Ownership

- Ownership是Rust中一个很重要的特征，它主要是让Rust能保证内存安全的情况下，免于使用garbage collector。有以下一些重要的概念：
  - borrowing
  - slices
- 预备知识：
  - stack: LIFO
  - heap：存入heap的时候是memory allocator找到一个空的地方，再返回一个pointer，points to the address. 它并不像stack那样存入的顺序是fixed的。
  - pushing value on stack往往比到heap上快，因为它不用单独再找一块空地。Accessing data in the heap也会比从stack上找要慢
  - Ownership的目的是为了manage heap data

Ownership rules:

1. Each value in Rust has a variable that's called its owner.
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped.

暂时还不是很理解，，看完例子再回来看看。

以String为例

- String allocated on the heap, 因此，我们在compile它的时候，是不知道它有多大的
- 在这种情况下，string是mutable的：

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

    s.push_str(", world!");

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

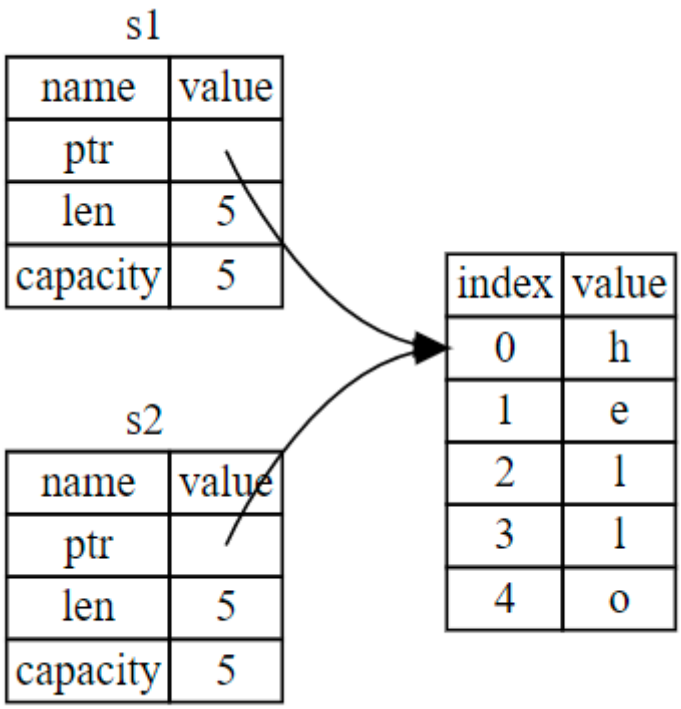
- Rust不需要Gabage collector或者我们自己alloct和free的原因是：the memory is automatically returned once the variable that owns it goes out of scope. 比如下面这个事例：

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

  - 再出去了scope后，rust会自动call一个drop函数
- `let s = String::from("hello")`本质上是得到了一个指向“hello”（存于heap）的pointer。所以，当我们这样赋值的时候：

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

它内存的实际情况不是像int，float那样是复制s1的值给s2，而是让s1，s2指向内存中的同一个地方：



- 值得注意的是，这个数据类型还存了len和capacity
- 当我们`let s2 = s1;`的时候，我们从copy的实际上是这个pointer。

- **但对于上面的例子，我们发现一个很严重的后果：因为对于rust中的每一个varible， 当它out of scope的时候都会调用一次drop(). s1, s2各调用一次的话就会造成double free.**
  - Rust是如何解决这个问题的方法是：
    - 当`let s2 = s1;`之后，Rust就认为s1不再valid了。比如对于下面的例子：

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

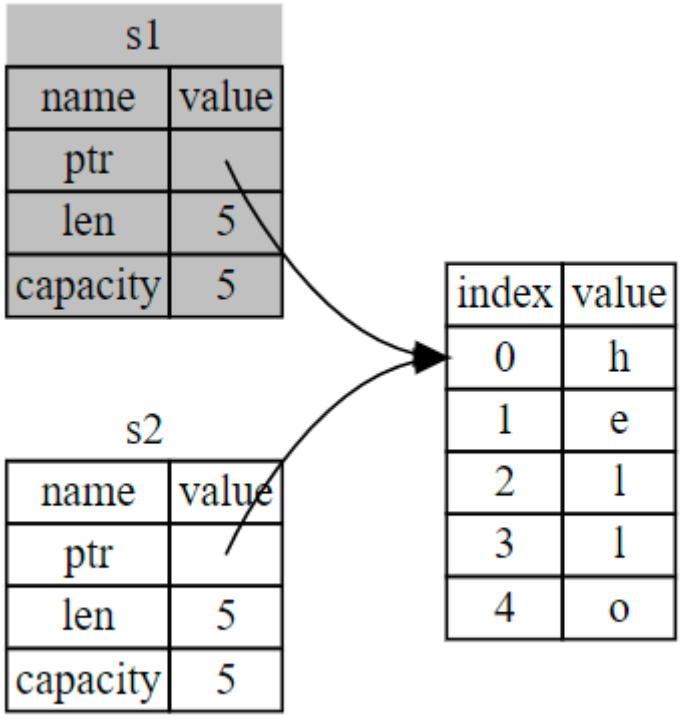
会得到这样error 的output：

```
error[E0382]: borrow of moved value: `s1`
  --> src/main.rs:10:24
   |
8  |         let s1 = String::from("Hi");
   |             -- move occurs because `s1` has type `String`, which
does not implement the `Copy` trait
9  |         let s2 = s1;
   |                  -- value moved here
10 |         println!("{}", s1);
   |                        ^^ value borrowed here after move

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

当然，如果s1不是pointer而是个integer的话是没问题的。因为它不用面临double free的难题。

- 所以，在`let s2 = s1;`后，就会变成：



这里，我们也能比较直观地理解`ownership`的含义了。-> 这块内存的ownership从s1转成了s2

Stack-only Data

- 我们上面也说了，像`i32`这样的值它是存在stack上的，复制了也不用担心double free的问题，所以以下的情况不会报错：

```
let i1 = 5;
let i2 = i1;
println!("{}", i2);
```

- Stack-only Data有：
  - All the integer types

- boolean type
- all the floating type
- character type
- tuples，但前提是tuple中包裹的是上述的stack only data

Return Values and Scope

- 返回值同样可以转移ownership，看下面的代码：

```
fn main() {
    let s1 = gives_ownership();

    let s2 = String::from("Hello");

    let s3 = takes_and_gives_back(s2);
}

fn gives_ownership() -> String{
    let some_string = String::from("yours");

    some_string
}

fn takes_and_gives_back(a_string: String) -> String{
    a_string
}
```

  - some_string指向的“your”在函数返回后，ownership被转移到了s1
  - `takes_and_gives_back()`先是a_string拿走了ownership，然后这个函数又通过return，把ownership交给了s3。
    - 但这样挺麻烦的，因为每次传给参给一个函数，都会把ownship转移给这个函数里的一个变量。这意味着当我们在函数外还想用这个值，还需要这个函数把ownship转移回来。我们可以使用这个值，而不改变ownership吗？ -> `refenrece`

- 另外，Rust也允许函数返回多个值：

```
fn main() {
    let s1 = gives_ownership();

    let s2 = String::from("Hello");

    let s3 = takes_and_gives_back(s2);

    let (s4, len) = calculate_length(s3);
    println!("The len of {} is {}", s4, len);
}

fn gives_ownership() -> String{
    let some_string = String::from("yours");

    some_string
}
```

```
fn takes_and_gives_back(a_string: String) -> String{
    a_string
}

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

    (s, length)
}
```