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

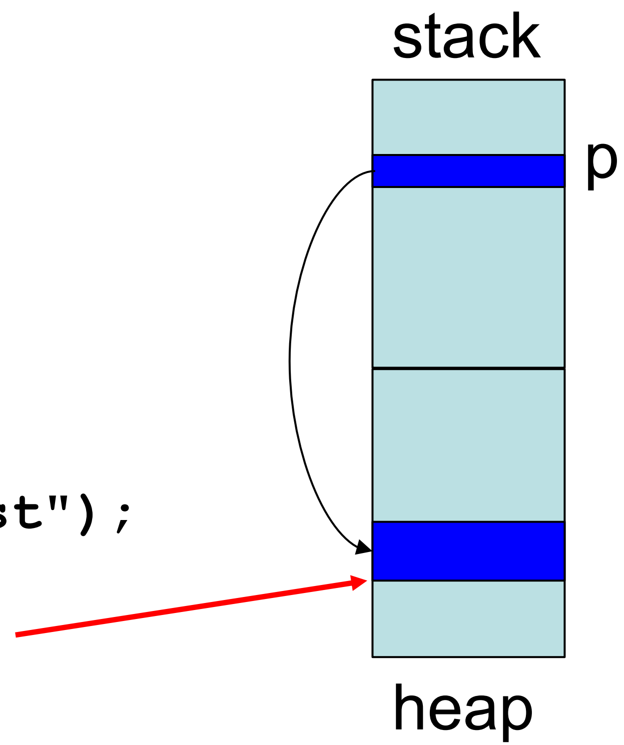
Memory: the Stack and the Heap

// C

```
char *p = malloc(10)
…
free(p);
```

// Java

```
String p = new String(”rust");
…
p = null;
```
//GC will collect later



p is deleted from stack when the function terminates

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)

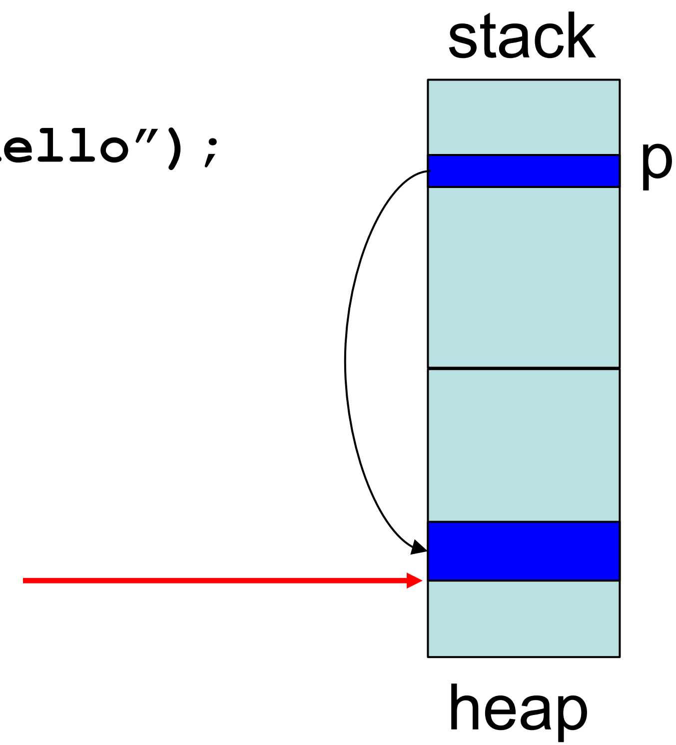
- Key features of Rust that ensure safety: **ownership** and **lifetimes**
  - Data has a single owner. Immutable aliases OK, but mutation only via owner or single mutable reference
  - How long data is alive is determined by a lifetime

Memory: the Stack and the Heap

// Rust

```
let p = String::from("hello”);
...
```

- Deleted when the owner p is out of scope.
- No manual free, no GC



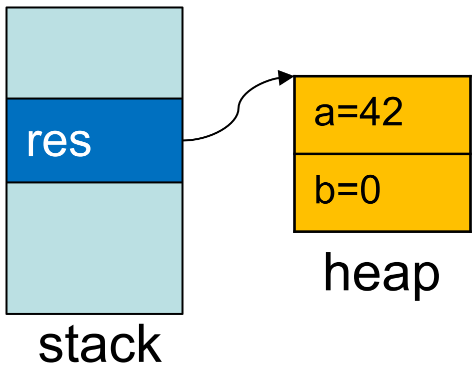
p is deleted from stack when the function terminates

Ownership

Only one "owner" of an object

- When the "owner" of the object goes out of scope, its data is automatically freed. No Garbage collection
- Can not access object beyond its lifetime (checked at compile-time)

```
fn foo() {
  let mut res = Box::new(Pair {
                              a: 0,
                              b: 0
                 });
 res.a = 42;
}
```



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!"); //appends to s

 println!("{}", s);
} //s's data is freed by calling s.drop()
```

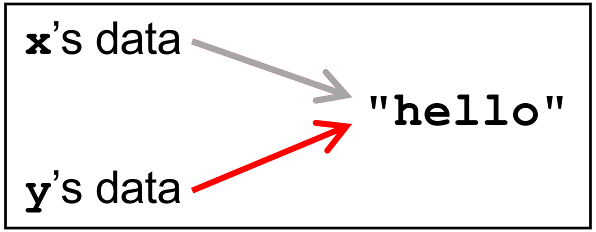
- **`s`** is the *owner* of this data
  - When **`s`** goes out of scope, its **`drop`** method is called, which frees the data

Assignment Transfers 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



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

*Avoids double **free()**!*

```
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 Transfer in Function Calls

```
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
}
```

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

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 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

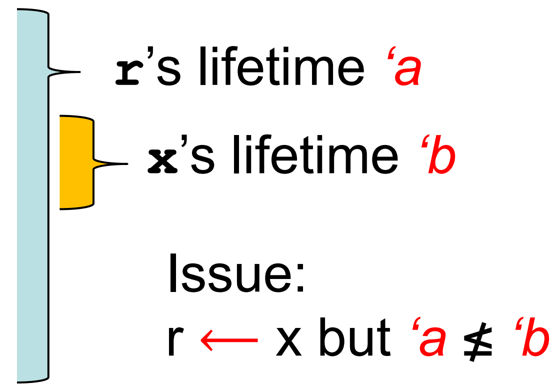
```
fn main() {
  let ref_invalid = dangle();
  println!(“what will happen … {}”,ref_invalid);
}
fn dangle() -> &String {
  let s1 = String::from(“hello”);
  &s1
} // bad! s1’s value has been dropped
```

  - 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

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

Traits Overview

- Traits allow us to abstract behavior that types can have in common
  - In situations where we use generic type parameters, we can use trait bounds to specify that the generic type must implement a trait

- Traits are a bit like Java interfaces
  - But we can implement traits over any type, anywhere in the code, not only at the point we define the type

Defining a Trait

- Here is a trait with a single function

```
pub trait Summarizable {
  fn summary(&self) -> String;
}
```

  - Specify **`&self`** for "instance" methods
    - Note: can also specify "associated" methods
      - Like **`static`** methods in Java
  - Equivalent in Java:

```
public interface Summarizable {
  String summary();
}
```

Implementing a Trait on a Type

name of trait

type on which we are implementing it

```
impl Summarizable for (i32,i32) {
    fn summary(&self) -> String {
        let &(x,y) = self;
        format!("{}",x+y)
    }
}
fn foo() {
    let y = (1,2).summary(); //"3"
    let z = (1,2,3).summary();//fails
}
```

trait method body

trait method invocation

Default Implementations

- Here is a trait with a default implementation

```
pub trait Summarizable {
   fn summary(&self) -> String {
     String::from("none")
   }
}
```

default impl

Impl uses default

```
impl Summarizable for (i32,i32,i32) {}
fn foo() {
    let y = (1,2).summary(); //"3"
    let z = (1,2,3).summary();//"none"
}
```

Trait Bounds

- With generics, you can specify that a type variable must implement a trait

```
pub fn notify<T: Summarizable>(item: T) {
  println!("Breaking news! {}",
                item.summary());
}
```

  – This method works on any type **`T`** that implements the **`Summarizable`** trait

- Can specify multiple Trait Bounds using **`+`**

  `fn foo<T:Clone + Summarizable>(…) -> i32 {…}` *or*

  `fn foo<T>(…) -> i32 where T:Clone + Summarizable {…}`

Standard Traits

- We have seen several standard traits already
  - `Clone` holds if the object has a clone() method
  - `Copy` holds if you can copy it
    - I.e., it's a primitive
  - `Deref` holds if you can dereference it
    - I.e., it's a reference
- There are other useful ones too
  - `Display` if it can be converted to a string
  - `PartialOrd` if it implements a comparison operator

Putting all Together

- Finds the largest element in an array slice
  - Generic in the type **T** of the contents of the array

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T
{
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

Requires Copy trait

Requires PartialOrd trait

Putting all Together

- Finds the largest element in an array slice
  - Generic in the type **T** of the contents of the array

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T
{…}
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

prints

```
The largest number is 100
The largest char is y
```

Notes

- Trait implementations can be generic too

```
pub trait Queue<T> {
    fn enqueue(&mut self, ele: T) -> (); …
}
impl <T> Queue<T> for Vec<T> {
    fn enqueue(&mut self, ele:T) -> () {…} …
}
```

- Generic method implementations of structs and enums can include trait bounds