PLC2025 Lecture 7, 4 Feb 2025

Strings

- Stored on the heap

In [2]:
```
let mut s = String::from("hello"); // allocates heap space for new String and initial
s.push_str(", world!"); // push_str() appends a literal to a String
println!("{}", s); // This will print `hello, world!`
println!("Again {}",s);
```

```
hello, world!
Again hello, world!
```

Copying values, stack

- Value is copied

In [3]:
```
let mut x = 7;
let mut y = x;
y = 77;
println!("x = {}, y = {}",x,y);
```

```
x = 7, y = 77
```

Copying values, heap

- Every value on the heap has a unique owner
- Assignment *moves* ownership
- Memory is freed as soon as scope of owner ends

In [4]:
```
let mut s1 = String::from("hello");
let mut s2 = s1;
s2.push_str(", world");
println!("s1 = {}, s2 = {}", s1, s2);
```

```
[E0382] Error: borrow of moved value: `s1`
   ╭─[command_4:1:1]
   │
 1 │ let mut s1 = String::from("hello");
   │     ──┬─  
   │       ╰─── move occurs because `s1` has type `String`, which does not implement
the `Copy` trait
 2 │ let mut s2 = s1;
   │              ─┬  
   │               ╰── value moved here
   │                 │
   │                 ╰─ help: consider cloning the value if the performance cost is acc
eptable: `.clone()`
   │
 4 │ println!("s1 = {}, s2 = {}", s1, s2);
   │                              ─┬  
   │                               ╰── value borrowed here after move
───╯
```

The `Copy` trait

- Traits are Rust's equivalent of Java interfaces and Python type classes
- For type that have `Copy` trait, values are copied without moving ownership
- All scalar types have this trait: `u16` , `i32` , `f64` , `bool` , `char` etc

Mutable parameters

- Need to declare `mut` to update in function

In [5]:
```
fn main(){
    let mut y = 77;
    update(y);
    println!("y is {}",y);
}

fn update(x:i32){
    x = x+5;
    println!("x is {}",x);
}
```

```
[E0384] Error: cannot assign to immutable argument `x`
   ┌─[command_5:1:1]
   │
 7 │ fn update(x:i32){
   │           │
   │           └─ help: consider making this binding mutable: `mut `
 8 │     x = x+5;
   │     ──┬──
   │       └──── cannot assign to immutable argument
   │
   │ Note: You can change an existing variable to mutable like: `let mut x = x;`
───┘
```

In [6]:
```
fn main(){
    let mut y = 77;
    update(y);
    println!("y is {}",y);
}

fn update(mut x:i32){
    x = x+5;
    println!("x is {}",x);
}
```

In [7]:
```
main()
```

```
x is 82
y is 77
```

Out[7]: ()

Cloning

- Makes a copy of a heap value

In [8]:
```
let s1 = String::from("hello");
let s2 = s1.clone();

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

```
s1 = hello, s2 = hello
```

Transferring ownership via function calls

- Move heap values back and forth

- Here, ownership of `s` moves to function `takes_ownership`

In [9]:
```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into function...
                                    // ... no longer valid here
} // s goes out of scope. Since s's value was moved, nothing special happens.

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

In [10]:
```
main()
```

```
hello
```

Out[10]:
```
()
```

If we try to use `s` after its value has moved, we get an error

In [11]:
```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into function...
                                    // ... no longer valid here
    println!("{}",s);
} // s goes out of scope. Since s's value was moved, nothing special happens.

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

```
[E0382] Error: borrow of moved value: `s`
   ╭─[command_11:1:1]
   │
 2 │     let s = String::from("hello");  // s comes into scope
   │         ┬  
   │         ╰── move occurs because `s` has type `String`, which does not implement t
he `Copy` trait
   │
 4 │     takes_ownership(s);             // s's value moves into function...
   │                     ┬  
   │                     ╰── value moved here
   │                      │
   │                      ╰── help: consider cloning the value if the performance cost
is acceptable: `.clone()`
   │
 6 │     println!("{}",s);
   │                   ┬  
   │                   ╰── value borrowed here after move
───╯
```

- For types with `Copy` trait, the value is copied to the function without moving ownership

In [12]:
```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into function...
                                    // ... no longer valid here
```

```
    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function, but
    println!("x is {}",x);          // i32 is Copy, so okay to still use x
} // x goes out scope, then s.
  // Since s's value was moved, nothing special happens.

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

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

In [13]:
```
main()
```

```
hello
5
x is 5
```

Out[13]: ()

- Examples of moving heap values in and out of functions
- In `gives_ownership`, the scope of `some_string` ends but the value created is moved to the calling scope by the return and hence persists after the function exits

In [14]:
```
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
}
```

- Transferring ownership requires clumsy mechanisms to "get back" parameters passed to functions

In [15]:
```
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)
}
```

In [16]:
```
main()
```

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

Out[16]:
```
()
```

References

- Point to a variable that contains a value on the heap
- Avoids moving ownership
- Creating a reference results in *borrowing* the value

In [17]:
```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

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

In [18]:
```
main()
```

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

Out[18]:
```
()
```

- Arguments passed as references are not automatically mutable
- Use `&mut` to denote a mutable reference

In [19]:
```
fn main() {
    let s = String::from("hello");
    change(&s);
}

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

```
[E0596] Error: cannot borrow `*some_string` as mutable, as it is behind a `&` referenc
e
   ┌─[command_19:1:1]
   │
 6 │ fn change(some_string: &String) {
   │                        ┬
   │                        └─ help: consider changing this to be a mutable referenc
e: `mut `
 7 │     some_string.push_str(", world");
   │     ──────┬──────
   │           └──────── `some_string` is a `&` reference, so the data it refers to cann
ot be borrowed as mutable
   │
   │ Note: You can change an existing variable to mutable like: `let mut x = x;`
───┘
```

In [20]:
```
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
    println!("s is {}",s);
}

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

In [21]:
```
main()
```

```
s is hello, world
```

Out[21]:
```
()
```

Constraints on mutable references

- One mutable reference is permitted

In [22]:
```
{
    let mut s = String::from("hello");
    let r1 = &mut s;
    println!("{}", r1);
}
```

```
hello
```

Out[22]:
```
()
```

- Cannot have two or more mutable references
- Avoids *race conditions* in concurrent programs

In [23]:
```
{
    let mut s = String::from("hello");

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

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

```
[E0499] Error: cannot borrow `s` as mutable more than once at a time
   ┌─[command_23:1:1]
   │
 4 │     let r1 = &mut s;
   │              ──┬──
   │                └──── first mutable borrow occurs here
 5 │     let r2 = &mut s;
   │              ──┬──
   │                └──── second mutable borrow occurs here
   │
 7 │     println!("{}, {}", r1, r2);
   │                        ─┬
   │                         └── first borrow later used here
───┘
```

- Here the second mutable reference is created after the first one goes out of scope, so this is fine

In [24]:
```
{
    let mut s = String::from("hello");
```

```
    {
        let r1 = &mut s;
    } // r1 goes out of scope here, so we can make a new reference with no problems.

    let r2 = &mut s;
}
```

Out[24]: ()

- Cannot mix immutable and mutable references
- Again to avoid race conditions

In [25]:
```
{
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // BIG PROBLEM

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

```
[E0502] Error: cannot borrow `s` as mutable because it is also borrowed as immutable
   ┌─[command_25:1:1]
   │
 4 │     let r1 = &s; // no problem
   │              ─┬
   │               └── immutable borrow occurs here
   │
 6 │     let r3 = &mut s; // BIG PROBLEM
   │              ──┬──
   │                └──── mutable borrow occurs here
   │
 8 │     println!("{}, {}, and {}", r1, r2, r3);
   │                                 ┬
   │                                 └── immutable borrow later used here
───┘
```

- Here the last use of `r1` and `r2` occurs before `r3` is declared
- Rust does sophisticated static analysis to determine this at compile time

In [26]:
```
{
    let mut s = String::from("hello");

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

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

}
```

```
hello and hello
hello
```

Out[26]: ()

- Unlike `gives_ownership` earlier, here `dangle` returns a reference
- Potential problem --- when `dangle` exits, `s` goes out of scope and `reference_to_nothing` becomes a *dangling pointer*, pointing to nothing

- Rust catches this as a compile-time error

In [27]:
```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

```
[E0106] Error: missing lifetime specifier
   ┌─[command_27:1:1]
   │
 5 │ fn dangle() -> &String {
   │                ┬┘
   │                └── expected named lifetime parameter
   │                 ┬
   │                 └── help: instead, you are more likely to want to return an owned
value: ``
   │                  │
   │                  └── help: consider using the `'static` lifetime, but this is uncom
mon unless you're returning a borrowed value from a `const` or a `static`: `'static `
───┘

[unused_variables] Error: unused variable: `reference_to_nothing`
   ┌─[command_27:1:1]
   │
 2 │     let reference_to_nothing = dangle();
   │         ──────────┬─────────
   │                   └─────────── warning: unused variable: `reference_to_nothing`
   │                   │
   │                   └─────────── help: if this is intentional, prefix it with an un
derscore: `_reference_to_nothing`
───┘

[E0515] Error: cannot return reference to local variable `s`
   ┌─[command_27:1:1]
   │
 7 │     &s
   │     ┬┘
   │     └── returns a reference to data owned by the current function
───┘
```

Slices

- A function to compute the length of the first word in a string
- `bytes.iter()` iterates through `bytes`, `enumerate()` returns a pair (index,reference to value), which is deomposed through pattern matching into `(i, &item)`
- `b' '` specifies a byte constant for the space character

In [28]:
```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

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

    s.len()
}
```

- In this function, Rust cannot recognize that the return value is an index into the string
- If we clear the string, the index is no longer valid, but cannot be flagged by compiler

In [29]:
```
fn main() {
    let mut s = String::from("hello world");

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

    s.clear(); // this empties the String, 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 totally invalid!
}
```

- Digression on references and scalar variables, to be resolved later

In [30]:
```
{
    let mut x = 5;
    let y = &mut x;
    *y = 7;
    println!("x is {}, y is {}",x,*y);
}
```

```
[unused_variables] Error: unused variable: `word`
[E0502] Error: cannot borrow `x` as immutable because it is also borrowed as mutable
    ┌─[command_30:1:1]
    │
  3 │     let y = &mut x;
    │             ──┬──
    │               └──── mutable borrow occurs here
    │
  5 │     println!("x is {}, y is {}",x,*y);
    │                                  ┬ ┬
    │                                  └──── immutable borrow occurs here
    │                                    │
    │                                    └── mutable borrow later used here
────┘
```

- A string slice is written similar to a slice in Python
- Gives a reference to a substring

In [31]:
```
{
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
}
```

Out[31]: ()

- Rewrite `first_word` to return slice corresponding to first word
- Will examine distinction between `&String` and `&str` later

In [32]:
```
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[..]
}
```

- Now, if we try to clear the "parent" string while holding a reference to a substring, it is a compile error
- Another example of combining immutable and mutable references --- the call `s.clear()` implicitly passes a mutable reference to `s` to `clear()`, while `word` currently holds an immutable reference

In [33]:
```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```

```
[E0502] Error: cannot borrow `s` as mutable because it is also borrowed as immutable
   ┌─[command_33:1:1]
   │
 4 │     let word = first_word(&s);
   │                           ─┬
   │                            └── immutable borrow occurs here
   │
 6 │     s.clear(); // error!
   │     ─────┬─
   │          └─── mutable borrow occurs here
   │
 8 │     println!("the first word is: {}", word);
   │                                       ──┬─
   │                                         └── immutable borrow later used here
───┘
```