CS 241: Systems Programming
Lecture 14. Structures

Fall 2025

Prof. Stephen Checkoway

struct

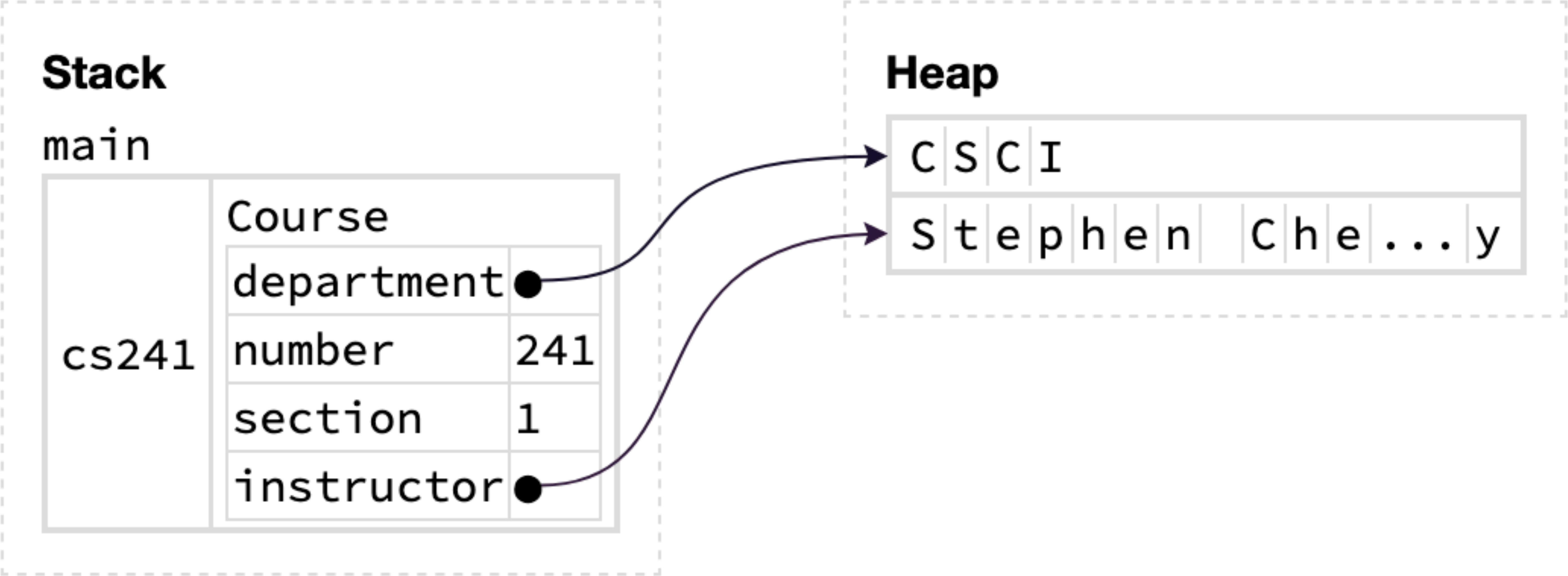
```
struct Course {
    department: String,
    number: i32,
    section: i32,
    instructor: String,
}

fn main() {
    let cs241 = Course {
        department: String::from("CSCI"),
        number: 241,
        section: 1,
        instructor: String::from("Stephen Checkoway"),
    };
}
```

struct

```rust
struct Course {
    department: String,
    number: i32,
    section: i32,
    instructor: String,
}

fn main() {
    let cs241 = Course {
        department: String::from("CSCI"),
        number: 241,
        section: 1,
        instructor: String::from("Stephen Checkoway"),
    };
}
```



Accessing members

```
struct Course {
    department: String,
    number: i32,
    section: i32,
    instructor: String,
}

fn main() {
    let cs241 = Course {
        department: String::from("CSCI"),
        number: 241,
        section: 1,
        instructor: String::from("Stephen Checkoway"),
    };
    println!("{} {}", cs241.department, cs241.number);
}
```

Modifying a member

```rust
struct Course {
    department: String,
    number: i32,
    section: i32,
    instructor: String,
}

fn main() {
    let mut cs241 = Course {
        department: String::from("CSCI"),
        number: 241,
        section: 1,
        instructor: String::from("Stephen Checkoway"),
    };

    cs241.department = String::from("Computer Science");
}
```

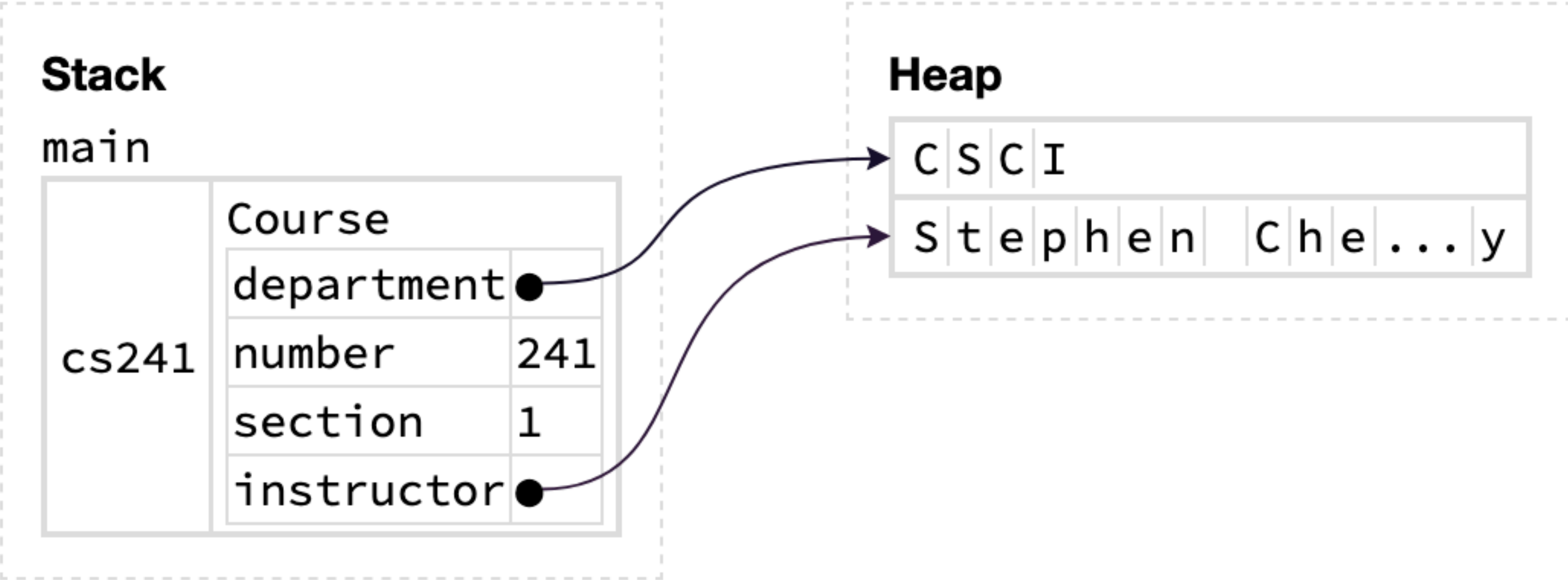
Old department String was dropped (and its contents deallocated)

Modifying a member

```
struct Course {
    department: String,
    number: i32,
    section: i32,
    instructor: String,
}

fn main() {
    let mut cs241 = Course {
        department: String::from("CSCI"),
        number: 241,
        section: 1,
        instructor: String::from("Stephen Checkoway"),
    };

    cs241.department = String::from("Computer Science");
}
```



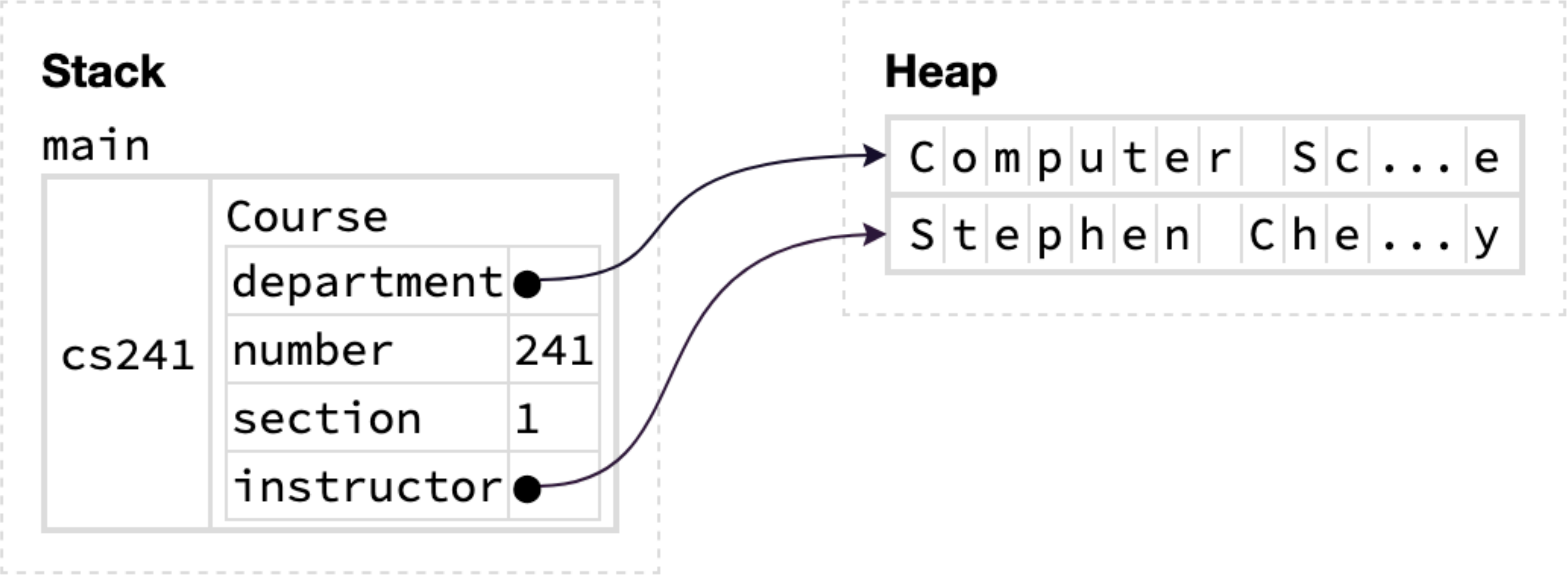
Old department String was dropped (and its contents deallocated)

Modifying a member

```
struct Course {
    department: String,
    number: i32,
    section: i32,
    instructor: String,
}

fn main() {
    let mut cs241 = Course {
        department: String::from("CSCI"),
        number: 241,
        section: 1,
        instructor: String::from("Stephen Checkoway"),
    };

    cs241.department = String::from("Computer Science");
}
```



Old department String was dropped (and its contents deallocated)

Field init shorthand

```rust
fn new_course(department: &str, number: i32) -> Course {
    Course {
        department: department.to_string(),
        number, // <- No need to write number: number
        section: 1,
        instructor: String::from("Staff"),
    }
}

fn main() {
    let cs241 = new_course("CSCI", 241);
    println!("{} {}", cs241.department, cs241.number);
}
```

You're designing a program for interacting with social media and you want to represent posts using a Post structure you're designing. Each Post needs an account name, contents, and a number of "likes." The account name and contents never change, but the number of likes can. Which structure definition best models this?

```
// A
struct Post {
    account: String,
    contents: String,
    likes: u64,
}
```

```
// B
struct Post {
    account: String,
    contents: String,
    likes: mut u64,
}
```

```
// C
struct Post {
    String account;
    String contents;
    u64 likes;
}
```

```
// D
struct Post {
    account: readonly String,
    contents: readonly String,
    likes: u64,
}
```

Update syntax

```
fn main() {
    let cs241 = new_course("CSCI", 241);
    let cs241_2 = Course {
        instructor: String::from("Mary Hogan"),
        section: 2,
        ..cs241
    };
}
```

Moves all of the remaining fields from cs241 into cs241_2 and drops cs241

What will this code print out?

A. CSCI Mary Hogan

B. CSCI Cynthia Taylor

C. This code will not compile

```
fn main() {
  let cs241 = Course {
  department: String::from("CSCI"),
  number: 241,
  section: 1,
  instructor: String::from(“Mary Hogan”),
  };

  let cs241_2 = Course {
  instructor: String::from("Cynthia Taylor"),
  section: 2,
  ..cs241
  };
  println!("{} {}", cs241.department, cs241_2.instructor);
}
```

Tuples

Tuples let us group multiple, heterogeneous types together

They have fixed size (no adding or removing elements)

```
let val: (i32, char, f64) = (42, '🦀', 6.022e23);
```

Tuples can be built from any types

Tuple structs

```rust
struct Point(i32, i32);

fn main() {
    let p = Point(4, 5);
    println!("{} {}", p.0, p.1);
}
```

Create an new instance by giving the name and field values

Refer to fields using .0, .1, .2, etc., just like tuples

Printing structs

Printing structs

We cannot print an instance of a struct with println!("{cs241}")

Printing structs

We cannot print an instance of a struct with println!("{cs241}")

```
error[E0277]: `Course` doesn't implement `std::fmt::Display`
```

Printing structs

We cannot print an instance of a struct with println!("{cs241}")

error[E0277]: `Course` doesn't implement `std::fmt::Display`

Display is a **trait** (like an interface in Java) that we can implement for our own types

Printing structs

We cannot print an instance of a struct with println!("{cs241}")

error[E0277]: `Course` doesn't implement `std::fmt::Display`

Display is a **trait** (like an interface in Java) that we can implement for our own types

For arrays and Vecs and Results, we printed the debug representation with println!("{cs241:?}")

Printing structs

We cannot print an instance of a struct with println!("{cs241}")

error[E0277]: `Course` doesn't implement `std::fmt::Display`

Display is a **trait** (like an interface in Java) that we can implement for our own types

For arrays and Vecs and Results, we printed the debug representation with println!("{cs241:?}")

error[E0277]: `Course` doesn't implement `Debug`

Deriving Debug

We can ask Rustc to produce an implementation of the Debug trait for us

```rust
#[derive(Debug)]
struct Course { … }

fn main() {
    let cs241 = new_course("CSCI", 241);
    println!("{cs241:?}");
    println!("{cs241:#?}");
}
```

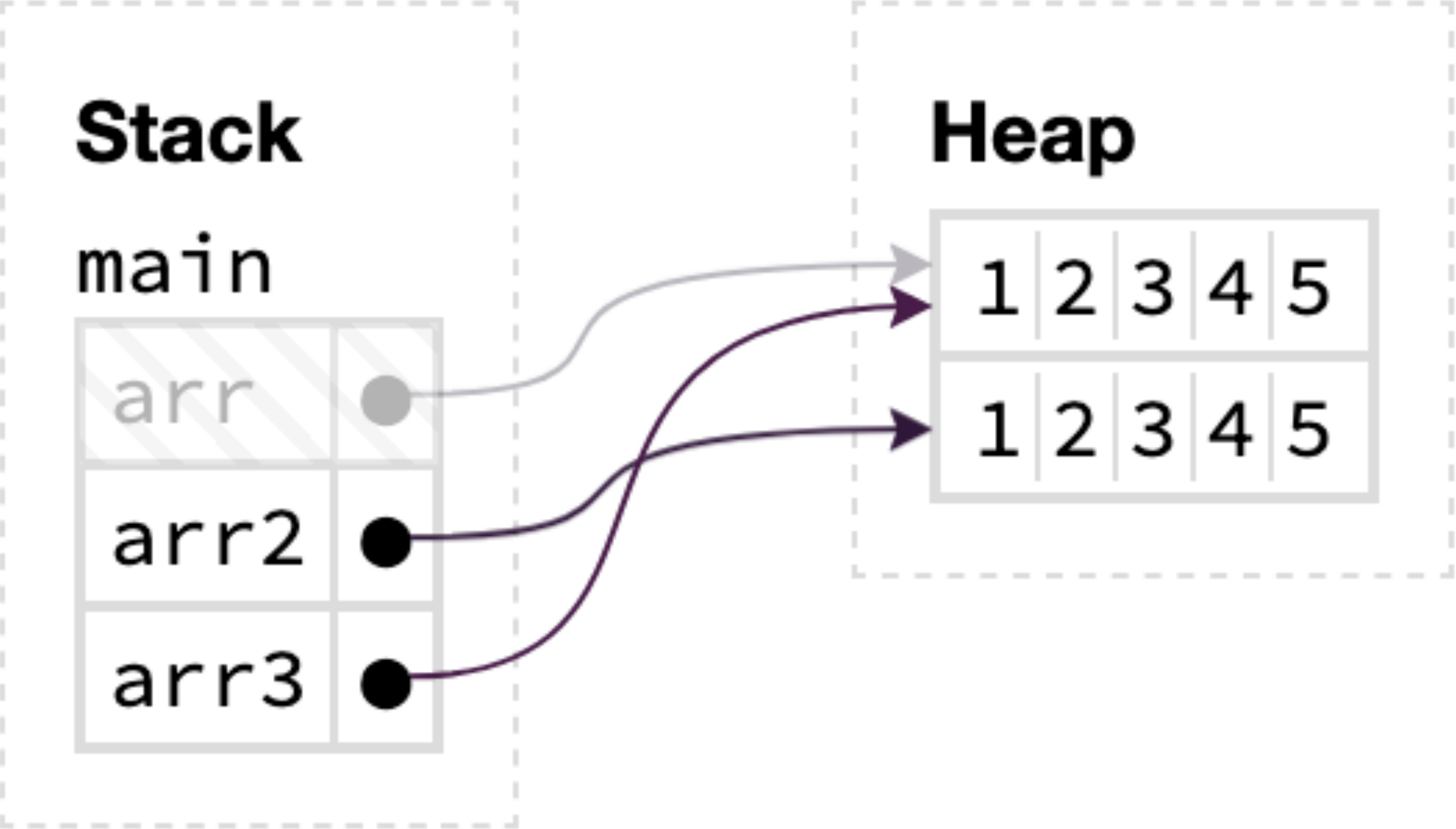
```
Output:
Course { department: "CSCI", number: 241, section: 1, instructor: "Staff" }
Course {
    department: "CSCI",
    number: 241,
    section: 1,
    instructor: "Staff",
}
```

Making copies via clone

The Clone trait has a .clone() method that makes a deep copy of objects

```
fn main() {
    let arr = vec![1, 2, 3, 4, 5];
    let arr2 = arr.clone();
    let arr3 = arr;
}
```

Most types implement Clone



Deriving Clone

```rust
#[derive(Debug, Clone)]
struct Course {
    department: String,
    number: i32,
    section: i32,
    instructor: String,
}
```

All of the members’ types must implement Clone in order to derive Clone

Methods

Methods

Methods are functions defined for a type that take an instance of the type as the first argument

- Similar to methods in object-oriented languages like Java and Python

Methods

Methods are functions defined for a type that take an instance of the type as the first argument

- Similar to methods in object-oriented languages like Java and Python

The first parameter is always named `self` and it is explicit (unlike Java and C++'s implicit `this` parameter)

Methods

Methods are functions defined for a type that take an instance of the type as the first argument

- Similar to methods in object-oriented languages like Java and Python

The first parameter is always named `self` and it is explicit (unlike Java and C++'s implicit `this` parameter)

We've used a bunch of examples of methods already including

- `.len()` for slices
- `.push()` for Strings and Vecs
- `.push_str()` for Strings
- `.chars()` to get an iterator over characters in a String
- `.iter()` to get an iterator over a collection (like a Vec)

Three types of methods

Three types of methods

There are three types of methods which are distinguished by the self parameter

Three types of methods

There are three types of methods which are distinguished by the self parameter

- `fn foo(&self) {}`   self is a shared reference to the instance

Three types of methods

There are three types of methods which are distinguished by the self parameter

- `fn foo(&self) {}` self is a shared reference to the instance
- `fn foo(&mut self) {}` self is a mutable reference to the instance

Three types of methods

There are three types of methods which are distinguished by the self parameter

- `fn foo(&self) {}` self is a shared reference to the instance
- `fn foo(&mut self) {}` self is a mutable reference to the instance
- `fn foo(self) {}` foo takes ownership of the instance

Methods taking shared refs

```
impl Course {
    fn name(&self) -> String {
        format!("{} {}", self.department, self.number)
    }

    fn full_name(&self) -> String {
        format!("{} {}-{}", self.department, self.number, self.section)
    }
}

fn main() {
    let cs241 = new_course("CSCI", 241);
    println!("{}", cs241.name());
}
```

Methods taking mutable refs

```
impl Course {
    fn set_instructor(&mut self, instructor: &str) {
        self.instructor = instructor.to_string();
    }
}

fn main() {
    let mut cs241 = new_course("CSCI", 241);
    cs241.set_instructor("Mary Hogan");
    println!("{}", cs241.instructor);
}
```

Methods taking ownership

Methods taking ownership

Two main use cases

- The type can be copied (like `i32`, `usize`, `bool`)
- The method is returning some lower-level implementation

Methods taking ownership

Two main use cases

- The type can be copied (like `i32`, `usize`, `bool`)
- The method is returning some lower-level implementation

i32 (and other integer types) have a bunch of methods that take self

- `fn abs(self) -> i32`
- `fn pow(self, exp: u32) -> i32`

Methods taking ownership

Two main use cases

- The type can be copied (like `i32`, `usize`, `bool`)
- The method is returning some lower-level implementation

i32 (and other integer types) have a bunch of methods that take self

- `fn abs(self) -> i32`
- `fn pow(self, exp: u32) -> i32`

Many types have `.into_foo()` methods that return implementation details

- String has `fn into_bytes(self) -> Vec<u8>`

Getters and setters are methods for getting or setting the value of a field. Imagine we have the following struct with a getter and a setter for the url field. Which of the three possible self parameters should we use for the url() and set_url() methods?

```
struct Foo {
    url: String,
}

impl Foo {
    fn url(SELF) -> &str { &self.url }
    fn set_url(SELF, url: String) { self.url = url; }
}
```

	url()	set_url()
A	&self	&mut self
B	self	mut self
C	self	&self
D	&mut self	&mut self
E	&self	&self

Method calls are syntactic sugar

```
cs241.set_instructor("Mary Hogan");
println!("{}", cs241.name);
```

is the same as

```
Course::set_instructor(&mut cs241, "Mary Hogan");
println!("{}", Course::name(&cs241));
```

Associated functions

Associated functions

Functions defined inside `impl` blocks are called associated functions

Associated functions

Functions defined inside `impl` blocks are called associated functions

Methods are one type of associated functions

Associated functions

Functions defined inside `impl` blocks are called associated functions

Methods are one type of associated functions

We can also have associated functions that don’t take an instance as an argument

- These are typically constructor functions
- Most types have a `new()` associated function that returns a new instance of the type

Associated functions

Functions defined inside `impl` blocks are called associated functions

Methods are one type of associated functions

We can also have associated functions that don’t take an instance as an argument

- These are typically constructor functions
- Most types have a `new()` associated function that returns a new instance of the type

Inside the `impl` block we can refer to the type as `Self`

Constructor

```
impl Course {
    fn new(department: &str, number: i32) -> Self {
        Self {
            department: department.to_string(),
            number,
            section: 1,
            instructor: String::from("Staff"),
        }
    }
}

fn main() {
    let cs241 = Course::new("CSCI", 241);
    println!("{}", cs241.name());
}
```

Examples from the standard library

- String::new() — Creates a new, empty String
- Vec::new() — Creates a new, empty Vec
- Vec::with_capacity(100) — Creates a new, empty Vec with capacity 100
- HashMap::new() — Creates a new, empty HashMap
- BufReader::new(inner) — Creates a new BufReader around some underlying type that implements the Read trait