Introduction to Rust

Please follow these instructions to get set up:

`github.com/rtfeldman/rust-1.51-workshop`

Introduction

What is Rust?

Who uses Rust?

Why & why not use Rust?

Workshop Structure

rust-lang.org

“ A language empowering everyone to build **reliable** and **efficient** software.”

Rust

compiles to either

machine code

```
0111010110
1011001111
0110110101
1001001011
```

Who uses Rust?

More at rust-lang.org/production

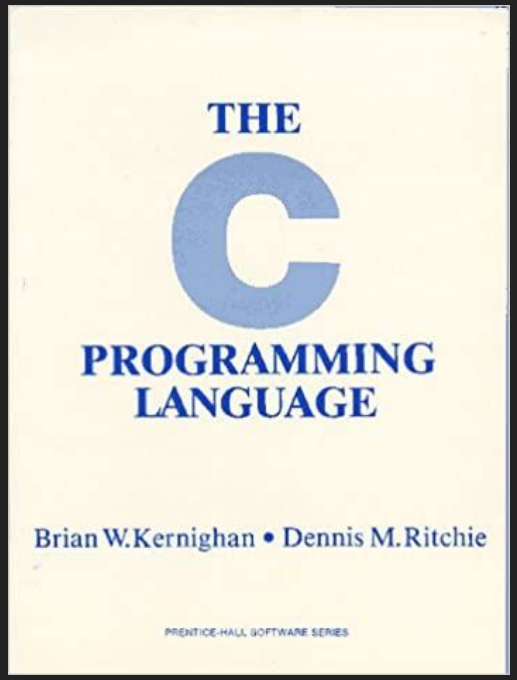
What can I build with Rust?

- Web servers
- Command-Line Interfaces
- Native desktop applications
- In-browser apps via WebAssembly
  - https://makepad.dev
- Performance-intensive libraries
- Operating systems (!)

Why use Rust?

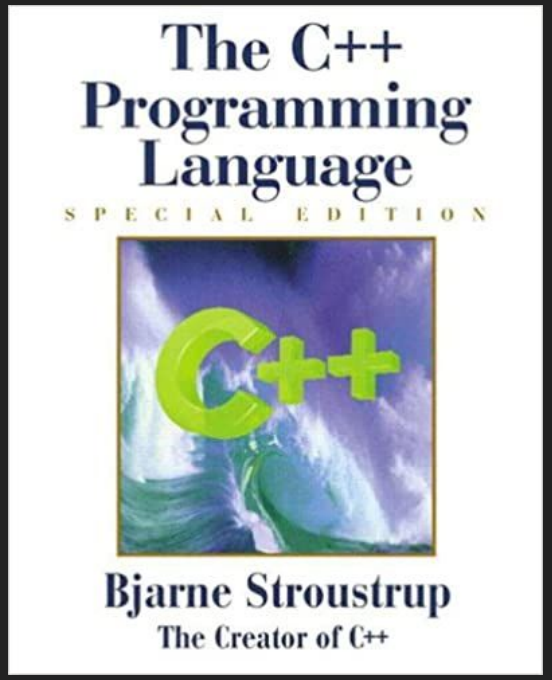
1. Speed
2. Performance
3. Going Real Fast

1972



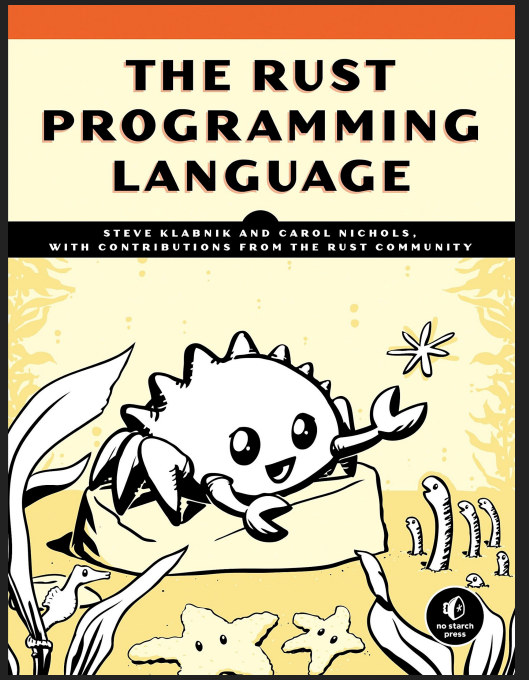
efficient

1985



efficient
OOP

2010



efficient
reliable
ergonomic

Why use Rust?

```
error[E0609]: no field `emial` on type `User`
  --> main.rs:12:22
   |
12 |     let email = user.emial;
   |                      ^^^^^ unknown field
   |
   = note: available fields are: `name`, `email`
```

```
main.cpp:14:22: error: no member named 'emial' in 'User'
    auto email = user.emial;
                      ~~~~ ^
```

```
error[E0609]: no field `emial` on type `User`
  --> main.rs:12:22
   |
12 |     let email = user.emial;
   |                      ^^^^^ unknown field
   |
   = note: available fields are: `name`, `email`
```

elm-lang.org

```
-- TYPE MISMATCH ---------------------------------------------------- Main.elm

This `user` record does not have a `emial` field:

14|             user.emial
                 ^^^^^

This is usually a typo. Here are the `user` fields that are most similar:

    { email : String
    , name : String
    }

So maybe emial should be email?
```

```
error[E0609]: no field `emial` on type `User`
  --> main.rs:12:22
   |
12 |     let email = user.emial;
   |                      ^^^^^ unknown field
   |
   = note: available fields are: `name`, `email`
```

```
main.cpp:14:22: error: no member named 'emial' in 'User'
    auto email = user.emial;
                 ~~~~ ^
```

Why use Rust?

C/C++ level performance, with…

…nice ergonomics and a language server

…automatic memory management

…a package manager and code formatter

…more compiler help with concurrency

…lots of compiler help for big code bases

Why not use Rust?

Rust is a big language - lots to learn!

smaller ecosystem than C/C++ (but FFI)

slower iteration cycle than most languages

- strict compiler
- satisfying (“fighting”) the borrow checker
- slow compile times for full builds
- tests can take awhile to build

safer than C++, less safe than e.g. pure FP

stackoverflow

“most loved programming language”

2016 - Rust
2017 - Rust
2018 - Rust
2019 - Rust
2020 - Rust

Workshop Structure

7 sections

- 30-40 minutes of slides (please ask questions!)
- 15-20 minutes of exercises (and solutions)

1 hour break for lunch

2-3 additional 10-15 minute breaks

Recap

What is Rust?

Who uses Rust?

Why & why not use Rust?

Workshop Structure

1. Primitives

strings

floats

integers

booleans

Hello, World!

app.rs

```
fn main() {
    println!("Hello, World!");
}
```

```
rustc app.rs
```

String Interpolation

```
fn main() {
    let greeting = "Hello";
    let subject = "World";

    println!("{}, {}!", greeting, subject);
}
```

```
Hello, World!
```

String Interpolation

```
let subject = "World";
let greeting = format!("Hello, {}!", subject);

fn main() {
    let crash_reason = "Server wanted a nap.";

    panic!("I crashed! {}", crash_reason);

    println!("This will never get run.");
}
```

Floats

```
fn main() {
    let x = 1.1;
    let y = 2.2;

    println!("x times y is {}", x * y);
}
```

```
x times y is 2.4200000000000004
```

Mutability

```
let x = 1.1;

x = 2.2;
```

```
let mut x = 1.1;

x = 2.2;
```

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> main.rs:3:5
  |
2 |     let x = 1.1;
  |         -
  |         |
  |         first assignment to `x`
  |         help: make this binding mutable: `mut x`
3 |     x = 2.2;
  |     ^^^^^^^ cannot assign twice to immutable variable
```

Numeric Types

```
let mut y = 2.2;

y = 3.1;
y = "three point one";
    y changed types!
```

Type Annotations

```
fn main() {
    let x = 1.1; // x is a float
    let y = 2.2;

    println!("x times y is {}", x * y);
}
```

Type Annotations

```
fn main() {
    let x: f64 = 1.1;
    let y = 2.2;

    println!("x times y is {}", x * y);
}
```

Type Annotations

"`main` takes no arguments and returns nothing"

```
fn main() {
    let answer = multiply_both(1.1, 2.2);

    println!("1.1 × 2.2 = ", answer);
}
```

`!` means `println` is a *macro*, not a *function*

"`multiply_both` takes two `f64` arguments and returns an `f64`"

```
fn multiply_both(x: f64, y: f64) -> f64 {
    return x * y;
}
```

Float Sizes

```
                f64 has 64 bits (8 bytes) of storage
let x: f64 = 10.0 / 3.0;
                    3.33333333…
let y: f32 = 10.0 / 3.0;
                f32 has 32 bits (4 bytes) of storage
```

more memory used ***allows for more precision***

more memory used ***may slow down the program***

Integers

```
let ninety = 90;

let negative_five = -5;

let one_thousand = 1_000;

let exactly_three = 10 / 3;

let this_will_panic = 5 / 0; // kaboom!
```

Integer Sizes

`i8`	8 bits (1B)	-127 to 128
`i16`	16 bits (2B)	-32,768 to 32,767
`i32`	32 bits (4B)	…
`i64`	64 bits (8B)	
`i128`	128 bits (16B)	

Unsigned Integers

`u8`	0-255
`u16`	0-65,535
`u32`	0-4,294,967,295
`u64`	0-18,446,744,073,709,551,615
`u128`	0-170,141,183,460,469,231,731,687,303,715,884,105,728

`char` a `u32` that's been Unicode validated

Converting Numbers with as

```
fn multiply(x: i64, y: u8) -> i64 {
    return x * (y as i64);
}

fn divide(x: i32, y: u16) -> f64 {
    return x as f64 / y as f64;
}
```

Booleans

```
let should_we_go_fast = true;
let should_we_go_slow = false;

true as u8     // evaluates to 1

false as u8    // evaluates to 0

1 == 2         // evaluates to false
```

Conditionals

```
if cats > 1 {
    println!("Multiple cats!");
} else {
    println!("Need more cats!");
}
```

Conditionals

```
if cats > 1 {
    println!("Multiple cats!");
} else if cats > 1_000 {
    println!("Too many cats!");
}
```

Statements and Expressions

An expression evaluates to a value

```
cats > 1_000
```

```
cats > count_cats(cat_areas)
```

A statement does not evaluate to a value

```
println!("Multiple cats!");
```

Statements and Expressions

```
fn multiply_both(x: f64, y: f64) -> f64 {
    return x * y;
}
```

expression

statement

```
fn multiply_both(x: f64, y: f64) -> f64 {
    x * y
}
```

expression

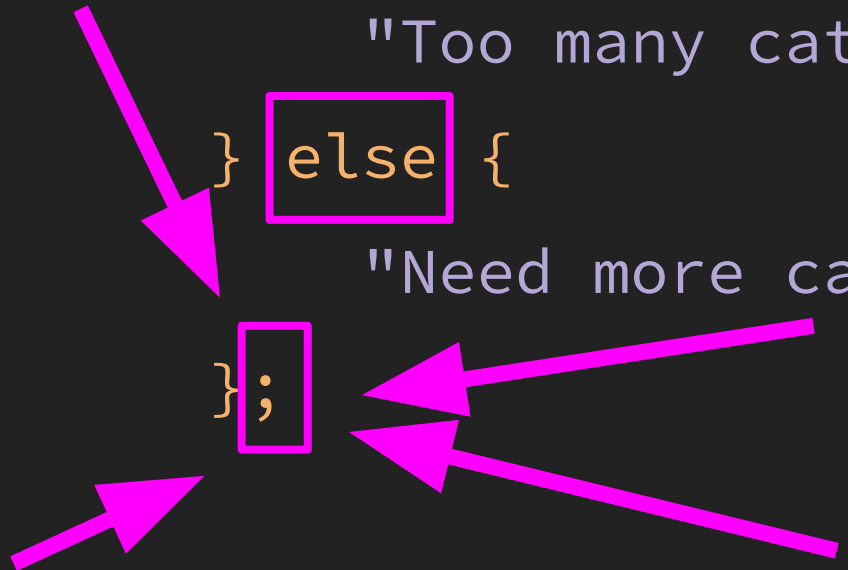
if a function ends with an expression,
it automatically returns that expression

Statements and Expressions

```
if cats > 1 {
    println!("Multiple cats!");
} else if cats > 1_000 {
    println!("Too many cats!");
} else {
    println!("Need more cats!");
}
```

Statements and Expressions

```
let message = if cats > 1 {
    "Multiple cats!"
} else if cats > 1_000 {
    "Too many cats!"
} else {
    "Need more cats!"
};
```



Statements and Expressions

```
let message = if cats > 1 {
    "Multiple cats!"
} else if cats > 1_000 {
    "Too many cats!"
} else {
    "Need more cats!"
};
```

Review of Part 1

strings

```
format!("Hi, {}!", name)
```

floats

```
let mut float: f64 = 1.234;
```

integers

```
let one: u32 = 1.99 as u32;
```

booleans

```
if x > 5 { true } else { false }
```

Exercises for Part 1

Follow the instructions in **`part1/README.md`**

2. Collections

tuples

structs

arrays

memory

Tuples

```rust
let point: (i64, i64, i64) = (0, 0, 0);

let x = point.0;
let y = point.1;
let z = point.2;

let (x, y, z) = point;
let (x, y, _) = point;
let (x, _, _) = point;
```

Tuples

```
let mut point: (i64, i64, i64) = (0, 0, 0);

point.0 = 17;
point.1 = 42;
point.2 = 90;
```

Unit

```
let unit: () = ();

fn main() {
    …
}

fn main() -> () {
    …
}

let println_return_val: () = println!("Hi!");
```

Structs

```
struct Point {
    x: i64,
    y: i64,
    z: i64,
}

fn new_point(x: i64, y: i64, z: i64) -> Point {
    Point { x: x, y: y, z: z }
}

fn new_point(x: i64, y: i64, z: i64) -> Point {
    Point { x, y, z }
}
```

Structs

```
struct Point {
    x: i64,
    y: i64,
    z: i64,
}

let point = Point { x: 1, y: 2, z: 3 };
let x = point.x;
let Point { x, y, z } = point;
let Point { x, y: _, z } = point;
let Point { x, z, .. } = point;
let Point { x, .. } = point;
```

Structs

```
struct Point {
    x: i64,
    y: i64,
    z: i64,
}

let mut point = Point { x: 1, y: 2, z: 3 };
point.x = 5;
```

Arrays

fixed-length

```
let mut years: [i32; 3] = [1995, 2000, 2005];

let first_year = years[0];
let [_, second_year, third_year] = years;

years[2] = 2010;
years[x] = 2010;  this will panic if it's out of bounds!
```

method call

```
for year in years.iter() {
    println!("Next year: {}", year + 1);
}
```

Arrays vs Tuples

```
let mut years: [i32; 3] = [1995, 2000, 2005];
```

```
    i32
for year in years.iter() {
    println!("Next year: {}", year + 1);
}
```

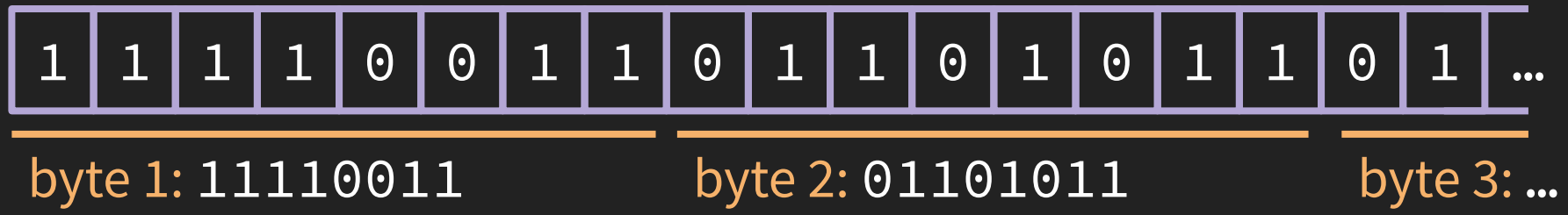
Arrays vs Tuples

Arrays can be iterated over
Tuples (and structs) cannot
Array elements must all have the same type

```
let years: (i32, i32, bool) =
                (1995, 2000, true);

for year in years.iter() {
    println!("Next year: {}", year + 1);
}
```

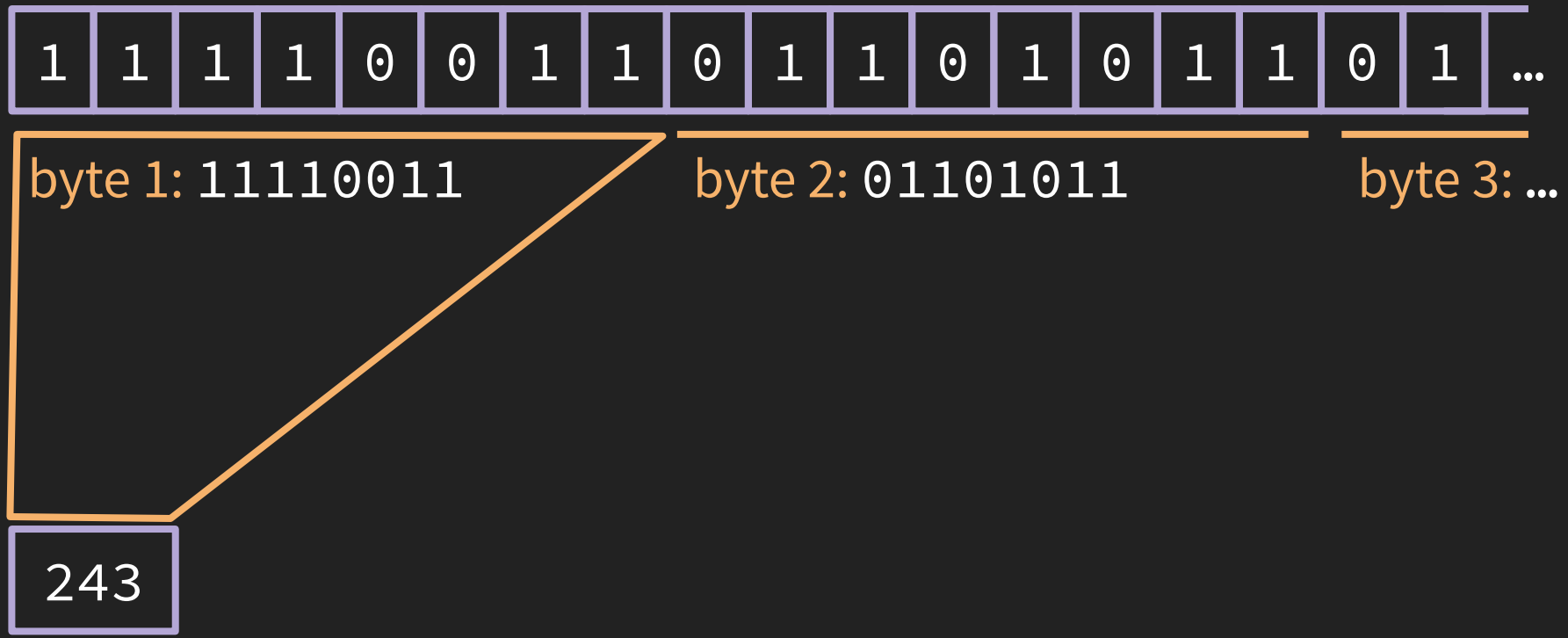
Memory

bits



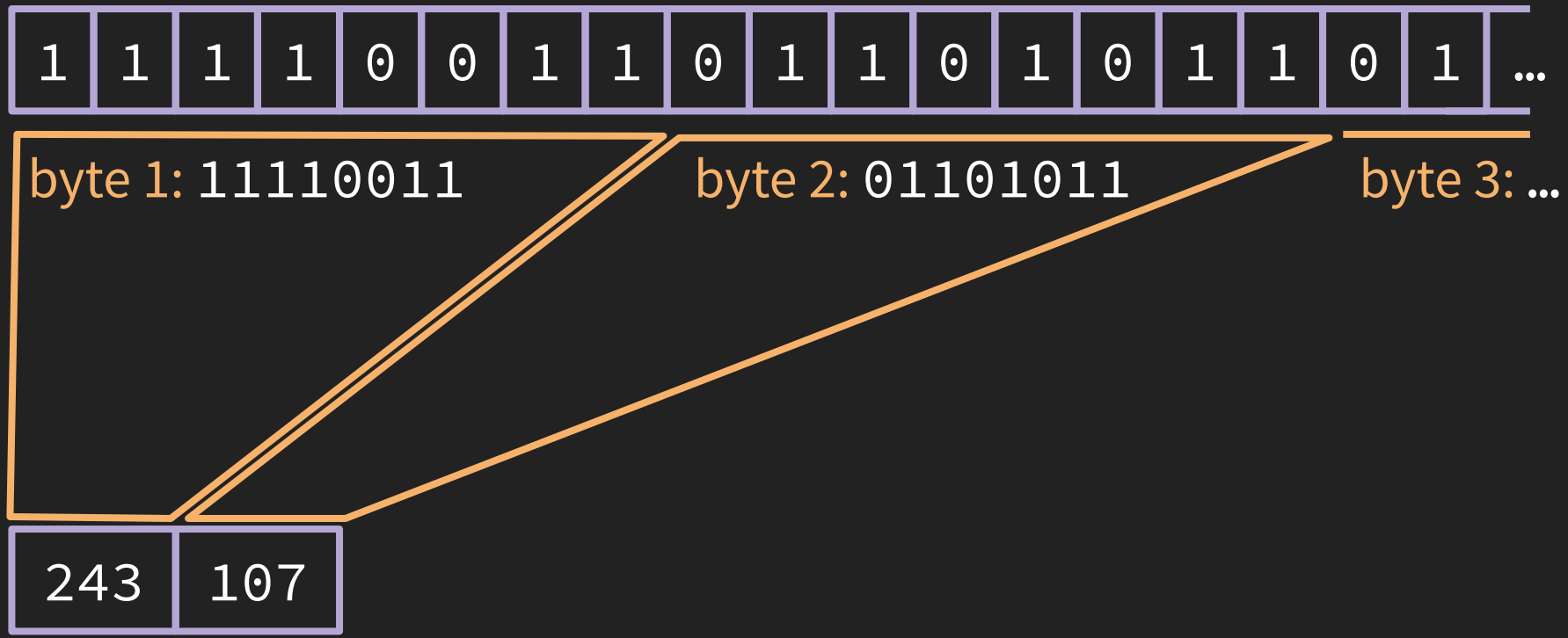
Memory

bits


1 1 1 1 0 0 1 1 0 1 1 0 1 0 1 1 0 1 ...
byte 1: 11110011
byte 2: 01101011
byte 3: ...
243

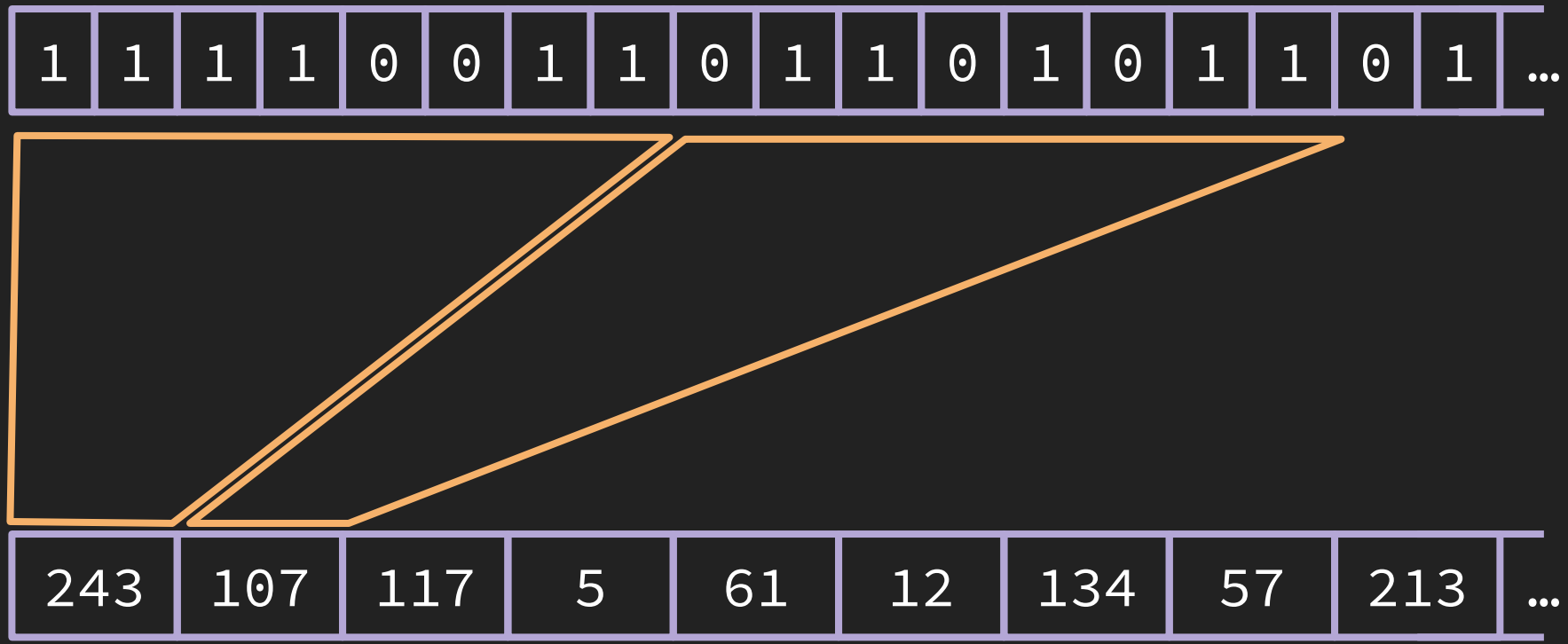
Memory

bits


1 1 1 1 0 0 1 1 0 1 1 0 1 0 1 1 0 1 ...
byte 1: 11110011
byte 2: 01101011
byte 3: ...
243
107

Memory

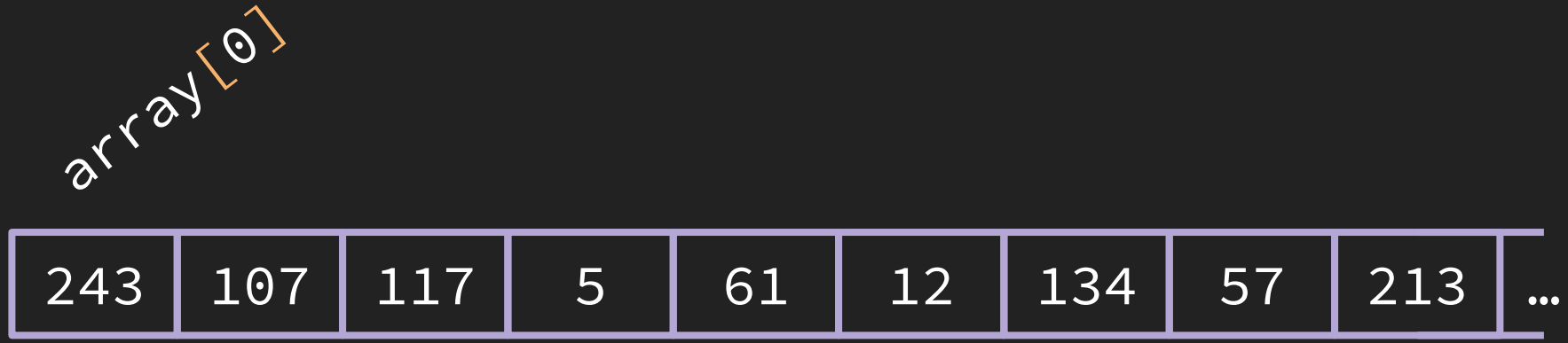
bits



bytes

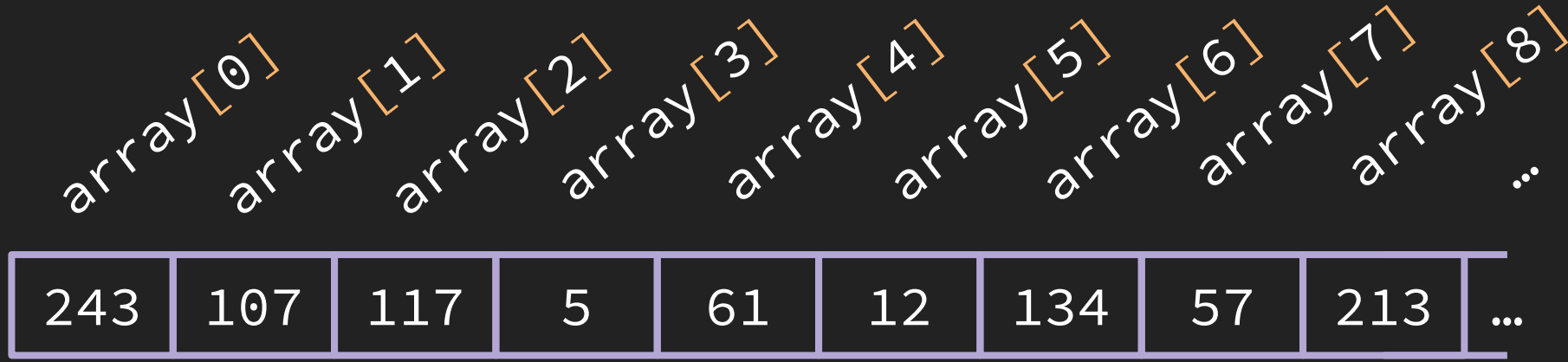
Memory

"u8 array"



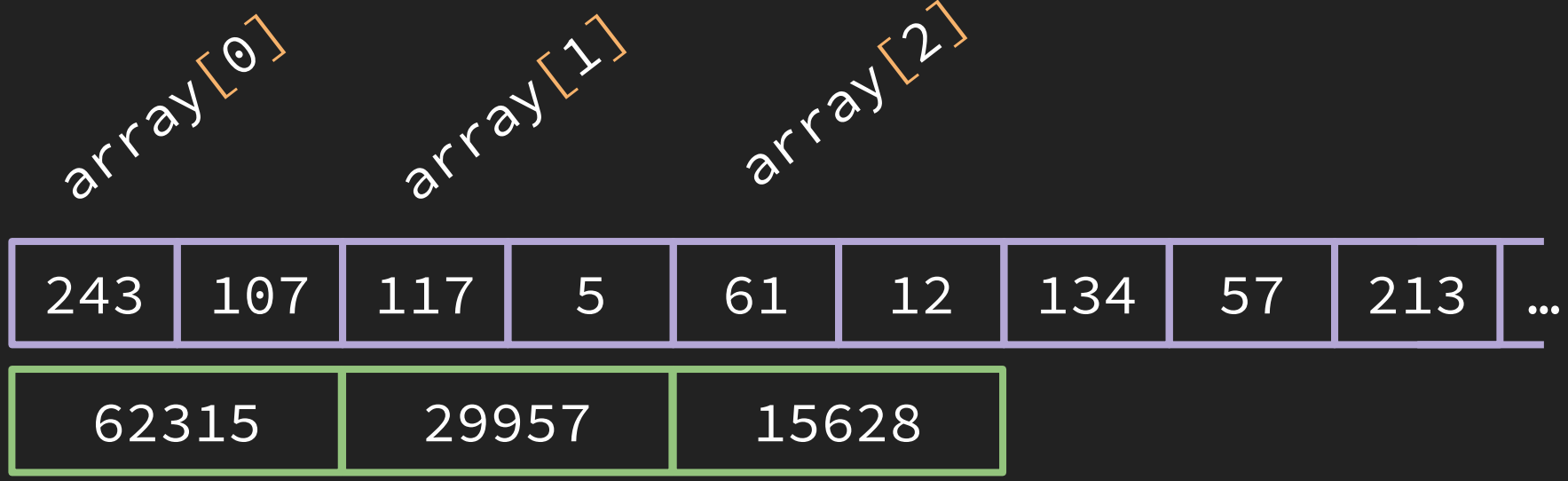
Memory

"u8 array"



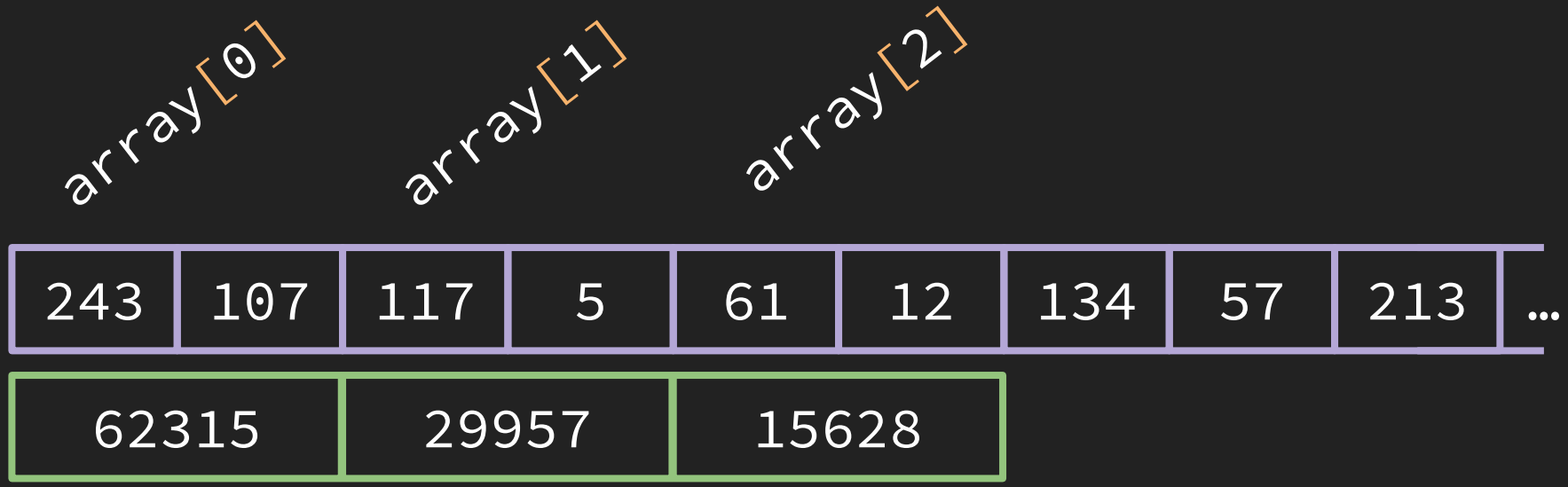
Memory

"`u16` array"



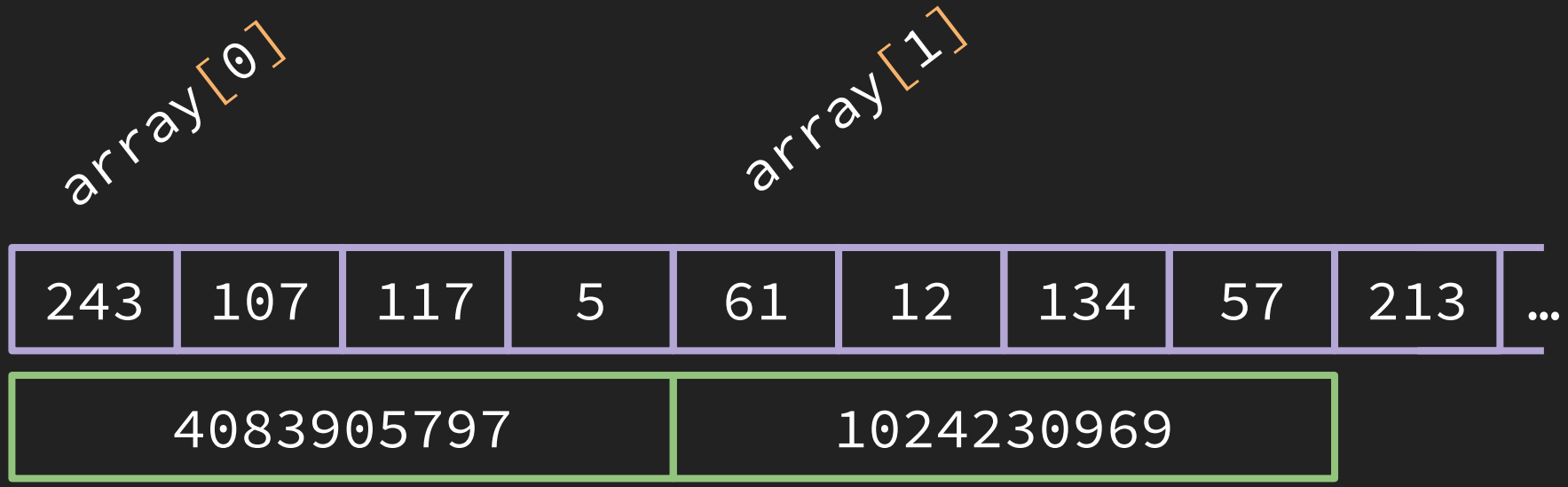
Memory

```
let array: [u16; 3] = [62315, 29957, 15628];
```



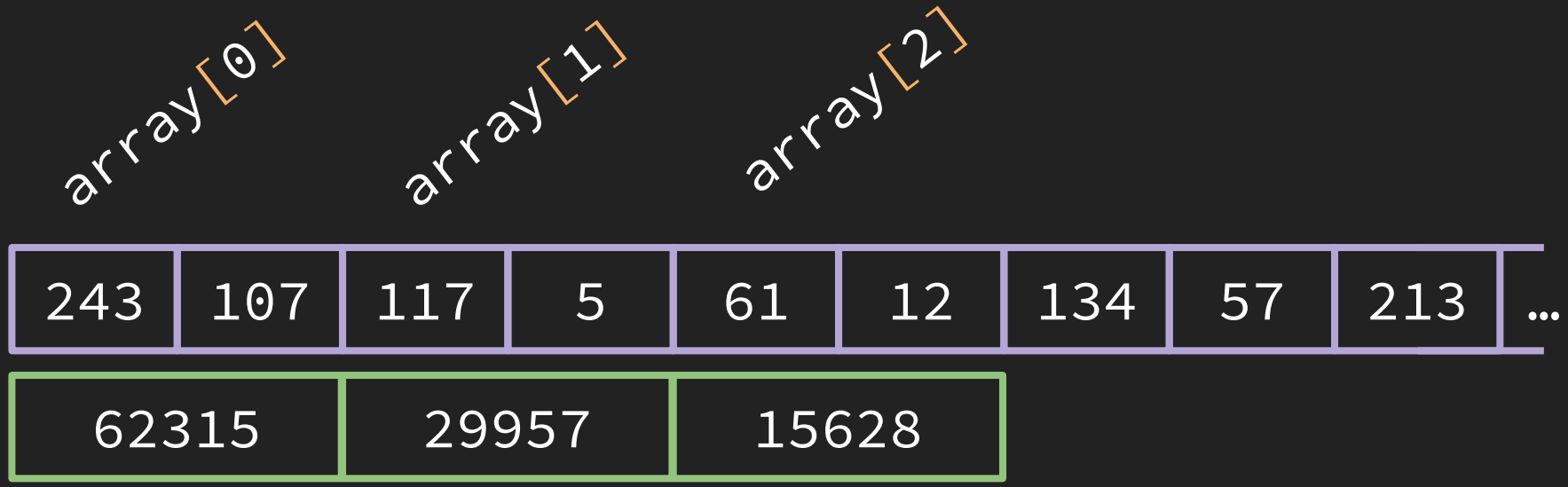
Memory

```
let array: [u32; 2] = [4083905797, 1024230969];
```



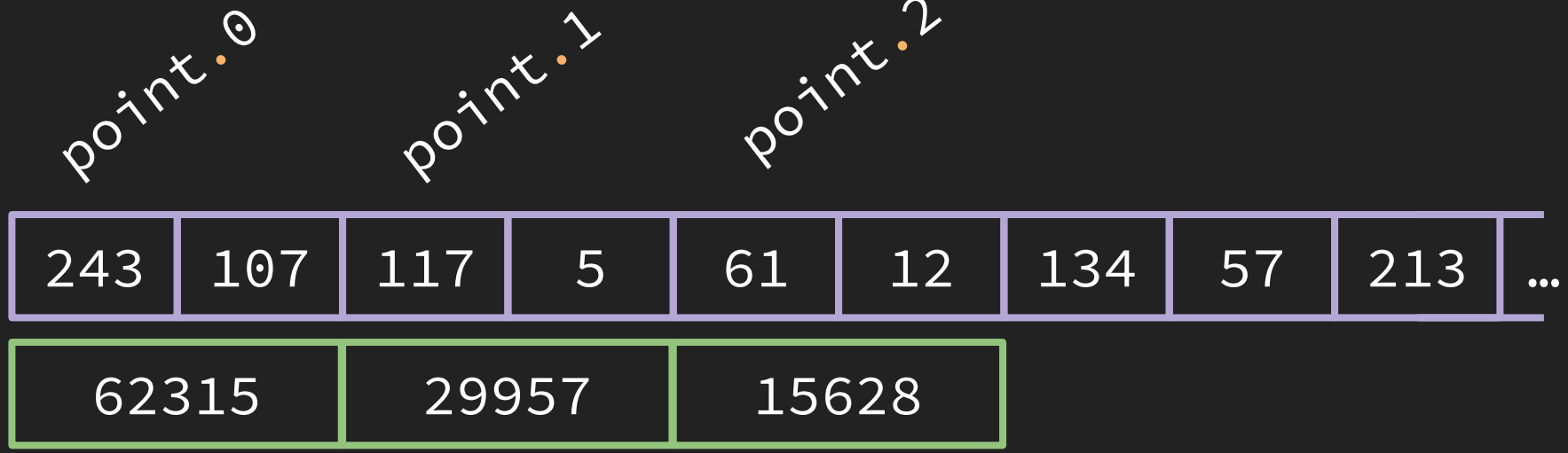
Memory

```
let array: [u16; 3] = [62315, 29957, 15628];
```



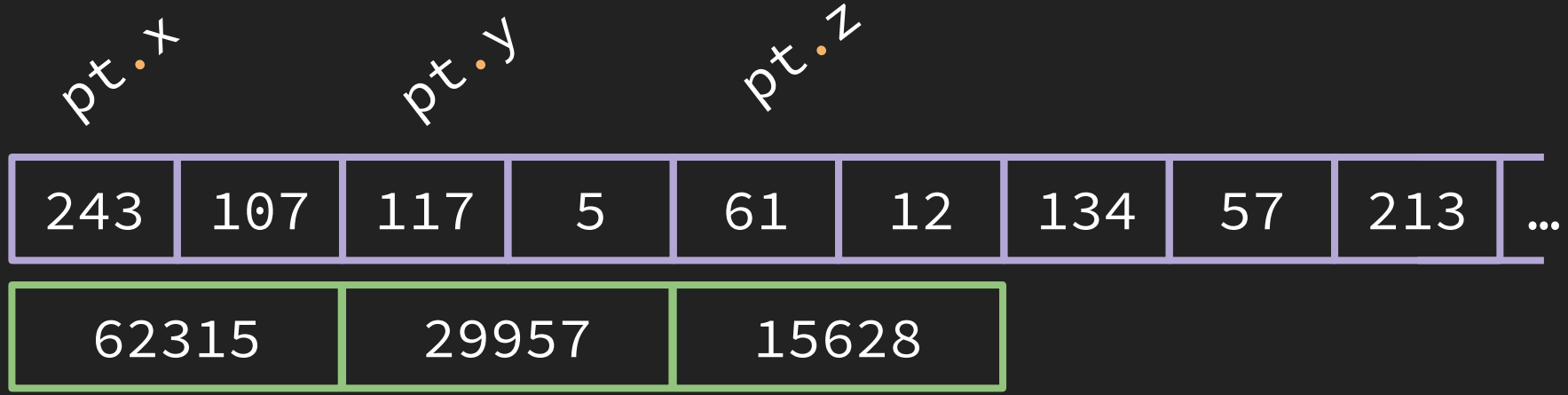
Memory

```
let point: (u16, u16, u16) =
                        (62315, 29957, 15628);
```



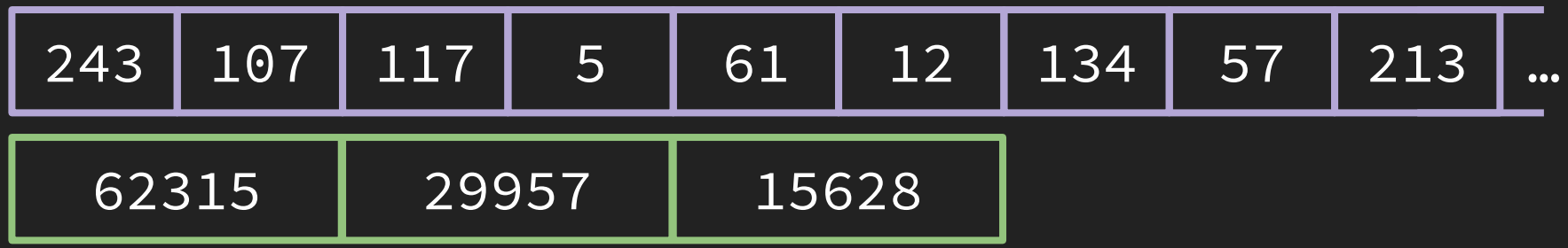
Memory

```
struct Point { x: u16, y: u16, z: u16 }


let pt: Point = Point {62315, 29957, 15628};
```



Memory

```
struct Point { x: u16, y: u16, z: u16 }

let point: (u16, u16, u16) =
                          (62315, 29957, 15628);
let pt: Point = Point {62315, 29957, 15628};
let array: [u16; 3] = [62315, 29957, 15628];
```



Tuples, Structs, Arrays

Arrays can be iterated over

Tuples and structs cannot

Array elements must all have the same type

Tuple and struct fields can be different types

In memory, they're all represented as adjacent bytes with no extra metadata

Review of Part 2

tuples `let foo: (i64, bool) = (1, true);`

structs `struct Foo { x: i64, is_up: bool }`

arrays `let arr: [u32; 3] = [1, 2, 3];`

memory `u8` is 8 bits (1 byte), `u16` is 16 bits (2 bytes), …

Exercises for Part 2

Follow the instructions in **`part2/README.md`**

3. Pattern Matching

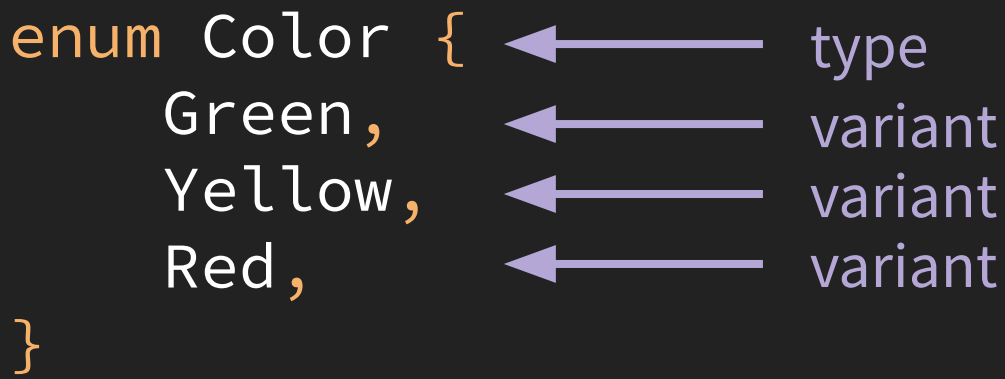
enums

pattern matching

methods

type parameters

Enums




```
let go = Color::Green;
let stop = Color::Red;
let slow_down = Color::Yellow;

let go: Color = Color::Green;
let stop: Color = Color::Red;
let slow_down: Color = Color::Yellow;
```

Enums

```
enum Color {
    Green,
    Yellow,
    Red,
    Custom { red: u8, green: u8, blue: u8 }
}

let go: Color = Color::Green;
let stop: Color = Color::Red;
let purple: Color = Color::Custom {
    red: 100, green: 0, blue: 250
};
```

Enums

```
enum Color {
    Green,
    Yellow,
    Red,
    Custom(u8, u8, u8)
}

let purple: Color = Color::Custom(100, 0, 250);
```

Pattern Matching

```
let current_color = Color::Yellow;

match current_color {
    Color::Green => {
        println!("It was green!");
    }
    Color::Yellow => {
        println!("It was yellow!");
    }
}
```

Pattern Matching

```
match current_color {
    Color::Green => {
        println!("It was green!");
    }
    Color::Yellow => {
        println!("It was yellow!");
    }
    Color::Custom { red, green, blue } => {
        println!("{} {} {}", red, green, blue);
    }
}
```

Pattern Matching

```
match current_color {
    Color::Green => {
        println!("It was green!");
    }
    Color::Yellow => {
        println!("It was yellow!");
    }
    Color::Custom(red, green, blue) => {
        println!("{} {} {}", red, green, blue);
    }
}
```

Pattern Matching

```
let color_str = match current_color {
    Color::Green => {
        "It was green!"
    }
    Color::Yellow => {
        "It was yellow!"
    }
};
```

```
match current_color {
      ^^^^^^^^^^^^^^^ patterns `Red` and `Custom { .. }` not covered
```

Pattern Matching

```
let color_str = match current_color {
    Color::Green => {
        "It was green!"
    }
    Color::Yellow => {
        "It was yellow!"
    }
    _ => {
        "It was something else!"
    }
};
```

Pattern Matching

```
let some_number = 5;

let number_str = match some_number {
    1 => {
        "It was one!"
    }
    2 => {
        "It was two!"
    }
    _ => {
        "It was something else!"
    }
};
```

Pattern Matching

```
let some_number = 5;

let number_str = match some_number {
    1 => {
        "It was one!"
    }
    2 => {
        "It was two!"
    }
    _ => {
        "It was something else!"
    }
};
```

Methods

```
enum Color { … }

impl Color {
    fn rgb(color: Color) -> (u8, u8, u8) { … }
    fn new(r: u8, g: u8, b: u8) -> Color { … }
}
}

let red = Color::new(250, 0, 0);
let purple = Color::new(100, 0, 250);
let (r, g, b) = Color::rgb(purple);
```

Methods

```
enum Color { … }

impl Color {
    fn rgb(self) -> (u8, u8, u8) { … }
    fn new(r: u8, g: u8, b: u8) -> Self { … }
}

let purple: Color = Color::new(100, 0, 250);
let (r, g, b) = Color::rgb(purple);
let (r, g, b) = purple.rgb();
```

same!

Type Parameters

```
let last_char = my_string.pop();
let last_char: char = my_string.pop();
```

Type Parameters

```
let last_char = my_string.pop();
let last_char: Option<char> = my_string.pop();

enum Option<T> {
    None,
    Some(T),
}

let email: Option<String> = Some(email_str);
let email: Option<String> = None;
```

(`None` is `Option::None` - the prefix is optional for `Option`)

Type Parameters

```
enum Result<O, E> {
    Ok(O),
    Err(E),
}

let success: Result<i64, String> = Ok(42);
let failure: Result<i64, String> = Err(str);
```

(the `Result::` prefix is also optional for `Result`)

Review of Part 3

enums

```
let purple = Color::Custom(1, 2, 3);
```

matching

```
match color {
    Color::Custom(r, g, b) => {
        r + g + b
    }
}
```

methods

```
color.rgb() == Color::rgb(color)
```

type params

```
let first: Option<char> = str.pop();
```

Exercises for Part 3

Follow the instructions in **`part3/README.md`**

4. Vectors

`Vec`

`usize`

stack memory

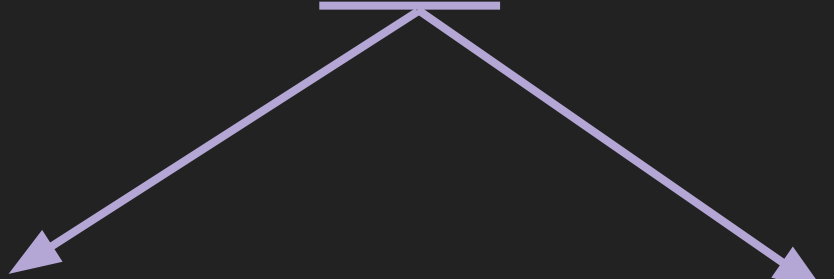
heap memory

Vectors

```
let mut years: Vec<i32> = vec![1995, 2000, 2005];

years.push(2010); // Now `years` has 4 elements,
                  // ending in 2010

years.push(2015); // Now `years` has 5 elements,
                  // ending in 2015

println!("Number of years: {}", years.len());
```

usize

```
let length: usize = years.len();
```



u32

u64

32-bit systems
(e.g. Web Assembly)

64-bit systems
(almost everything else)

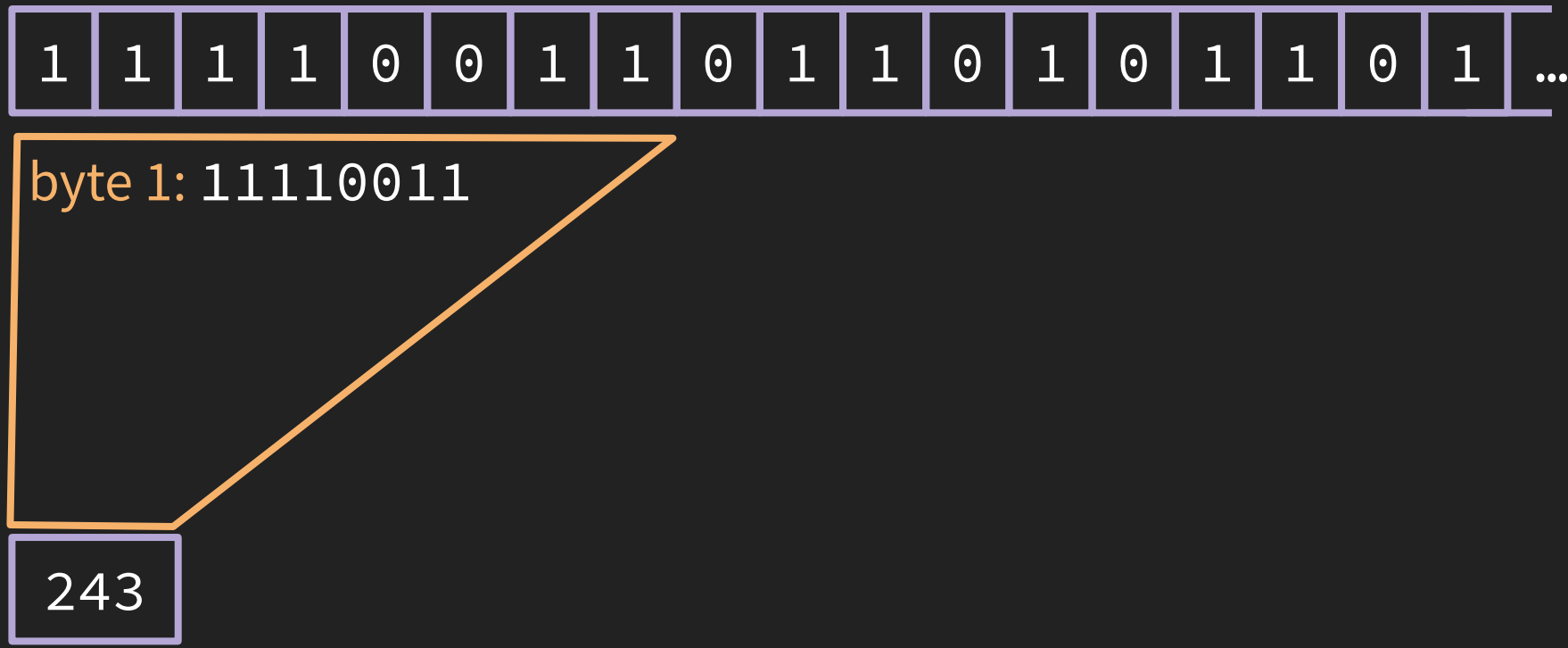
Vectors vs Arrays

```
let mut nums: [u8; 3] = [1, 2, 3];
let mut nums: Vec<u8> = vec![1, 2, 3];

for num in nums { … }
```

the tradeoffs here set the stage for
the biggest factor in language performance!

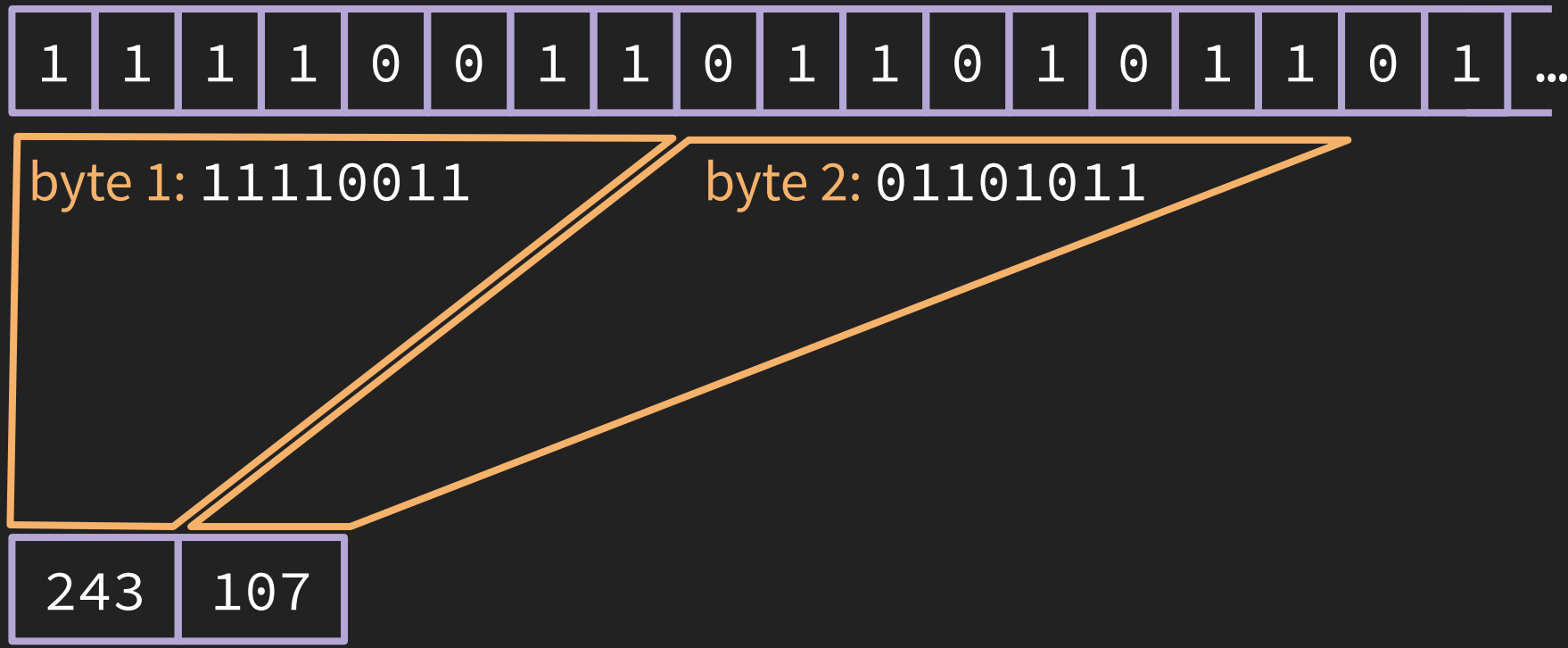
Memory

bits



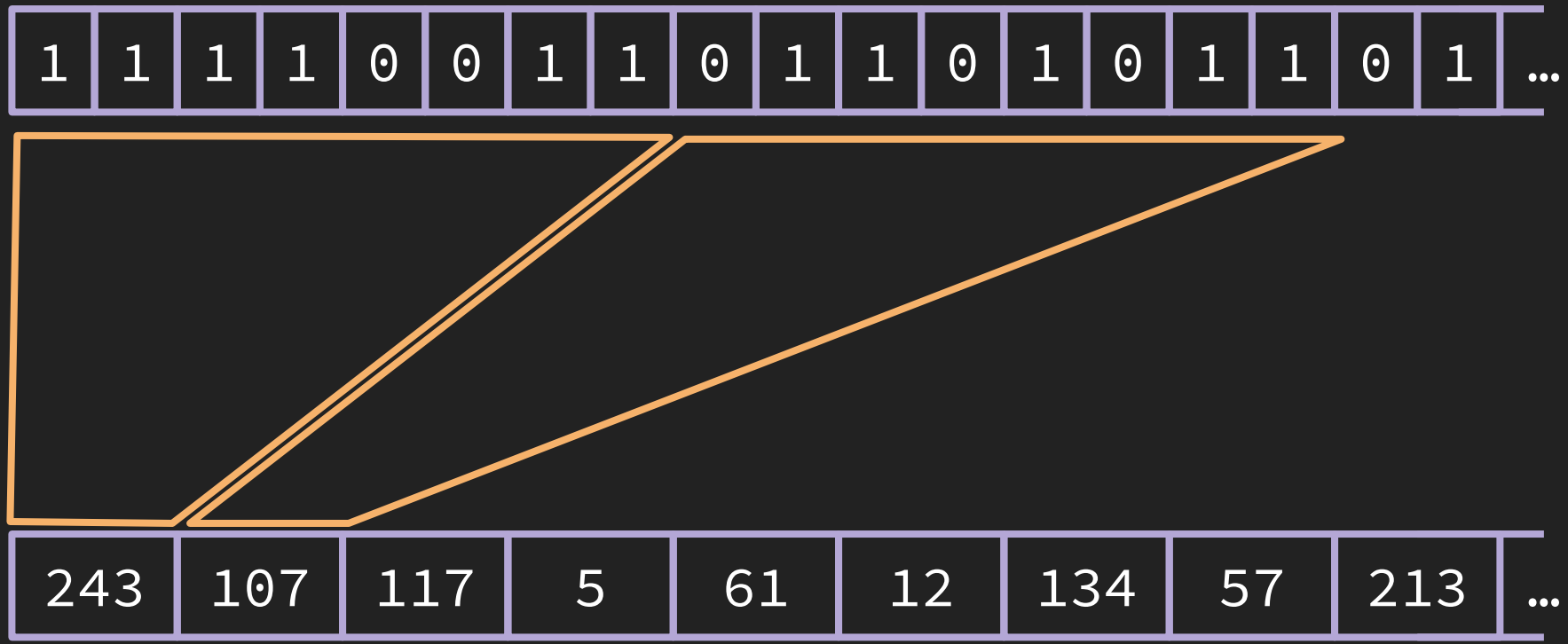
Memory

bits


1 1 1 1 0 0 1 1 0 1 1 0 1 0 1 1 0 1 …
byte 1: 11110011
byte 2: 01101011
243
107

Memory

bits



bytes

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

```
stack_length: 0
```

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

`stack_bytes[stack_length - 1]`

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

`stack_length: 1`

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

`stack_bytes[stack_length - 2]`

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

`stack_bytes[stack_length - 1]`

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

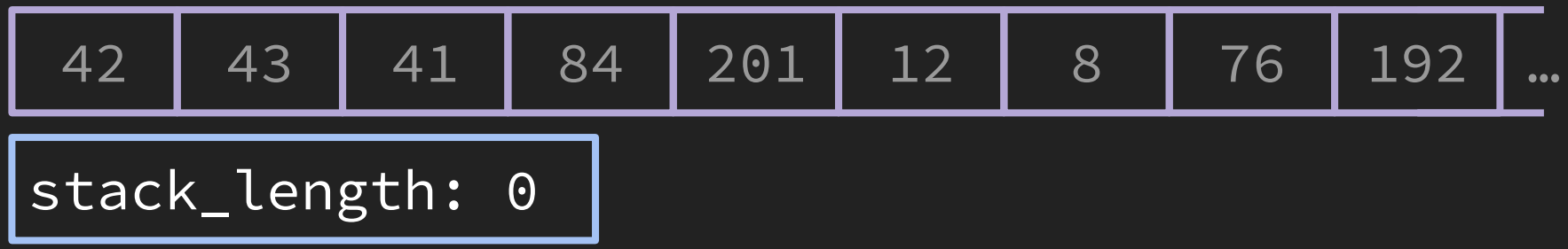
```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```

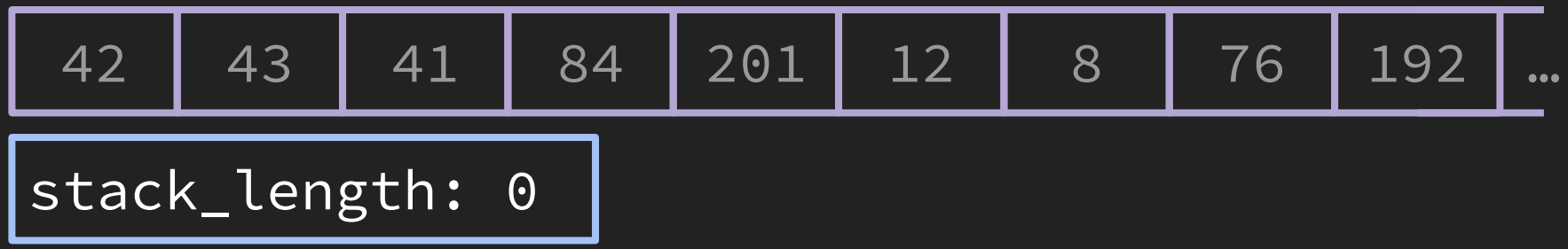
The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(42);
```



The Stack

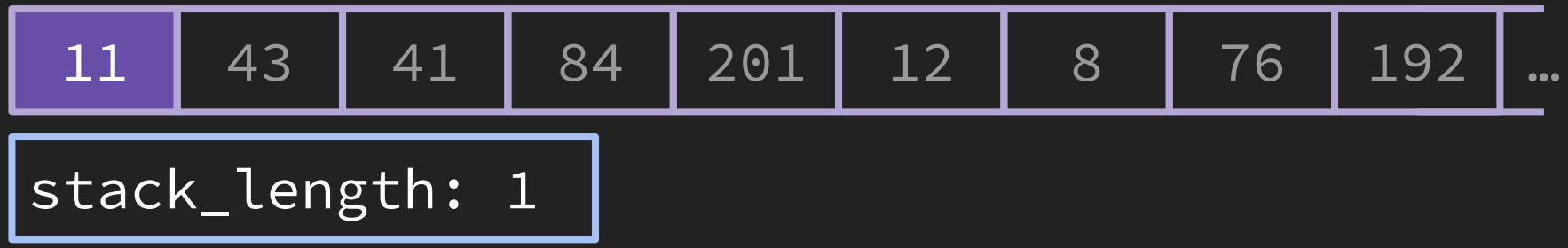
```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(11);
```



The Stack

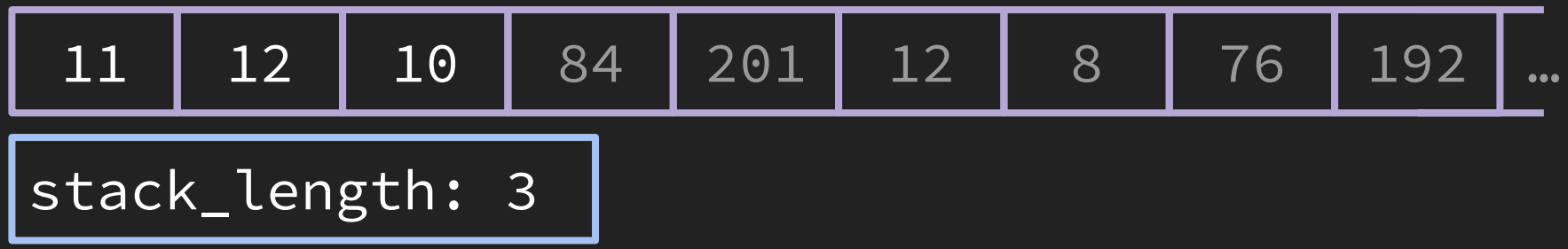
`stack_bytes[stack_length - 1]`

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(11);
```



The Stack

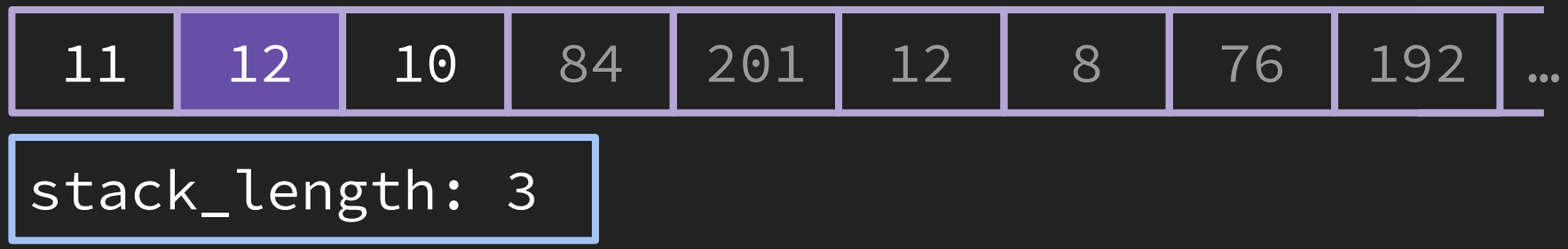
```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(11);
```



The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(11);
```

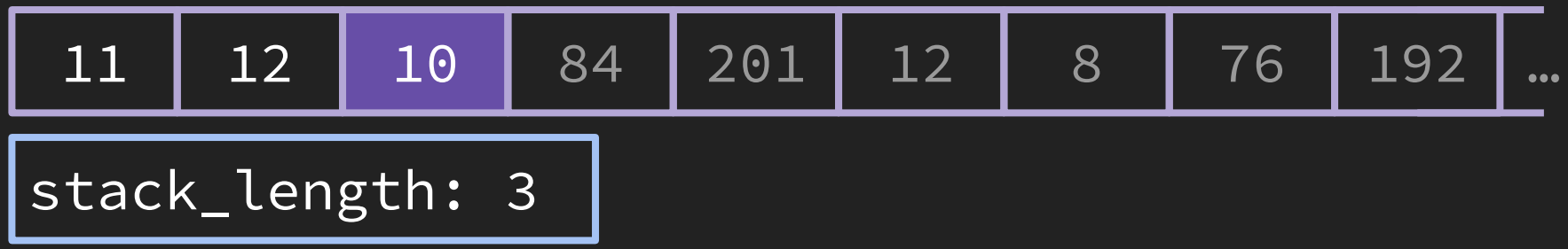
`stack_bytes[stack_length - 2]`



The Stack

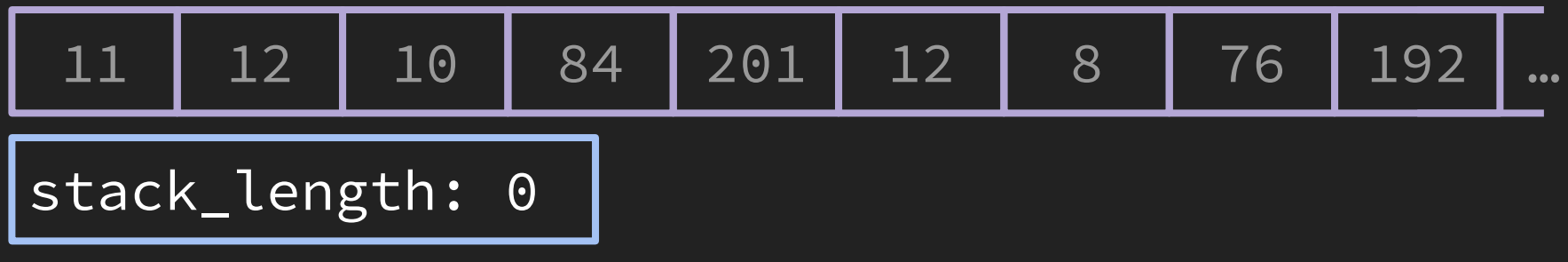
```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(11);
```

`stack_bytes[stack_length - 1]`



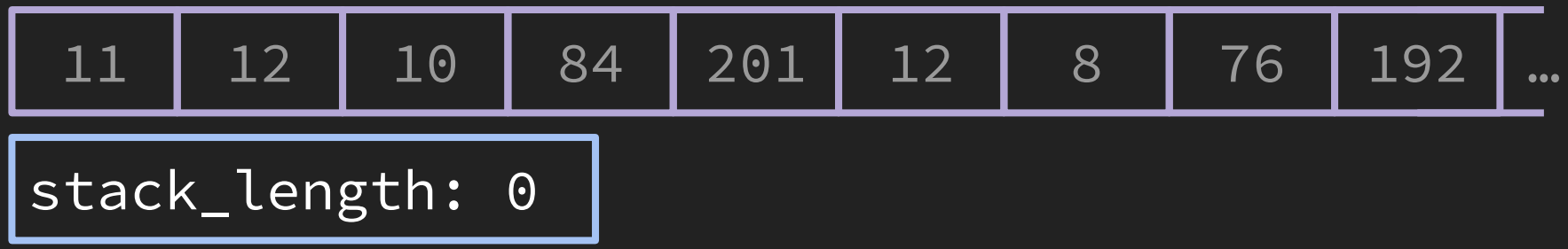
The Stack

```
fn increment_decrement(num: u8) {
    print_nums(num + 1, num - 1);
}

fn print_nums(x: u8, y: u8) { … }

increment_decrement(11);
```



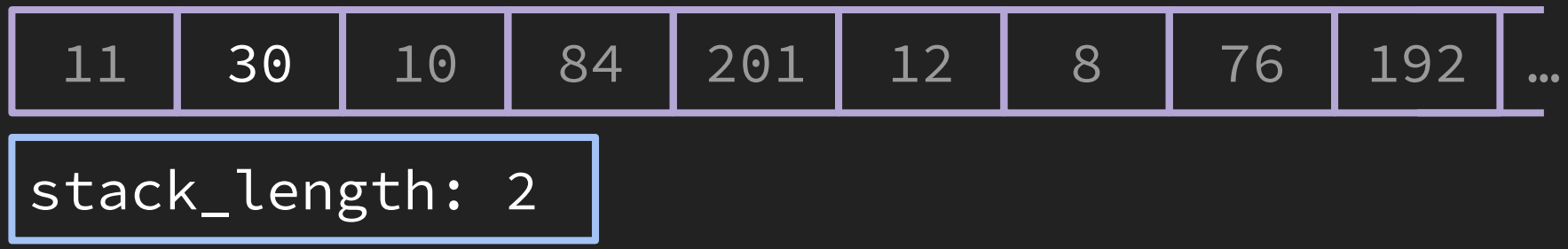
The Stack

```
fn double_and_return(num: u8) -> u8 {
    return num * 2;
}

let x = double_and_return(30);
```



The Stack

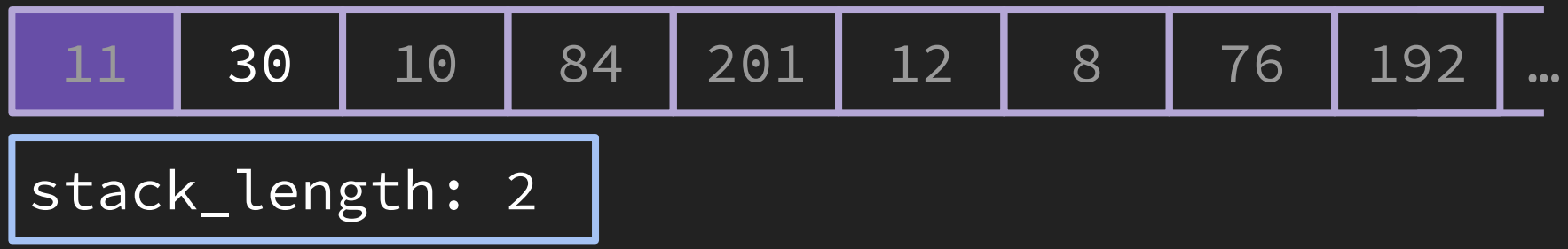
```
fn double_and_return(num: u8) -> u8 {
    return num * 2;
}

let x = double_and_return(30);
```



The Stack

```
fn double_and_return(num: u8) -> u8 {
    return num * 2;
}

let x = double_and_return(30);
```

reserved for return value

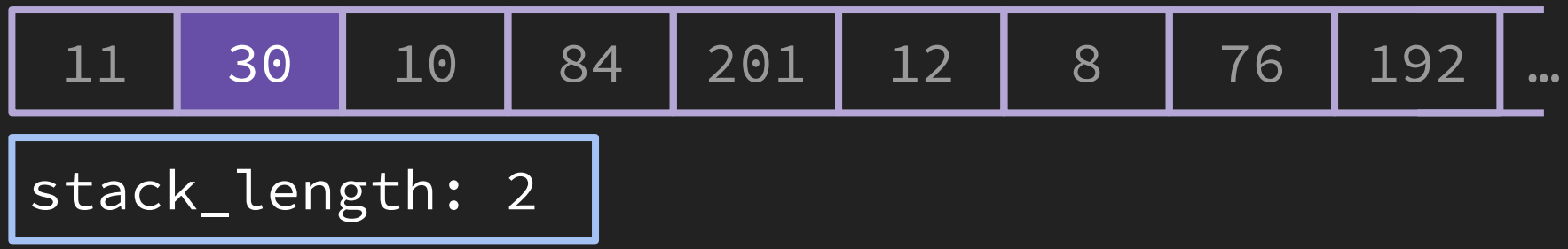


The Stack

`stack_bytes[stack_length - 1]`

```
fn double_and_return(num: u8) -> u8 {
    return num * 2;
}

let x = double_and_return(30);
```

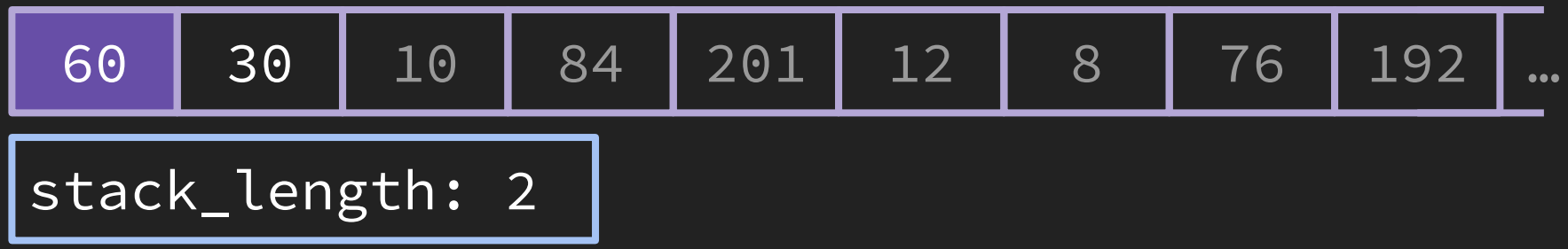
reserved for return value



The Stack

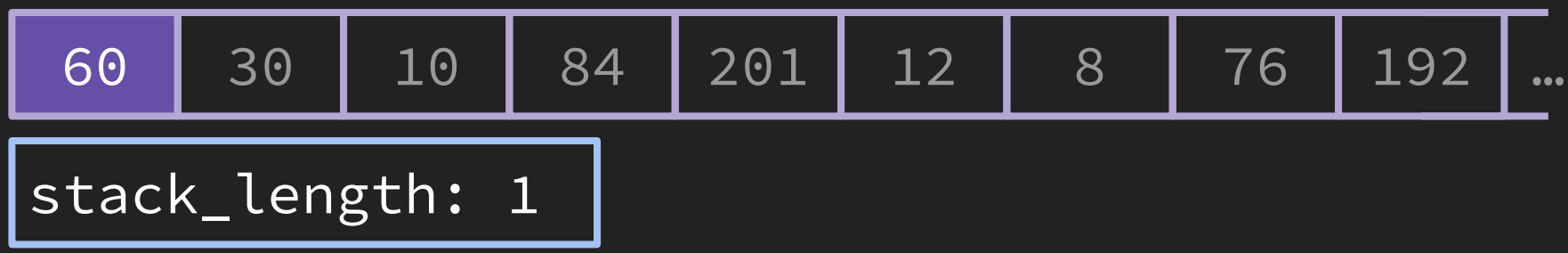
```
fn double_and_return(num: u8) -> u8 {
    return num * 2;
}

let x = double_and_return(30);
```

reserved for return value



The Stack

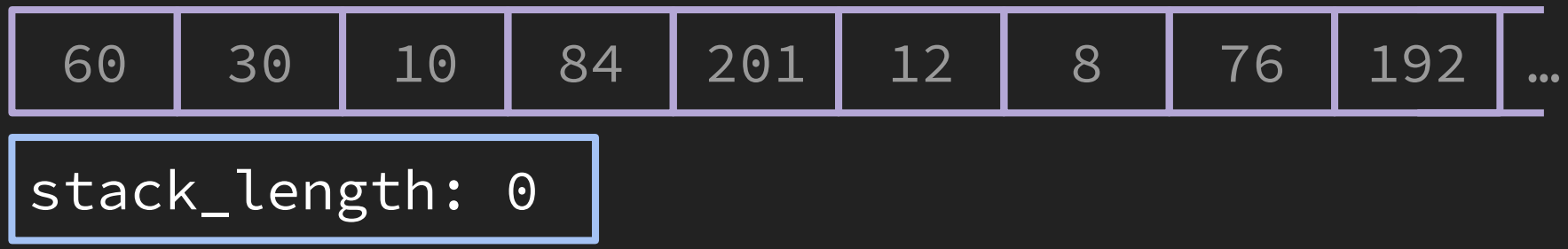
```
fn double_and_return(num: u8) -> u8 {
    return num * 2;
}

let x = double_and_return(30);
```



The Stack

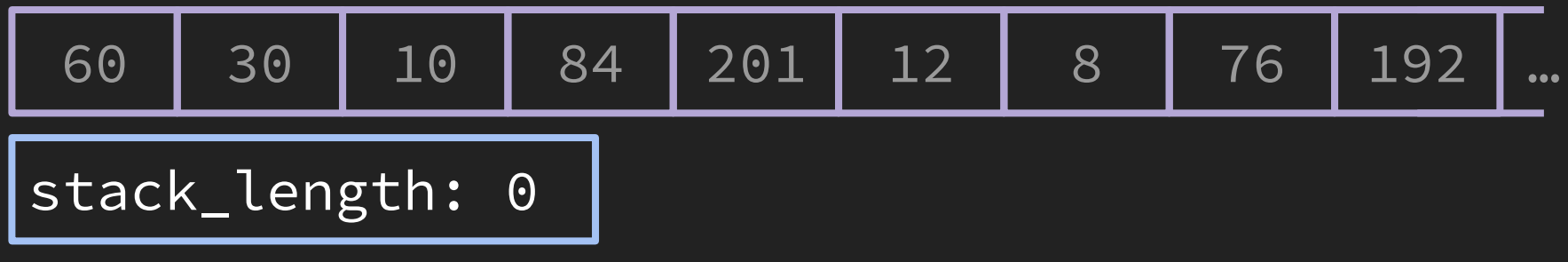
```
fn double_and_return(num: u8) -> u8 {
    return num * 2;
}

let x = double_and_return(30);
```

reserved for return value



The Stack

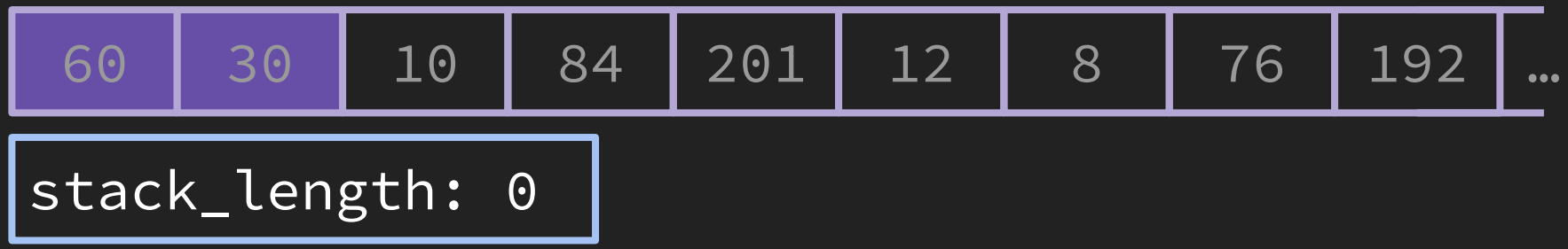
```
fn double_twice(num: u8) -> (u8, u8) {
    return (num * 2, num * 2);
}

let (x, y) = double_twice(30);
```



The Stack

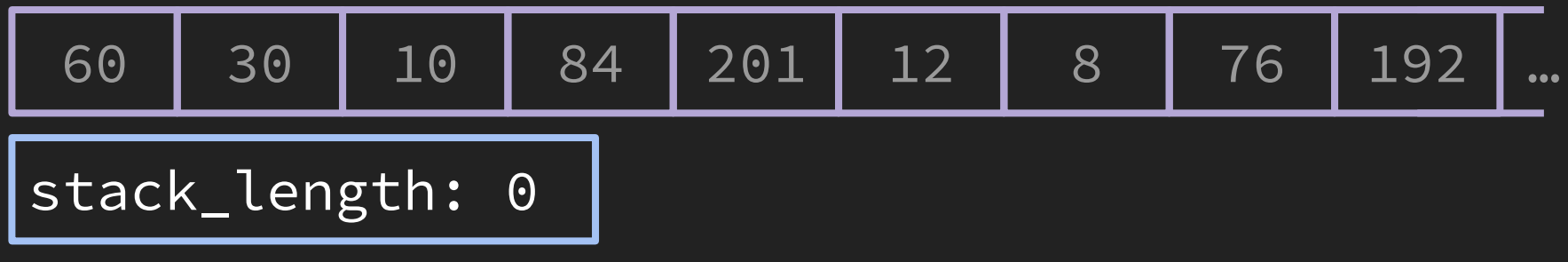
```
fn double_twice(num: u8) -> (u8, u8) {
    return (num * 2, num * 2);
}

let (x, y) = double_twice(30);
```

reserved for return value
reserved for return value



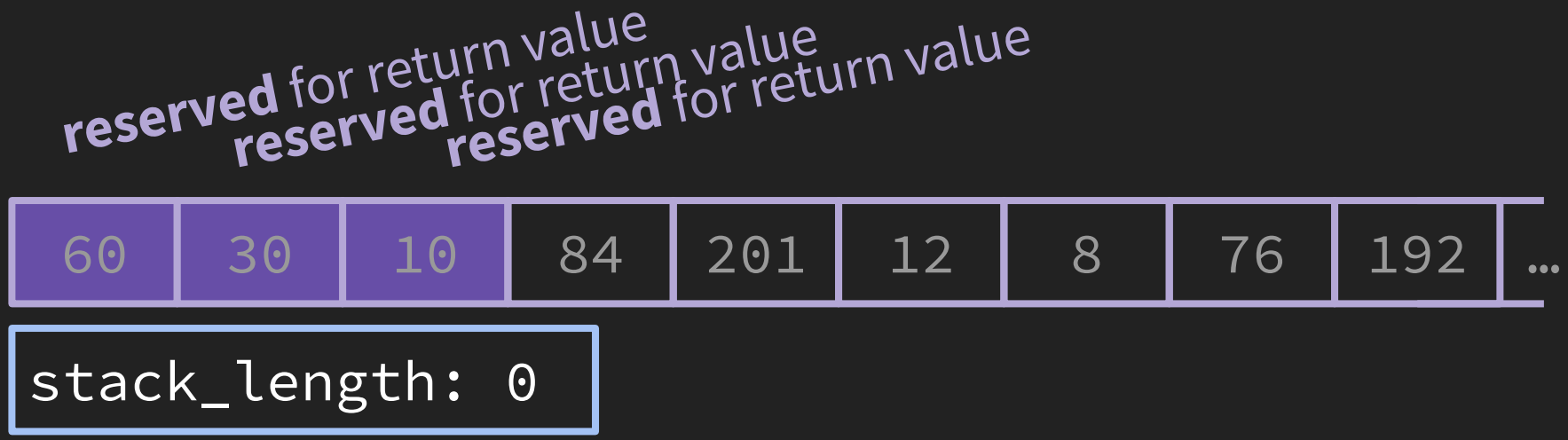
The Stack

```
fn double_thrice(num: u8) -> (u8, u8, u8) {
    return (num * 2, num * 2);
}

let (x, y, z) = double_thrice(30);
```



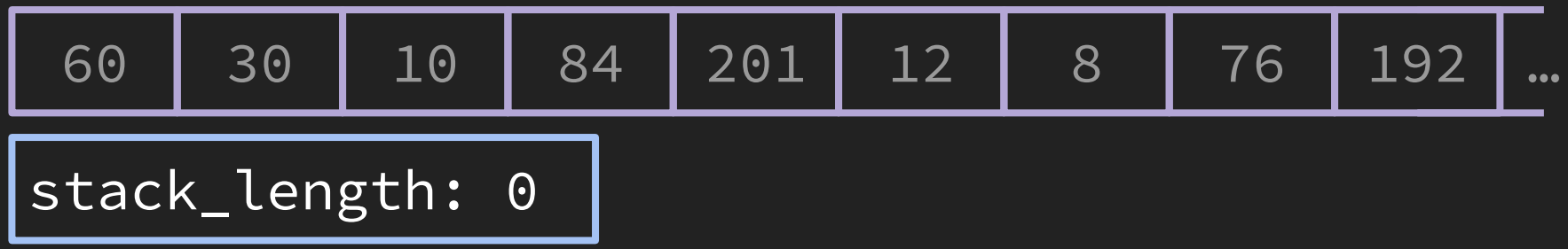
The Stack

```
fn double_thrice(num: u8) -> (u8, u8, u8) {
    return (num * 2, num * 2);
}

let (x, y, z) = double_thrice(30);
```



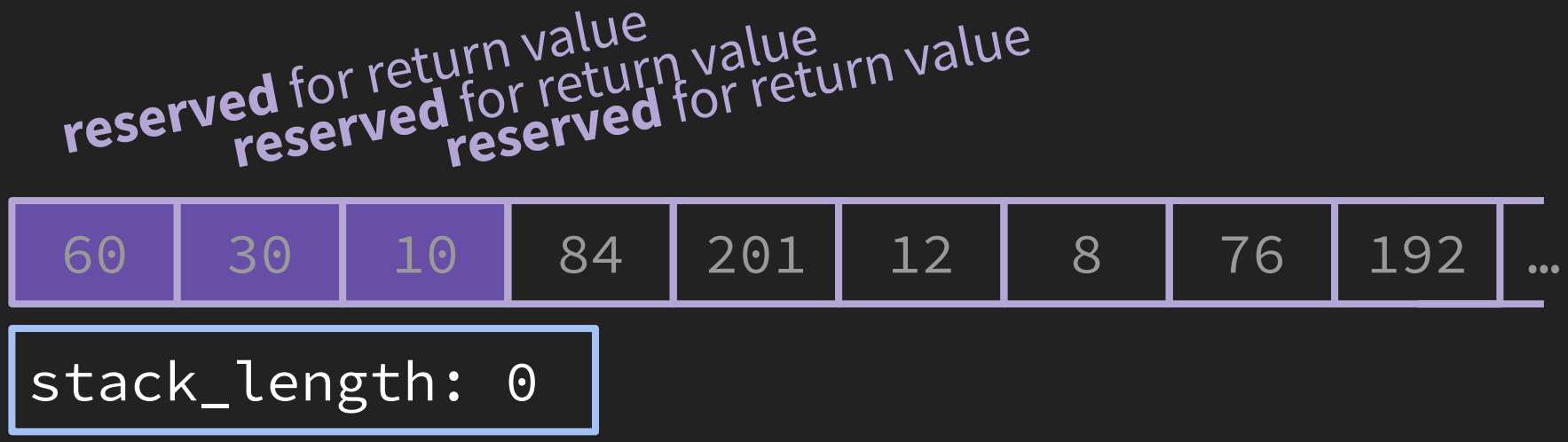
The Stack

```
fn double_thrice(num: u8) -> [u8; 3] {
    …
}

let [x, y, z] = double_thrice(30);
```



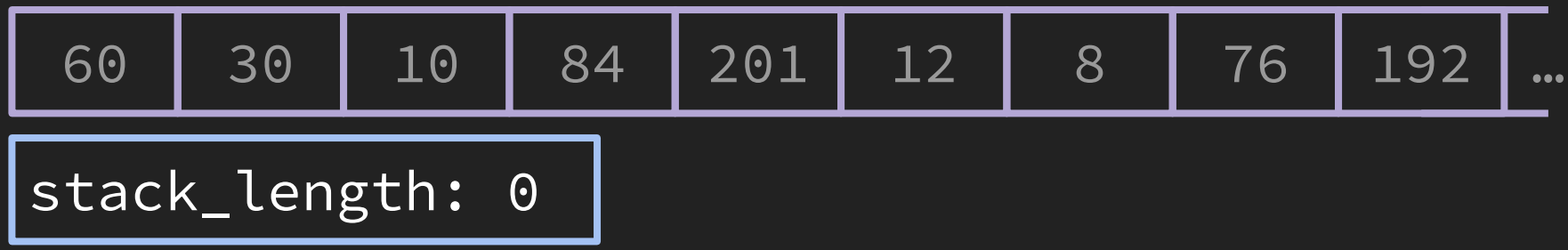
The Stack

```
fn double_thrice(num: u8) -> [u8; 3] {
    …
}

let [x, y, z] = double_thrice(30);
```



The Stack

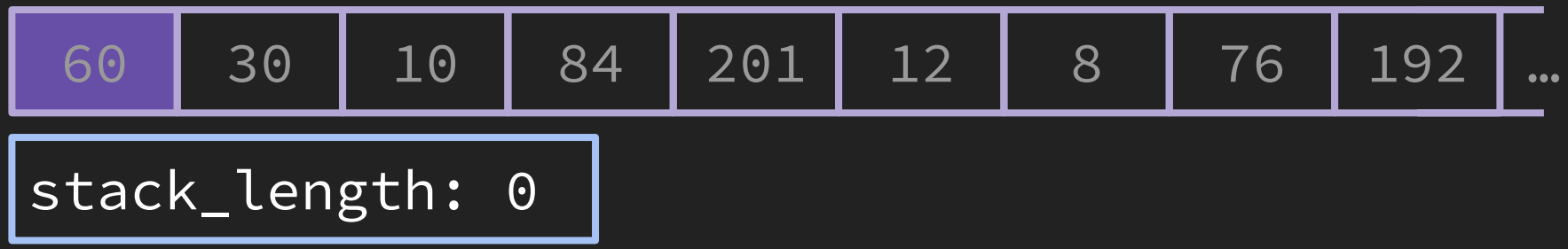
```
fn double_many(num: u8) -> Vec<u8> {
    …
}

let nums = double_many(30);
```



The Stack

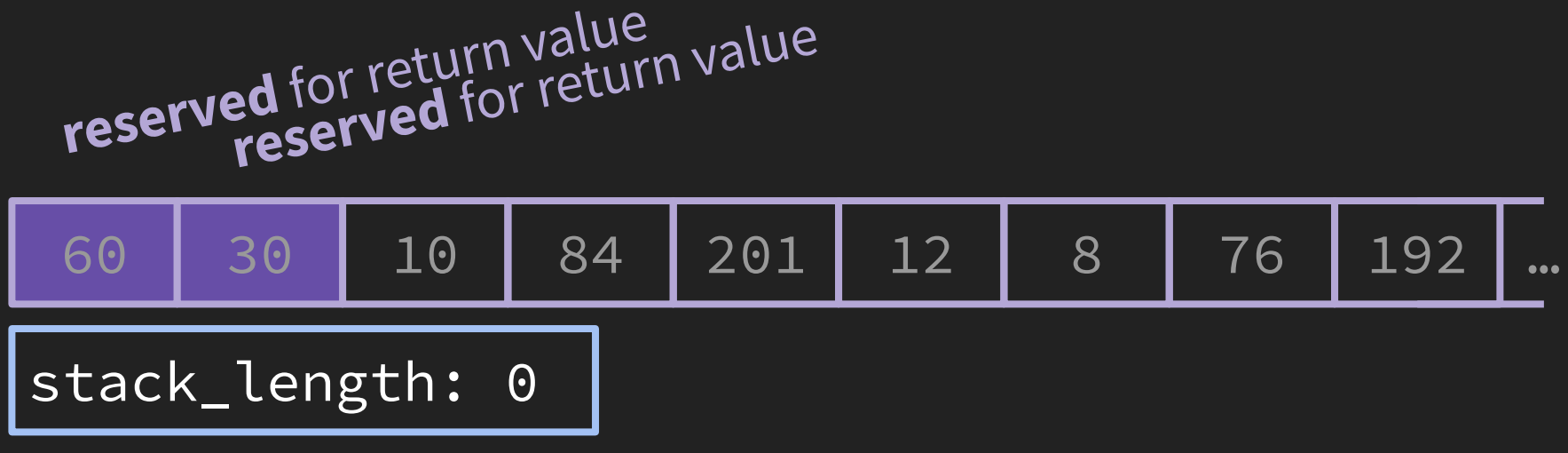
```
fn double_many(num: u8) -> Vec<u8> {
    …
}

let nums = double_many(30);
```

reserved for return value



The Stack

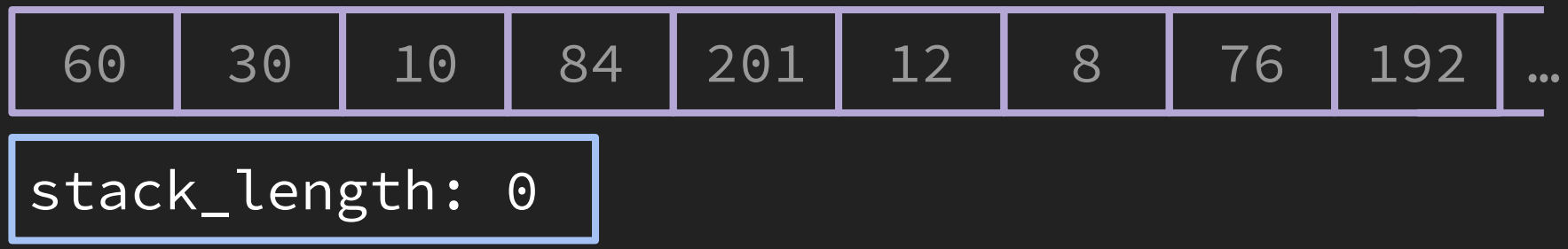
```
fn double_many(num: u8) -> Vec<u8> {
    …
}

let nums = double_many(30);
```



The Stack

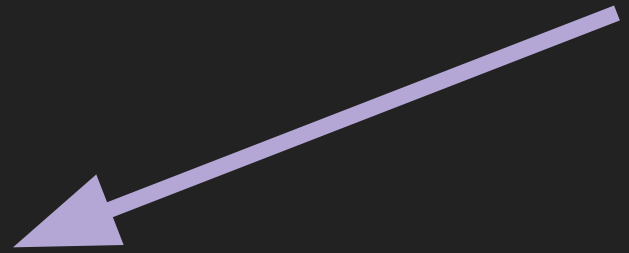
```
fn double_many(num: u8) -> Vec<u8> {
    …
}

let nums = double_many(30);
```

to be returnable, **size must be known** at compile time



The Stack

```
fn double_many(num: u8) -> Vec<u8> {
    …
}
```



```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

to be returnable, **size must be known** at compile time

The Heap

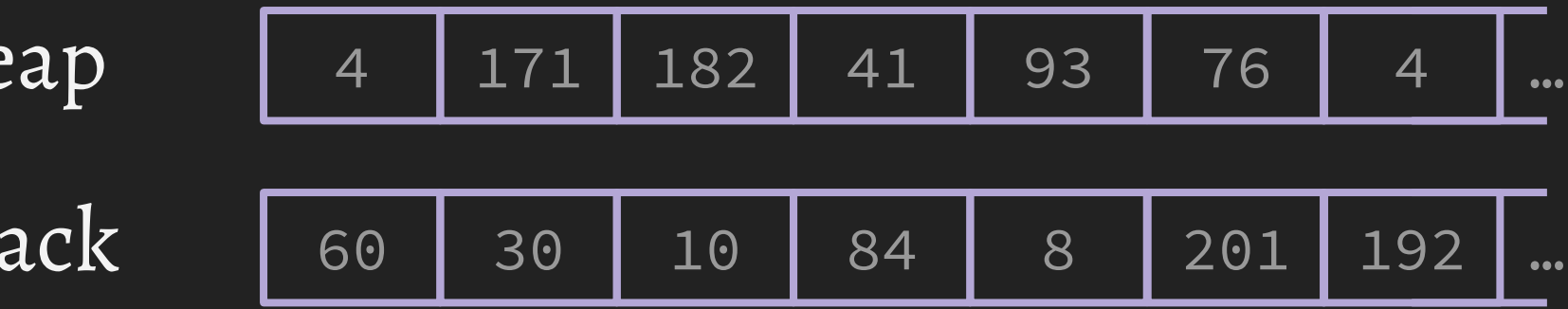
```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

stack

The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

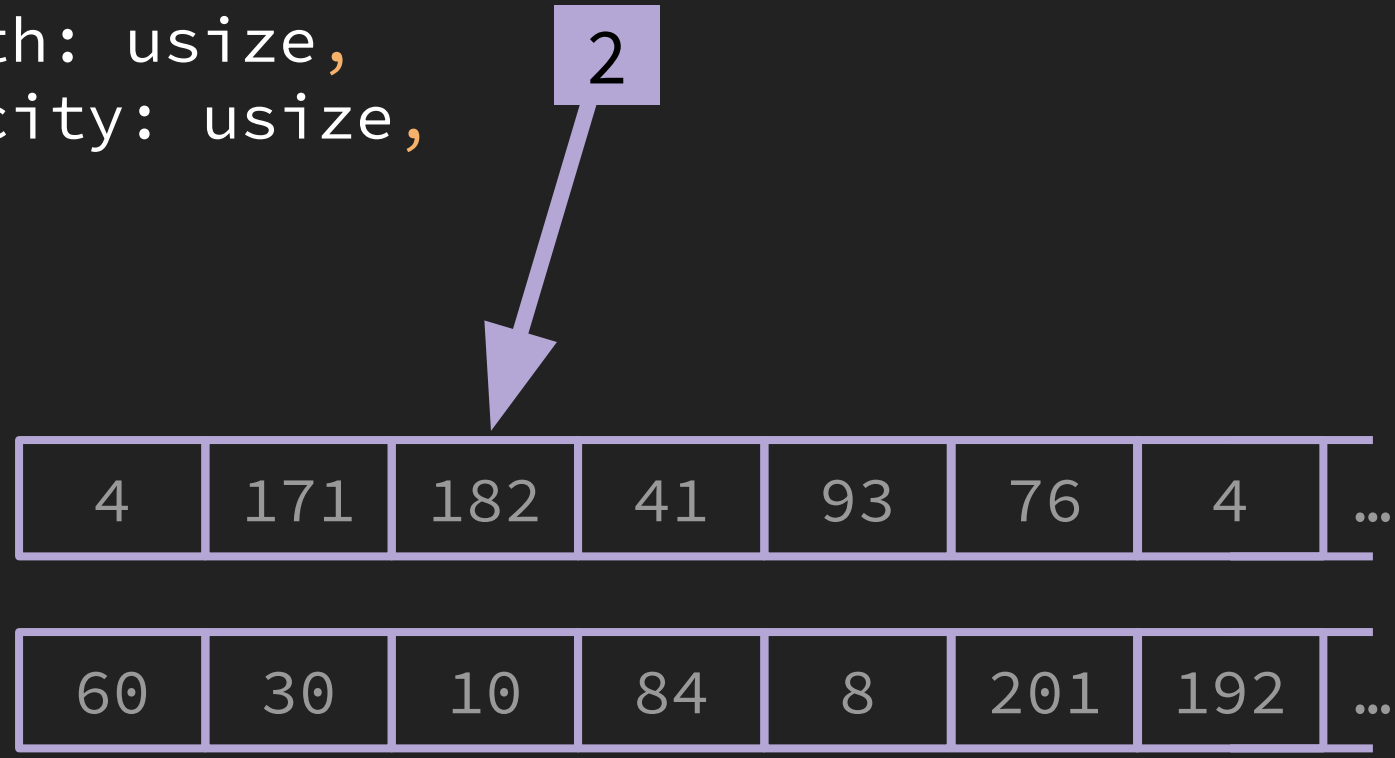
```
let nums = vec![1, 2, 3];
```



The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

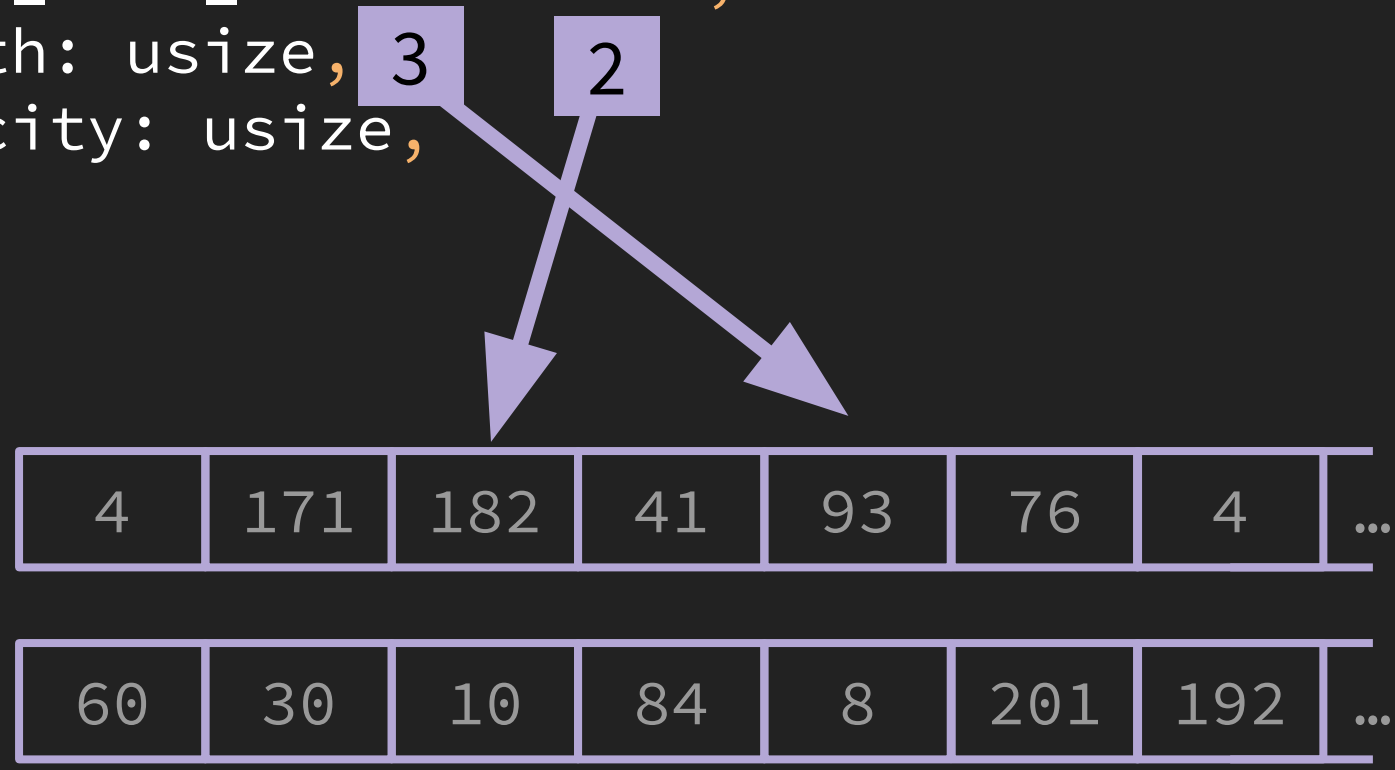
```
let nums = vec![1, 2, 3];
```



The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

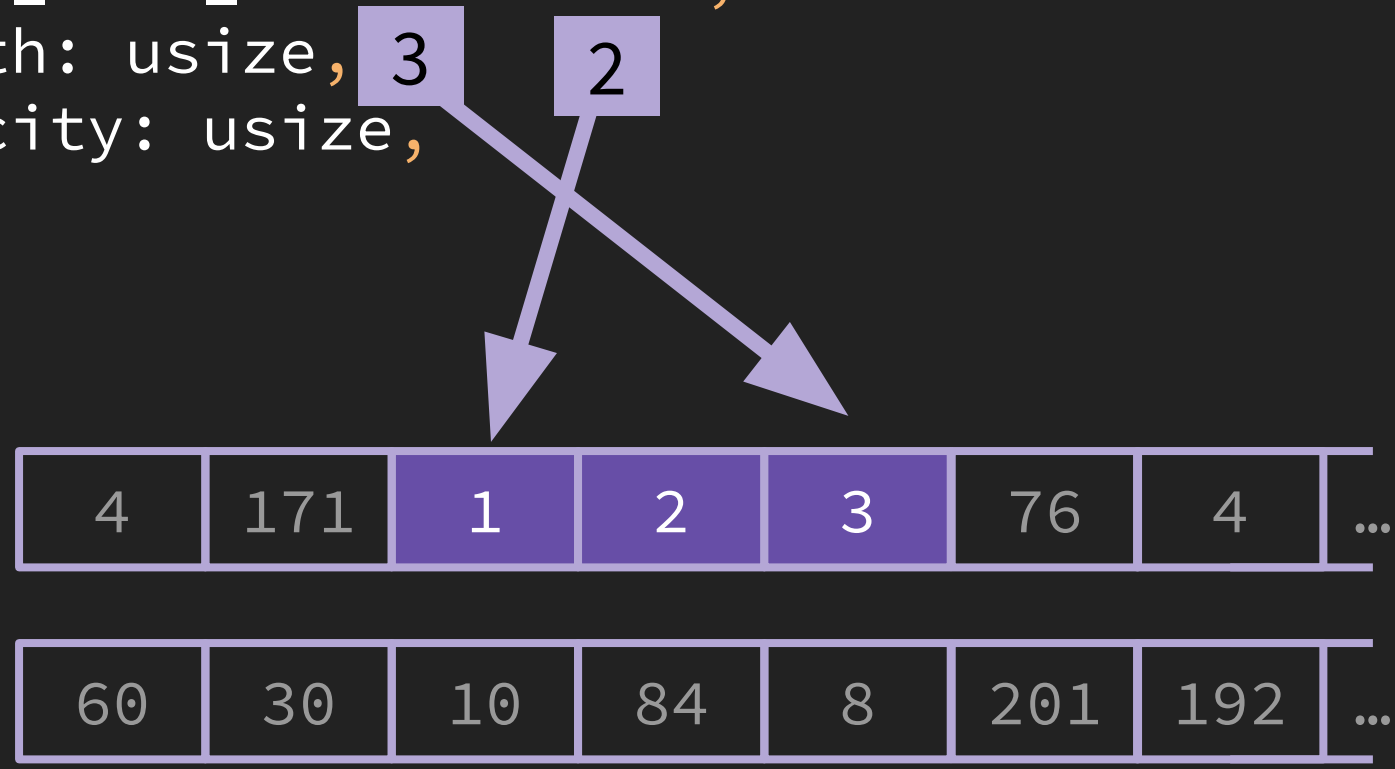
```
let nums = vec![1, 2, 3];
```



The Heap

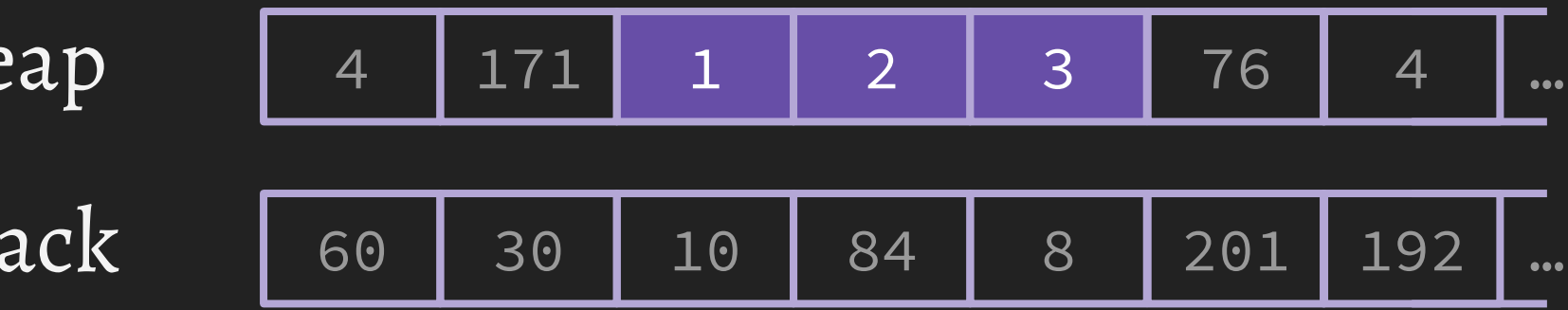
```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

```
let nums = vec![1, 2, 3];
```



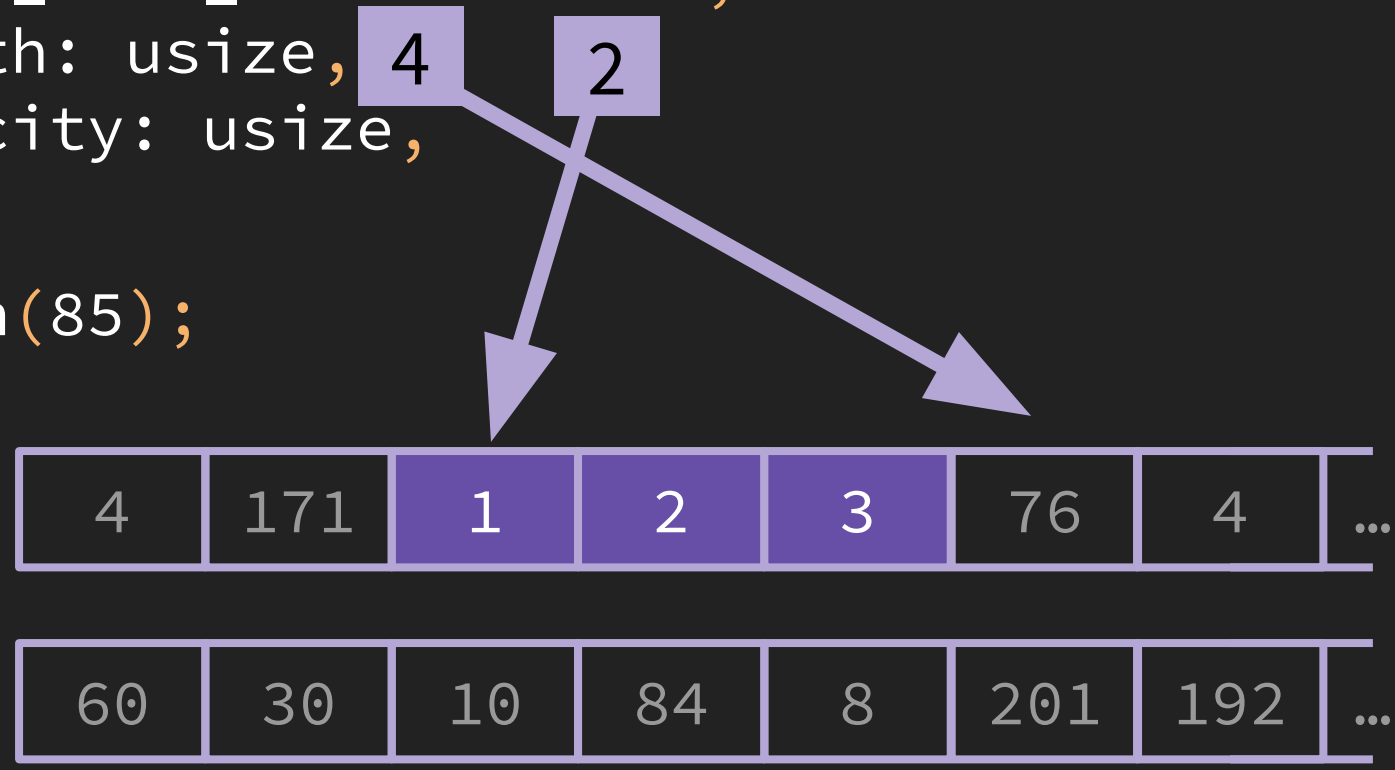
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}

nums.push(85);
```



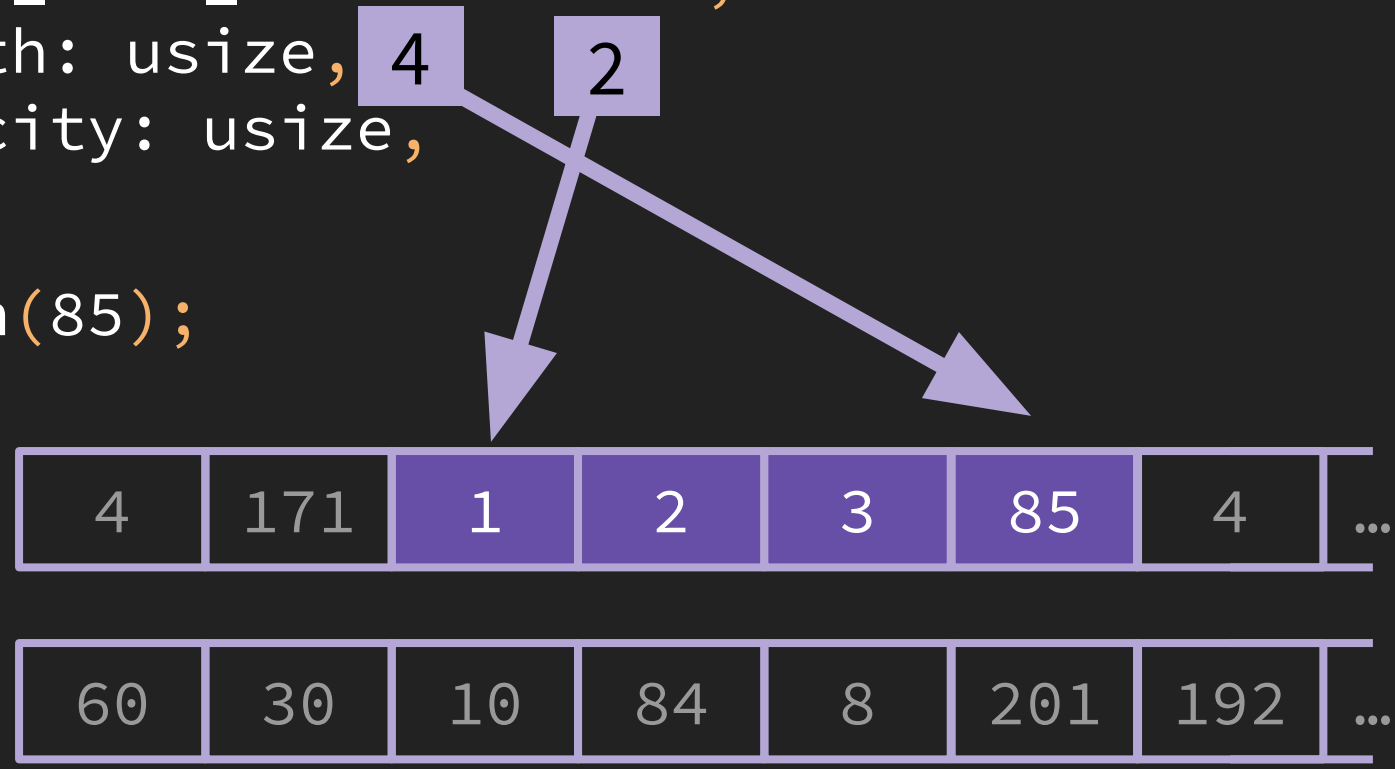
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}

nums.push(85);
```



The Heap

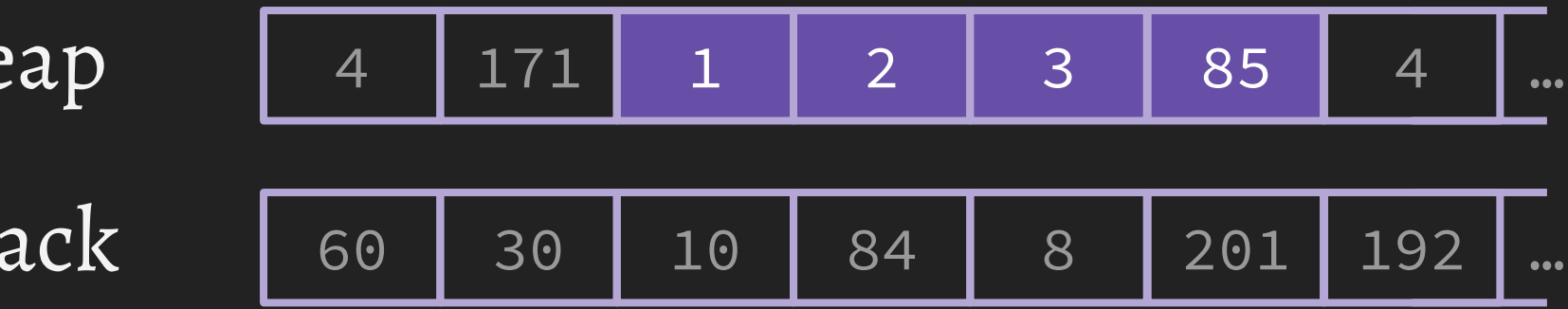
```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}

nums.push(85);
```



The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

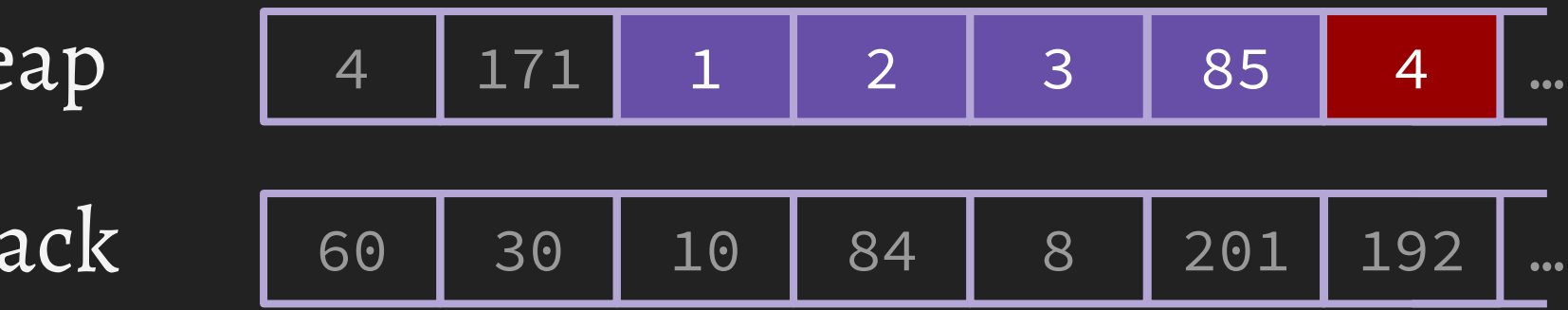
```
nums.push(85);
nums.push(29);
```



The Heap

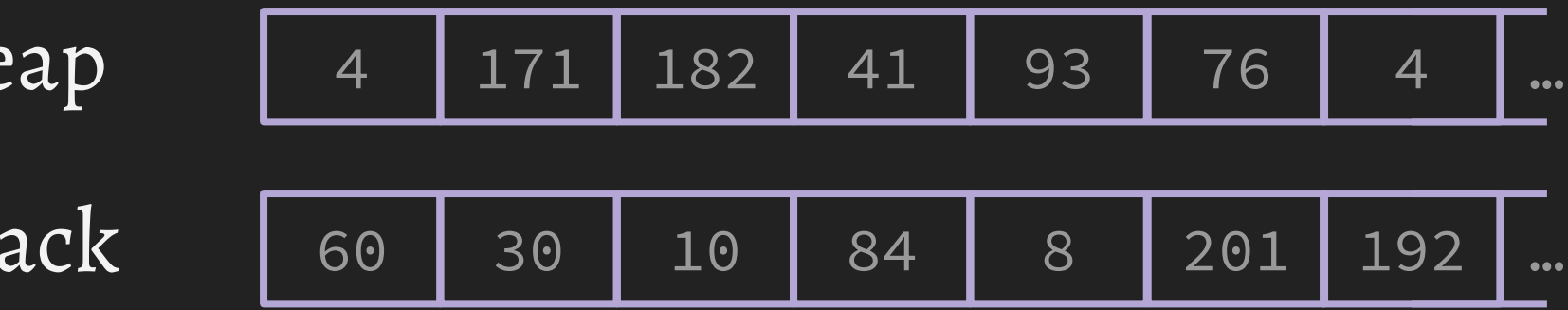
```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

```
nums.push(85);
nums.push(29);
```



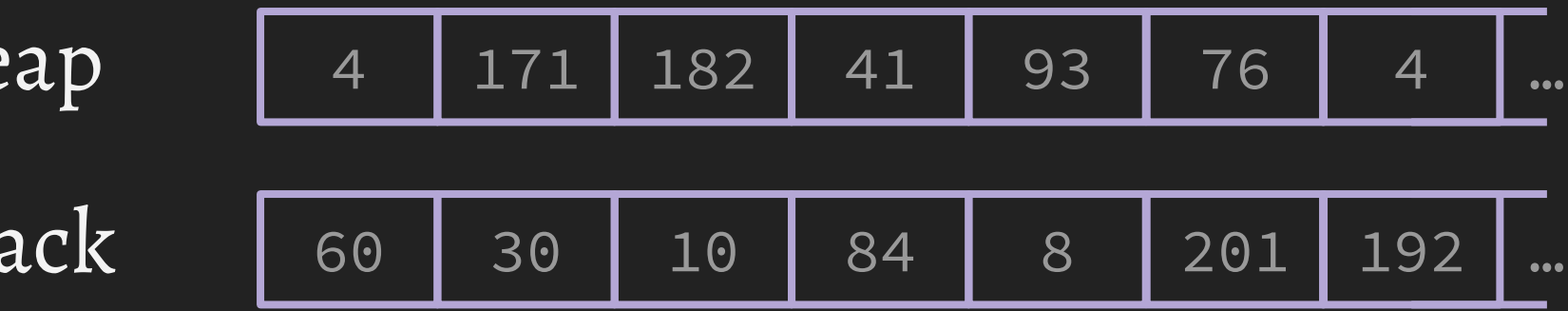
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}

let nums = vec![1, 2, 3];
```



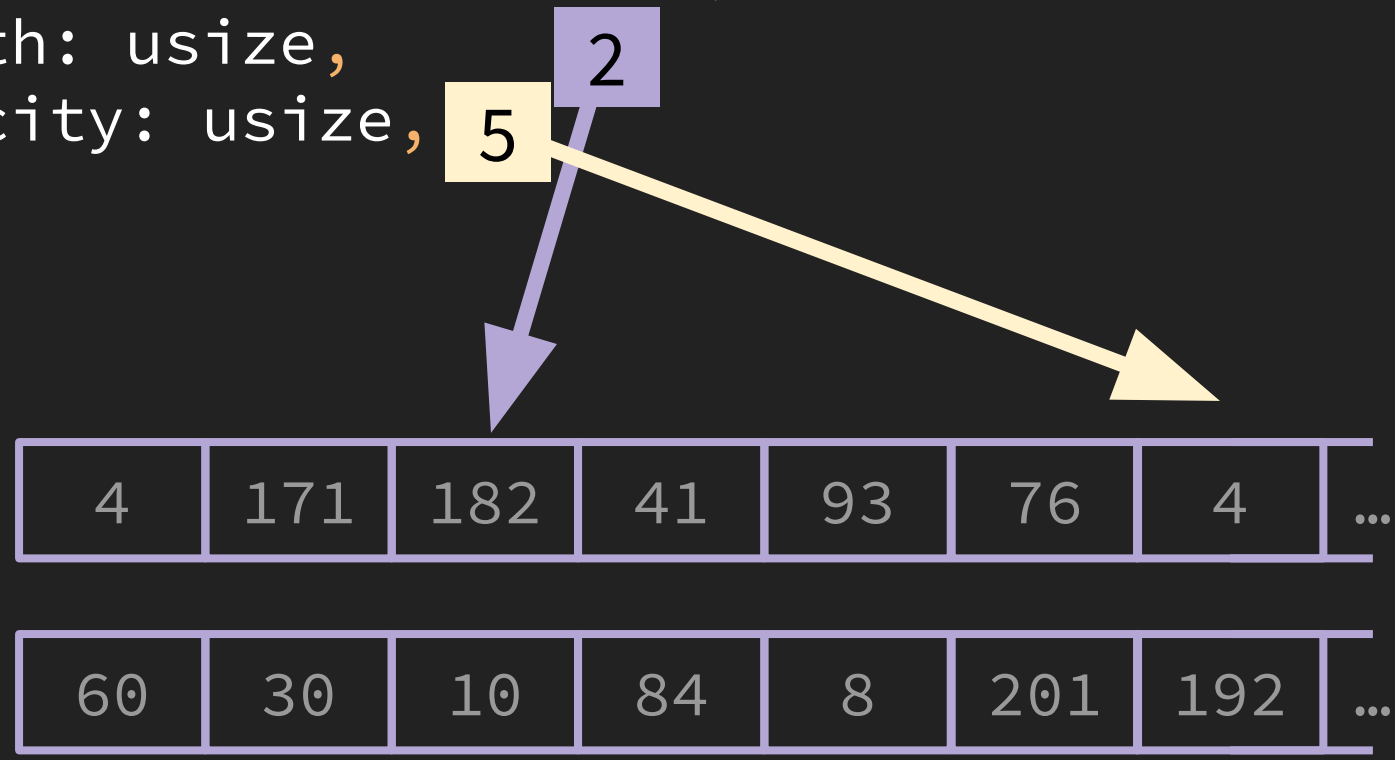
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}

let nums = Vec::with_capacity(5);
```



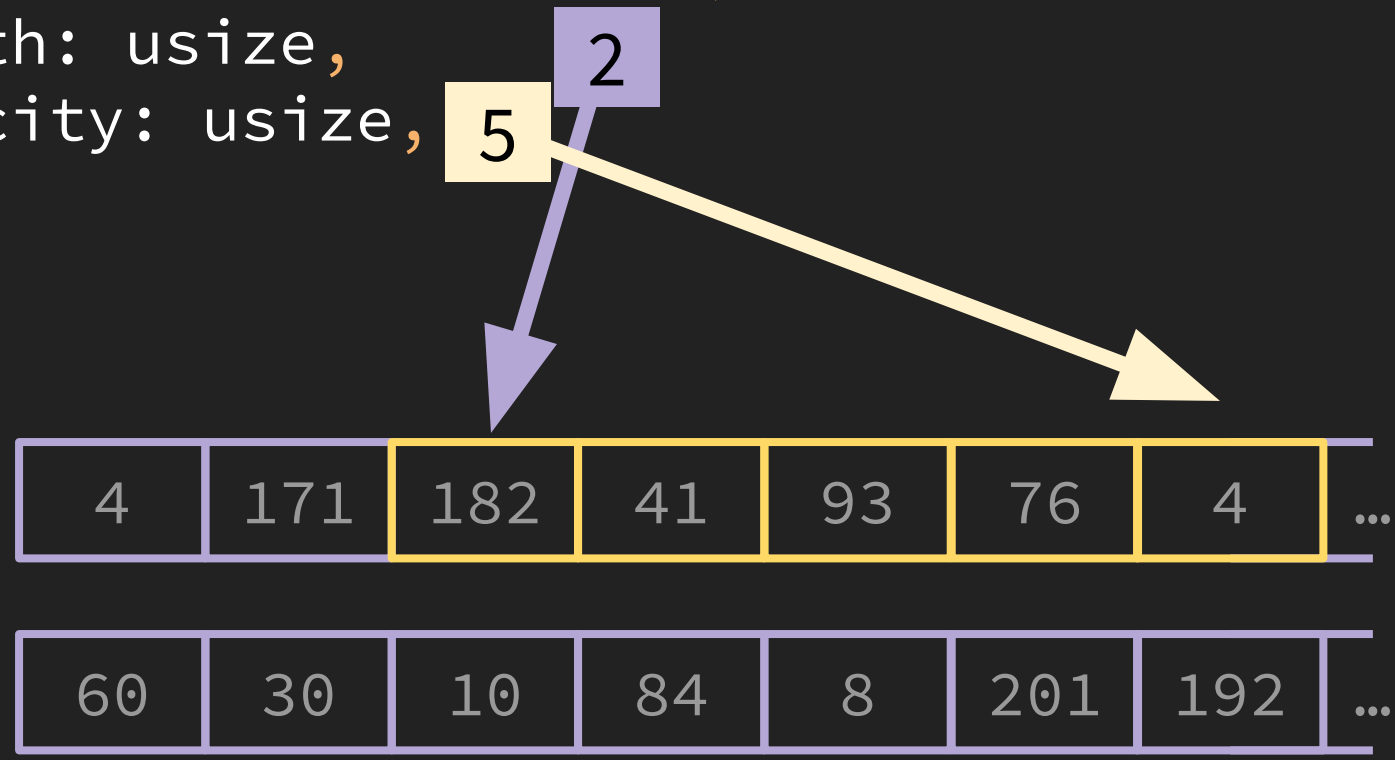
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```



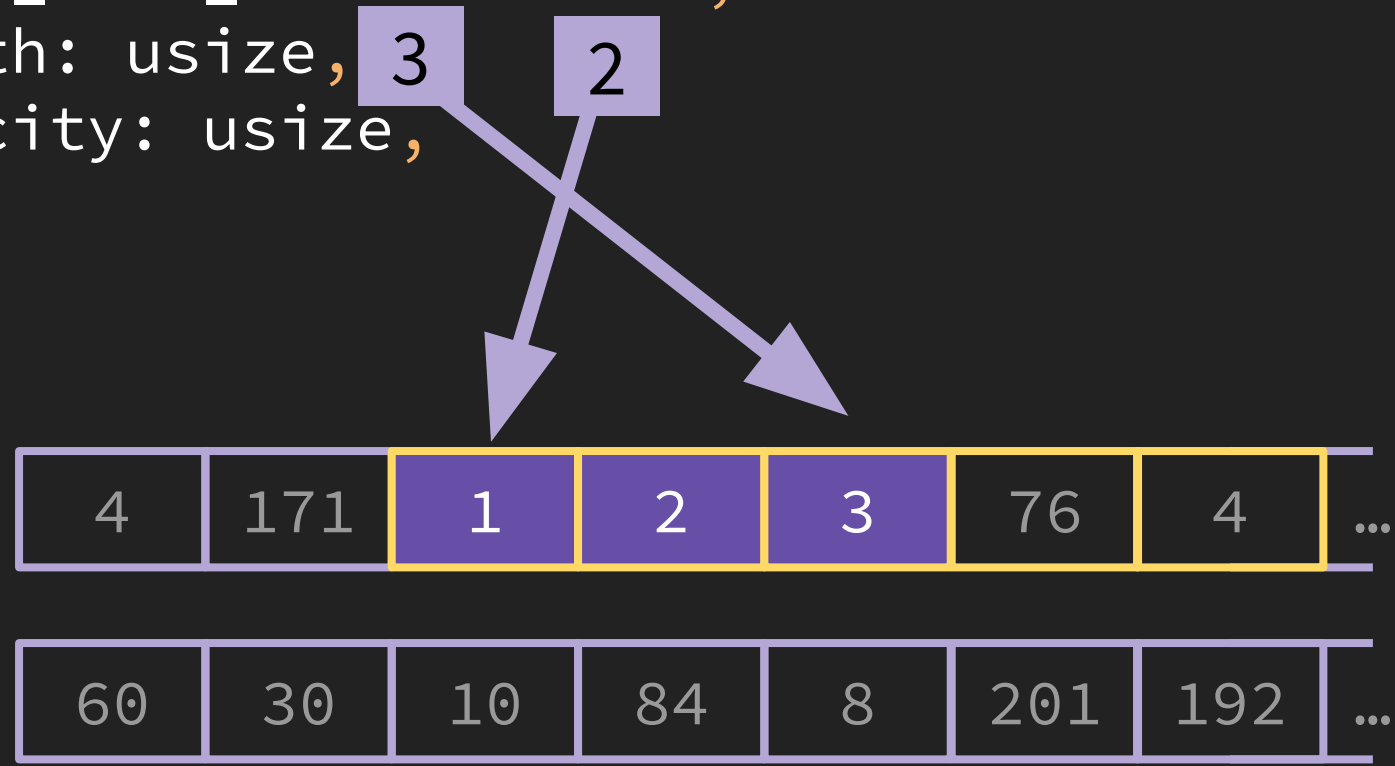
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```



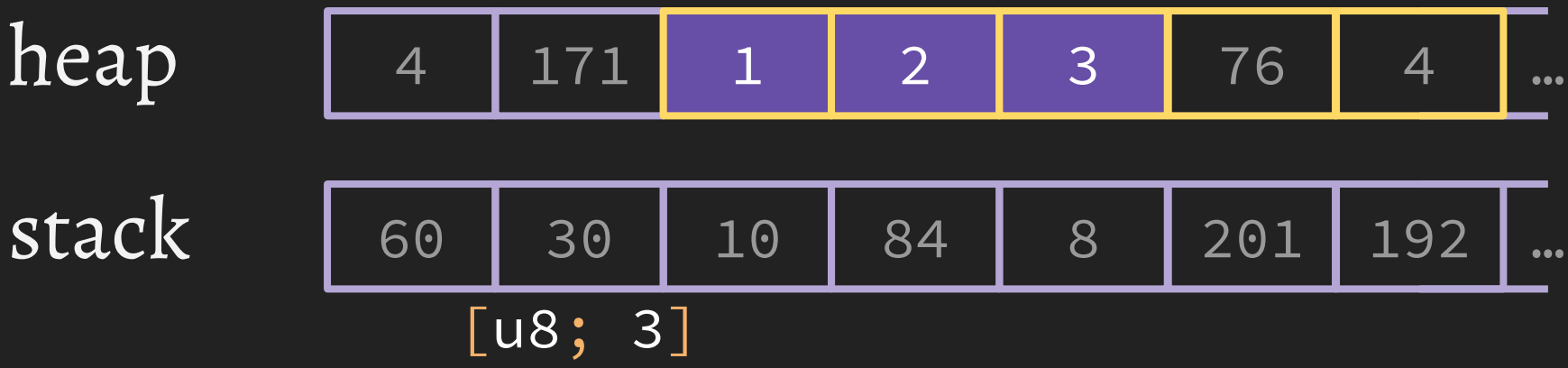
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```



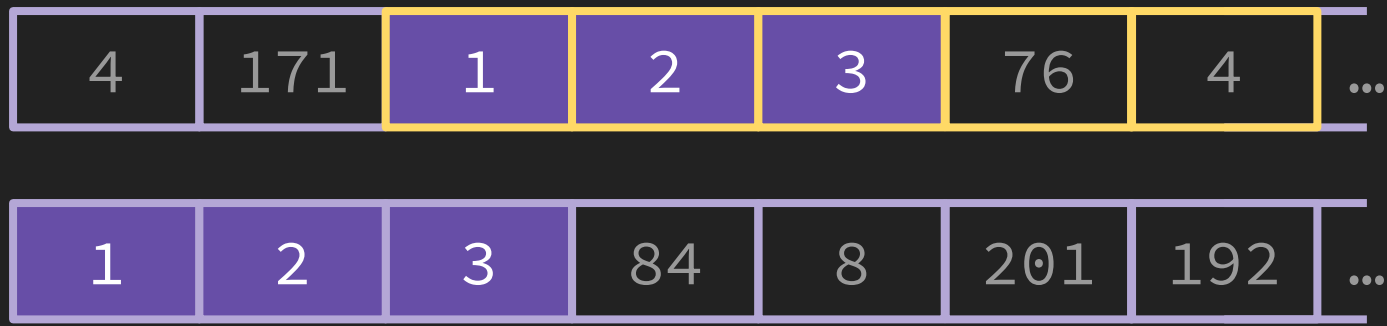
The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```



The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```



Vectors vs Arrays

stack-allocated

```
let mut array: [u8; 3] = [1, 2, 3];
let mut vector: Vec<u8> = vec![1, 2, 3];
```

heap-allocated

the tradeoffs here set the stage for
the biggest factor in language performance!

Review of Part 4

Vec	`let nums: Vec<u8> = vec![1, 2];`
usize	`u64` on 64-bit systems, `u32` on 32-bit
stack memory	`stack_bytes[stack_length - 1]`
heap memory	`heap_bytes[index_of_first_elem]`

Exercises for Part 4

Follow the instructions in **`part4/README.md`**

5. Ownership

Manual memory management

Automatic memory management

Ownership

Cloning

The Heap

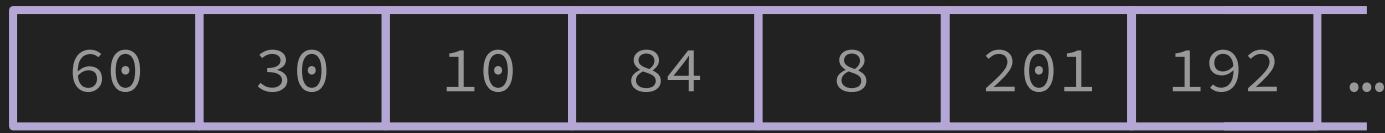
```rust
let nums = vec![1, 2, 3];
```

The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

```
let nums = vec![1, 2, 3];
```

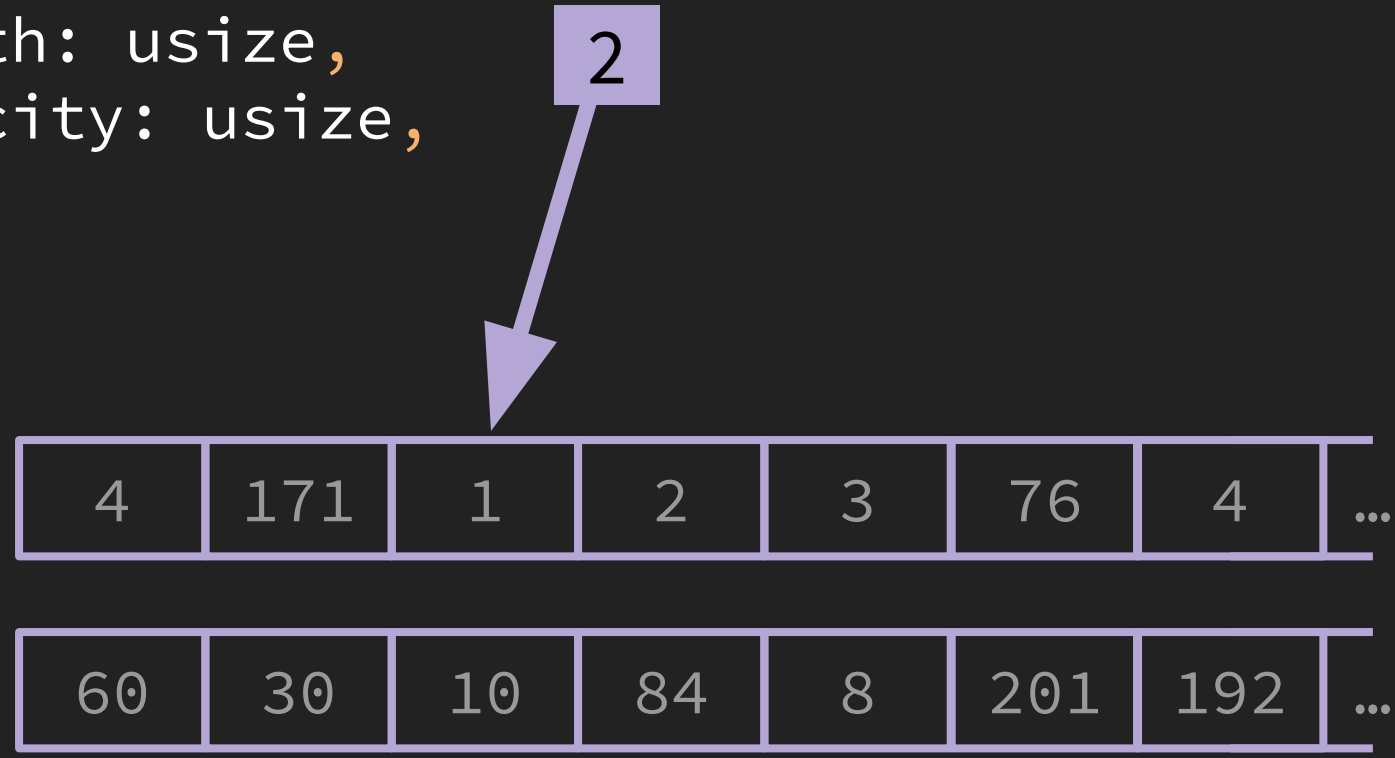
stack



The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

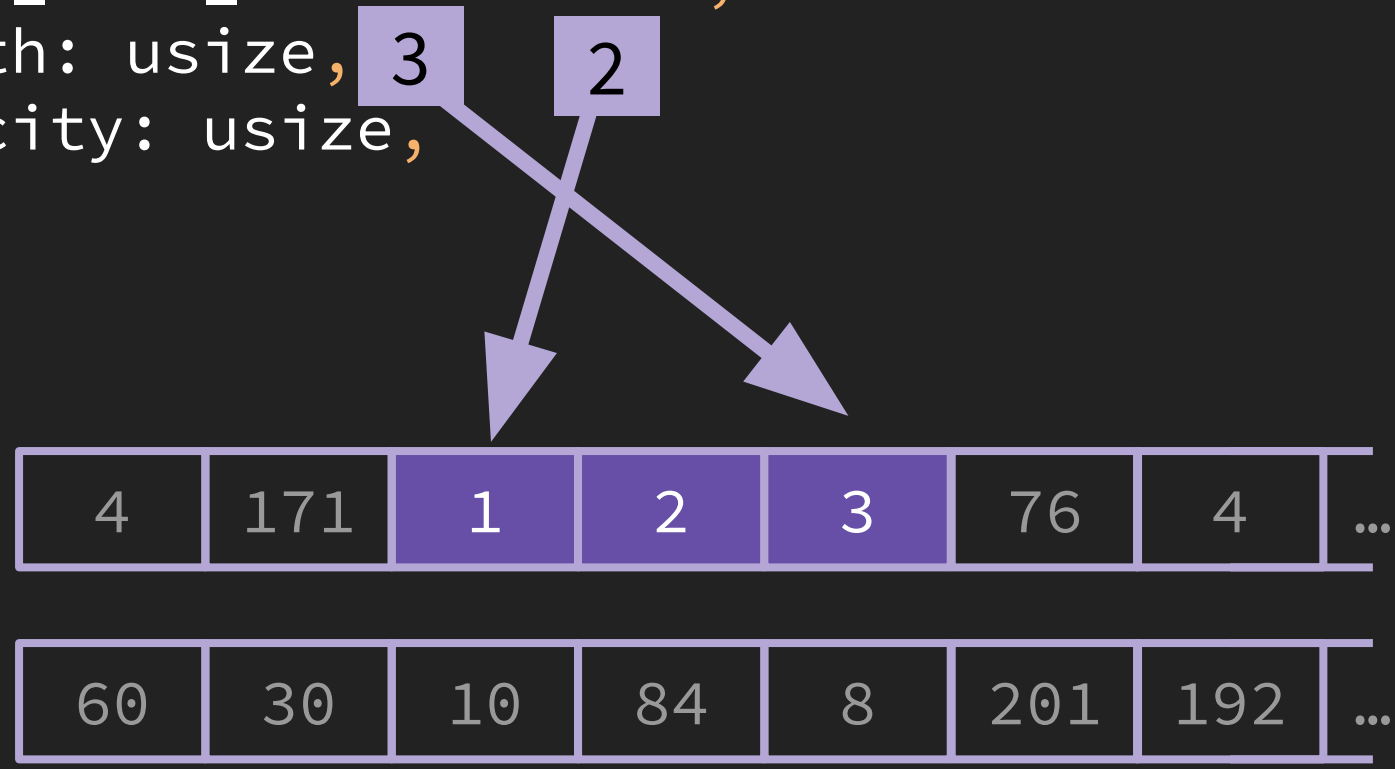
```
let nums = vec![1, 2, 3];
```



The Heap

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

```
let nums = vec![1, 2, 3];
```

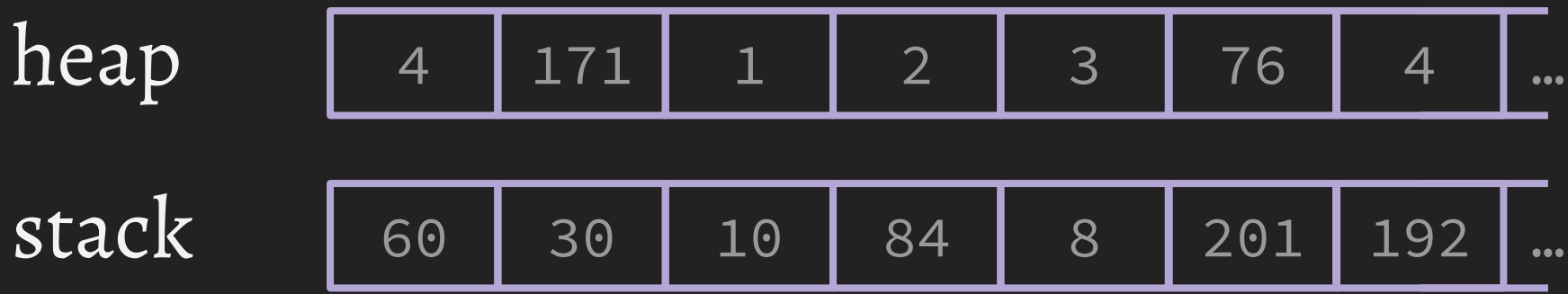


Heap Bookkeeping

```
let nums = vec![1, 2, 3];
```

`vec!` calls `alloc(3)`

"find 3 **unused heap bytes** in a row,
and mark them as in-use"



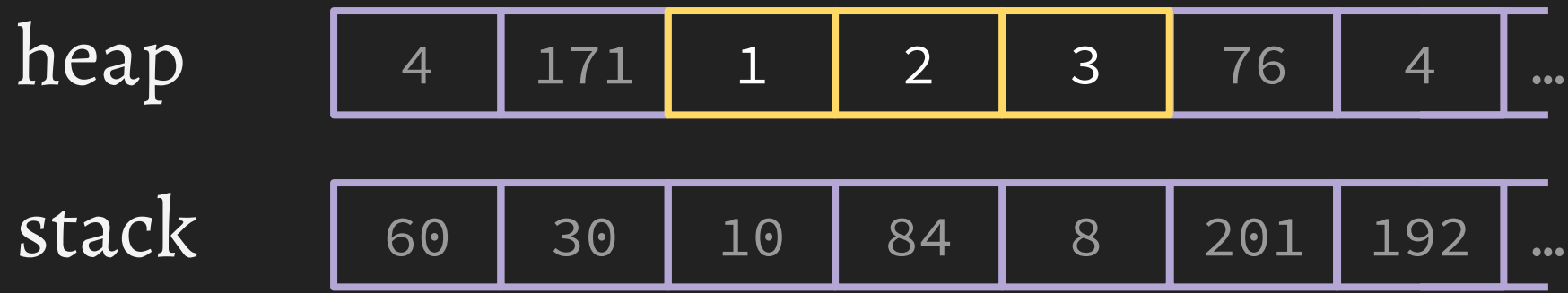
Heap Bookkeeping

```
let nums = vec![1, 2, 3];
```

...how does the bookkeeping system know when those bytes are no longer in use?

`vec!` calls `alloc(3)`

"find 3 **unused heap bytes** in a row, and mark them as in-use"



Manual Memory Management

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3, 4];
    let mut total_orders = 0;

    for order in orders.iter() {
        total_orders += orders;
    }

    let final_orders = finish(total_orders);

    return final_orders;
}
```

Manual Memory Management

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3, 4]; // alloc
    let mut total_orders = 0;

    for order in orders.iter() {
        total_orders += orders;
    }

    let final_orders = finish(total_orders);

    return final_orders;
}
```

Manual Memory Management

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3, 4]; // alloc
    let mut total_orders = 0;

    for order in orders.iter() {
        total_orders += orders;
    }

    let final_orders = finish(total_orders);
    dealloc(orders); // free these bytes!
    return final_orders;
}
```

Use-After-Free

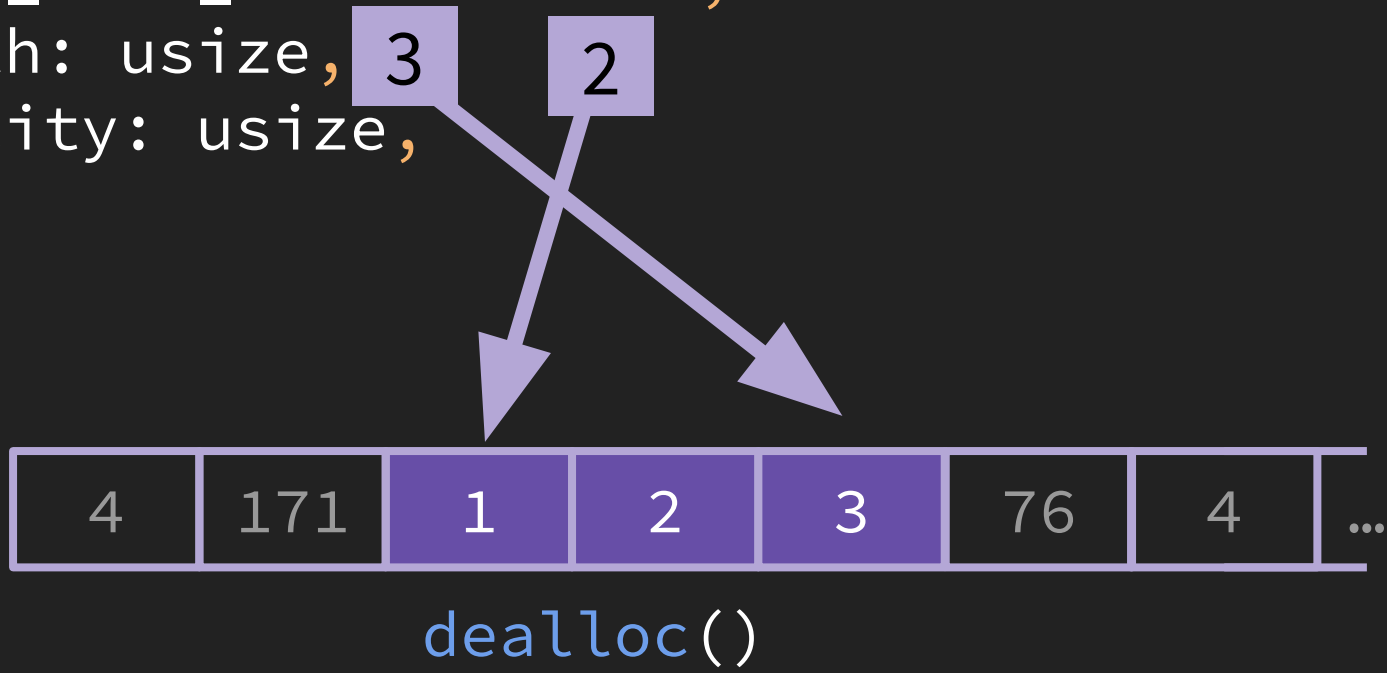
```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3, 4]; // alloc
    let mut total_orders = 0;
    dealloc(orders);
    for order in orders.iter() {
        total_orders += orders;
    }

    let final_orders = finish(total_orders);

    return final_orders;
}
```

***use-after-free** bug*

Use-After-Free

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

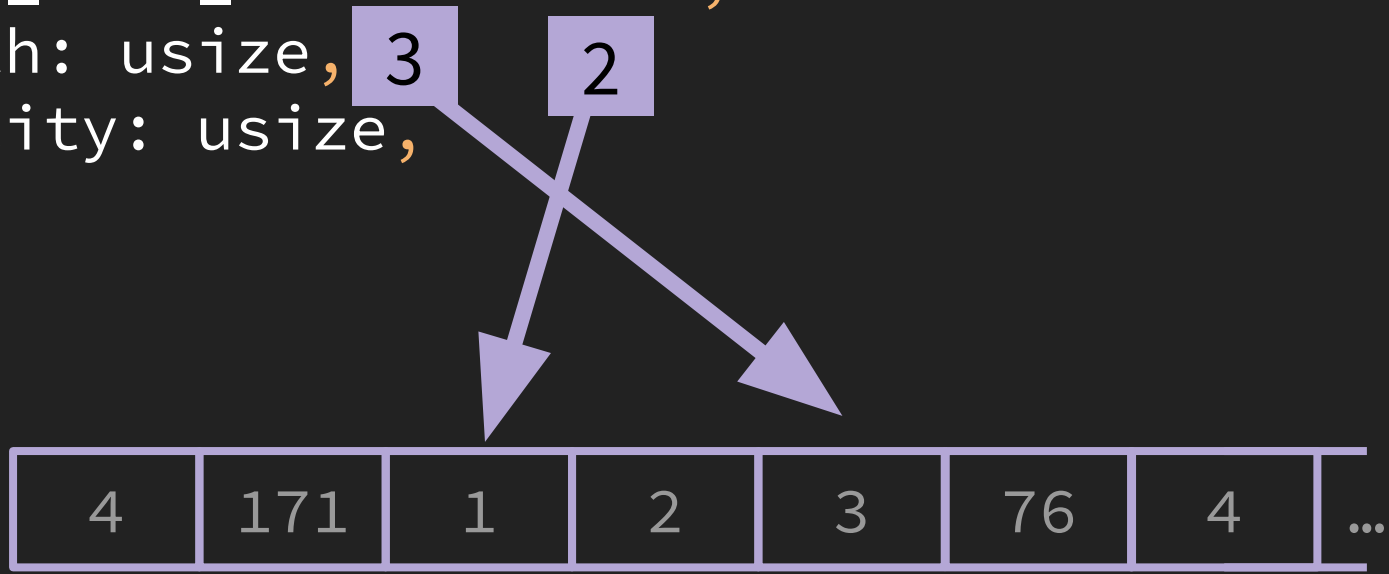
```
let nums = vec![1, 2, 3];
```



Use-After-Free

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

```
let nums = vec![1, 2, 3];
```

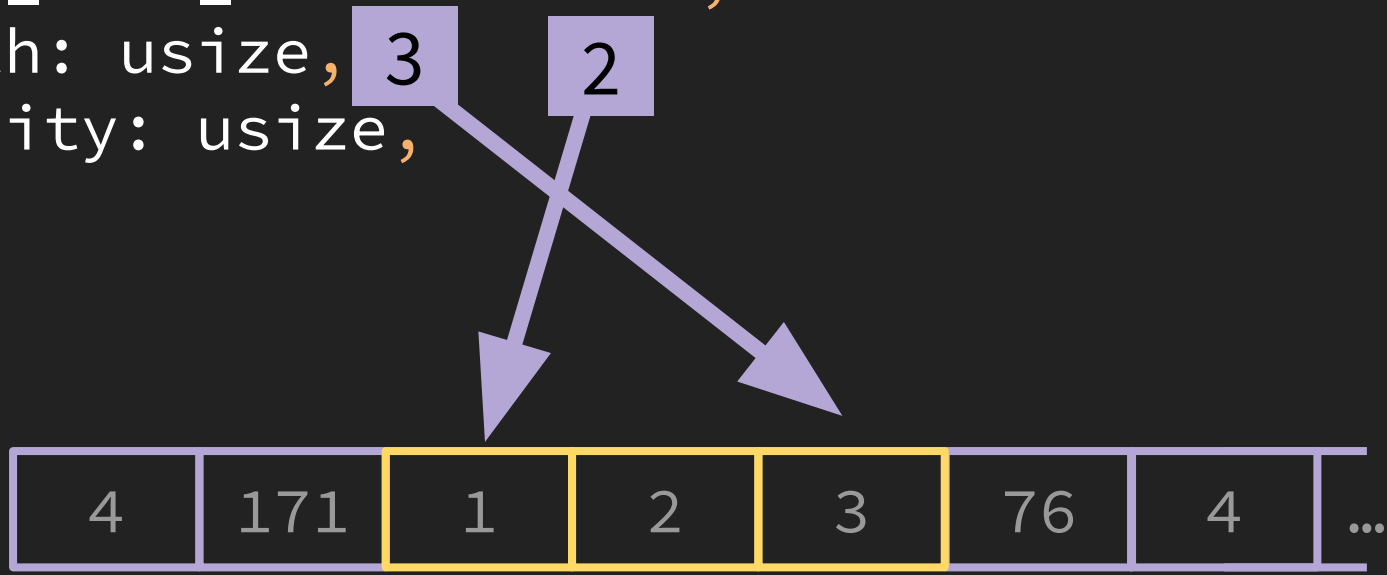



```
let things = vec![72, 49, 200];
alloc(3)
```

Use-After-Free

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

```
let nums = vec![1, 2, 3];
```

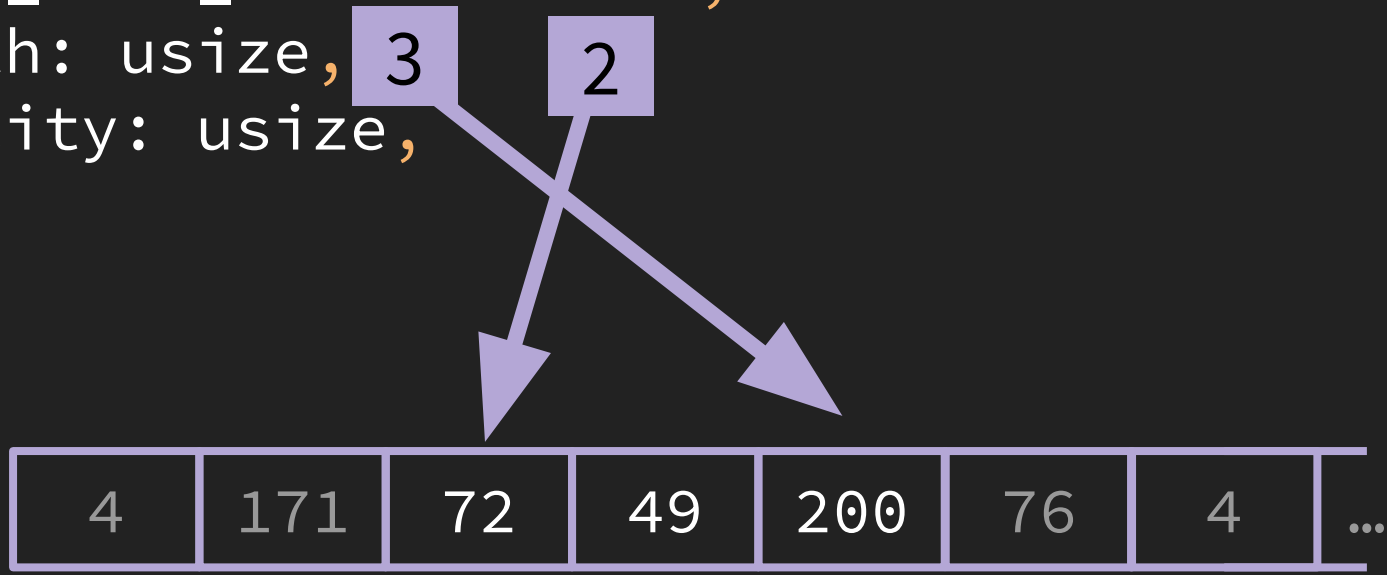



```
let things = vec![72, 49, 200];
alloc(3)
```

Use-After-Free

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

```
let nums = vec![1, 2, 3];
```




```
let things = vec![72, 49, 200];
for num in nums.iter() { … }
```

Use-After-Free

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3]; // alloc
    let mut total_orders = 0;
    dealloc(orders);
    for order in orders.iter() {
        total_orders += orders;
    }

    let final_orders = finish(total_orders);

    return final_orders;
}
```

***use-after-free** bug*

Use-After-Free

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3]; // alloc
    let mut total_orders = 0;

    for order in orders.iter() {
        total_orders += orders;
    }

    let final_orders = finish(total_orders);
    dealloc(orders); // free these bytes!
    return final_orders;
}
```

Double Free

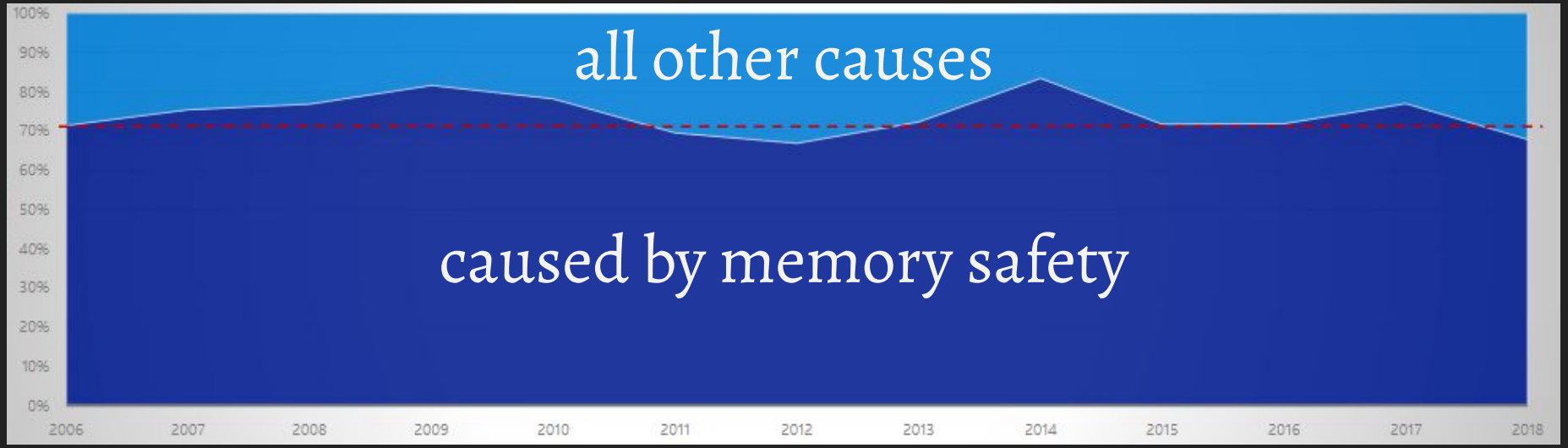
```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3]; // alloc
    let mut total_orders = 0;

    for order in orders.iter() {
        total_orders += orders;
    }
    dealloc(orders); // free these bytes!
    let final_orders = finish(total_orders);
    dealloc(orders); // free these bytes!
    return final_orders;
}
```

Double Free

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3]; // alloc
    let mut total_orders = 0;

    for order in orders.iter() {
        total_orders += orders;
    }
    dealloc(orders); // free heap_bytes[2]
    let final_orders = finish(total_orders);
    dealloc(orders); // free heap_bytes[2]
    return final_orders; double free bug
}  what if something else is using those bytes now?
```

Manual Memory Management

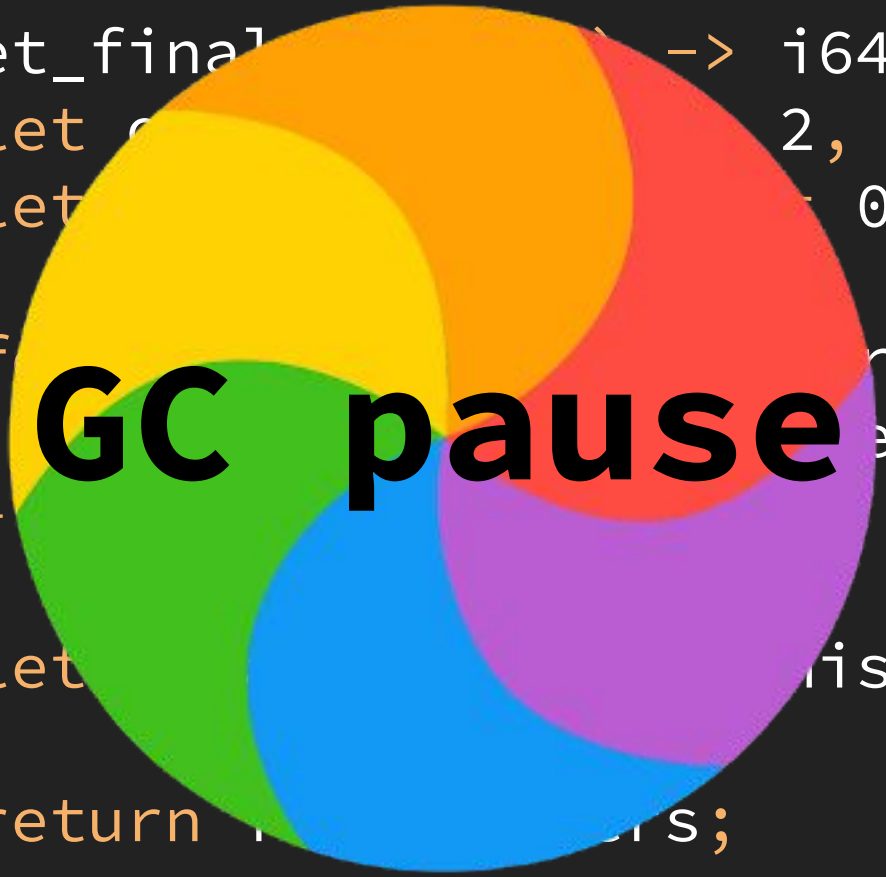
2018 Microsoft study of critical vulnerability causes



manual memory management: *it's error-prone!*

Garbage Collection

```
fn get_fina                    -> i64 {
    let o                      2, 3]; // GC alloc
    let                        0;

    f                          r() {
                               ers;
    }

    let                        nish(total_orders);

    return                     rs;
}
```



How Rust Manages Memory

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3]; // alloc
    let mut total_orders = 0;
    ✓ guaranteed no use-after-free
    for order in orders.iter() {
        total_orders += orders;
    }
    ✓ guaranteed no double free
    let final_orders = finish(total_orders);
    // dealloc(orders)  because it went out of scope
    return final_orders;
}   ✓ guaranteed no GC pause
```

How Rust Manages Memory

```
fn get_final_orders() -> i64 {
    let orders = vec![1, 2, 3]; // alloc
    let mut total_orders = 0;

    for order in orders.iter() {
        total_orders += orders;
    }
    could have safely freed the memory sooner!
    let final_orders = finish(total_orders);
    // dealloc(orders)
    return final_orders;
}
```

How Rust Manages Memory

```
fn get_final_orders() -> i64 {
    let mut total_orders = 0;

    {
        let orders = vec![1, 2, 3]; // alloc

        for order in orders.iter() {
            total_orders += orders;
        }
    } // dealloc(orders) because it went out of scope

    return finish(total_orders);
}
```

Ownership

```
fn get_years() -> Vec<i32> {
    let years = vec![1995, 2000, 2005, 2010];

    return years;
}


fn main() {
    let years = get_years();
}
```

Ownership

```
fn get_years() -> Vec<i32> {
    let years = vec![1995, 2000, 2005, 2010];
                // alloc
    return years;
} // dealloc(years)  because it went out of scope


fn main() {
    let years = get_years();
}       use-after-free bug!
```

Ownership

```
fn get_years() -> Vec<i32> {
    let years = vec![1995, 2000, 2005, 2010];
                // alloc
    return years;
}


fn main() {
    let years = get_years();
}
```

Ownership

```
fn get_years() -> Vec<i32> {
    let years = vec![1995, 2000, 2005, 2010];
                // alloc (this scope "owns" years)
    return years;
}


fn main() {
    let years = get_years();
}
```

Ownership

```
fn get_years() -> Vec<i32> {
    let years = vec![1995, 2000, 2005, 2010];
                // alloc (this scope "owns" years)
    return years;
}


fn main() {
    let years = get_years();
}
```

Ownership

```
fn get_years() -> Vec<i32> {
    let years = vec![1995, 2000, 2005, 2010];
                // alloc (this scope "owns" years)
    return years; // transfer ownership to main
}                           ("move" years to main's scope)


fn main() {
    let years = get_years(); // take ownership
} // dealloc(years) because it went out of scope
                    without being moved elsewhere
```

Ownership

```
fn get_years() -> Vec<i32> {
    let years = vec![1995, 2000, 2005, 2010];
                // alloc
    return years; // transfer ownership to main
}
```

✓ guaranteed no use-after-free

✓ guaranteed no double free

```
fn main() {
    let years = get_years(); // take ownership
} // dealloc(years)
```

✓ guaranteed no GC pause

Ownership

takes ownership of `years` from caller

```
fn print_years(years: Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
} // dealloc(years)
```

Ownership

```
fn print_years(years: Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
} // dealloc(years)

fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    print_years(years); years got deallocated
    print_years(years); use-after-free bug!
}
```

Use-after-move compiler error

```
fn print_years(years: Vec<i32>) { … }
```

```
error[E0382]: use of moved value: `years`
10 |     print_years(years);
   |                 ----- value moved here
11 |     print_years(years);
   |                 ^^^^^ value used here after move
```

```
fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    print_years(years);
    print_years(years);
}
```

Use-after-move compiler error

```
fn print_years(years: Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
} // dealloc(years)
```

Use-after-move compiler error

```
fn print_years(years: Vec<i32>) -> Vec<i32>{
    for year in years.iter() {
        println!("Year: {}", year);
    }
    return years; // transfer ownership
}                 // to the caller's scope

fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    let years2 = print_years(years);
    let years3 = print_years(years2);
} // dealloc(years3)
```

`.clone()` is your friend

```
fn print_years(years: Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
} // dealloc(years)

fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    print_years(years);
    print_years(years);
}
```

`.clone()` is your friend

```
fn print_years(years: Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
} // dealloc(years)

fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    print_years(years.clone());
    print_years(years);
}
```

Review of Part 5

manual	`dealloc()`, use-after-free, double free
automatic	GC pauses vs. Rust's scope-based heuristics
ownership	compiler errors prevent use-after-free
cloning	addressing compiler errors by sacrificing perf

Exercises for Part 5

Follow the instructions in **`part5/README.md`**

6. Borrowing

References & Borrowing

Mutable References

Slices

References & Borrowing

```
fn print_years(years: Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
} // dealloc(years)

fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    print_years(years); // compiler error
    print_years(years);
}
```

References & Borrowing

```
fn print_years(years: Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
}

fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    print_years(years);
    print_years(years);
}
```

References & Borrowing

"a **reference** to a `Vec<i32>`"

```
fn print_years(years: &Vec<i32>) {
    for year in years.iter() {
        println!("Year: {}", year);
    }
}
```

"a **reference** to a `self`"

```
fn len(&self) -> usize
```

```
fn main() {
    let years = vec![1990, 1995, 2000, 2010];

    print_years(&years);
    print_years(&years);
}
```

temporarily give print_years access to years (`&years` - "borrow years")

Use-after-move compiler error

"borrow checker error"

```
error[E0382]: use of moved value: `years`
10 |     print_years(years);
   |                 ----- value moved here
11 |     print_years(years);
   |                 ^^^^^ value used here after move
```

You can't "turn off the borrow checker" in Rust

an article by Rust Core Team member Steve Klabnik

Mutable References

```
let mut years: Vec<i32> = vec![1990, 1995];
let mutable_years: &mut Vec<i32> = &mut years;
let length = mutable_years.len();
mutable_years.clear();
// clear() removes all elements from the Vec
```

```
fn clear(&mut self) {
  // set self's length to 0
}
```

```
fn len(&self) -> usize
```

Mutable Reference Restrictions

```
let years: Vec<i32> = vec![1990, 1995];

let years_ref1: &Vec<i32> = &years;
let years_ref2: &Vec<i32> = &years;
```

Mutable Reference Restrictions

```
let years: Vec<i32> = vec![1990, 1995];

let years2: &mut Vec<i32> = &mut years;
```

```
error[E0596]: cannot borrow `years` as mutable, as it is not declared as mutable
 --> main.rs:4:33
  |
2 |     let years = vec![1990, 1995, 2000, 2010];
  |         ----- help: consider changing this to be mutable: `mut years`
```

Mutable Reference Restrictions

```
let mut years: Vec<i32> = vec![1990, 1995];

let years2: &mut Vec<i32> = &mut years;
let years3: &mut Vec<i32> = &mut years;
```

```
error[E0499]: cannot borrow `years` as mutable more than once at a time
 --> main.rs:5:33
  |
4 |     let years2: &mut Vec<i32> = &mut years;
  |                                  ---------- first mutable borrow occurs here
5 |     let years3: &mut Vec<i32> = &mut years;
  |                                  ^^^^^^^^^^ second mutable borrow occurs here
```

Mutable Reference Restrictions

```
let mut years: Vec<i32> = vec![1990, 1995];

let years2: &Vec<i32> = &years;
let years3: &mut Vec<i32> = &mut years;
```

```
error[E0502]: cannot borrow `years` as mutable because it is also borrowed as immutable
 --> main.rs:5:33
  |
4 |     let years2: &Vec<i32> = &years;
  |                              ------ immutable borrow occurs here
5 |     let years3: &mut Vec<i32> = &mut years;
  |                                 ^^^^^^^^^^ mutable borrow occurs here
```

Mutable Reference Restrictions

```
let mut years: Vec<i32> = vec![1990, 1995];

let years2: &mut Vec<i32> = &mut years;
let years3: &Vec<i32> = &years;
```

```
error[E0502]: cannot borrow `years` as immutable because it is also borrowed as mutable
 --> main.rs:5:29
  |
4 |     let years2: &mut Vec<i32> = &mut years;
  |                                  ---------- mutable borrow occurs here
5 |     let years3: &Vec<i32> = &years;
  |                             ^^^^^^ immutable borrow occurs here
```

Mutable Reference Restrictions

```
let years: Vec<i32> = vec![1990, 1995, 2005];
let mutable_years: &mut Vec<i32> = &mut years;

mutable_years.clear();
// clear() removes all elements from the Vec
```

```
fn clear(&mut self) {
  // set self's length to 0
}

fn len(&self) -> usize
```

Slices

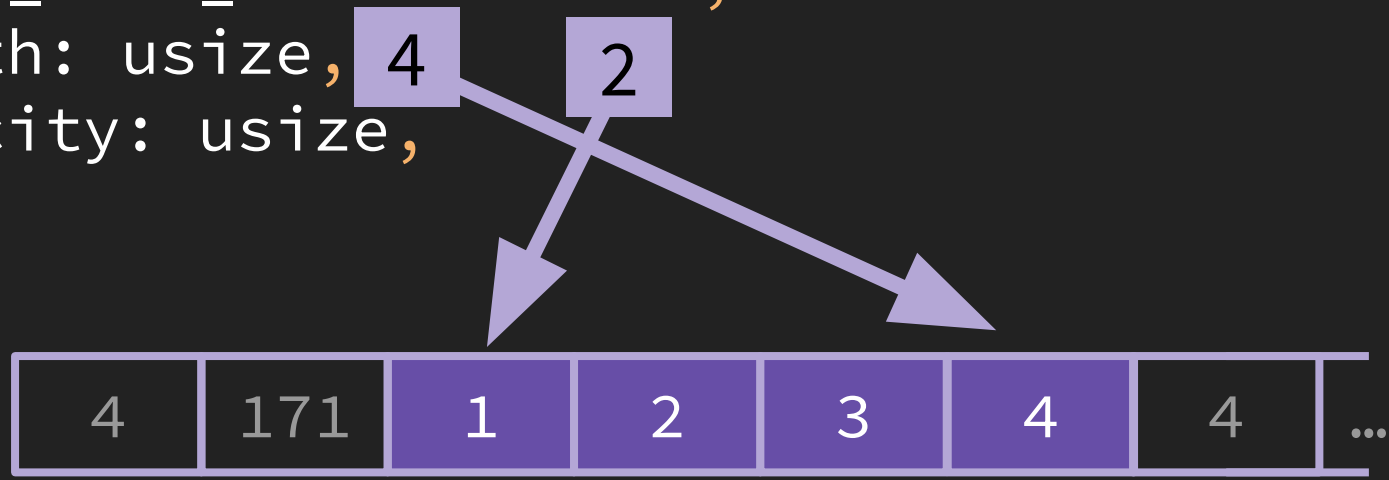
```
let nums = vec![1, 2, 3];
```

```
struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

Slices

```
struct SliceMetadata {
    first_elem_index: usize,
    length: usize
}

struct VecMetadata {
    first_elem_index: usize,
    length: usize,
    capacity: usize,
}
```

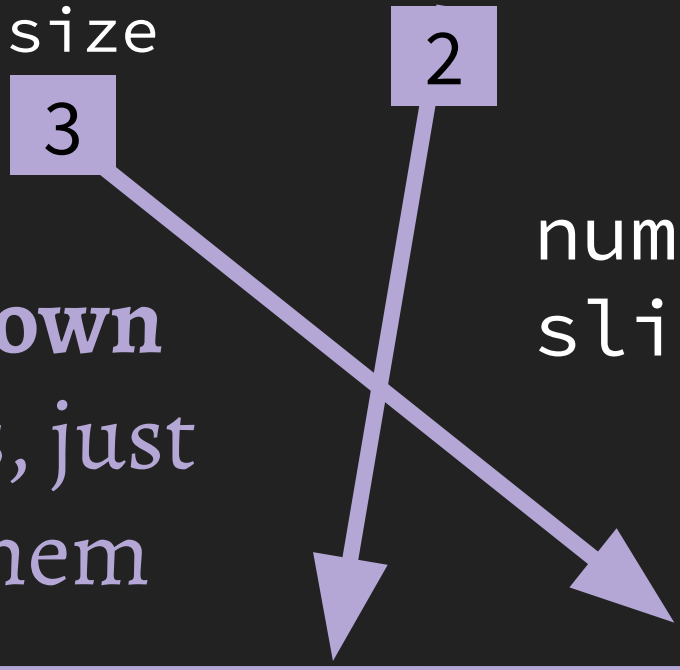
```
let nums = vec![1, 2, 3];
let slice = &nums[0..2];
```



Slices

```
struct SliceMetadata {
    first_elem_index: usize,
    length: usize
}
```

```
let nums = vec![1, 2, 3];
let slice = &nums[0..3];
```




```
nums: Vec<u8>
slice: &[u8]
```

slice doesn't **own** the elements, just **references** them

heap

Slices

```
let str_slice = string[3..7];
```

slice doesn't **own** the elements, just **references** them

```
nums: Vec<u8>
slice: &[u8]
```

Slices

```
let str_slice: &str = string[3..7];
```

slice doesn't **own** the elements, just **references** them

```
nums: Vec<u8>
slice: &[u8]
```

```
nums.as_slice()
```

```
string.as_str()
```

Review of Part 6

borrowing

```
let years_ref: &Vec<i32> = &years;
```

mut refs

```
let x: &mut Vec<i32> = &mut years;
```

slices

```
let slice: &[i32] = &years[1..3];
```

```
let foo: &str = &string[1..3];
```

Exercises for Part 6

Follow the instructions in **`part6/README.md`**

7. Lifetimes

Lifetimes

Lifetime Annotations

Lifetime Elision

The Static Lifetime

Lifetimes

```
let years: Vec<i64> = vec![
    1980, 1985, 1990, 1990, 2000, 2005, 2010
];

let eighties: &[i64] = &years[0..2];
let nineties: &[i64] = &years[2..4];

println!(
    "We have {} years in the nineties",
    nineties.len()
);
```

Lifetimes

```
fn jazz_releases(years: &[i64]) -> Releases {
    let eighties: &[i64] = &years[0..2];
    let nineties: &[i64] = &years[2..4];

    Releases {
        years,
        eighties,
        nineties,
    }
}
```

```
struct Releases {
    years: &[i64],
    eighties: &[i64],
    nineties: &[i64],
}
```

Lifetimes

```
fn jazz_releases(years: &[i64]) -> Releases {…}

let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz_releases(&all_years)
};

let eighties =
    releases.eighties;
```

use-after-free!

```
struct Releases {
    years: &[i64],
    eighties: &[i64],
    nineties: &[i64],
}
```

Lifetimes

```
fn jazz_releases(years: &[i64]) -> Releases {…}

let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz_releases(&all_years)
}; // dealloc(all_years)
 slice of all_years.releases
let eighties =
    releases.eighties;
```

~~use-after-free!~~ compiler error!

```
struct Releases {
   years: &[i64],
   eighties: &[i64],
   nineties: &[i64],
}
```

Lifetimes

the ***lifetime*** of `all_years`

```
let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz_releases(&all_years)
}; // dealloc(all_years)

let eighties =
    releases.eighties;
```

refers to `all_years`
after its lifetime has ended

Lifetimes

```
let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz_releases(&all_years)
}; // dealloc(all_years)
```

Lifetimes

the ***lifetime*** of `all_years`

```
let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz_releases(&all_years)
}; // dealloc(all_years)

let eighties =
    releases.eighties;
```

refers to `all_years`
after its lifetime has ended

Lifetimes

```
fn jazz_releases(years: &[i64]) -> Releases {…}

let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz_releases(&all_years)
}; // dealloc(all_years)

let eighties =
    releases.eighties;
```

refers to `all_years`
after its lifetime has ended

Lifetimes

still within its lifetime

```
fn jazz_releases(years: &[i64]) -> Releases {…}
```

Releases *depends on* years

```
let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz_releases(&all_years)
}; // dealloc(all_years)

let eighties =
    releases.eighties;
```

refers to `all_years`
after its lifetime has ended

Lifetimes

```
fn jazz(years: &[i64]) -> Releases {
    let eighties: &[i64] = &years[0..2];
    let nineties: &[i64] = &years[2..4];

    Releases {
        years,
        eighties,
        nineties,
    }
}
```

```
struct Releases {
    years: &[i64],
    eighties: &[i64],
    nineties: &[i64],
}
```

Lifetimes

a lifetime parameter named `'a`

```
fn jazz<'a>(years: &'a [i64]) -> Releases {
    let eighties: &[i64] = &years[0..2];
    let nineties: &[i64] = &years[2..4];

    Releases {
        years,
        eighties,
        nineties,
    }
}
```

```
struct Releases {
    years: &[i64],
    eighties: &[i64],
    nineties: &[i64],
}
```

Lifetimes

a lifetime parameter named `'a`

```
fn jazz<'a>(years: &'a [i64]) -> Releases {
    let eighties: &'a [i64] = &years[0..2];
    let nineties: &'a [i64] = &years[2..4];

    Releases {
        years,
        eighties,
        nineties,
    }
}
```

```
struct Releases {
    years: &[i64],
    eighties: &[i64],
    nineties: &[i64],
}
```

Lifetimes

a lifetime parameter named 'a

```
fn jazz<'a>(years: &'a [i64]) -> Releases {
    let eighties: &'a [i64] = &years[0..2];
    let nineties: &'a [i64] = &years[2..4];

    Releases {
        years,
        eighties,
        nineties,
    }
}
```

```
struct Releases<'y> {
    years: &'y [i64],
    eighties: &'y [i64],
    nineties: &'y [i64],
}
```

a lifetime parameter named 'y

Lifetimes

```
fn jazz<'a>(years: &'a [i64]) -> Releases<'a>

let releases = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz(&all_years)
}; // dealloc(all_years)

let eighties =
    releases.eighties;
```

the ***lifetime*** of `all_years`

refers to `all_years`
after its lifetime has ended

Lifetimes

```
fn jazz<'a>(years: &'a [i64]) -> Releases<'a>

let releases: Releases<'a> = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz(&all_years)
}; // dealloc(all_years)

let eighties =
    releases.eighties;
```

refers to `Releases<'a>` after `'a` has ended

Lifetimes

```
fn jazz<'a>(years: &'a [i64]) -> Releases<'a>

let releases: Releases<'a> = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz(&all_years)
};
```

lifetime annotations
are required in all structs
that hold references

```
struct Releases<'y> {
    years: &'y [i64],
    eighties: &'y [i64],
    nineties: &'y [i64],
}
```

Lifetime Elision

```
fn jazz<'a>(years: &'a [i64]) -> Releases<'a>

let releases: Releases<'a> = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz(&all_years)
};
```

Lifetime Elision

```
fn jazz<'a>(years: &'a [i64]) -> Releases<'a>

let releases: Releases<'_> = {
    let all_years: Vec<i64> = vec![
        1980, 1985, 1990, 1995, 2000, 2000
    ];
    jazz(&all_years)
};
```

The Static Lifetime

```
let name = "Sam";

let name: &'static str = "Sam";

let name: &str = "Sam";
```

Review of Part 7

lifetimes	time between a value is allocated and deallocated
annotations	`struct Foo<'a> { x: &'a [i64] }`
elision	`fn len<'a>(&'a self) ⇒ fn len (&self)`
&'static	`let hi: &'static str = "Hello, World!"`

Exercises for Part 7

Follow the instructions in **`part7/README.md`**

Wrap-Up

Summary

Additional Resources

rust-lang.org

“ A language empowering everyone to build **reliable** and **efficient** software.”

Rust

compiles to either

machine code

```
0111010110
1011001111
0110110101
1001001011
```

Who uses Rust?

More at rust-lang.org/production

Why use Rust?

1. Speed
2. Performance
3. Going Real Fast

1. Primitives

strings

```
format!("Hi, {}!", name)
```

floats

```
let mut float: f64 = 1.234;
```

integers

```
let one: u32 = 1.99 as u32;
```

booleans

```
if x > 5 { true } else { false }
```

2. Collections

tuples

```
let foo: (i64, bool) = (1, true);
```

structs

```
struct Foo { x: i64, is_up: bool }
```

arrays

```
let arr: [u32; 3] = [1, 2, 3];
```

memory

`u8` is 8 bits (1 byte), `u16` is 16 bits (2 bytes), …

3. Pattern Matching

enums

```
let purple = Color::Custom(1, 2, 3);
```

matching

```
match color {
    Color::Custom(r, g, b) => {
        r + g + b
    }
}
```

methods

```
color.rgb() == Color::rgb(color)
```

type params

```
let first: Option<char> = str.pop();
```

4. Vectors

Vec	`let nums: Vec<u8> = vec![1, 2];`
usize	`u64` on 64-bit systems, `u32` on 32-bit
stack memory	`stack_bytes[stack_length - 1]`
heap memory	`heap_bytes[index_of_first_elem]`

5. Ownership

manual	`dealloc()`, use-after-free, double free
automatic	GC pauses vs. Rust's scope-based heuristics
ownership	compiler errors prevent use-after-free
cloning	addressing compiler errors by sacrificing perf

6. Borrowing

borrowing

```
let years_ref: &Vec<i32> = &years;
```

mut refs

```
let x: &mut Vec<i32> = &mut years;
```

slices

```
let slice: &[i32] = &years[1..3];
```

```
let foo: &str = &string[1..3];
```

7. Lifetimes

lifetimes	time between a value is allocated and deallocated
annotations	`struct Foo<'a> { x: &'a [i64] }`
elision	`fn len<'a>(&'a self) ⇒ fn len (&self)`
&'static	`let hi: &'static str = "Hello, World!"`

Additional Resources

rust-lang.org/learn

rust-lang.org/community

rust-analyzer.github.io (LSP extension)

The lessons/ directory in this workshop's repo

@rtfeldman