

CMSC 330: Organization of Programming Languages

Ownership, References, and Lifetimes in Rust

Memory: the Stack and the Heap

- The stack
 - constant-time, automatic (de)allocation
 - Data size and lifetime must be known at compile-time
 - Function parameters and locals of known (constant) size
- The heap
 - Dynamically sized data, with non-fixed lifetime
 - Slightly slower to access than stack; i.e., via a pointer
 - **GC**: automatic deallocation, adds space/time overhead
 - **Manual** deallocation (C/C++): low overhead, but non-trivial opportunity for **devastating bugs**
 - Dangling pointers, double free – instances of memory corruption

Memory Management Errors

- May forget to free memory (**memory leak**)

```
{ int *x = (int *) malloc(sizeof(int)); }
```
- May retain ptr to freed memory (**dangling pointer**)

```
{ int *x = ...malloc();  
  free(x);  
  *x = 5; /* oops! */  
}
```
- May try to free something twice (**double free**)

```
{ int *x = ...malloc(); free(x); free(x); }
```

 - This may corrupt the memory management data structures
 - E.g., the memory allocator maintains a **free list** of space on the heap that's available

GC-less Memory Management, Safely

- Rust's heap memory **managed without GC**
- **Type checking** ensures **no dangling pointers** or **double frees**
 - **unsafe** idioms are **disallowed**
 - **memory leaks *not prevented*** (not a safety problem)
- Key features of Rust that ensure safety: **ownership** and **lifetimes**
 - Data has a single **owner**. **Immutable** aliases OK, but mutation only via owner or **single mutable reference**
 - How long data is alive is determined by a **lifetime**

Rules of Ownership

1. Each value in Rust has a variable that's 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** (freed)

String: Dynamically sized, mutable data

```
{
  let mut s = String::from("hello");
  s.push_str(", world!");
  println!("{}", s);
} //s's data is freed by calling s.drop()
```

namespace (like Java package)

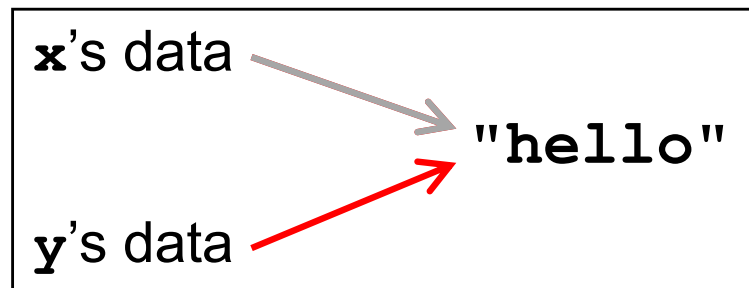
- **s**'s contents allocated on the heap
 - Pointer to data is internal to the **String** object
 - When appending to **s**, the old data is freed and new data is allocated
- **s** is the *owner* of this data
 - When **s** goes out of scope, its **drop** method is called, which frees the data
 - No garbage collection!

Transferring Ownership

- Heap allocated data is copied by reference

```
let x = String::from("hello");  
let y = x; //x moved to y
```

- Both **x** and **y** point to the same underlying data



*Avoids double
free()!*

- A move leaves only **one owner**: **y**

```
println!("{}", world!", y); //ok  
println!("{}", world!", x); //fails
```

Deep Copying Retains Ownership

- Make **clones** (copies) to avoid ownership loss

```
let x = String::from("hello");  
let y = x.clone(); //x no longer moved  
println!("{}", world!", y); //ok  
println!("{}", world!", x); //ok
```

- Primitives copied automatically

- `i32`, `char`, `bool`, `f32`, tuples of these types, etc.

```
let x = 5;  
let y = x;  
println!("{}", = 5!", y); //ok  
println!("{}", = 5!", x); //ok
```

- These have the **Copy** trait; more on traits later

Ownership and Function Calls

- On a call, ownership passes from:
 - argument to called function's parameter
 - returned value to caller's receiver

```
fn main() {
  let s1 = String::from("hello");
  let s2 = id(s1); //s1 moved to arg
  println!("{}",s2); //id's result moved to s2
  println!("{}",s1); //fails
}
fn id(s:String) -> String {
  s // s moved to caller, on return
}
```

References and Borrowing

- Create an alias by making a **reference**
 - An explicit, non-owning pointer to the original value
 - Called **borrowing**. Done with **&** operator
- **References are immutable** by default

```
fn main() {
  let s1 = String::from("hello");
  let len = calc_len(&s1); //lends pointer
  println!("the length of '{}' is {}",s1,len);
}
fn calc_len(s: &String) -> usize {
  s.push_str("hi"); //fails! refs are immutable
  s.len()          // s dropped; but not its referent
}
```

Quiz 1: Owner of s data at *HERE* ?

```
fn foo(s:String) -> usize {  
  let x = s;  
  let y = &x;  
  let z = x;  
  let w = &y;  
  \\ HERE  
}
```

- A. x
- B. y
- C. z
- D. w

Quiz 1: Owner of s data at *HERE* ?

```
fn foo(s:String) -> usize {  
  let x = s;  
  let y = &x;  
  let z = x;  
  let w = &y;  
  \\ HERE  
}
```

A. x

B. y

C. z

D. w

Rules of References

1. At any given time, you can have *either* but not both of
 - One mutable reference
 - Any number of immutable references
2. References must always be valid (pointed-to value not dropped)

Borrowing and Mutation

- Make **immutable references** to **mutable** values
 - Shares **read-only access** through owner and borrowed references
 - Same for immutable values
 - **Mutation disallowed** on original value until **borrowed reference(s)** dropped

```
{ let mut s1 = String::from("hello");  
  { let s2 = &s1;  
    println!("String is {} and {}",s1,s2); //ok  
    s1.push_str(" world!"); //disallowed  
  } //drops s2  
  s1.push_str(" world!"); //ok  
  println!("String is {}",s1);} //prints updated s1
```

Mutable references

- To permit mutation via a reference, use **&mut**
 - Instead of just **&**
 - But **only OK for mutable variables**

```
let mut s1 = String::from("hello");
{ let s2 = &s1;
  s2.push_str(" there"); //disallowed; s2 immut
} //s2 dropped
let mut s3 = &mut s1; //ok since s1 mutable
s3.push_str(" there"); //ok since s3 mutable
println!("String is {}", s3); //ok
```

Quiz 2: What does this evaluate to?

```
{ let mut s1 = String::from("Hello!");  
  {  
    let s2 = &s1;  
    s2.push_str("World!");  
    println!("{}", s2)  
  }  
}
```

- A. "Hello!"
- B. "Hello! World!"
- C. Error
- D. "Hello!World!"

Quiz 2: What does this evaluate to?

```
{ let mut s1 = String::from("Hello!");  
  {  
    let s2 = &s1;  
    s2.push_str("World!");  
    println!("{}", s2)  
  }  
}
```

- A. "Hello!"
- B. "Hello! World!"
- C. Error; s2 is not mut**
- D. "Hello!World!"

Quiz 3: What is printed?

```
fn foo(s: &mut String) -> usize{
    s.push_str("Bob");
    s.len()
}
fn main() {
    let mut s1 = String::from("Alice");
    println!("{}",foo(&mut s1))
}
```

- A. 0
- B. 8
- C. Error
- D. 5

Quiz 3: What is printed?

```
fn foo(s: &mut String) -> usize{
    s.push_str("Bob");
    s.len()
}
fn main() {
    let mut s1 = String::from("Alice");
    println!("{}",foo(&mut s1))
}
```

- A. 0
- B. 8**
- C. Error
- D. 5

Ownership and Mutable References

- Can make **only one mutable reference**
- Doing so **blocks use** of the original
 - **Restored** when reference is dropped

```
let mut s1 = String::from("hello");
{ let s2 = &mut s1; //ok
  let s3 = &mut s1; //fails: second borrow
  s1.push_str(" there"); //fails: second borrow
} //s2 dropped; s1 is first-class owner again
s1.push_str(" there"); //ok
println!("String is {}",s1); //ok
```

implicit borrow

(**self** is a reference)

Immutable and Mutable References

- **Cannot make a mutable reference** if immutable references exist
 - Holders of an immutable reference assume the object will not change from under them!

```
let mut s1 = String::from("hello");
{ let s2 = &s1; //ok: s2 is immutable
  let s3 = &s1; //ok: multiple imm. refs allowed
  let s4 = &mut s1; //fails: imm ref already
} //s2-s4 dropped; s1 is owner again
s1.push_str(" there"); //ok
println!("String is {}",s1); //ok
```

Aside: Generics and Polymorphism

- Rust has support like that of Java and OCaml
 - Example: The `std` library defines `Vec<T>` where `T` can be **instantiated** with a variety of types
 - `Vec<char>` is a vector of characters
 - `Vec<&str>` is a vector of string slices
- You can define polymorphic functions, too
 - Rust:

```
fn id<T>(x:T) -> T { x }
```
 - Java:

```
static <T> T id(T x) { return x; }
```
 - Ocaml:

```
let id x = x
```
- More later...

Dangling References

- References must always be to **valid memory**
 - Not to memory that **has been dropped**

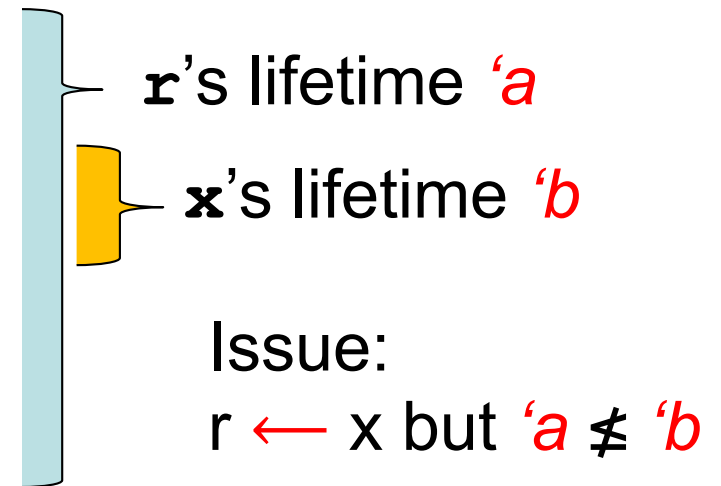
```
fn main() {  
    let ref_invalid = dangle();  
    println!("what will happen ... {}", ref_invalid);  
}  
fn dangle() -> &String {  
    let s1 = String::from("hello");  
    &s1  
} // bad! s1's value has been dropped
```

- Rust type checker will disallow this using a concept called **lifetimes**
 - A **lifetime** is a type-level parameter that **names the scope in which the data is valid**

Lifetimes: Preventing Dangling Refs

- Another way to view our prior example

```
{  
  let r; // deferred init  
  {  
    let x = 5;  
    r = &x;  
  }  
  println!("r: {}", r); //fails  
}
```



- The Rust type checker observes that **x** goes out of scope while **r** still exists
 - A **lifetime** is a *type variable* that identifies a scope
 - **r's lifetime 'a exceeds x's lifetime 'b**

Lifetimes and Functions

- Lifetime of a reference not always visible
 - E.g., when passed as an **argument to a function**

String slice
(more later)

```
fn longest(x:&str, y:&str) -> &str {  
    if x.len() > y.len() { x } else { y }  
}
```

- What could **go wrong** here?

```
{ let x = String::from("hi");  
  let z;  
  { let y = String::from("there");  
    z = longest(&x, &y); //will be &y  
  } //drop y, and thereby z  
  println!("z = {}", z); //yikes!  
}
```

Quiz 4: What is printed?

```
{ let mut s = &String::from("dog");
  {
    let y = String::from("hi");
    s = &y;
  }
  println!("s: {}", s);
}
```

- A. dog
- B. hi
- C. Error – y is immutable
- D. Error – y dropped while still borrowed

Quiz 4: What is printed?

```
{ let mut s = &String::from("dog");  
  {  
    let y = String::from("hi");  
    s = &y;  
  }  
  println!("s: {}", s);  
}
```

- A. dog
- B. hi
- C. Error – y is immutable
- D. Error – y dropped while still borrowed**

Lifetime Parameters

- Each reference of type t has a **lifetime parameter**
 - $\&t$ (and $\&\text{mut } t$) – lifetime is implicit
 - $\&'a t$ (and $\&'a \text{mut } t$) – lifetime $'a$ is explicit
- Where do the lifetime names come from?
 - When left implicit, they are generated by the compiler
 - Global variables have lifetime 'static
- Lifetimes can also be **generic**

```
fn longest<'a>(x:&'a str, y:&'a str) -> &'a str {  
    if x.len() > y.len() { x } else { y }  
}
```

- Thus: x and y must have the same lifetime, and the returned reference shares it

Lifetimes FAQ

- When do we use **explicit lifetimes**?
 - When more than one var/type needs the same lifetime (like the `longest` function)
- How does **lifetime subsumption** work?
 - If lifetime ``a` is longer than ``b`, we can use ``a` where ``b` is expected; can require this with ``b: `a`.
 - Permits us to call `longest(&x, &y)` when `x` and `y` have different lifetimes, but one outlives the other
 - Just like subtyping/subsumption in OO programming
- Can we use **lifetimes in data definitions**?
 - Yes; we will see this later when we define `structs`, `enums`, etc.

Recap: Rules of References

1. At any given time, you can have *either* but not both of
 - One mutable reference
 - Any number of immutable references
2. References must always be valid
 - A reference must never outlive its referent