CMSC330: Rust Basics

Chris Kauffman

Last Updated:
Thu Nov 16 04:22:46 AM EST 2023

Logistics

Reading

The Rust Programming Language

- Official tutorial guide from Rust foundation
- Chapters 1-10 should be sufficient for the course

Assignments

- Project 6: Lambda Calculus Interpreter, Due 15-Nov
- Project 7: Rust Basics, later this week
- **No class on Tue 21-Nov by order of President Pines**

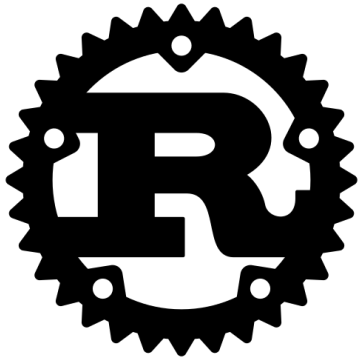
Goals

- Introduction to Rust
- Language Features / Relations
- Memory Ownership

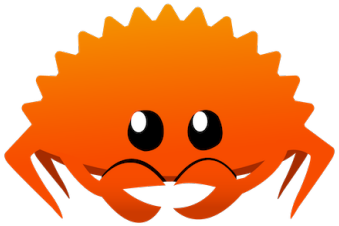
Announcements: None

A Short History of Rust

Graydon Hoare according to Crunchbase



Rust Logo



Rustacean "Ferris", mascot

- Started ~2006 as a side project by Graydon Hoare while working at Mozilla (makes of Firefox and other fine tools)
- Started after a software failure incapacitated an elevator at Hoare's apartment requiring him to climb 21 flights of stairs and inspiring a desire for a "safe" programming language
- Compiler originally written in OCaml, became self-hosting in 2010
- After Mozilla divested developers in 2020, Rust Foundation was created to manage language and community
- Stack Overflow Developer Survey named Rust the "most loved programming language" every year from 2016 to 2023 (Wikip)
- Hoare named Rust "... after a group of remarkably hardy fungi" (MIT Tech Review)

Exercise: Collatz Computation An Introductory Example

- `collatz.rs` prompts for an integer and computes the Collatz Sequence starting there
- The current number is updated to the next in the sequence via

  ```
  if cur is EVEN cur=cur/2; else cur=cur*3+1
  ```

- This process is repeated until it converges to 1 (mysteriously) or the maximum iteration count is reached
- The code demonstrates a variety of Python features and makes for a great crash course intro
- With a neighbor, study this code and identify the features you should look for in every programming language

Exercise: Collatz Computation An Introductory Example

```rust
// collatz.rs:
use std::io;
use std::str::FromStr;

const VERBOSE : bool = true;

fn collatz (start: i32, maxsteps: i32)
            -> (i32,i32)
{
  let mut cur = start;
  let mut step = 0;
  if VERBOSE {
    println!("start: {start}");
    println!("Step  Current");
    println!("{step:3} {cur:5}");
  }
  while cur != 1 && step < maxsteps {
    step += 1;
    if cur % 2 == 0{
      cur = cur / 2;
    }
    else{
      cur = cur*3 + 1;
    }
    if VERBOSE {
      println!("{step:3} {cur:5}");
    }
  }
  (cur,step)              // return val
  // return (cur,step); // alt
}
```

```rust
fn main() { // entry point
  println!("Collatz start val: ");
  let mut instr = String::new();
  match io::stdin().read_line(&mut instr) {
    Err(why) => panic!("Input failed: {}",why),
    Ok(_) => {}                    // freak on error
  };                               // proceed on an ok
  instr.pop();                     // remove trailing \n
  let start = match i32::from_str(instr.as_str()) {
    Err(why) => panic!("Bad int '{instr}': {}",why),
    Ok(anint) => anint             // freak on error
  };                               // o/w return the int
  let (last,steps) = collatz(start, 500);
  println!("Reached {last} after {steps} iters");
}
```

Look for... Comments, Statements/Expressions, Variable Types, Assignment, Basic Input/Output, Function Declarations, Conditionals, Iteration, Aggregate Data, Library System

Answers: Collatz Computation An Introductory Example

- ☒ Comments: `// comment to end of line`
- ☒ Expressions `x+1 a&&b t<m`, Statements `if xyz { ... }` or `println!("Hi");` statement lines end with semicolons
- ☒ Variable Types: `i32` integer, boolean, some types mentions others inferred
- ☒ Assignment: via `let x = expr;` or `let mut x = expr;` or `x = 3*x+1;`
- ☒ Basic Input/Output: `println!()` and `...` oh boy... `io::stdin.read_line()`
- ☒ Function Declarations:

  `fn funcname(param1: type1, param2: type2) -> RetType {`
- ☒ Conditionals (if-else): `if cond { ...} else {...}` ; also `match{ }` conditions
- ☐ Iteration (loops): clearly `while cond {...}`, also `for iter {}`
- ☐ Aggregate data (arrays, records, objects, etc): `(rust,has,tuples)` and `Variant(types)`
- ☐ Library System: `use std::io;` is like `import std.io`

Compile and Run

```
>> rustc collatz.rs

>> file collatz
collatz: ELF 64-bit LSB pie executable, x86-64,
version 1 (SYSV), dynamically linked, ...

>> collatz
Collatz start val:
10
start: 10
Step  Current
  0    10
  1     5
  2    16
  3     8
  4     4
  5     2
  6     1
Reached 1 after 6 iters

>> ./collatz
Collatz start val:
apple
thread 'main' panicked at collatz.rs:66:17:
Bad int 'apple': invalid digit found in string
note: run with `RUST_BACKTRACE=1` environment
variable to display a backtrace
```

- `rustc` compiles code with a `main()` method to executables named after the file
- `collatz` fails on bad input like `apple` via the `panic!()` macro though code running an elevator may wish to take a different tack...

Rust's Prime Directive: Be Safe!

- Avoid memory bugs at all costs
- Provide mechanisms for error handling but force programs to contend with errors
- If failing, fail predictably

Memory Safe Languages

- You may be told that Rust is a memory safe language; this is true and often said to contrast it to C/C++ which may segfault as they give the power of unrestricted pointer operations
- You might respond to the speaker that there are a few other memory safe languages such as Java, Python, OCaml, Scheme, Racket, Clojure, Shell Script, Haskell, Fortran, Javascript, etc. and basically all other languages that don't have unrestricted pointer operations
- You might indicate to the speaker that perhaps they meant Rust is memory safe without using a Garbage Collector which is unusual
- You might end by mentioning that if being memory safe is good and lots of PLs have that quality, having a Garbage Collector might also be good and make a language more usable

Borrowing Ideas from C/C++

- **No Garbage Collector**: GC costs at runtime so avoid it
  - C/C++ handle this with manual management, e.g. `malloc()/free()` or `new / delete`
  - Rust follows a different model
- Aim for "zero cost abstractions": high-level looking code compiles down to very efficient machine instructions, no runtime penalties
- Use of the `<T>` syntax for generic / templated code
- Similar syntax for namespace navigation

  `some::package::file::run_function(a,b)`
- Shared with C++: allow operator overloading so that `a+b` can be overloaded for any types
- Shared with C++: variety of ways to pass parameter, as is, as refs, as mutable refs, etc.
- Shared with C++: add a LOT of stuff to the language; small, tightly integrated features are less fun than playing with everything and the kitchen sink

Borrowing Ideas from OCaml

- Default to immutable data as it is more easily shared and enables concurrency more readily
- Explicitly label data as `mut` to indicate mutability which comes with benefits and costs
- Strongly typed
- Some degree of type inference supported but certain places, particularly function signatures, require explicit typing
- Some degree of polymorphism supported though the model is closer to C++ Templates / Java Generics
- Support Pattern `match`-ing with rich variant/algebraic types, called `enum` in Rust; use these in the standard library

Borrowing Ideas from Python

- Provide a featurful array-like data structure (List in Python, Vector in Rust)
- Favor iterators in `for` loops and ensure that most standard container types support them for ease
- Like Python (and Java), makes use of code annotations such as `#[test]` to denote a function is a test case or `#[derive(Debug)]` to automatically derive some functionality for a data type associated with debugging

Borrowing Ideas from Java

- Uses method dispatch:
- Rust is NOT object-oriented, but then again Gosling would have removed class inheritance from Java given a second chance
- Java also features Interfaces which are a collection of methods implemented by a class
- Rust follows this model: data types `impl` collections of methods referred to as **Traits** allowing the data type to be used any place something with the given trait is present

Borrowing Ideas from Lisp

- Provide for powerful macro creation that enables manipulation of the syntax tree during compilation
- Macros like `println!(...)` aren't functions, rather they generate code in place which can make things more efficient and allow for compile time safety with convenience such as in `println!("x:  {x}")`

Cautions and Disclosures

Cautions

- Rust is 17 years old with wide public attention for $<$10 years
- During that time it has undergone radical changes with much breakage to older code, a trend that is likely to continue
- It combines many features from many other languages
- It's only feature of real note is its memory model: no Garbage Collector, manage memory at compile time as possible

Disclaimer

I don't know Rust particularly well but I don't like what I see. In the name of safety, it makes the creation of linked data structures like lists and trees nearly impossible. I don't think this is a good tradeoff and would not select Rust for my projects at this time.

I will try hard not to left my negative view overly influence our discussion as there are still interesting things to learn.

But it doesn't have a ruddy REPL. WTF^{M}?

Ownership of Memory Locations

> *Ownership is Rust's most unique feature and has deep implications for the rest of the language. It enables Rust to make memory safety guarantees without needing a garbage collector, so its important to understand how ownership works.*
> – *The Rust Programming Language, Ch 4*

Also from the book:

- Each value (memory block) in Rust has an owner
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped (de-allocated)

Ownership Relates to Scoping

- PLs provide bindings of names to values in memory
- Each PL has semantics about when names go out of scope and what becomes of the memory bound to it
- Below example shows a simple example in 3 related languages
- Rust is similar to others with Stack/Heap allocation BUT detects when values are no longer reachable and immediately de-allocates them
- This strategy has myriad consequences that we'll discuss

```c
// C version
void print_str(){
  int i = 5;
  char s[6] = "hello";
  char *h = malloc(6);
  strcpy(h,"there");
  printf("%d %s %s\n",
         i,s,h);
}
// i stack-allocated
// s stack-allocated
// automatically de-allocated
// h heap-allocated
// h out of scope: heap ref
// is lost, memory leak
```

```java
// Java version
public static void printStr(){
  int i = 5;
  String s = "hello";
  String h = new String("there");

  System.out.printf("%d %s %s\n",
                    i,s,h);
}
// i stack-allocated
// s refs const
// automatically de-allocated
// h heap-allocated
// h out of scope, subject
// to be GC'd later
```

```rust
// Rust Version
fn printStr(){
  let i = 5;
  let s = "hello";
  let h = String::new("there");

  println!("{i} {h} {s}")

}
// i stack-allocated
// s refs static str
// automatically de-allocated
// h heap-allocated
// h out of scope, immediately
// "dropped" to de-allocate it
```

Ownership Basics

To get a start on Ownership, examine `ownership_basics.rs` which has a series of 4 examples

1. `i32` integers as params
2. Broken `String` ownership
3. Working `String` ownership with cloning
4. Working `String` ownership with references

These will start to give a sense of the rules the Rust compiler enforces on ownership

```
>> rustc ownership_basics.rs

>> ./ownership_basics
2 plus 3 is 5
3 plus 2 is 5
two plus three is two_three
two plus three is three_two
```

Ownership Basics 1 / 4: Copy-able Values

- Some types like `i32` (32-bit signed integers) copy their bits when assigned or passed as parameters[1]
- Identical semantics to C / Java / OCaml
- Copying means everyone owns distinct copies

```
1  // ownership_basics: working int version
2  fn add2(x: i32, y: i32) -> i32{
3    let z = x+y;
4    return z;
5  }
6  fn show_add() {                    // ALWAYS WORKS
7    let a = 2;                       // allocate ints
8    let b = 3;
9    let ab = add2(a,b);              // pass ints as params
10   let ba = add2(b,a);              // due to copying, retain ownership
11   println!("{a} plus {b} is {ab}");
12   println!("{b} plus {a} is {ba}");
13 }
```

[1] Rust denotes this "copyable" quality with its Copy Trait which is implemented by `i32` the type of integers. We'll look at Traits and supporting them in the near future though likely not Copy.

Ownership Basics 2 / 4: Problems

- Strings in Rust are heap-allocated, passed as pointers to the heap location just as in C / Java / OCaml
- **Only one owner of String** can exist and ownership can change hands
- This breaks the code below

```
1  // ownership_basics: string append Version 1 (broken)
2  fn show_append() {
3    let s = String::from("two");  // allocate strings
4    let t = String::from("three");
5    let st = append2(s,t);        // append2() assumes ownership of s and t
6    let ts = append2(t,s);        // ownership lost and compiler forbids re-use
7    println!("{s} plus {t} is {st}");
8    println!("{s} plus {t} is {ts}");
9  }
10 fn append2(x: String, y: String) -> String{
11   let mut z = String::new();
12   z.push_str(&x);
13   z.push_str(&y);
14   return z;
15 } // x,y now dropped and de-allocated
```

Ownership Basics 2.5 / 4: Compiler Errors

`rustc ownership_basics.rs` generates some loud errors

```
>> rustc ownership_basics.rs
error[E0382]: use of moved value: `t`
  --> ownership_basics.rs:35:20
   |
33 | let t = String::from("three");
   |     - move occurs because `t` has type `String`, which does not implement
   |       the `Copy` trait
34 | let st = append2(s,t);        // append2() assumes ownership of s and t
   |                    - value moved here
35 | let ts = append2(t,s);        // ownership lost and compiler forbids re-use
   |                  ^ value used here after move
```

These are frequent in Rust development, requires learning to follow the compiler and "borrow checker" rules

Ownership Basics 3 / 4: Cloning

A fix for the ownership problems is to Clone the Strings using the `clone()` method from its Clone Trait, identical in name and semantics to Java's idea of Clone.

```
1  // ownership_basics: string append Version 2 (works via cloning)
2  fn show_append() {
3    let s = String::from("two");
4    let t = String::from("three");
5    let st = append2(s.clone(),t.clone()); // append2() gets its own copies
6    let ts = append2(t.clone(),s.clone()); // of s and t
7    println!("{s} plus {t} is {st}");
8    println!("{s} plus {t} is {ts}");
9  }
10 fn append2(x: String, y: String) -> String {
11   let mut z = String::new();
12   z.push_str(&x);
13   z.push_str("_");
14   z.push_str(&y);
15   return z;
16 }
```

Cloning works but is dissatisfying as data must be duplicated every time a function is called. Rust provides a more efficient alternative.

Ownership Basics 4 / 4: References

A more efficient fix for the ownership problem is to adjust the function parameters and call site to use a **Reference** with notation

- `&myvar` at a call site or in an assignment generates a reference to `myvar`
- `param:&String` for a function parameter type of `param` is a String reference

```
1  // ownership_basics: string append Version 3 (works via refs)
2  fn show_append() {
3    let s = String::from("two");
4    let t = String::from("three");
5    let st = append2(&s,&t);          // append2() gets references to s/t
6    let ts = append2(&t,&s);          // show_append() retains ownership
7    println!("{s} plus {t} is {st}");
8    println!("{s} plus {t} is {ts}");
9  }
10 fn append2(x: &String, y: &String) -> String {
11   let mut z = String::new();     // params x,y are refs to String
12   z.push_str(&x);
13   z.push_str("_");
14   z.push_str(&y);
15   return z;
16 }
```

Working with refs is as essential to Rust as pointers are to C

References

Reference in Rust use the & syntax as in

```
{
  call_func(&some_var);          // pass a ref
  let x = &some_var;             // assign a ref
}
{
  let myvar : &some_type = ...; // variable has ref type
}

fn myfunc(param: &some_type) {} // param has ref type
```

- Refs are like pointers in they give access to the data pointed at
- They differ from full pointers in that they do not give the ability to end the life of a memory block which is restricted to the owner of the block
- Refs allow **borrowing** of memory blocks

Exercise: Motivating References from C

What is wrong with the following C program

```
1  void use_arr(int *arr, int len){
2    printf("arr: [");
3    for(int i=0; i<len; i++){
4      printf("%d ",arr[i]);
5    }
6    printf("]\n");
7  }
8
9  void use_up_arr(int *arr, int len){
10   printf("arr: [");
11   for(int i=0; i<len; i++){
12     printf("%d ",arr[i]);
13   }
14   printf("]\n");
15   free(arr);
16 }
```

```
17 int main(int argc, char *argv[]){
18   int len = 5;
19   int *a = malloc(sizeof(int) * len);
20   for(int i=0; i<len; i++){
21     a[i] = (i+1)*10;
22   }
23
24   use_arr(a,len);
25   use_up_arr(a,len);
26   use_arr(a,len);
27
28   return 0;
29 }
```

Answers: Motivating References from C

```c
// c_mem_problems.c:
void use_up_arr(int *arr, int len){
  printf("arr: [");
  for(int i=0; i<len; i++){
    printf("%d ",arr[i]);
  }
  printf("]\n");
  free(arr);
}
int main(int argc, char *argv[]){
  int len = 5;
  int *a = malloc(sizeof(int) * len);
  for(int i=0; i<len; i++){
    a[i] = (i+1)*10;
  }

  use_arr(a,len);        // okay
  use_up_arr(a,len);     // free'd
  use_arr(a,len);        // not okay

  return 0;
}
```

- A classic use after free error
- 2nd call to `use_arr()` accesses a `free()`'d block
- Java / OCaml don't allow user free()'s, GC does this

Exercise: How Rust "Fixes" the C Mistakes

```rust
// rust_owner_problems.rs:
fn use_arr(arr: Vec<i32>){
  print!("arr: [");   // arr owned
  for x in arr{
    print!("{x} ");
  }
  println!("]");
}                     // arr dropped

fn use_up_arr(arr: Vec<i32>){
  print!("arr: [");   // arr owned
  for x in arr{
    print!("{x} ");
  }
  println!("]");
}                     // arr dropped

fn main(){
  let len = 5;
  let mut a = vec![];
  for i in 0..len {
    a.push((i+1)*10);
  }
  use_arr(a);     // ownership lost
  use_up_arr(a); // compiler error
  use_arr(a);
}
```

Side Notes

- Check out `vec![]` macro to create Vector
- `push(x)` to add on to a vector
- Iteration over a range via `start..stop`

Ownership

- Rust fixes the C problem by passing ownership of a memory block to functions by default
- Once passed into the first function, a is lost and cannot be used

```
>> rustc rust_owner_problem.rs
error[E0382]: use of moved value: `a`
  --> rust_owner_problem.rs:29:14
25 |   use_arr(a);                    // ownership lost
   |           - value moved here
26 |   use_up_arr(a);                 // compiler error
   |               ^ value used here after move
```

What's the fix for this in Rust

Answers: How Rust "Fixes" the C Mistakes

```rust
// rust_owner_borrow.rs:
fn use_arr(arr: &Vec<i32>){
  print!("arr: [");  // arr borrowed
  for x in arr{
    print!("{x} ");
  }
  println!("]");
}                    // arr not dropped

fn use_up_arr(arr: &Vec<i32>){
  print!("arr: [");  // arr borrowed
  for x in arr{
    print!("{x} ");
  }
  println!("]");
}                    // arr not dropped

fn main(){
  let len = 5;
  let mut a = vec![];
  for i in 0..len {
    a.push((i+1)*10);
  }
  use_arr(&a);       // ownership retained
  use_up_arr(&a);    // ownership retained
  use_arr(&a);       // ownership retained
}                    // a is now dropped
```

Fixes

```
27 |   use_arr(a);          // ownership lost
   |           - value moved here
28 |   use_up_arr(a);       // compiler error
   |              ^ value used here after move
   |

note: consider changing this parameter type in
function `use_arr` to borrow instead if owning the
value isn't necessary
```

- Adjust parameter to be reference types
- Adjust calls pass references to functions
- References do not affect ownership nor cause drops (deallocation)

Mutable References

- Synatx `&mut x` may be used in place of `&` to indicate reference may be mutated
- Multi-threaded programs are restricted to use only 1 mutable reference at a time OR as many immutable refs as desired

```
1  // mut_ref.rs:
2  fn add_some(vec: &mut Vec<i32>){
3    for i in 1..=3 {
4      vec.push(i);                  // alters param vector
5    }
6  }
7  fn main(){
8    let mut v = vec![10,11];
9    add_some(&mut v);               // pass with ability to mutate
10   add_some(&mut v);               // and again
11   println!("{:?}",v);             // use hand debug print formating
12 }
```

```
>> rustc mut_ref.rs
>> ./mut_ref
[10, 11, 1, 2, 3, 1, 2, 3]
```

`str` and `String`

The `str` type, also called a string slice, is the most primitive string type. It is usually seen in its borrowed form, `&str`. It is also the type of string literals, `&'static str`. . . . A `&str` is made up of two components: a pointer to some bytes, and a length.

– *Rust Docs for str*

The `String` type is the most common string type that has ownership over the contents of the string. It has a close relationship with its borrowed counterpart, the primitive `str`.

– *Rust Docs for String*

Defining Data Types: `struct`

Implementing Methods: `impl`

Traits: like Java Interfaces

Trait	Type	??	Notes
New	New	Yes	Can implement MyTrait for MyData
Exists	New	Yes	Can implement Iterator for MyData
New	Exists	Yes	Can implement MyTrait for i32
Exists	Exists	No	Cannot implement Iterator for i32