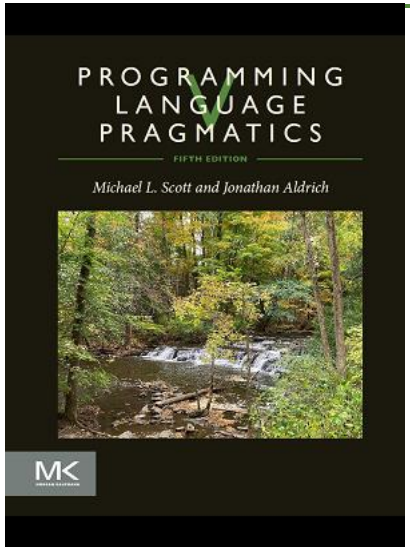
Ownership in Rust

This material is based heavily on the Rust book, as adapted by Will Crichton et al.

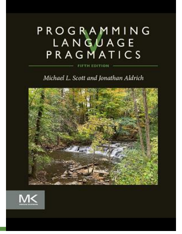


Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich

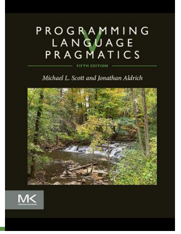
Copyright © 2024 by Jonathan Aldrich and Michael L. Scott.

Limitations of garbage collection



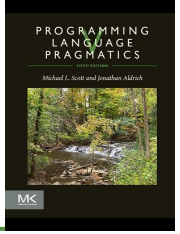
- Garbage collection (GC) provides safe, convenient memory management
  - Ideal for applications programming
- But GC can be limiting for systems programming
  - May not be willing to pay GC’s time and space costs
  - May need control of memory layout and use
  - GC failure modes can be problematic
    - Under GC, servers slow down when running out of memory
    - Crashing with “out of memory” may be better—can just restart the server
- Manual memory allocation may be needed in these conditions
  - But in most languages, this sacrifices safety

Rust provides safe manual memory management



- Rust is a systems-oriented programming language
  - manual memory management
  - concurrency and parallelism
- Rust's *ownership type system* ensures safety
  - from memory leaks and dangling references (today)
  - from data races in concurrent programs (later)

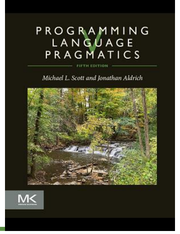
Safety in Rust



- Safety in Rust means a lack of undefined behavior
- Example of undefined behavior (from the Rust book):

```
fn read(y: bool) {
    if y {
        println!("y is true!");
    }
}

fn main() {
    read(x); // oh no! x isn't defined!
    let x = true;
}
```

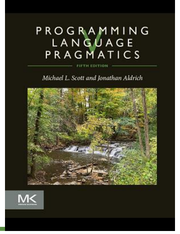
- It is undefined behavior to read a variable before it is defined
- Why is undefined behavior bad?
  - Well, it might execute just fine
  - But the program above could read garbage data, making results unpredictable
  - In general, undefined behavior can cause crashes or security vulnerabilities



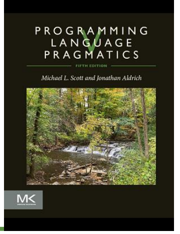
How Rust ensures safety

- Key goal of Rust: ensure program don't have undefined behavior
  - Combination of static and dynamic checks
  - Check as much as possible statically
- Bugs are still possible! But certain kinds of bugs can't happen.
- Ownership in Rust is a discipline for using memory
- Ownership prevents undefined behavior related to memory (except in *unsafe* code)
  - Reading uninitialized memory
  - Using memory after it is freed
  - Freeing memory twice
  - Memory leaks (forgetting to free memory)

Ownership in Rust

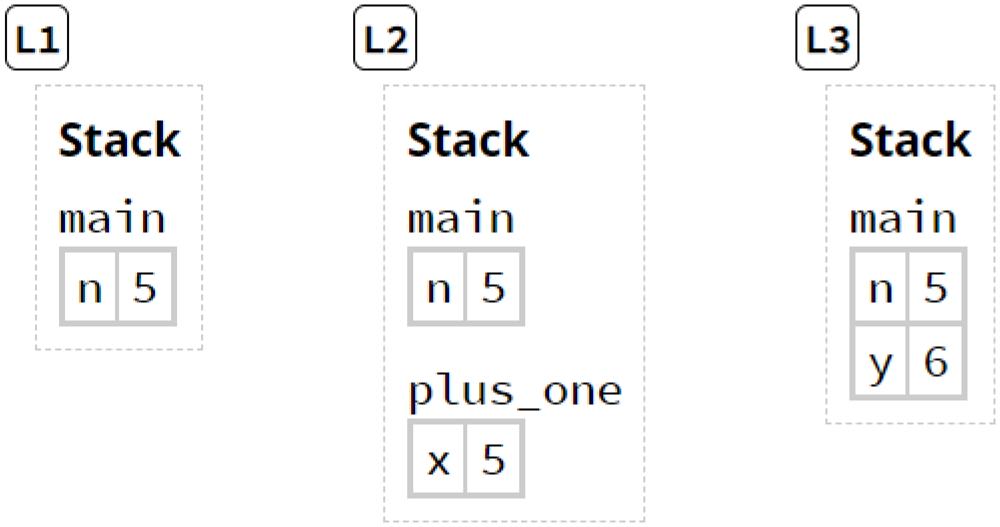


- Rust uses a value model of variables, like C/C++
  - Any typed data that is currently in use (i.e. *live*) is a value
  - A value's *lifetime* lasts from when is allocated until it is freed
- Value allocation
  - on the stack if lifetime matches a function call and size is known
  - on the heap otherwise
- Invariant: every value has exactly one *owner*
  - A variable
  - Some other value that is (or contains) a smart pointer
- Values on the heap are reclaimed when they lose their owner
  - e.g. because their owning smart pointer goes out of scope, or is reassigned



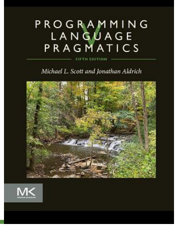
Conceptual model of variables of primitive types on the stack

```
fn main() {
    let n = 5; L1
    let y = plus_one(n); L3
    println!("The value of y is: {y}");
}

fn plus_one(x: i32) -> i32 {
    L2 x + 1
}
```



example from the Rust book (Brown version)

Assigning one variable to another copies the data if the type implements the Copy trait (includes `i32`)




```
let a = 5; L1
let mut b = a; L2
b += 1; L3
```

L1

Stack

main

a	5

L2

Stack

main

a	5
b	5

L3

Stack

main

a	5
b	6

But this is very expensive if the data is large →

```
let a = [0; 1_000_000]; L1
let b = a; L2
```

L1

Stack

main

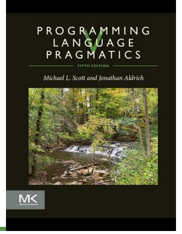
a	0	0	0	0	0	0	0	0	0	0	0	...	0

L2

Stack

main

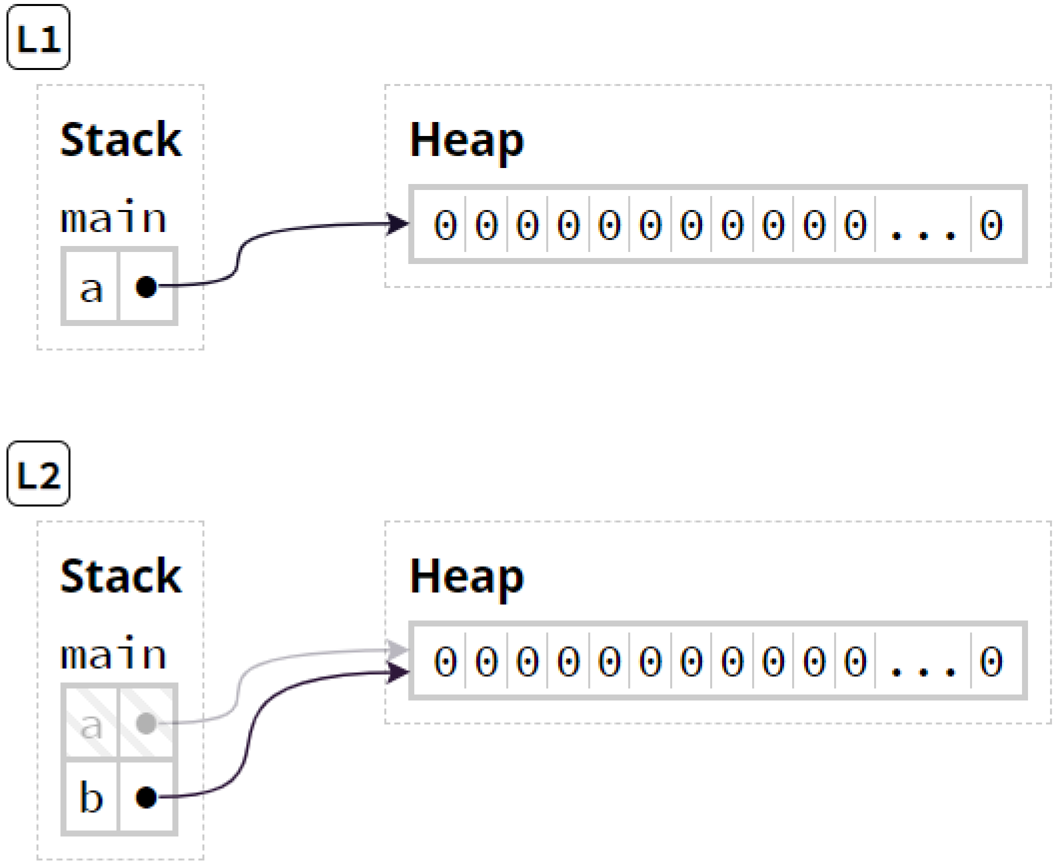
a	0	0	0	0	0	0	0	0	0	0	0	...	0
b	0	0	0	0	0	0	0	0	0	0	0	...	0



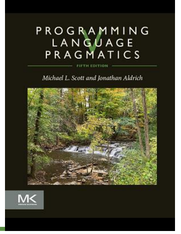
Boxes allocate memory in the heap

- Now a is of Box (pointer) type
- Assigning b to the value of a *moves* the pointer, because Box does not implement Copy
- We say that a *owns* the data before the move, and b owns it afterward
- We cannot use a after the move

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```



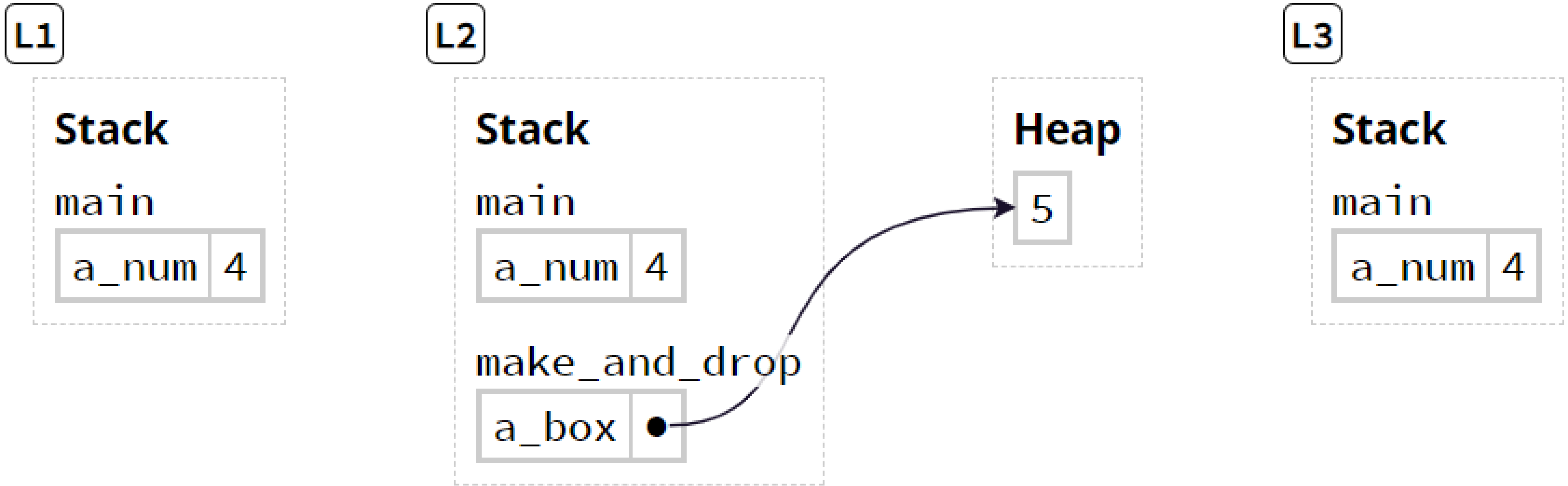
Owners deallocate boxes



If a variable owns a box, when Rust deallocates the variable's frame, then Rust deallocates ("drops") the box's heap memory.

The box holding 5 is deallocated at the end of `make_and_drop`

```
fn main() {
    let a_num = 4; L1
    make_and_drop(); L3
}

fn make_and_drop() {
    let a_box = Box::new(5); L2
}
```

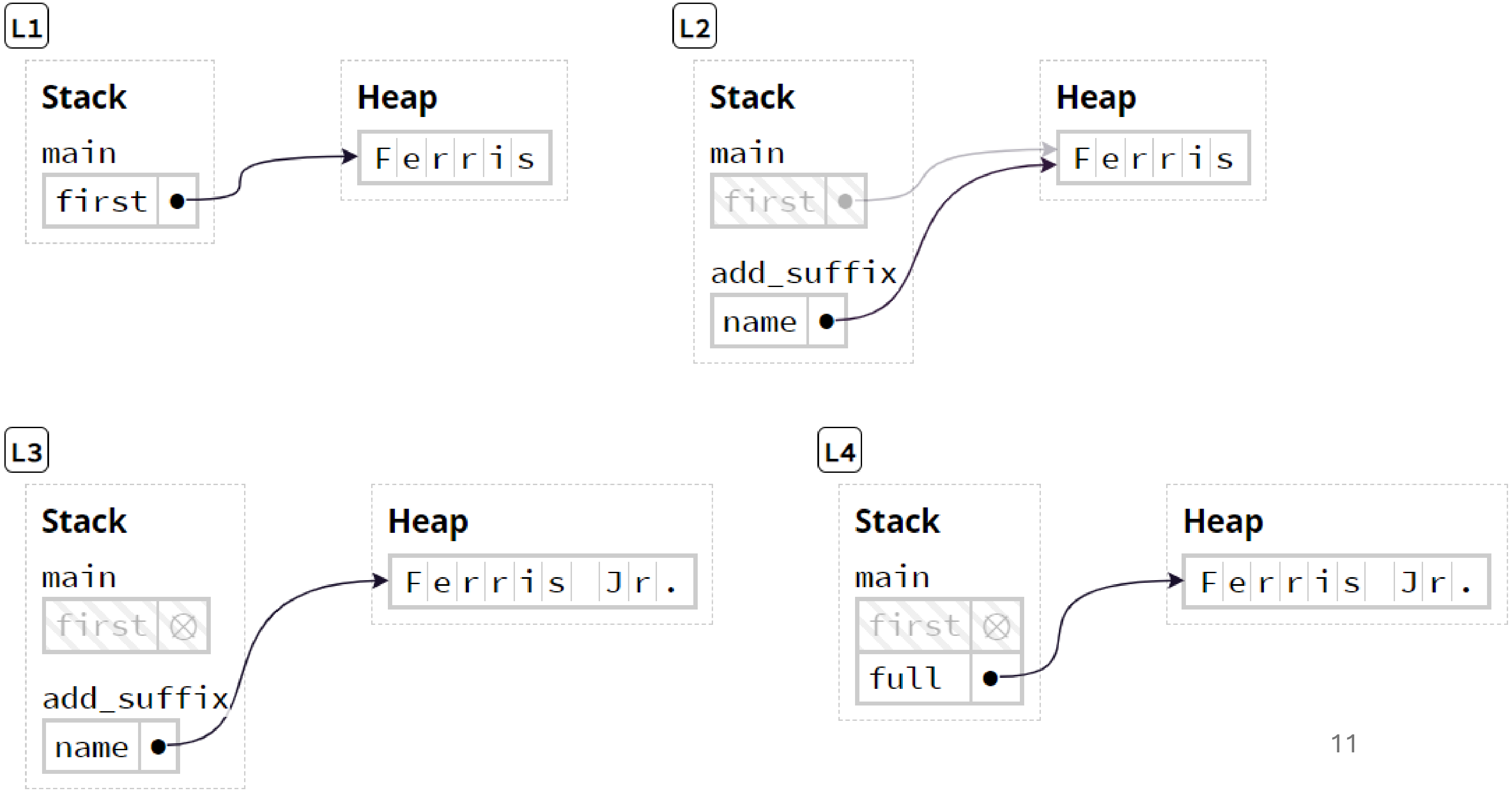


Ownership

Here's an example involving string manipulation

Note that it would be an error to use `first` after the pointer is moved in the call to `from`

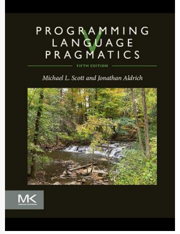
```
fn main() {
    let first = String::from("Ferris"); L1
    let full = add_suffix(first); L4
    println!("{full}");
}

fn add_suffix(mut name: String) -> String {
    L2 name.push_str(" Jr."); L3
    name
}
```

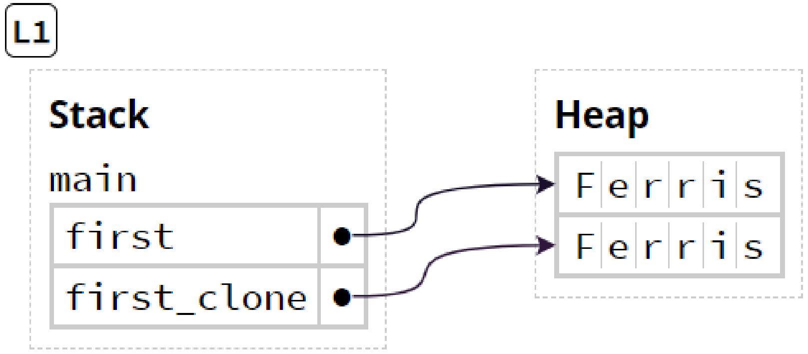


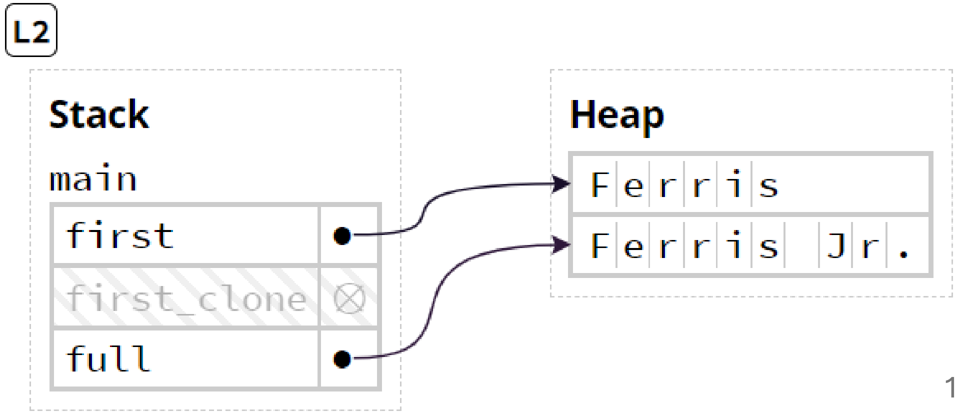
Cloning

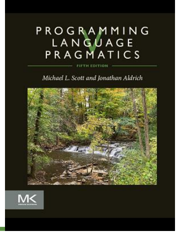
- If we want to continue to use the `first` string, we can *clone* it before moving the pointer
- The `clone` method makes a *deep copy* of the string (the data on the heap is copied, not just the pointer)

```
fn main() {
    let first = String::from("Ferris");
    let first_clone = first.clone(); L1
    let full = add_suffix(first_clone); L2
    println!("{full}, originally {first}");
}

fn add_suffix(mut name: String) -> String {
    name.push_str(" Jr.");
    name
}
```







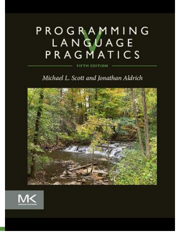


Ownership quiz

- Does this program compile? Why or why not?
- If it compiles, what is the result when it runs?

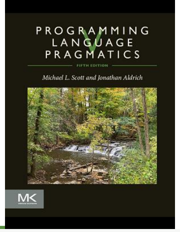
```
fn main() {
    let s = String::from("hello");
    let s2;
    let b = false;
    if b {
        s2 = s;
    }
    println!("{}", s);
}
```

Ownership quiz (SOLUTION)



- Does this program compile? Why or why not?
- Answer: no, it does not compile, because `s` might be moved to `s2` inside the `if` statement, so `s` cannot be used in the `println!` call.
- Rust doesn't try to figure out whether if statements will execute (that's undecidable in general)

```
fn main() {
    let s = String::from("hello");
    let s2;
    let b = false;
    if b {
        s2 = s;
    }
    println!("{}", s);
}
```

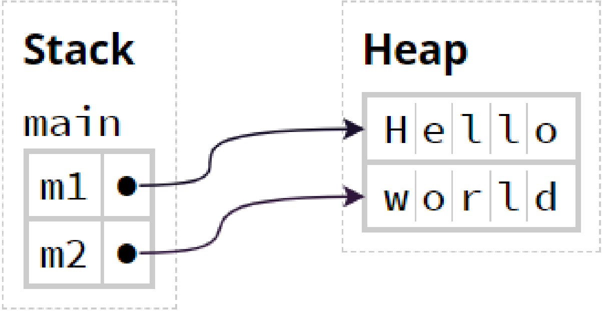


Using pointers after passing them to a function

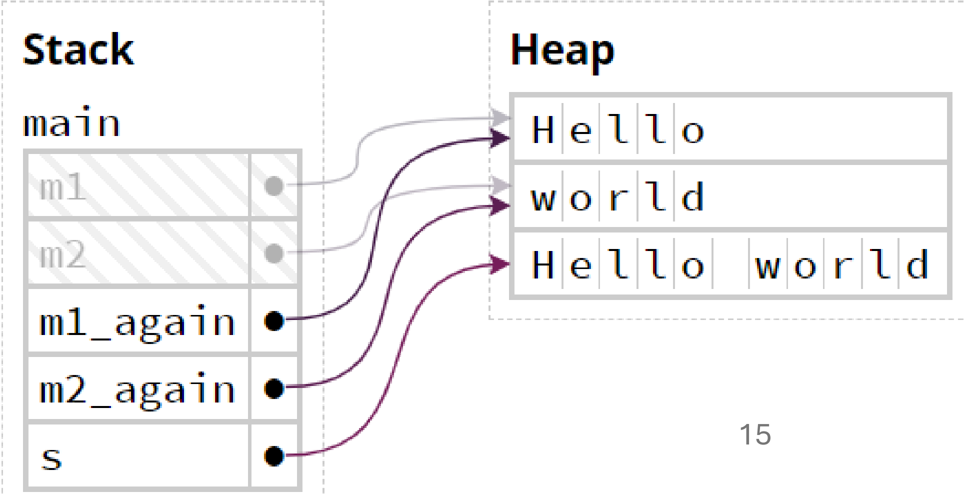
Moving owned pointers can be inconvenient

```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    let (m1_again, m2_again) = greet(m1, m2);
    let s = format!("{} {}", m1_again, m2_again); L2
}

fn greet(g1: String, g2: String) -> (String, String) {
    println!("{} {}!", g1, g2);
    (g1, g2)
}
```

L1



L2

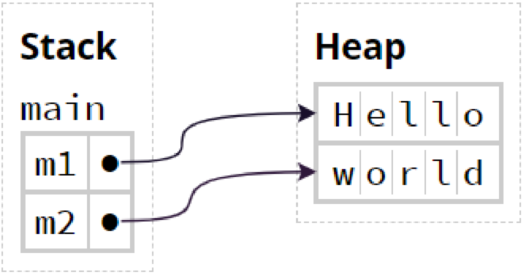


References

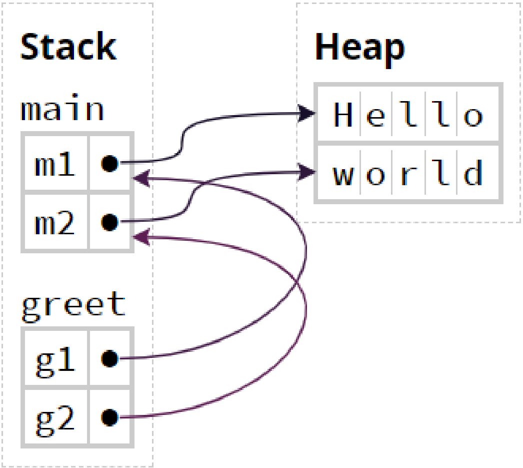
- A *reference* is a non-owning pointer
- The expression `&m1` *borrows* `m1`
- `g1` and `g2` are not deallocated at the end of `greet`, because they are not owned

```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    greet(&m1, &m2); L3  // note the ampersands
    let s = format!("{} {}", m1, m2);
}

fn greet(g1: &String, g2: &String) { // note the ampersands
    L2 println!("{} {}!", g1, g2);
}
```

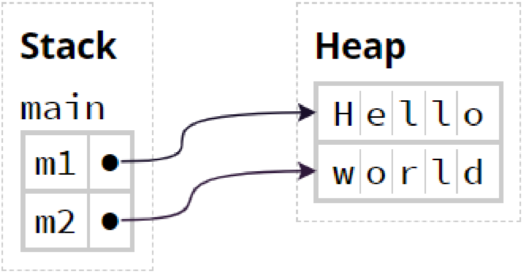
L1

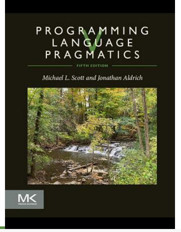


L2



L3



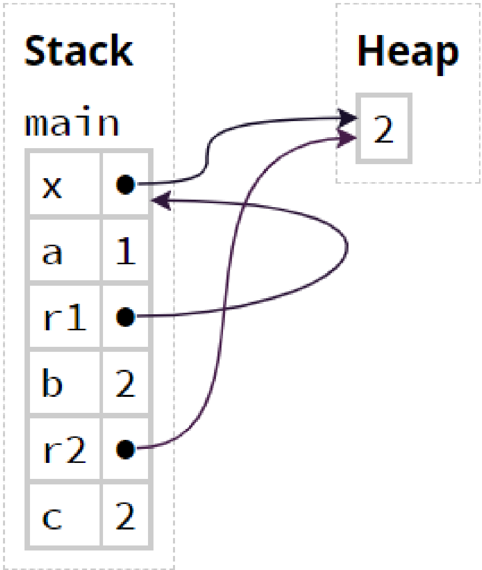


Dereferencing pointers

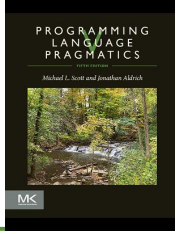
The * operator is used to access the data a pointer refers to

```
let mut x: Box<i32> = Box::new(1);
let a: i32 = *x;           // *x reads the heap value, so a = 1
*x += 1;                   // *x on the left-side modifies the heap value,
                           //     so x points to the value 2

let r1: &Box<i32> = &x;    // r1 points to x on the stack
let b: i32 = **r1;         // two dereferences get us to the heap value

let r2: &i32 = &*x;        // r2 points to the heap value directly
let c: i32 = *r2; L1       // so only one dereference is needed to read it
```

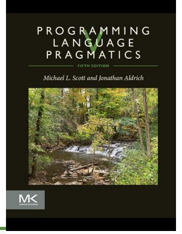
L1



Rust inserts (de)references automatically when the . operator is used



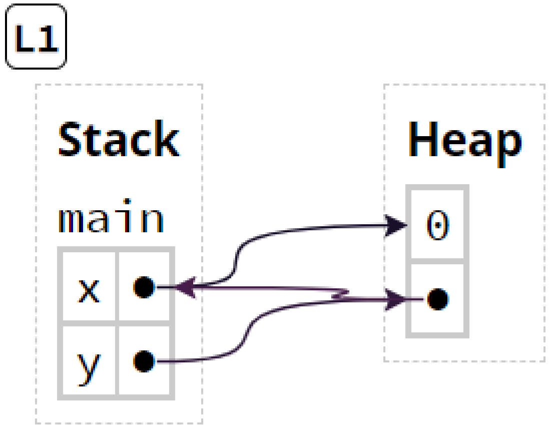

```
let x: Box<i32> = Box::new(-1);
let x_abs1 = i32::abs(*x); // explicit dereference
let x_abs2 = x.abs();      // implicit dereference
assert_eq!(x_abs1, x_abs2);

let r: &Box<i32> = &x;
let r_abs1 = i32::abs(**r); // explicit dereference (twice)
let r_abs2 = r.abs();       // implicit dereference (twice)
assert_eq!(r_abs1, r_abs2);

let s = String::from("Hello");
let s_len1 = str::len(&s); // explicit reference
let s_len2 = s.len();      // implicit reference
assert_eq!(s_len1, s_len2);
```



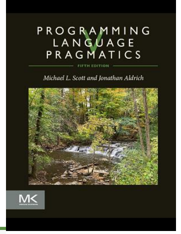
(De)referencing quiz

- Consider the following program, showing the state of memory after the last line:

```
let x = Box::new(0);
let y = Box::new(&x); L1
```

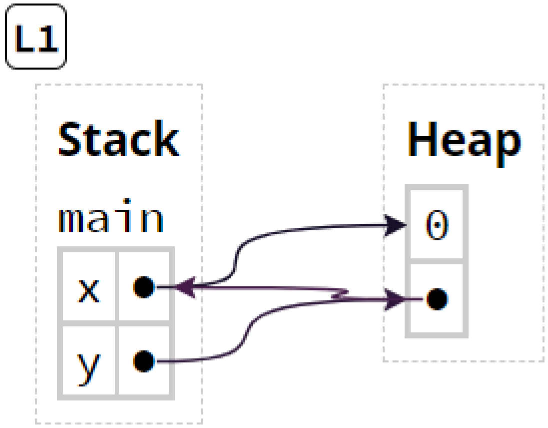


- If you wanted to copy out the number 0 through y, how many dereferences would you need to use?
  - For example, if the correct expression is *y, then the answer is 1.

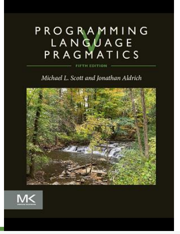


(De)referencing quiz (SOLUTION)

```
let x = Box::new(0);
let y = Box::new(&x); L1
```



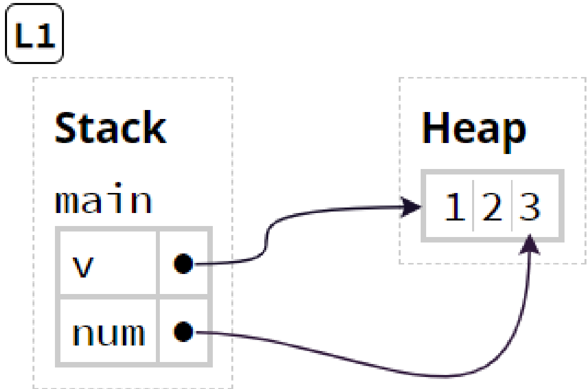
- Consider the following program, showing the state of memory after the last line:
- If you wanted to copy out the number 0 through y, how many dereferences would you need to use?
  - For example, if the correct expression is *y, then the answer is 1.
- Answer: 3 (***y)
  - One dereference for each pointer in the diagram
  - Also: one for each new, one for each &

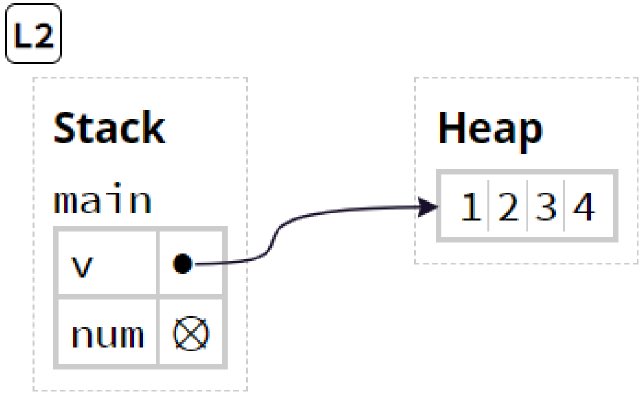


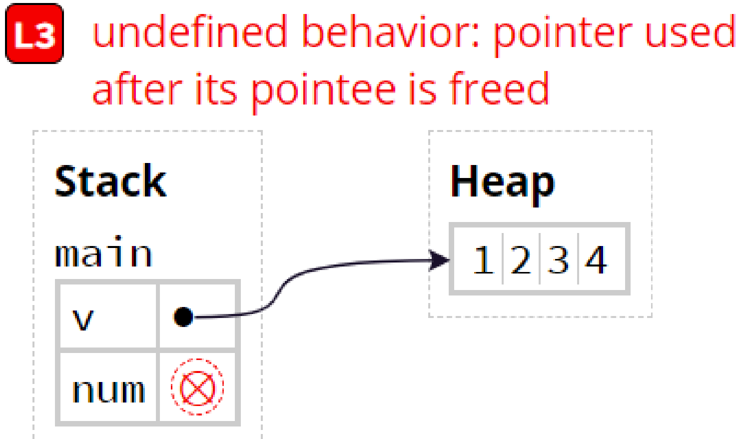
Rust prohibits simultaneous aliasing and mutation

- At L2, the alias num points to v[2]
- We write to v at L2 and read from num at L3
- This is a problem because the Vec's memory is re-allocated at L2, so the pointer used at L3 points to deallocated memory. Undefined behavior!

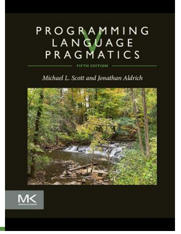
```
let mut v: Vec<i32> = vec![1, 2, 3];
let num: &i32 = &v[2]; L1
v.push(4); L2
println!("Third element is {}", *num); L3
```





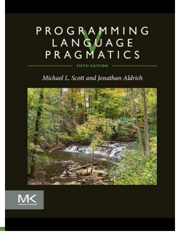


"Puzzled Ferris" means this code does not compile



Rust's ***borrow checker*** ensures reference safety

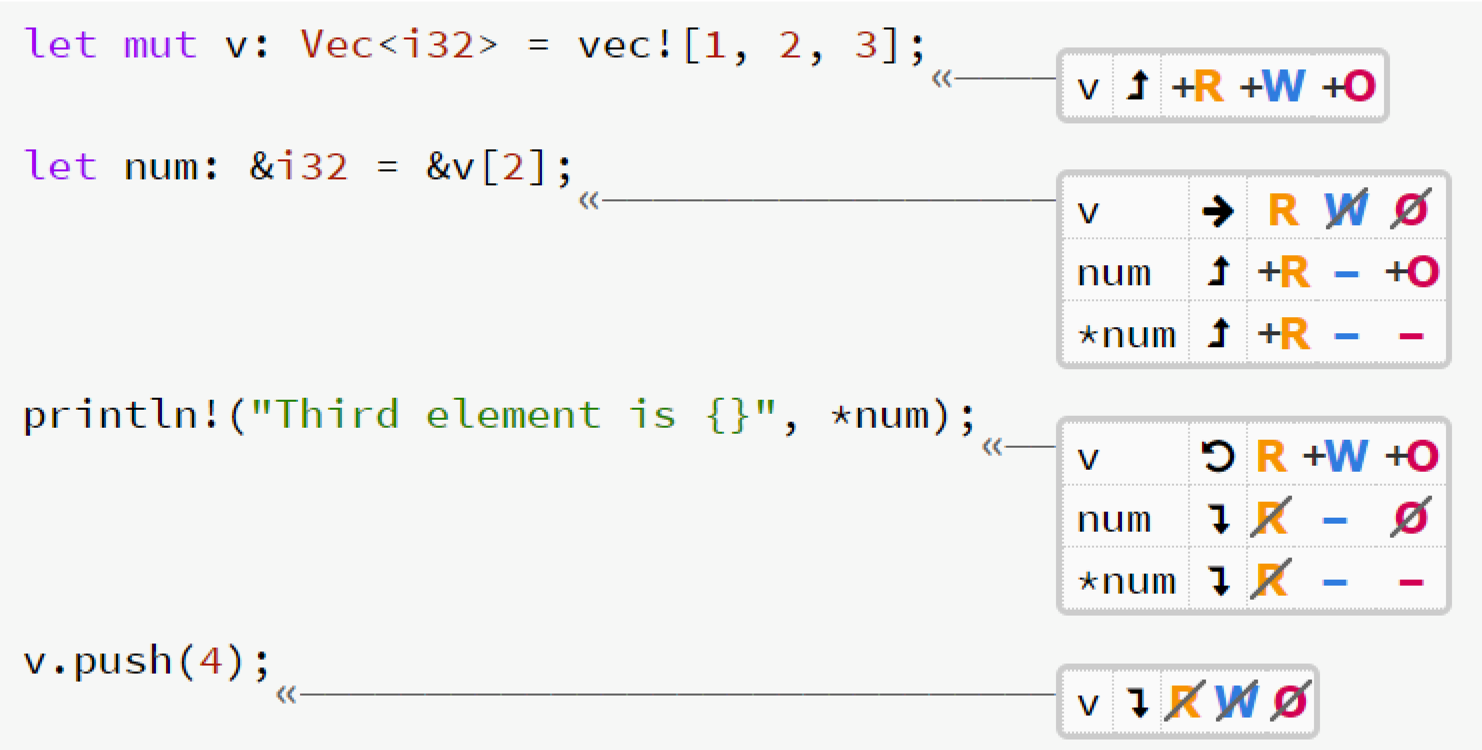
- Ensures that data is never aliased and mutated at the same time
- Tracks the permissions associated with each variable:
  - **Read** (**R**): data can be copied to another location.
  - **Write** (**W**): data can be mutated in-place (`let mut` vars)
    - Invariant: there can be at most one **W** permission to a piece of data at any time
  - **Own** (**O**): data can be moved or dropped.
- Creating a reference can temporarily remove these permissions



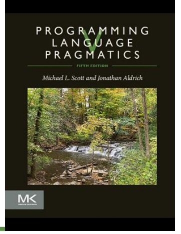
Example: how borrow checking works

Notes:

- Different permissions for num and *num
  - manipulating the reference vs. accessing the data
- Permissions are defined on *paths*
  - num, *num, `v[2]`, `a.field`, `*((*a)[0].1)`
- Permissions are lost when a mutually exclusive permission must be used
  - e.g. **W** on v is needed at `v.push(4)` so **R** on num is lost



A borrow checking error




```
let mut v: Vec<i32> = vec![1, 2, 3];
```

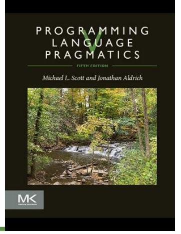
v	↥	+R	+W	+O

```
let num: &i32 = &R v[2];
```

v	→	R	~~W~~	~~O~~
num	↥	+R	–	+O
*num	↥	+R	–	–

```
v(R W).push(4);
println!("Third element is {}", *num);
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> test.rs:4:1
  |
3 | let num: &i32 = &v[2];
  |                 - immutable borrow occurs here
4 | v.push(4);
  | ^^^^^^^^^ mutable borrow occurs here
5 | println!("Third element is {}", *num);
  |                                 ---- immutable borrow later used here
```



We can also borrow mutably with &mut

```
let mut v: Vec<i32> = vec![1, 2, 3];

let num: &mut i32 = &mut v[2];

*num += 1;
println!("Third element is {}", *num);

println!("Vector is now {:?}", v);
```

After `let mut v: Vec<i32> = vec![1, 2, 3];`

v	↥	+R	+W	+O

After `let num: &mut i32 = &mut v[2];`

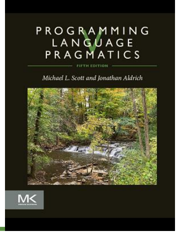
v	→	~~R~~	~~W~~	~~O~~
num	↥	+R	–	+O
*num	↥	+R	+W	–

After `println!("Third element is {}", *num);`

v	↻	+R	+W	+O
num	↧	~~R~~	–	~~O~~
*num	↧	~~R~~	~~W~~	–

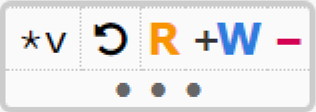
After `println!("Vector is now {:?}", v);`

v	↧	~~R~~	~~W~~	~~O~~



Permissions are returned when a reference's lifetime ends

```
fn ascii_capitalize(v: &mut Vec<char>) {
    let c = &v[0];

    if c.is_ascii_lowercase() {
        let up = c.to_ascii_uppercase();

        v[0] = up;
    } else {

        println!("Already capitalized: {:?}", v);
    }
}
```



- Control flow can make this interesting!

Borrowing quiz

In the example, explain why `strs` loses and regains write (W) permissions

```rust
fn get_first(v: &Vec<String>) -> &str {
    &v[0]
}

fn main() {
    let mut strs = vec![
        String::from("A"), String::from("B")
    ];

    let first = get_first(&strs);

    if first.len() > 0 {
        strs.push(String::from("C"));
    }
}
```

After `fn get_first(v: &Vec<String>) -> &str {`:

v	↑	+R	–	+O
*v	↑	+R	–	–

After `&v[0]`:

v	↓	~~R~~	–	~~O~~
*v	↓	~~R~~	–	–

After `let mut strs = vec![...];`:

strs	↑	+R	+W	+O

After `let first = get_first(&strs);`:

strs	→	R	~~W~~	~~O~~
first	↑	+R	–	+O
*first	↑	+R	–	–

After `if first.len() > 0 {`:

strs	↻	R	+W	+O
first	↓	~~R~~	–	~~O~~
*first	↓	~~R~~	–	–

After `strs.push(String::from("C"));`:

strs	↓	~~R~~	~~W~~	~~O~~

After `}`:

strs	↓	~~R~~	~~W~~	~~O~~

Borrowing quiz

In the example, explain why `strs` loses and regains write (W) permissions

ANSWER: `get_first` returns an immutable reference to data within `strs`, so `strs` is not writable while `first` is live

```
fn get_first(v: &Vec<String>) -> &str {
    &v[0]
}

fn main() {
    let mut strs = vec![
        String::from("A"), String::from("B")
    ];
    let first = get_first(&strs);
    if first.len() > 0 {
        strs.push(String::from("C"));
    }
}
```

After `fn get_first(v: &Vec<String>) -> &str {`

v	↑	+R	–	+O
*v	↑	+R	–	–

After `&v[0]`

v	↓	~~R~~	–	~~O~~
*v	↓	~~R~~	–	–

After `];`

strs	↑	+R	+W	+O

After `let first = get_first(&strs);`

strs	→	R	~~W~~	~~O~~
first	↑	+R	–	+O
*first	↑	+R	–	–

After `if first.len() > 0 {`

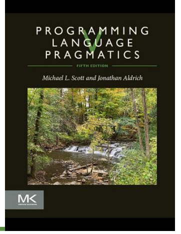
strs	↻	R	+W	+O
first	↓	~~R~~	–	~~O~~
*first	↓	~~R~~	–	–

After `strs.push(String::from("C"));`

strs	↓	~~R~~	~~W~~	~~O~~

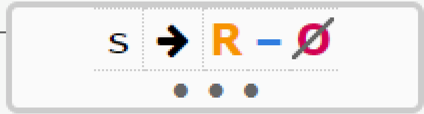
After `}`

strs	↓	~~R~~	~~W~~	~~O~~

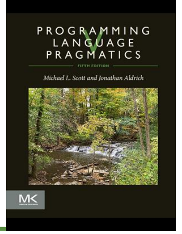


Invariant: data must outlive its references

```
let s = String::from("Hello world");

let s_ref = &s;


drop(s);
println!("{}", s_ref);
```



- The drop function explicitly releases a data structure
  - Any pointers in the data structure will be freed
- But drop requires an ownership (O) permission and we do not have that for s while s_ref is live



Fixing borrow checking errors

- The following code has a borrow error:

```
/// Returns a person's name with "Ph.D." added as a title
fn award_phd(name: &String) -> String {
    let mut name = *name;
    name.push_str(", Ph.D.");
    name
}
```

- What's the best fix?

A

```
fn award_phd(name: &String) -> String {
    let mut name = name.clone();
    name.push_str(", Ph.D.");
    name
}
```

B

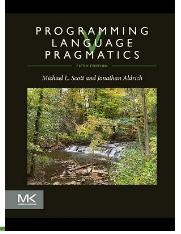
```
fn award_phd(mut name: String) -> String {
    name.push_str(", Ph.D.");
    name
}
```

C

```
fn award_phd(name: &mut String) {
    name.push_str(", Ph.D.");
}
```

D

```
fn award_phd(name: &String) -> String {
    let mut name = &*name;
    name.push_str(", Ph.D.");
    name
}
```

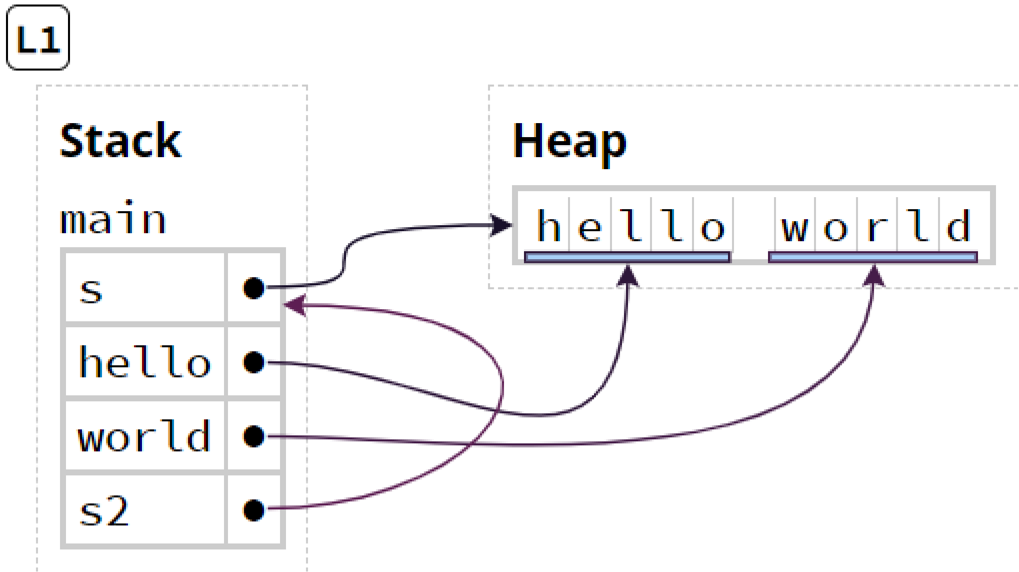


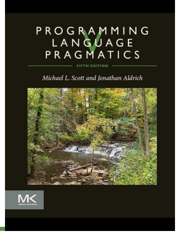
String slices

- A string slice of type `&str` points to a range of characters in a string
  - `&str` is the type of string literals in Rust!
- A slice knows its length—access beyond the length is a run time error
- Slices are references, so taking a slice changes the permission to the underlying data
  - If s were `mut` then we couldn't mutate it while `hello` is live
- You can also take slides of arrays

```
let a = [1, 2, 3, 4, 5];
let slice : &[i32] = &a[1..3];
```

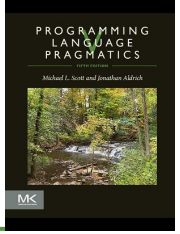
```
let s = String::from("hello world");

let hello: &str = &s[0..5];
let world: &str = &s[6..11];
let s2: &String = &s;  L1
```





Sometimes Rust can’t tell the lifetime of a reference

```rust
// does longest return x or y?
// unclear -- and it matters if they have different lifetimes
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}

fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";
    let result = longest(string1.as_str(), string2);
    println!("The longest string is {result}");
}
```

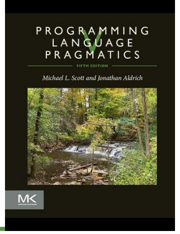


Lifetime annotations can help

```
&i32         // a reference
&'a i32      // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

- We don't need to write lifetime annotations everywhere—just when we need to compare the lifetimes of different references (e.g. in a function signature)

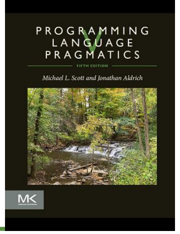
Using lifetime annotations




```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

- This signature tells Rust that for some lifetime `'a`, the arguments must live at least as long as `‘a`.
- Also, the value returned by `longest` will live at least as long as `'a`.

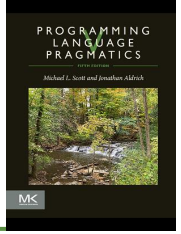
We can put lifetime annotations in structs




```rust
struct ImportantExcerpt<'a> { part: &'a str, }
fn main() {
  let novel = String::from("Call me Ishmael. Some years ago...");
  let first_sentence = novel.split('.').next().unwrap();
  let i = ImportantExcerpt { part: first_sentence, };
}
```

- The lifetime parameter of `ImportantExcerpt` tracks how long the `part` reference lives.
- Rust checks that `i` isn't used after `novel` goes out of scope

Lifetime annotations can often be elided



- If you don’t provide them, Rust acts as if they were specified according to the following rules
  - Every lifetime in the input type gets its own lifetime parameter

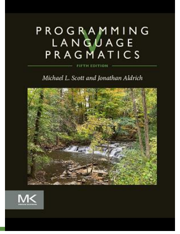
```
fn foo(x: &i32, y: &i32) → foo<'a, 'b>(x: &'a i32, y: &'b i32)
```

  - If there is exactly one lifetime parameter, that lifetime is assigned to all output lifetime parameters

```
fn foo(x: &i32) -> &i32 → fn foo<'a>(x: &'a i32) -> &'a i32
```

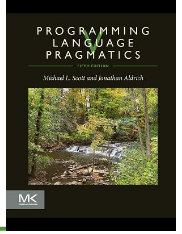
  - [Methods only]: If there are multiple input lifetime parameters, but one of them is `&self` or `&mut self`, that lifetime is used for all output lifetime parameters

The `'static` lifetime



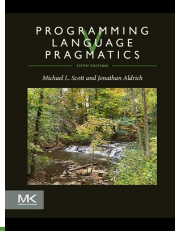
- The `'static` lifetime is for things that live for the entire execution of the program
  - Example: string literals
- Only use it if you know the underlying data lives indefinitely!

More about `Drop`



- When a value goes out of scope, we say it is *dropped*
  - If the value's type implements the `Drop` trait, the `drop` function is called
- Every heap allocated value is owned by exactly one box
  - Initially the box that allocates it
  - Ownership is transferred if one variable is assigned to another
    - the old box variable cannot be used
- Every box value is dropped exactly once
  - its drop function deallocates the heap data it owns
- Because of unique ownership
  - there cannot be any memory leaks
  - there cannot be any dangling pointers

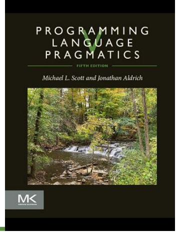
We can define binary trees with Box



- We use a struct to define a tree node with a key and left and right subtrees
- Rust does not have null pointers, so we use *options*
  - An `Option` is either `Some` (storing a value) or `None` (no value, like `null`)
  - In this case, the option holds a `Box<TNode>` since the subtree is allocated separately

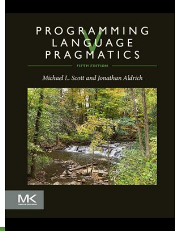
```
struct TNode {
  key: i32,
  left: Tree,
  right: Tree,
}
type Tree = Option<Box<TNode>>;
```

Now we define `insert`




```
struct TNode {
  key: i32,
  left: Tree,
  right: Tree,
}
type Tree = Option<Box<TNode>>;
fn insert(tree: &mut Tree, key: i32) {
  match tree {
    None => {
      *tree = Some(Box::new(TNode { key, left: None, right: None }))
    },
    Some(t) => {
      if key < t.key {
        insert(&mut t.left, key)
      } else {
        insert(&mut t.right, key)
      }
    },
  }
}
```

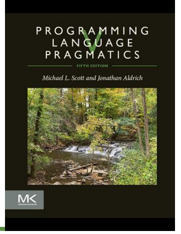
```
fn main() {
  let mut tree = None;
  insert(&mut tree, 5);
  insert(&mut tree, 3);
  insert(&mut tree, 4);
  insert(&mut tree, 10);
  // the whole tree is deallocated here
}
```



The Rc library allows sharing

- A value in the heap can only be pointed to by one box at a time
  - this is what enables us to avoid leaks and dangling references
  - but we can only create data structures like trees that have no sharing
- To create graphs, we need sharing in the heap
- The Rc library provides reference-counted shared pointers
  - The reference count is incremented and decremented when Rc pointers are created and dropped
  - When an Rc pointer is dropped and the reference count goes down to zero, the heap value is dropped and the storage is reclaimed

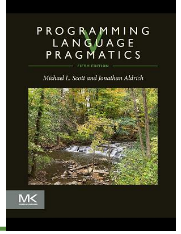
We can define graphs with Rc



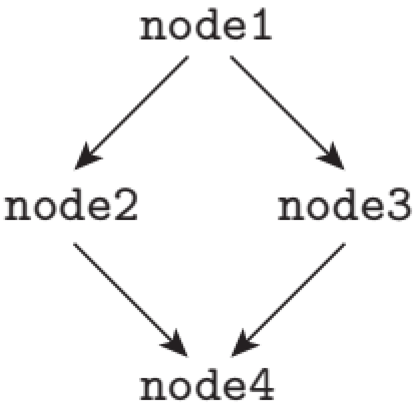

```
struct GNode {
  key: i32,
  edges: Vec<Edge>,
}

type Edge = Rc<RefCell<GNode>>;
```

- `Vec` is an expandable array type with bound checking
- Recall that Rust doesn't allow shared values to be mutable
  - But we want to change graph nodes, e.g. while building the graph
- `RefCell` implements unique, mutable borrowing
  - The contents can be borrowed temporarily
  - A run-time check verifies there are no other borrowers

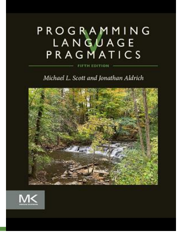
Now we can create a graph




```
struct GNode {
  key: i32,
  edges: Vec<Edge>,
}

type Edge = Rc<RefCell<GNode>>;

fn new_node(key: i32) -> Edge {
  Rc::new(RefCell::new(
    GNode { key,
            edges : Vec::new()
          }))
}
```

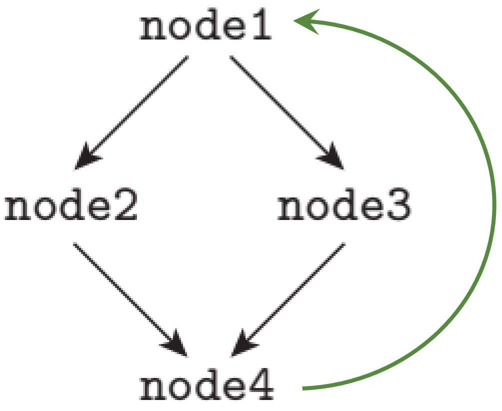



```
fn main() {
  let node1: Edge = new_node(1);
  let node2: Edge = new_node(2);
  let node3: Edge = new_node(3);
  let node4: Edge = new_node(4);
  node1.borrow_mut().edges.push(node2.clone());
  node1.borrow_mut().edges.push(node3.clone());
  node2.borrow_mut().edges.push(node4.clone());
  node3.borrow_mut().edges.push(node4.clone());
}
```

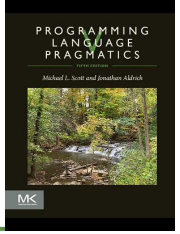


Cyclic graphs won’t be collected

```
struct GNode {
  key: i32,
  edges: Vec<Edge>,
}

type Edge = Rc<RefCell<GNode>>;

fn new_node(key: i32) -> Edge {
  Rc::new(RefCell::new(
    GNode { key,
            edges : Vec::new()
          }))
}
```




```
fn main() {
  let node1: Edge = new_node(1);
  let node2: Edge = new_node(2);
  let node3: Edge = new_node(3);
  let node4: Edge = new_node(4);
  node1.borrow_mut().edges.push(node2.clone());
  node1.borrow_mut().edges.push(node3.clone());
  node2.borrow_mut().edges.push(node4.clone());
  node3.borrow_mut().edges.push(node4.clone());
  node4.borrow_mut().edges.push(node1.clone());
}
```



Rust provides safe manual memory management

- It does so by enforcing 3 invariants
  - every value has exactly one *owner*
  - there can be at most one **W** permission to a piece of data at any time
  - data must outlive its references