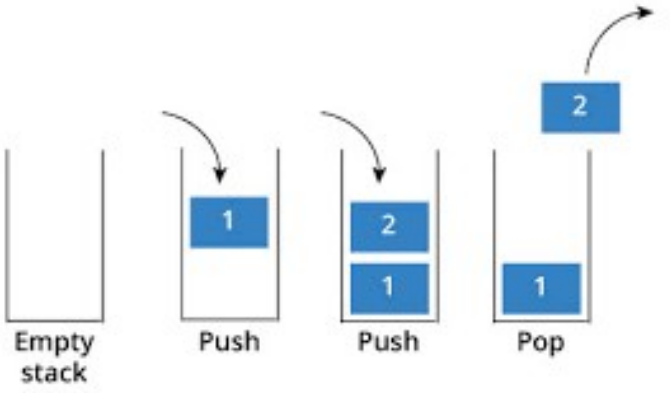
In a **systems programming language** like **Rust**,
whether a **value** is **on the stack** or **the heap**
has an enormous effect on how the language behaves
and why you have to make certain decisions.

STACK & HEAP

→ parts of memory

STACK

- stores values in the order it gets them and removes the values in the opposite order
- is referred to as **last in, first out**
- add data = push onto the stack
- remove data = pop off the stack
- all data stored on the stack must have a known, fixed size



HEAP

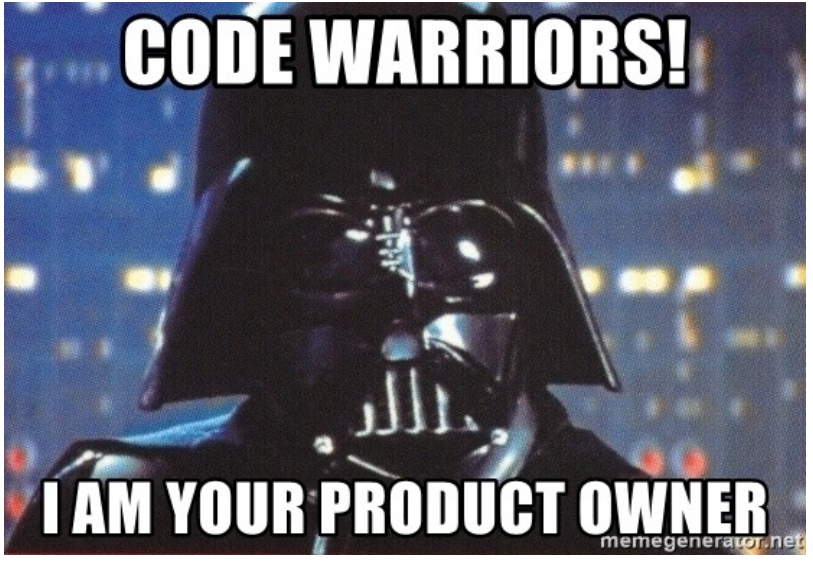
- **stores data** with
  an **unknown size** at compile time
  or a **size that might change**
- is less organized
- **allocating on the heap**
  (operating system finds an empty spot in the heap that is big enough, marks it as being in use, and returns a pointer, which is the address of that location)

Stack or heap?

- pushing to the stack is **faster** than allocating on the heap
- allocating space on the heap requires more work, because the operating system must first find a big enough space to hold the data
- accessing data in the heap is **slower** than accessing data on the stack

What is ownership?

When you understand it, you'll have a solid foundation for understanding the features that **make Rust unique**



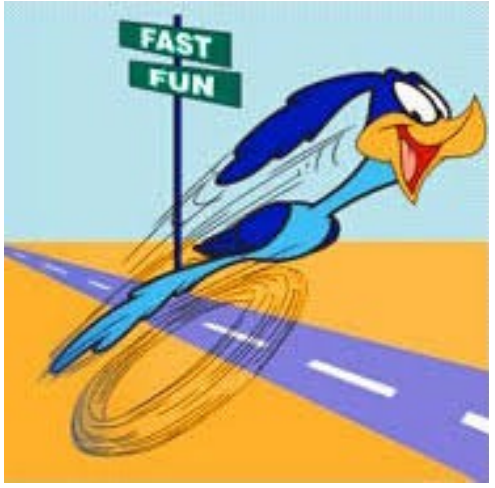
Ownership

→ **keeping track** of what parts of code are using what data on the heap

→ **minimizing** the amount of duplicate data on the heap

→ **cleaning up** unused data on the heap so you don't run out of space

None of the ownership features slow down your program while it is running!



Ownership rules

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

Variable scope

```
let s = "hello";
```

```
{                       // s is not valid here, it's not yet declared
    let s = "hello";    // s is valid from this point forward

    // do stuff with s
}                       // this scope is now over, and s is no longer valid
```

When ***s*** into scope comes, valid it is.

It remains valid until out of scope it goes.

The **String** type

String literals are convenient, but they aren’t suitable for every situation in which we may want to use text (they are immutable).

→ Rust has a **second string type**, String - it is allocated on the heap and as such is able to store an amount of text that is unknown to us at compile time.

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

This kind of string can be mutated:

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

s.push_str(", world!"); // push_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`
```

Memory and Allocation

String::from

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                  // this scope is now over, and s is no
                                   // longer valid
```

What when **s** goes out of scope?

→ Rust calls a special function for us: **drop**, and it is where the author of String can put the code to free memory.

Rust calls drop automatically at the closing curly bracket.

Ways Variables and Data Interact: **Move**

In Rust multiple variables can interact with the same data in different ways

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

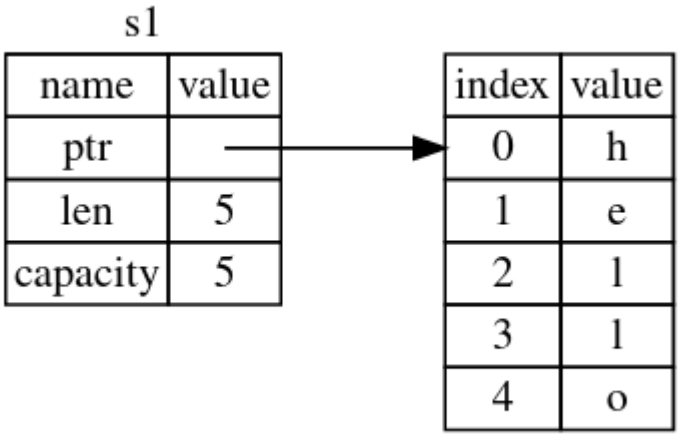
Integers are simple values with a known, fixed size - these two 5 values are pushed onto the stack

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

With **String**, there is a different situation – it is made up of three parts: a **pointer to the memory** that holds the contents of the string, a **length**, and a **capacity**. This group of data is stored on the stack.

But the content of the string is stored on the heap.

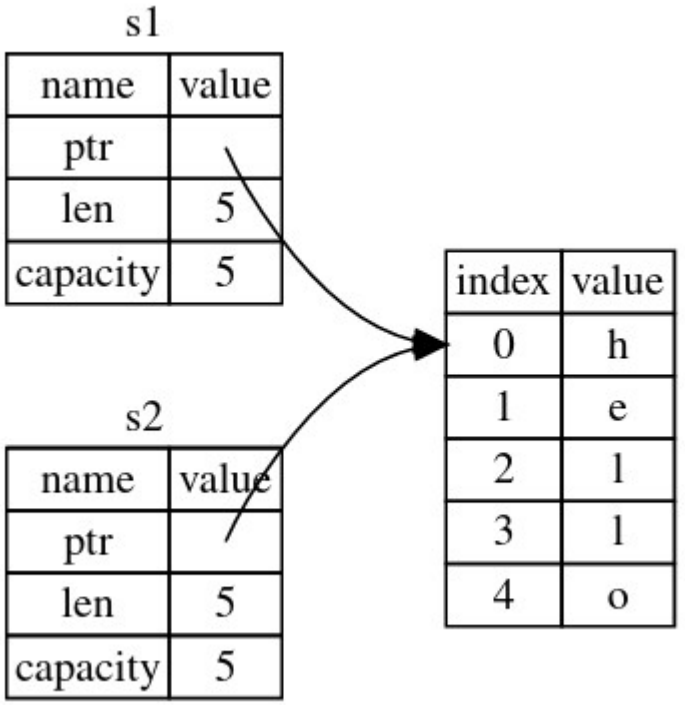
Representation in memory of a String holding the value "hello" bound to s1



→ **length** - how much memory, in bytes, the contents of the String is currently using

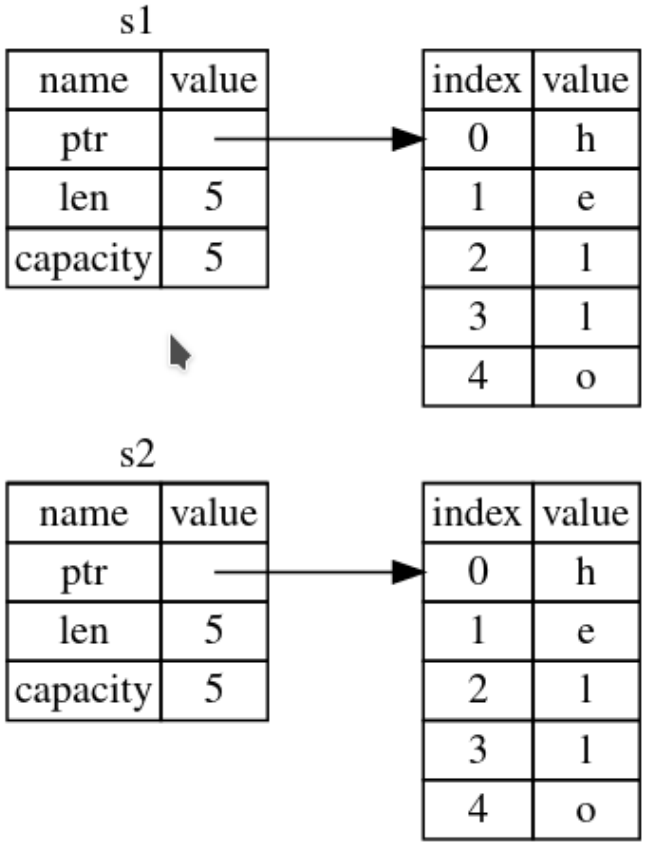
→ **capacity** - the total amount of memory, in bytes, that the String has received from the operating system

Representation in memory of the variable s2 that has a copy of the pointer, length, and capacity of s1



When we **assign s1 to s2**, the **String metadata 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.

Another possibility for what **s2 = s1** might do if Rust copied the heap data as well



What when Rust instead copied the heap data as well?

The operation **s2 = s1** could be **very expensive** in terms of runtime performance, if the data on the heap were large.

What is ***double free error*** ?

→ When **s2 and s1 go out of scope**, they will both try to free the same memory.

→ This is one of the **memory safety bugs**.

→ Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities.

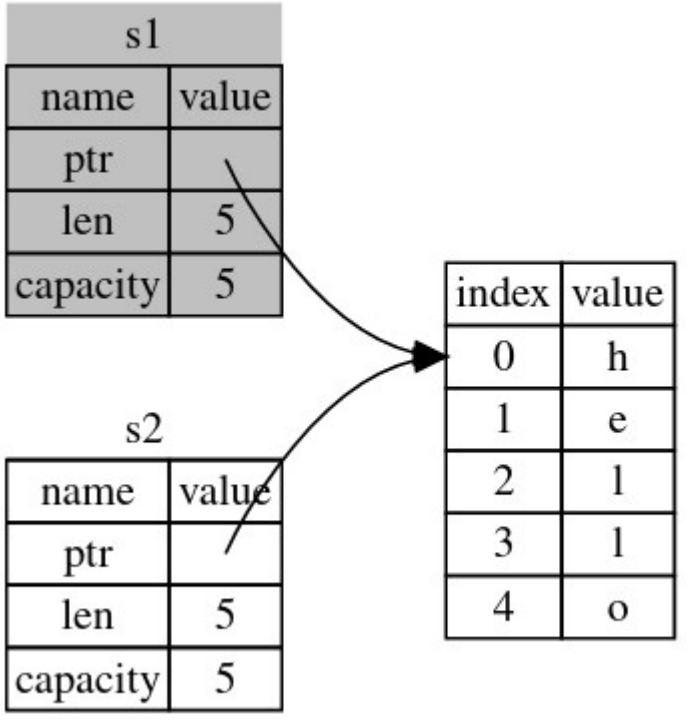
```
let s1 = String::from("hello");
let s2 = s1;

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

Error example for such a situation:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `std::string::String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership`.

To learn more, run the command again with --verbose.
```

Representation in memory after s1 has been invalidated

- The concept of copying the pointer, length, and capacity without copying the data 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**.
- 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.



Ways Variables and Data Interact: **Clone**

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

println!("s1 = {}, s2 = {}", s1, s2);
```

Clone is a method, which is used when we want to deeply copy the heap data of the String, not just the stack data.

Stack-Only Data: **Copy**

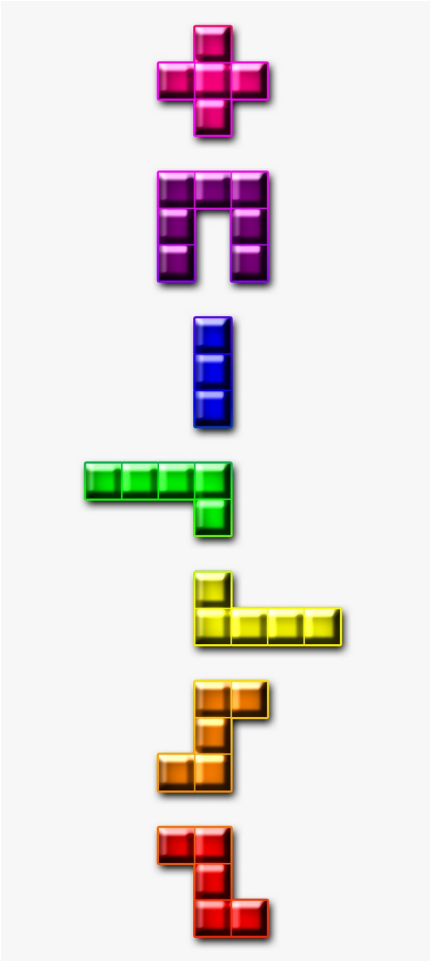
What with this code, which is valid, but seems to contradict what we just learned?
We don't have a call to clone, but x is still valid and wasn't moved into y.

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

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

The reason is that types such as integers that have a **known size at compile time** are stored entirely on the stack.

Rust has a special annotation called the **Copy** trait that we can place on types like integers that are stored on the stack.

If a type has the **Copy** trait, an older variable is still usable after assignment.



So what types are 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 are also Copy. For example, (i32, i32) is Copy, but (i32, String) is not.

Ownership and Functions

```
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.
```

Return Values and Scope - transferring ownership of return values

```rust
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 goes out of scope but 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("hello"); // some_string comes into scope

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

// takes_and_gives_back will take a String and return 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
}
```

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 the data has been moved to be owned by another variable.

Returning ownership of parameters
- it is possible to return multiple values using a tuple

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

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

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

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

    (s, length)
}
```

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 this concept, called **references** :)