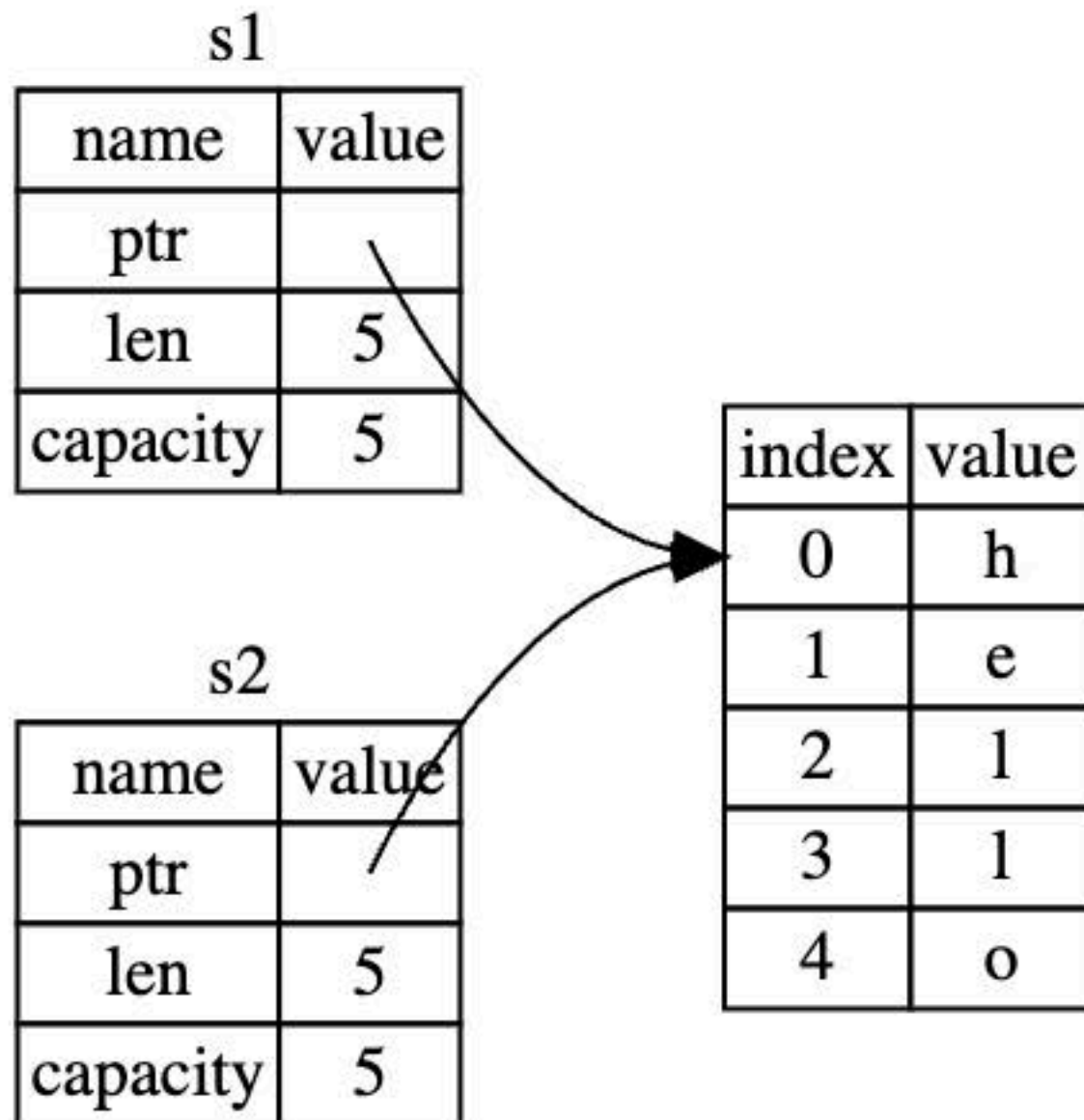



The length is how much memory, in bytes, the contents of the `String` are currently using. The capacity is the total amount of memory, in bytes, that the `String` has received from the allocator. The difference between length and capacity matters, but not in this context, so for now, it's fine to ignore the capacity.

When we assign `s1` to `s2`, the `String` data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the pointer refers to. In other words, the data representation in memory looks like Figure 4-2.




The representation does *not* look like Figure 4-3, which is what memory would look like if Rust instead copied the heap data as well. If Rust did this, the operation `s2 = s1` could be very expensive in terms of runtime performance if the data on the heap were large.

s1

name	value
ptr	
len	5
capacity	5

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

s2

name	value
ptr	
len	5
capacity	5

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

Earlier, we said that when a variable goes out of scope, Rust automatically calls the `drop` function and cleans up the heap memory for that variable. But Figure 4-2 shows both data pointers pointing to the same location. This is a problem: when `s2` and `s1` go out of scope, they will both try to free the same memory. This is known as a *double free* error and is one of the memory safety bugs we mentioned previously. Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities.

To ensure memory safety, after the line `let s2 = s1;`, Rust considers `s1` as no longer valid. Therefore, Rust doesn't need to free anything when `s1` goes out of scope. Check out what happens when you try to use `s1` after `s2` is created; it won't work:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{s1}, world!");
```



You'll get an error like this because Rust prevents you from using the invalidated reference:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
--> src/main.rs:5:15
2 |   let s1 = String::from("hello");
  |           -- move occurs because `s1` has type `String`, which does not implement the
3 |   let s2 = s1;
  |           -- value moved here
4 |
5 |   println!("{s1}, world!");
  |           ^^^^^ value borrowed here after move

= note: this error originates in the macro `$crate::format_args_nl` which comes from the
help: consider cloning the value if the performance cost is acceptable
3 |   let s2 = s1.clone();
  |           ++++++
```

For more information about this error, try ``rustc --explain E0382``.
error: could not compile `ownership` (bin "ownership") due to 1 previous error

If you've heard the terms *shallow copy* and *deep copy* while working with other languages, the concept of copying the pointer, length, and capacity without copying the data probably sounds like making a shallow copy. But because Rust also invalidates the first variable, instead of being called a shallow copy, it's known as a *move*. In this example, we would say that `s1` was *moved* into `s2`. So, what actually happens is shown in Figure 4-4.

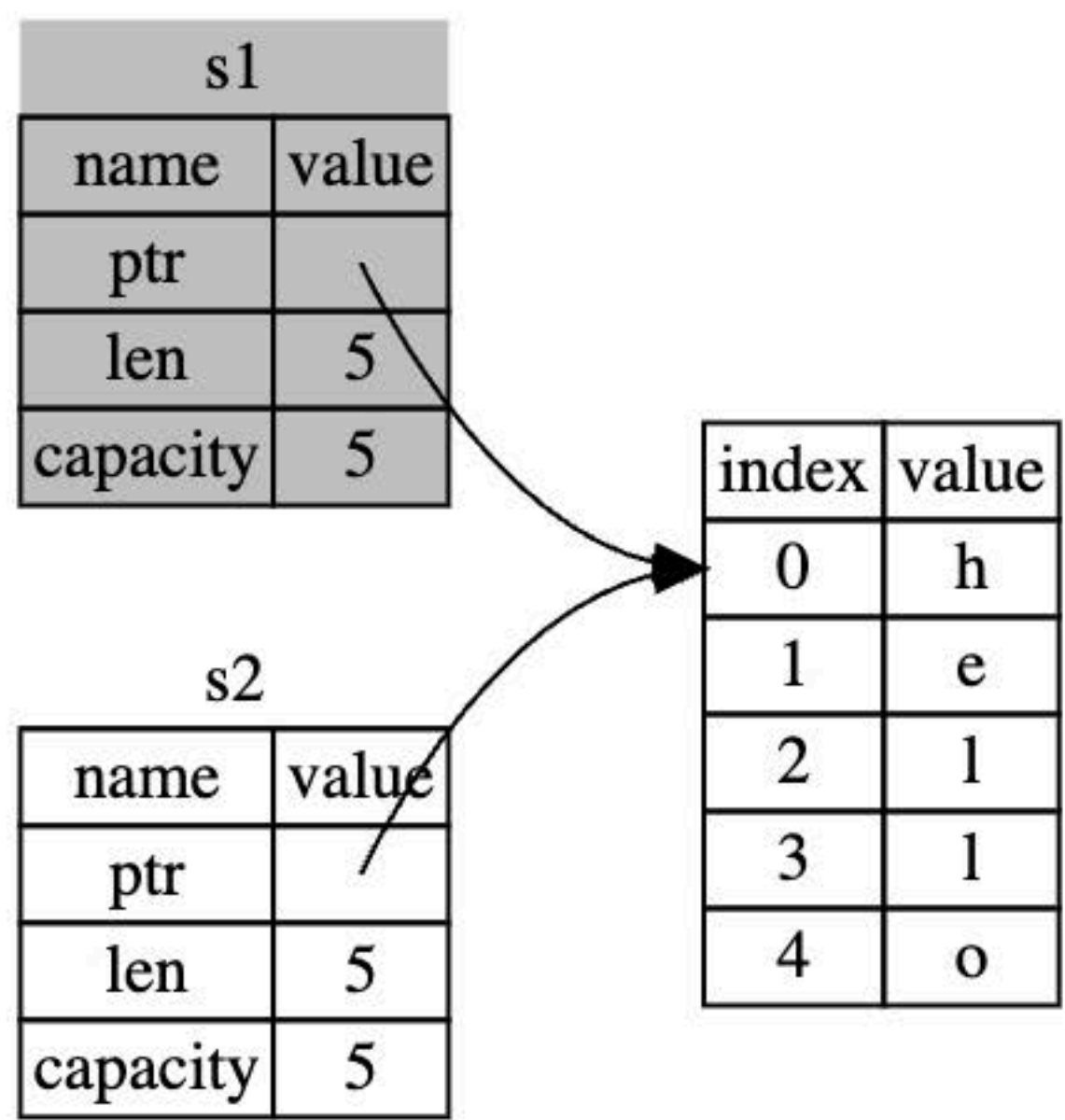


Figure 4-4: Representation in memory after `s1` has been invalidated

That solves our problem! With only `s2` valid, when it goes out of scope it alone will free the memory, and we're done.

In addition, there's a design choice that's implied by this: Rust will never automatically create "deep" copies of your data. Therefore, any *automatic* copying can be assumed to be inexpensive in terms of runtime performance.

Scope and Assignment

The inverse of this is true for the relationship between scoping, ownership, and memory being freed via the `drop` function as well. When you assign a completely new value to an existing variable, Rust will call `drop` and free the original value's memory immediately. Consider this code, for example:

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

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

We initially declare a variable `s` and bind it to a `String` with the value `"hello"`. Then we immediately create a new `String` with the value `"ahoy"` and assign it to `s`. At this point, nothing is referring to the original value on the heap at all.

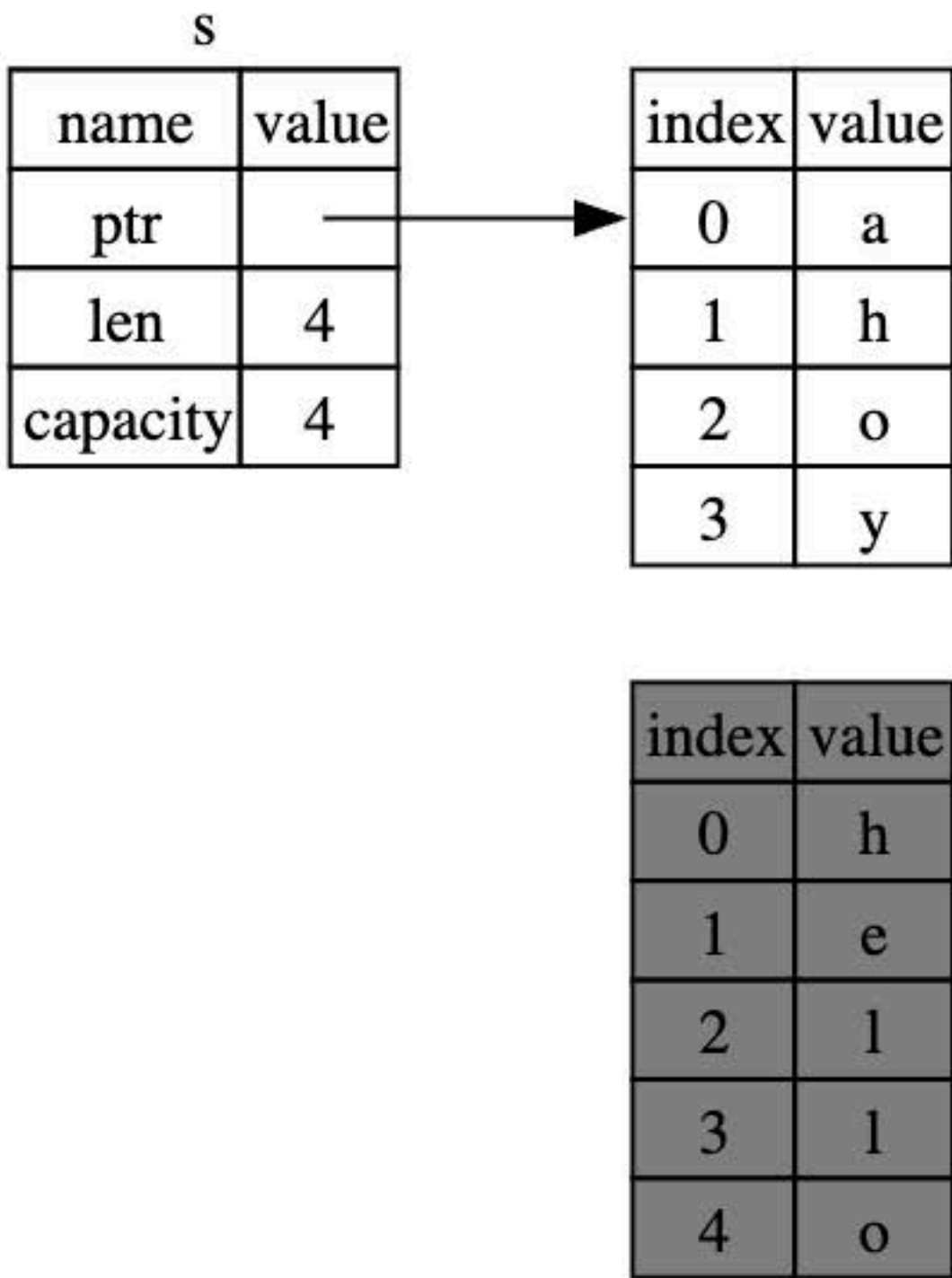


Figure 4-5: Representation in memory after the initial value has been replaced in its entirety.

The original string thus immediately goes out of scope. Rust will run the `drop` function on it and its memory will be freed right away. When we print the value at the end, it will be `"ahoy, world!"`.

Variables and Data Interacting with Clone

If we *do* want to deeply copy the heap data of the `String`, not just the stack data, we can use a common method called `clone`. We'll discuss method syntax in Chapter 5, but because methods are a common feature in many programming languages, you've probably seen them before.

Here's an example of the `clone` method in action:

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

This works just fine and explicitly produces the behavior shown in Figure 4-3, where the heap data *does* get copied.

When you see a call to `clone`, you know that some arbitrary code is being executed and that code may be expensive. It's a visual indicator that something different is going on.

Stack-Only Data: Copy

There's another wrinkle we haven't talked about yet. This code using integers—part of which was shown in Listing 4-2—works and is valid:

```
let x = 5;
let y = x;

println!("x = {x}, y = {y}");
```

But this code seems to contradict what we just learned: we don't have a call to `clone`, but `x` is still valid and wasn't moved into `y`.

The reason is that types such as integers that have a known size at compile time are stored entirely on the stack, so copies of the actual values are quick to make. That means there's no reason we would want to prevent `x` from being valid after we create the variable `y`. In other words, there's no difference between deep and shallow copying here, so calling `clone` wouldn't do anything different from the usual shallow copying, and we can leave it out.

Rust has a special annotation called the `Copy` trait that we can place on types that are stored on the stack, as integers are (we'll talk more about traits in [Chapter 10](#)). If a type implements the `Copy` trait, variables that use it do not move, but rather are trivially copied, making them still valid after assignment to another variable.

Rust won't let us annotate a type with `Copy` if the type, or any of its parts, has implemented the `Drop` trait. If the type needs something special to happen when the value goes out of scope and we add the `Copy` annotation to that type, we'll get a compile-time error. To learn about how to add the `Copy` annotation to your type to implement the trait, see ["Derivable Traits"](#) in Appendix C.

So, what types implement the `Copy` trait? You can check the documentation for the given type to be sure, but as a general rule, any group of simple scalar values can implement `Copy`, and nothing that requires allocation or is some form of resource can implement `Copy`. Here are some of the types that implement `Copy`:

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating-point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain types that also implement `Copy`. For example, `(i32, i32)` implements `Copy`, but `(i32, String)` does not.

Ownership and Functions

The mechanics of passing a value to a function are similar to those when assigning a value to a variable. Passing a variable to a function will move or copy, just as assignment does. Listing 4-3 has an example with some annotations showing where variables go into and out of scope.

Filename: src/main.rs

```
fn main() {
    let s = String::from("hello"); // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                 // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward
} // Here, x goes out of scope, then s. But because s's value was moved, nothing
   // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{some_string}");
} // Here, some_string goes out of scope and `drop` is called. The backing
   // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // Here, some_integer goes out of scope. Nothing special happens.
```

Listing 4-3: Functions with ownership and scope annotated

If we tried to use `s` after the call to `takes_ownership`, Rust would throw a compile-time error. These static checks protect us from mistakes. Try adding code to `main` that uses `s` and `x` to see where you can use them and where the ownership rules prevent you from doing so.

Return Values and Scope

Returning values can also transfer ownership. Listing 4-4 shows an example of a function that returns some value, with similar annotations as those in Listing 4-3.

Filename: src/main.rs

```
fn main() {
    let s1 = gives_ownership();           // gives_ownership moves its return
                                         // value into s1

    let s2 = String::from("hello");      // s2 comes into scope

    let s3 = takes_and_gives_back(s2);   // s2 is moved into
                                         // takes_and_gives_back, which also
                                         // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {        // gives_ownership will move its
                                         // return value into the function
                                         // that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                          // some_string is returned and
                                         // moves out to the calling
                                         // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                         // scope

    a_string // a_string is returned and moves out to the calling function
}
```

Listing 4-4: Transferring ownership of return values

The ownership of a variable follows the same pattern every time: assigning a value to another variable moves it. When a variable that includes data on the heap goes out of scope, the value will be cleaned up by `drop` unless ownership of the data has been moved to another variable.

While this works, taking ownership and then returning ownership with every function is a bit tedious. What if we want to let a function use a value but not take ownership? It's quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well.

Rust does let us return multiple values using a tuple, as shown in Listing 4-5.

Filename: src/main.rs

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{s2}' is {len}.");
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Listing 4-5: Returning ownership of parameters

But this is too much ceremony and a lot of work for a concept that should be common. Luckily for us, Rust has a feature for using a value without transferring ownership, called *references*.