Ownership and the String Type

Performant Software Systems with Rust — Lecture 4

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

The Dreaded error[E0382]

```
error[E0382]: borrow of moved value
```

Let’s start from the very beginning

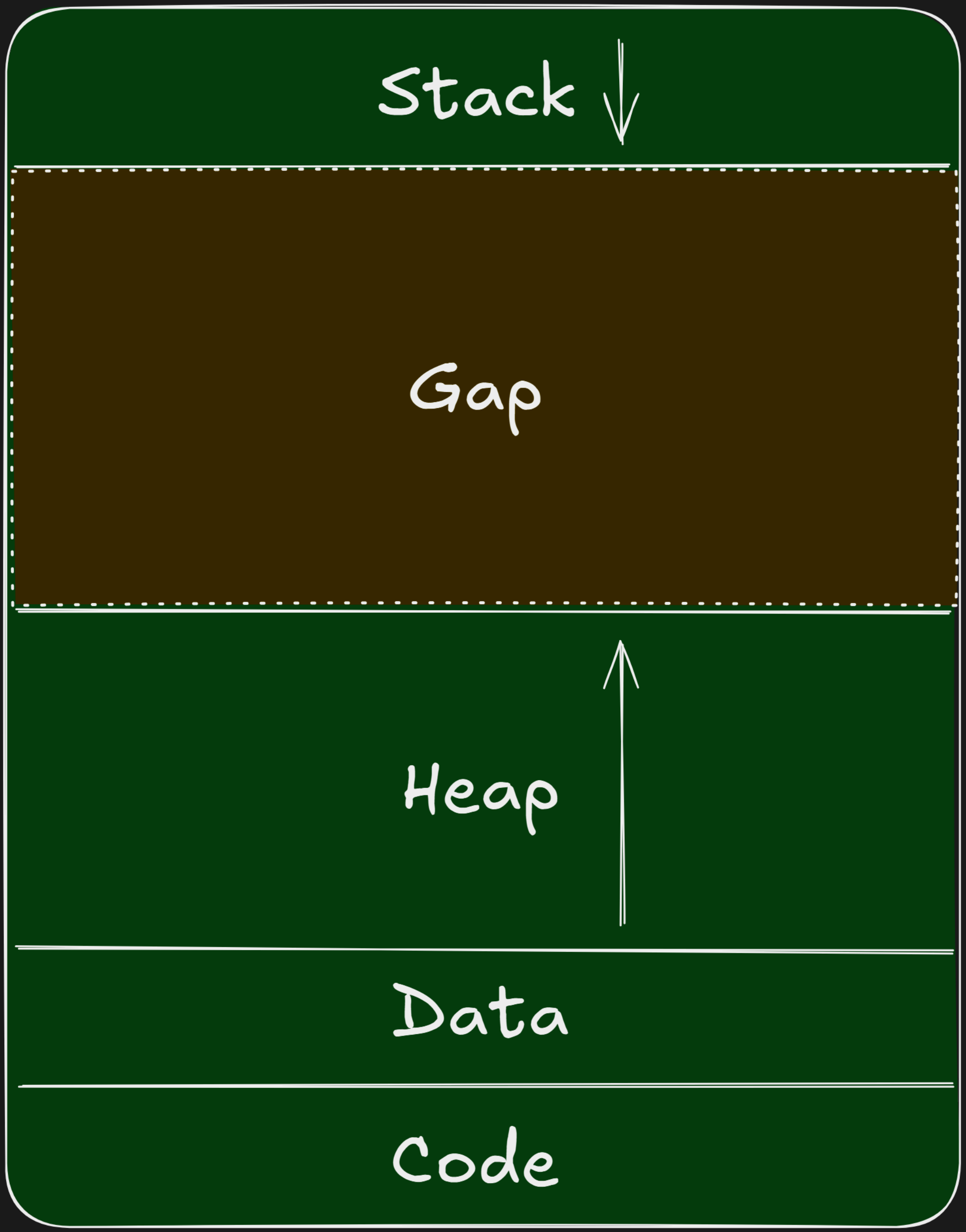
The Stack

- Variables and parameters on the stack are **local** to the function scope they are declared in
- Function calls proceed in a **last-in, first-out** order
  - When we call a function, we **push** its stack frame onto the stack
  - When we return from it, we **pop** its stack frame off
- The stack is **fast** — we only need to move the **stack pointer**

The Heap

- The heap is where **dynamic memory** is allocated — we need to explicitly request and release memory
  - A request (with `malloc()` in C or `new` in C++) returns a pointer to allocated memory on the heap
  - This pointer can be stored in a variable on the stack
  - A release (with `free()` in C or `delete` in C++) deallocates the memory at a pointer
- Slower than the stack because we need to **search** for a block of memory that is large enough

The stack grows downwards to lower memory addresses, and the heap grows upwards to higher memory addresses


Stack
Gap
Heap
Data
Code

Ownership Rules

- Each value in Rust has a variable that is its **owner**
- There can only be **one** owner at a time
- When the owner goes out of scope, the value will be **dropped**

Revisiting Variable Scope

```rust
{   // s is not valid here, it’s not yet declared
    let s = "hello";
    // s comes into scope, it is valid from this point forward
    println!("{s}");
    // s remains valid until it goes out of scope
}   // this scope is now over, s is no longer valid
```

```
hello
```

Fun fact: This runtime output is produced by running the code directly when compiling this presentation, with a Jupyter Kernel for Rust.

To continue to study **ownership**, we need a data type that is stored on the heap

The String Type

- We've seen string literals before, but they are **immutable**
- What if we want to store a string that is **unknown** at compile time?

```rust
// a mutable String allocated on the heap
let mut s = String::from("hello");
s.push_str(", world!"); // appends a literal to an existing String
println!("{s}");
```

```
hello, world!
```

Allocating and Releasing Strings on the Heap

- We allocated memory for the String `s` manually using `String::from`
- When the variable `s` goes out of scope, the memory is automatically released

Rust vs. C/C++

- In Rust, there is no need to manually call `free` or `delete` explicitly
  - Wastes memory if we do it too late (or forget to do it — **memory leaks**)
  - Invalid memory access if we do it too early
  - Double free if we do it twice, corrupting the memory allocator

Rust vs. Go/JavaScript

- In Rust, there is no need to use a **garbage collector** to clean up unused memory
- Allocated memory is **automatically** released once the variable that owns it goes out of scope

```rust
{   // s is not valid here, it’s not yet declared
    let mut s = String::from("hello");
    // s comes into scope, it is valid from this point forward
    s.push_str(", world!");
    // s remains valid until it goes out of scope
    println!("{s}");
}   // this scope is now over, s is no longer valid and is `dropped'
```

Resource Acquisition Is Initialization (RAII) in C++

- Rust's idea is not new — it is inspired by RAII in C++
- With RAII, resources are acquired in the constructor and released in the destructor of an object
- This is why C++ has **smart pointers** like `std::unique_ptr` and `std::shared_ptr`
- The main difference is
  - Rust doesn't have `new` and `delete`
  - Rust enforces the ownership rules at compile time

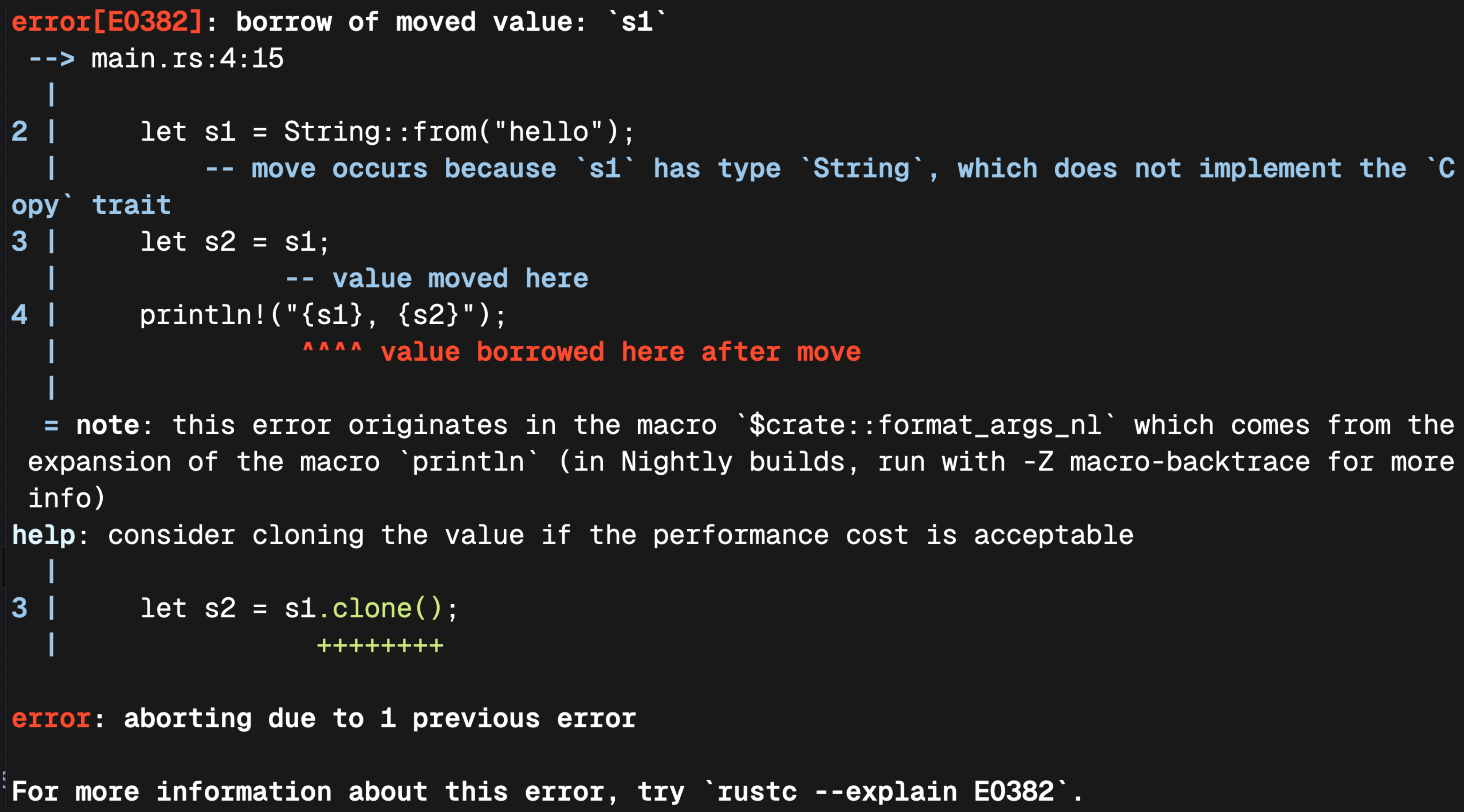
Difference Between Scalar Types and Strings

```rust
let x = 5; // bind the value 5 to x
let y = x; // make a copy of the value in x and bind it to y
println!("{x}, {y}");
```

```
5, 5
```

```rust
let s1 = String::from("hello");
let s2 = s1;
println!("{s1}, {s2}");
```

?

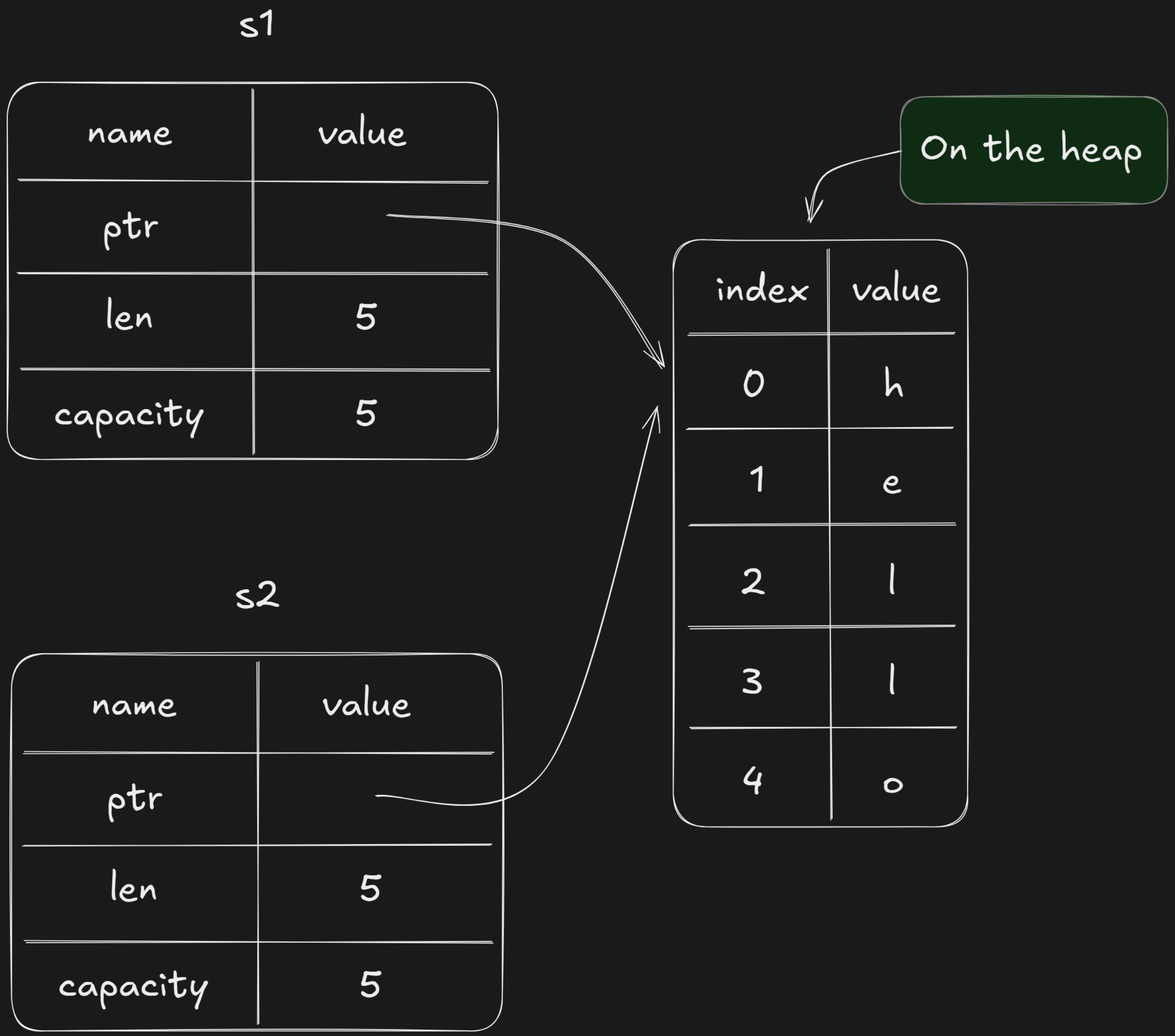


```
error[E0382]: borrow of moved value: `s1`
 --> main.rs:4:15
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `C
opy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{s1}, {s2}");
  |               ^^^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the
 expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more
 info)
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0382`.
```

What Happens When You Copy a String?



name	value
ptr	
len	5
capacity	5

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

Back to Our Example

```rust
{
    let s1 = String::from("hello");
    let s2 = s1;
    println!("{s1}, {s2}");
} // s1 and s2 go out of scope at the same time
// but should both of them be dropped? --- ``double free'' error!
```

Making a **Move**

We should make a **shallow copy** and then **invalidate** the original variable copied — which is called a **move**

```rust
{
    let s1 = String::from("hello");
    let s2 = s1; // s1 is moved into s2, and is no longer valid
    println!("{s1}, {s2}"); // error: borrow of moved value: `s1`
} // s1 and s2 go out of scope at the same time
// but s1 does not need to be freed, as it is no longer valid
```

What happens with this code?

```rust
let mut s = String::from("hello");
s = String::from("ahoy");

println!("{s}, world!");
```

Making a Clone

```rust
{
    let s1 = String::from("hello");
    let s2 = s1.clone(); // s2 has a deep copy of s1
    println!("{s1}, {s2}"); // no more errors
} // s1 and s2 go out of scope at the same time, are are both freed
```

Back to Our Example Again

```rust
let x = 5; // bind the value 5 to x
let y = x; // make a copy of the value in x and bind it to y
println!("{x}, {y}");
```

```
5, 5
```

We haven't called `clone()` — why do we make a copy of `x` here?

- `x` is an integer on the stack only, and such stack-only data is not expensive to copy
- There is no reason to invalidate `x` and move it into `y` here
- `Copy` trait is implemented for all types that are stored on the stack only, just like integers
  - If a type implements the `Copy` trait, variables that use it do not move, but rather are trivially copied
  - If a type implements the `Drop` trait, it cannot implement the `Copy` trait

Passing a Value to a Function Can Transfer Ownership

```rust
fn main() {
    let s = String::from("hello"); // s comes into scope
    takes_ownership(s); // s's value moves into the function,
    // s is no longer valid here
    let x = 5; // x comes into scope
    makes_copy(x); // x is an integer that implements the Copy trait,
                   // copies into the function and okay to use later
} // both s and x goes out of scope
// Because s's value was moved, nothing special happens
```

```rust
fn takes_ownership(s: String) {
    println!("{s}");
} // s goes out of scope and is dropped

fn makes_copy(n: i32) {
    println!("{n}");
} // n goes out of scope, and nothing special happens
```

Returning a Value from a Function Can Also Transfer Ownership

```rust
fn main() {
    let s1 = gives_ownership(); // moves its return value into s1
    let s2 = String::from("hello");
    let s3 = takes_and_gives_back(s2);
    // s2 moved into the function; its return value moved into s3
} // s1 and s3 are dropped, s2 was moved and nothing happens

fn gives_ownership() -> String {
    let s = String::from("yours");
    s // moves `s` out of the function
}

fn takes_and_gives_back(s: String) -> String {
    s // moves into the function and then out of the function
}
```

Allowing a Function to Use a Value Without Taking Ownership

```rust
fn main() {
    let s1 = String::from("hello");
    let (s1, len) = calculate_length(s1);
    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: String) -> (String, usize) {
    (s, s.len())
}
```

Any issues you can see here?

Demo: error[E0382]: borrow of moved value

Allowing a Function to Use a Value Without Taking Ownership

```rust
fn main() {
    let s1 = String::from("hello");
    let (s1, len) = calculate_length(s1);
    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len();
    (s, length)
}
```

```
The length of 'hello' is 5.
```

References and Borrowing

- **References** allow you to **borrow** a value without taking ownership of it
  - Represented by &
  - Can **derefence** them with *
  - Unlike C pointers, they can never be null or “dangling”

References and Borrowing

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1); // pass a reference to s1
    println!("The length of '{s1}' is {len}.");
}

fn calculate_length(s: &String) -> usize {
    s.len()
} // Here, s goes out of scope. But because s does not have ownership of wh
  // it refers to, the String is not dropped.
```

Trying to Modify What We Borrow

```rust
fn main() {
    let s = String::from("hello");

    change(&s);
    println!("{s}");
}

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

Demo: error[**E0596**]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference

Mutable References

- **Mutable references** allow you to change the value of a reference
  - Represented by `&mut`

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

    change(&mut s); // pass a mutable reference to s
    println!("{s}");
}

fn change(some_string: &mut String) { // take a mutable reference
    some_string.push_str(", world");
}
```

```
hello, world
```

Can we create **two** mutable references to the same value?

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

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

Demo: error[**E0499**]: cannot borrow `s` as mutable more than once at a time

Can we have one mutable reference and an immutable reference?

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

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // error[E0502]

println!("{}, {}, and {}", r1, r2, r3);
```

Demo: error[**E0502**]: cannot borrow `s` as mutable because it is also borrowed as immutable

Mutable References: House Rules

- **Cannot** borrow as a **mutable reference** more than once
- **Cannot** have a **mutable reference** while also having an **immutable reference**
- **Can** have **multiple immutable references** at the same time
- **Summary:** Can either have **one mutable reference** or **multiple immutable references**, but not both
- But why?

Data Races: The Most Elusive Bugs in Concurrent Software

- Two or more pointers access the same data at the same time
- At least one of the pointers is being used to write to the data
- There’s no mechanism being used to synchronize access to the data

Data races in Rust are **compile-time** errors

Fixing the Compile-Time Error

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

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{r1} and {r2}");
// variables r1 and r2 will not be used after this point

let r3 = &mut s; // no problem
println!("{r3}");
```

```
hello and hello
hello
()
```

Dangling References

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

fn dangle() -> &String { // dangle returns a reference to a String
    let s = String::from("hello"); // s is a new String
    &s // we return a reference to the String, s
}
// s goes out of scope, and is dropped.
```

Demo: error[**E0106**]: missing lifetime specifier

Fixing the Compile-Time Error

```rust
fn no_dangle() -> String {
    let s = String::from("hello");

    s // ownership is moved out, nothing is dropped
}
```

References: House Rules

- References must always be valid
- At any given time, you can have either one mutable reference or any number of immutable references
- Data will not go out of scope before the reference to the data does

Problem: Write a function that takes a string of words, and returns the first word it finds in the string

```rust
fn first_word(s: &String) -> ?
```

Returning an **index** of the end of the first word as an integer?

Let’s try this idea.

```rust
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes(); // convert to an array of bytes

    // .iter(): an iterator that returns each element
    // .enumerate(): returns each element as a tuple
    // (index, reference to element)
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

But there is no **guarantee** that the index returned will be **valid** in the future

```rust
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word will get the value 5

    s.clear(); // this empties s, making it equal to ""

    // `word` still has the value 5 here, but there's no more
    // string that we could meaningfully use the value 5 with.
    // `word` is now invalid.
}
```

String Slices

- `&str`: **Immutable** references to a part of a `String`
- Created using a range within square brackets
  - `[starting_index..ending_index]`

```rust
let s = String::from("hello");
let slice = &s[0..2];
let slice = &s[..2];
let slice = &s[3..];
let slice = &s[..];
```

- Stored internally with the **starting reference** and **length** of the slice

String Literals as Slices

```rust
let s = "Hello, world!"; // the type of s is &str
```

Fixing Our Solution to `first_word()`

```rust
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}

fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s);
    s.clear(); // compile-time error[E0502]
    println!("the first word is: {word}");
}
```

Demo: error[**E0502**]: cannot borrow `s` as mutable because it is also borrowed as immutable

String Slices as Parameters

Instead of `&String`, we can use `&str` as the parameter type

```rust
fn first_word(s: &str) -> &str {
```

String Slices as Parameters

```rust
fn main() {
    let my_string = String::from("hello world");

    // works on slices of `String`s (partial or whole)
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // also works on references to `String`s, which are
    // equivalent to whole slices of `String`s
    let word = first_word(&my_string);

    let my_string_literal = "hello world";
    // works on slices of string literals (partial or whole)
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // Because string literals are string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
```

Creating Slices of Collection Types

```rust
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3]; // `slice` has a type `&[i32]`
println!("{}", slice == &[2, 3]);
```

```
true
()
```

An in-depth look at the `String` type

What is a String?

- A **vector** of bytes

```rust
pub struct String {
    vec: Vec<u8>,
}
```

What is a `String`?

- UTF-8-encoded and **growable**

```rust
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שלום");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

Creating and Initializing a New String

```rust
let mut s = String::new(); // creates a new, empty string
```

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

```rust
let s = "hello".to_string();
```

Appending to a String

```rust
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2); // appends a string slice
println!("s1 is {s1}, s2 is {s2}");
```

```
s1 is foobar, s2 is bar
()
```

```rust
let mut s = String::from("lo");
s.push('l'); // appending a single character
```

Concatenating Strings with +

```rust
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
// using deref coearcion to turn `&s2` into `&s2[..]`
let s3 = s1 + &s2;
// s1 has been moved and is no longer valid
println!("{}, {}", s3, s2);
```

```
Hello, world!, world!
()
```

```rust
fn add(self, s: &str) -> String {
```

Concatenating Strings with the `format!` Macro

```rust
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{s1}-{s2}-{s3}");
println!("{}", s);
```

```
tic-tac-toe
()
```

Indexing into Strings is Not Allowed

```rust
let s1 = String::from("hello");
let h = s1[0];
```

```
error[E0277]: the type `str` cannot be indexed by `{integer}`
 --> main.rs:3:16
  |
3 |     let h = s1[0];
  |                ^ string indices are ranges of `usize`
  |
  = help: the trait `SliceIndex<str>` is not implemented for `{integer}`, w
  = note: you can use `.chars().nth()` or `.bytes().nth()`
          for more information, see chapter 8 in The Book: <https://doc.rus
  = help: the trait `SliceIndex<[_]>` is implemented for `usize`
  = help: for that trait implementation, expected `[_]`, found `str`
  = note: required for `String` to implement `Index<{integer}>`
```

Strings are encoded internally with UTF-8 — a character can take one or two bytes

Use Slices Instead

```rust
let hello = String::from("😃🤗📘");
let s = &hello[0..4];
println!("{}", s);
```

```
😃
()
```

Internal UTF-8 Encoding Values

```rust
let hello = String::from("😀🤗📘");
let s = &hello[0..4];

for c in hello.chars() {
    print!("{c} ");
}

for b in s.bytes() {
    println!("{b}");
}
```

```
😀 🤗 📘 240
159
152
128
()
```

Required Additional Reading

The Rust Programming Language, Chapter 4, 8.2