references to the same piece of data

A Note on the Philosophy of Rust

```
int main(void) {
  char* x = "hello world";
  char* y = x;
  printf("%s\n", x);
  printf("%s\n", y);
  return 0;
}
```

The type/borrow checker disallows a lot of "natural" programs

Working with your hand tied behind your back makes you better with that one hand

Outline

Drop

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

For data on the heap, when a variable goes out of scope, Rust calls a function called **drop** on its value to return the memory

It's like adding **free(x)** at the end of the block

Drop

```
fn main() {
    let mut x = String::from("x");
    x = String::from("y");
    println!("{x}");
}
```

There is also an implicit drop call when a value is replaced

Again, drop applies to values

Move

```
// THIS DOES NOT COMPILE
fn main() {
    let x = String::from("x")
        let y = x;
    println!("{x}");
    println!("{y}");
}
```

For data on the heap, memory needs to be returned when the owner goes out of scope

Data on the heap must be **moved** on assignment (really, the pointer must be given up)

In this example, **y** owns the one copy of the string that **x** originally owned

Move

```
fn foo(mut x : String) -> String {
    x.push_str("y");
    x
}

fn main() {
    let x = String::from("x");
    let y = foo(x);
    println!("{0}", y);
}
```

Moves also happen at return values

Ownership is transferred to the parameter of **foo**, and then given to **y** as the return value of **foo**

Copy

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

For data on the stack, there is no memory to return

Data on stack can be **copied** on assignment

x and **y** both own a copy of the value **5**

What's copied and what's moved?

Short answer: Stack data is copied, heap data is moved

Long answer: Everything is moved except for those types which implement the Copy trait

We'll talk about traits later, they're like *type classes* or *interfaces*

Outline

Immutable References

```
fn length(x : &String) -> i32 {
    let mut count = 0;
    for _ in x.chars() {
        count += 1
    }
    count
}

fn main() {
    let x = String::from("xyz");
    let y = length(&x);
    println!("{}", y);
}
```

A reference is like a pointer, except that it's *guaranteed* to point at a valid value

A Note on Dereferencing

```
fn foo(x : &String) {
let _ : &String = x;
let _ : String = *x;
let _ : str = **x;
}
```

It is also possible to dereference, and this looks a bit more like a pointer, but the behavior can sometimes be unclear

Deref is a trait (like Copy) and the behavior of dereferencing can include implicit coercions

Mutable References

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

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

Mutable references are the same, except that we're allowed to update the associated value

We can only have one mutable reference at a time

No Data Races

```
fn main() {
    let mut s = String::from("hello");
    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s;
    println!("{}, {}, and {}", r1, r2, r3);
}
```

There can be no immutable references if there is a single mutable reference

No immutable reference can get different "views" of the same data

No Dangling Pointers

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

We cannot use references data within the scope of the function as return values

We'll see that *lifetimes* are actually what cause the compile-time error

Summary

Ownership allows for simple (but restrictive) memory management

References gives us a convenient (but restrictive) interface to owned values without having to pass around ownership

We're allowed *either* one mutable reference *or* multiple immutable references

These restrictions give us strong guarantees about memory allocation

Outline

Introduction

The Stack

The Heap

Memory Management

References

Borrowing

Workshop

Task

- Finish up Assignment 1
- Get ahead of reading for the next class