

CMSC 330: Organization of Programming Languages

Safe, Low-level Programming with **Rust**

What choice do programmers have today?

C/C++

- Low level
- More control
- Performance over safety
- Memory managed manually
- No periodic garbage collection
- ...

Java, OCaml, Go, Ruby...

- High level
- Secure
- Less control
- Restrict direct access to memory
- Run-time management of memory via periodic garbage collection
- No explicit malloc and free
- Unpredictable behavior due to GC
- ...

Rust: Type safety and low-level control

- Begun in 2006 by Graydon Hoare
- Sponsored as full-scale project and announced by Mozilla in 2010
 - Changed a lot since then; source of frustration
 - But now: **most loved programming language** in Stack Overflow annual surveys of **2016**, **2017**, and **2018**
- Takes ideas from **functional** and **OO** languages, and **recent research**
- Key properties: **Type safety** despite use of **concurrency** and **manual memory management**
 - And: **No data races**

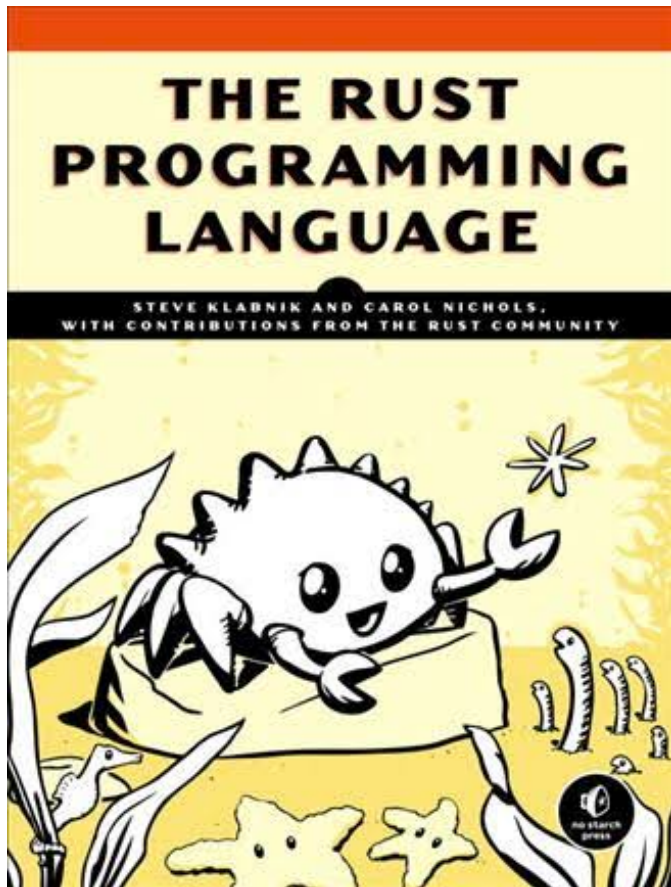
Features of Rust

- Lifetimes and Ownership
 - Key feature for ensuring safety
- Traits as core of object(-like) system
- Variable default is *immutability*
- Data types and *pattern matching*
- Type inference
 - No need to write types for local variables
- Generics (aka *parametric polymorphism*)
- First-class functions
- Efficient C bindings

Rust in the real world

- Firefox Quantum and Servo components
 - <https://servo.org>
- REmacs port of Emacs to Rust
 - <https://github.com/Wilfred/remacs>
- Amethyst game engine
 - <https://www.amethyst.rs/>
- Magic Pocket filesystem from Dropbox
 - <https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/>
- OpenDNS malware detection components
- <https://www.rust-lang.org/en-US/friends.html>

Information on Rust



- **Rust book** free online
 - <https://doc.rust-lang.org/book/>
 - **We will follow it in these lectures**
- More references via Rust site
 - <https://www.rust-lang.org/en-US/documentation.html>
- Rust Playground (REPL)
 - <https://play.rust-lang.org/>

Installing Rust

- Instructions, and stable installers, here:

<https://www.rust-lang.org/en-US/install.html>

- On a Mac or Linux (VM), open a terminal and run

```
curl https://sh.rustup.rs -sSf | sh
```

- On Windows, download+run [rustup-init.exe](#)

<https://static.rust-lang.org/rustup/dist/i686-pc-windows-gnu/rustup-init.exe>

Rust compiler, build system

- Rust programs can be compiled using `rustc`
 - Source files end in suffix `.rs`
 - Compilation, by default, produces an executable
 - No `-c` option
- Preferred: Use the `cargo` package manager
 - Will invoke `rustc` as needed to build files
 - Will download and build dependencies
 - Based on a `.toml` file and `.lock` file
 - You won't have to mess with these for this class
 - Like `ocamlbuild` or `dune`

Using rustc

- Compiling and running a program

main.rs:

```
fn main() {  
    println!("Hello, world!")  
}
```

```
% rustc main.rs
```

```
% ./main
```

```
Hello, world!
```

```
%
```

Using cargo

- Make a project, build it, run it

```
% cargo new hello_cargo --bin
```

```
% cd hello_cargo
```

```
% ls
```

```
Cargo.toml  src/
```

```
% ls src
```

```
main.rs
```

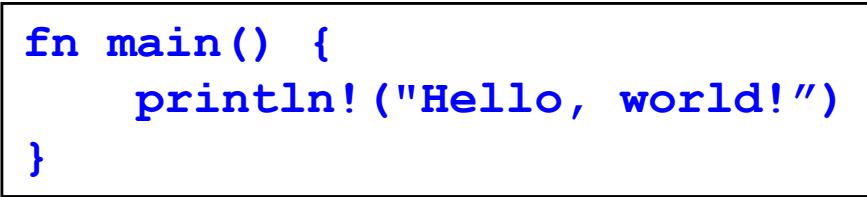
```
% cargo build
```

```
Compiling hello_cargo v0.1.0 (file:///...)
```

```
Finished dev [unoptimized + debuginfo] ...
```

```
% ./target/debug/hello_cargo
```

```
Hello, world!
```



```
fn main() {  
    println!("Hello, world!")  
}
```

Rust, interactively

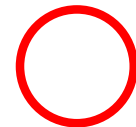
- Rust has no top-level *a la* OCaml or Ruby
- There is an in-browser execution environment
 - See, for example, <https://doc.rust-lang.org/stable/rust-by-example/hello.html>

Hello World

This is the source code of the traditional Hello World program.

```
// This is the main function
fn main() {
    // The statements here will be executed when the compiled binary is called

    // Print text to the console
    println!("Hello World!");
}
```



```
Hello World!
```

Rust Documentation

- Your go-to to learn about Rust is the Rust documentation page
 - <https://doc.rust-lang.org/stable/>
- This contains links to
 - the Rust Book (on which most of our slides are based),
 - the reference manual, and
 - short manuals on the compiler, cargo, and more

Rust Basics

Functions

```
// comment
fn main() {
    println!("Hello, world!");
}
```

Hello, world!

Factorial in Rust (recursively)

```
fn fact(n:i32) -> i32
```

```
{  
  if n == 0 { 1 }  
  else {  
    let x = fact(n-1);  
    n * x  
  }  
}
```

```
fn main() {  
  let res = fact(6);  
  println!("fact(6) = {}", res);  
}
```

fact(6) = 720

If *Expressions* (not Statements)

```
fn main() {  
    let n = 5;  
    if n < 0 {  
        print!("{} is negative", n);  
    } else if n > 0 {  
        print!("{} is positive", n);  
    } else {  
        print!("{} is zero", n);  
    }  
}
```

5 is positive

Let Statements

- By default, Rust variables are immutable
 - Usage checked by the compiler
- **mut** is used to declare a resource as mutable.

```
fn main() {  
    let a: i32 = 0;  
    a = a + 1;  
    println!("{}", a);  
}
```

```
fn main() {  
    let mut a: i32 = 0;  
    a = a + 1;  
    println!("{}", a);  
}
```

Compile error

Let Statements

```
fn main() {  
    let x = 5;  
  
    let x: i32 = 5; //type annotation  
  
    let mut x = 5; //mutable x: i32  
    x = 10;  
}
```

Using Mutation

- Mutation is useful when performing iteration
 - As in C and Java

```
fn fact(n: u32) -> u32 {  
  let mut x = n;  
  let mut a = 1;  
  loop {  
    if x <= 1 { break; }  
    a = a * x;  
    x = x - 1;  
  }  
  a  
}
```

infinite loop
(break out)

Data: Scalar Types

- Integers
 - `i8`, `i16`, `i32`, `i64`, `isize`
 - `u8`, `u16`, `u32`, `u64`, `usize`
 - Characters (unicode)
 - `char`
 - Booleans
 - `bool` = { `true`, `false` }
 - Floating point numbers
 - `f32`, `f64`
 - Note: arithmetic operators (+, -, etc.) *overloaded*
-
- Machine word size
- Defaults (from inference)

Fun Fact

- The original Rust compiler was written in **OCaml**
 - Betrays the sentiments of the language's designers!
- Now the Rust compiler is written in ... **Rust**
 - How is this possible? Through a process called **bootstrapping**:
 - The first Rust compiler written in Rust is compiled by the Rust compiler written in OCaml
 - Now we can use the binary from the Rust compiler to compile itself
 - We discard the OCaml compiler and just keep updating the binary through self-compilation
 - So don't lose that binary! 😊

CMSC 330: Organization of Programming Languages

Ownership, References, and Lifetimes in Rust

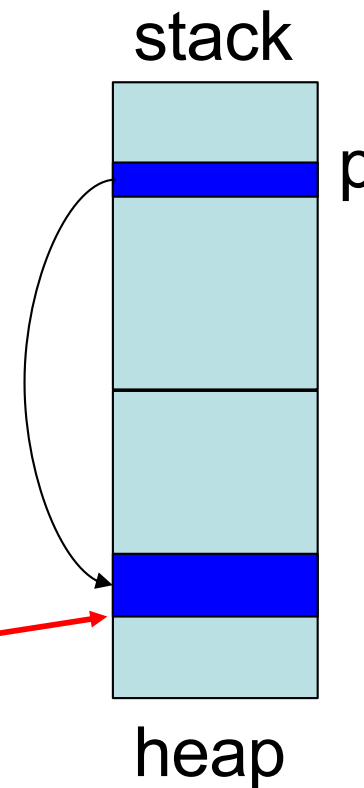
Memory: the Stack and the Heap

- 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

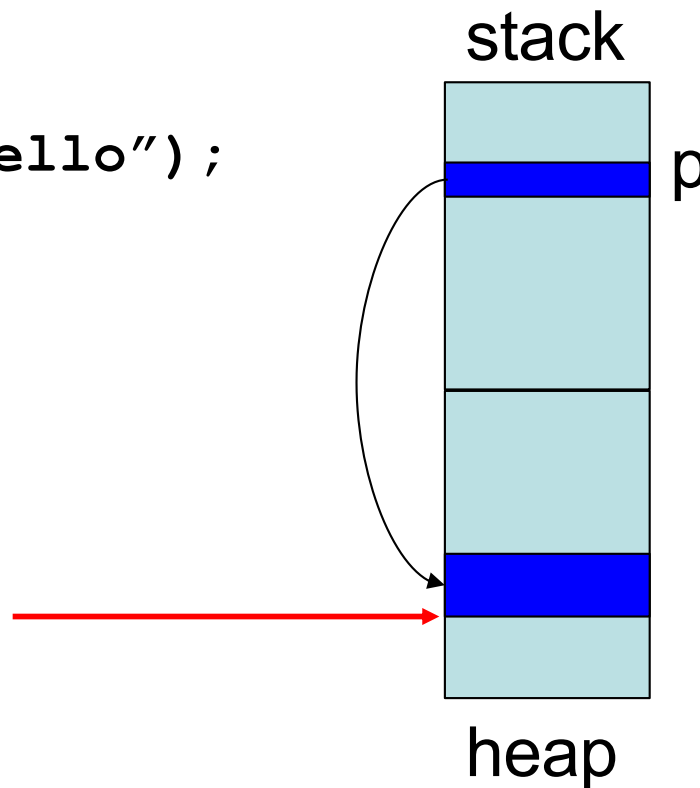
GC-less Memory Management, Safely

- 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

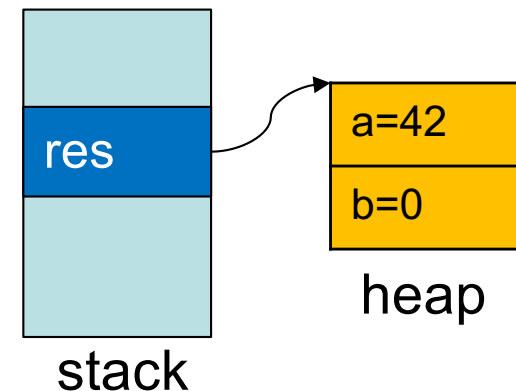


Ownership

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()
```

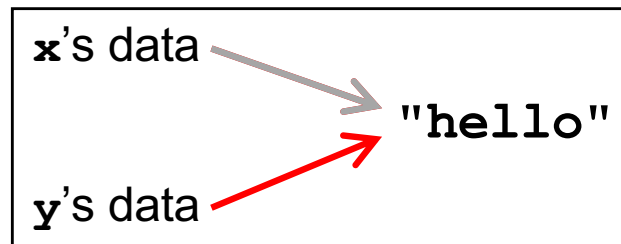
- **s** is the *owner* of this data
 - When **s** goes out of scope, its **drop** method is called, which frees the data

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



*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 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