# CMSC330: Rust Basics

Chris Kauffman

*Last Updated:*
*Thu Dec 7 09:14:29 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 8: Rust Spell Checker
  Released Due Mon 11-Dec
- Quiz 4: Fri 08-Dec during Discussion

## Goals

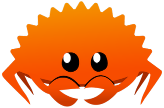
- ☒ Introduction to Rust
- ☒ Language Features / Relations
- ☒ Memory Ownership
- ⊟ Datatypes and Traits
- ☐ Extras

# 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

```
// 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
}
```

```
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$^{\text{M}}$?

# 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

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

[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

```
// ownership_basics: string append Version 1 (broken)
fn show_append() {
  let s = String::from("two");  // allocate strings
  let t = String::from("three");
  let st = append2(s,t);        // append2() assumes ownership of s and t
  let ts = append2(t,s);        // ownership lost and compiler forbids re-use
  println!("{s} plus {t} is {st}");
  println!("{s} plus {t} is {ts}");
}
fn append2(x: String, y: String) -> String{
  let mut z = String::new();
  z.push_str(&x);
  z.push_str(&y);
  return z;
} // 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.

```rust
// ownership_basics: string append Version 2 (works via cloning)
fn show_append() {
  let s = String::from("two");
  let t = String::from("three");
  let st = append2(s.clone(),t.clone()); // append2() gets its own copies
  let ts = append2(t.clone(),s.clone()); // of s and t
  println!("{s} plus {t} is {st}");
  println!("{s} plus {t} is {ts}");
}
fn append2(x: String, y: String) -> String {
  let mut z = String::new();
  z.push_str(&x);
  z.push_str("_");
  z.push_str(&y);
  return z;
}
```

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

```rust
// ownership_basics: string append Version 3 (works via refs)
fn show_append() {
  let s = String::from("two");
  let t = String::from("three");
  let st = append2(&s,&t);        // append2() gets references to s/t
  let ts = append2(&t,&s);        // show_append() retains ownership
  println!("{s} plus {t} is {st}");
  println!("{s} plus {t} is {ts}");
}
fn append2(x: &String, y: &String) -> String {
  let mut z = String::new();    // params x,y are refs to String
  z.push_str(&x);
  z.push_str("_");
  z.push_str(&y);
  return z;
}
```

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

```
void use_arr(int *arr, int len){
  printf("arr: [");
  for(int i=0; i<len; i++){
    printf("%d ",arr[i]);
  }
  printf("]\n");
}

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);
  use_up_arr(a,len);
  use_arr(a,len);

  return 0;
}
```

# **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_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

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

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

# Vectors

Vectors are Rust's goto data structure, an extensible array in the same vein as Python Lists / Java `ArrayList`

```
// vec_demo.rs:
  let mut v : Vec<i32> = Vec::new();   // required type annotation for new()
  v.push(10); v.push(20); v.push(30);  // extend vector
  let mut v2 = vec![10,20,30];         // vec! macro is commonly used
  v2[1] = 40;                          // standard [] indexing
  println!("v2[1]: {}", v2[1]);

  // Oh so many ways to iterate
  for x in &v2 {                       // implicit slice
    print!("{x} ");
  }
  for x in &v2[..] {                   // explicit slice
    print!("{x} ");
  }
  for x in v2.iter() {                 // explicit iterator
    print!("{x} ");
  }
  for i in 0..v2.len(){                // traditional via range
    print!("v2[{i}]: {}  ",v2[i]);
  }
  for (i,x) in v2.iter().enumerate() { // iterator + index
    print!("v2[{i}]: {x}  ");
  }
  // Vectors are Generic / Polymorphic; type annotation below is optional
  let vs1 : Vec<&str> = vec!["katz:","all","your","bass","..."];
  let vs2             = vec!["are:","belong","to","us"];
...
```

# Slices in Rust

- Vectors support **slices**, a borrowed portion of a data structure
- Allows for efficient borrowing of portions of Vectors / DS's
- Syntax for slice types is... interesting

| Type | Rust Parlance | Elems | C Equiv |
|---|---|---|---|
| `&[i32]` | slice of `i32` | `i32` | array of `int` |
| `&[&str]` | slice of `&str` | `&str` | array of `char[]` |
| `&str` | string slice (bleck!) | `char` | plain `char[]` |

```
// vec_demo.rs: SLICES: borrowed portions of vectors
  let v100: Vec<i32> = (0..100).collect(); // range to vector
  println!("v100[50]: {}",v100[50]);

  let v20_40: &[i32] = &v100[20..40];  // SLICE of vector w/ explicit
  for x in v20_40 {                    // type annotation
    print!("{x} ");
  }
  let v50_70 = &v100[50..70];          // SLICE omitting type annotation
  for x in v50_70 {
    print!("{x} ");
  }
  let vs =                             // vector of primitive str
    vec!["katz:","all","your","bass","are","belong","to","us"];
  let sl_vs : &[&str] = &vs[1..4];     // SLICE of vector, type annotation optional
  for x in sl_vs {                     // iterate over slice
    print!("{x} ");
  }
...
```

# `str` and `String` 1 / 2

*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*

# str and String 2 / 2

```
// string_vs_str.rs:
fn main(){
  let a = "hello world";                  // primitive str not meant to grow in size
  let b = String::from("hello world");    // standard String buffer of characters

  let mut c = "goodbye mut";              // both can be made mutable
  let mut d = String::from("goobye mut");

  for (i,ch) in c.chars().enumerate() {   // iterate over characters in a str
    println!("c[{i}]: {ch}");             // .iter() pops of chars while
  }                                       // .enumerate() gives index,char pairs
  for (i,ch) in d.chars().enumerate() {   // likewise for characters in a String
    println!("d[{i}]: {ch}");
  }

  let cs1 : &str = &c[2..11];             // slice of &str is &str
  let ds1 : &str = &d[2..11];             // slice of String is &str

  let clen = c.len();                     // length methods for both str and String
  let dlen = d.len();

  // c.push_str(" again");      // no ability to grow a str
  d.push_str(" again");         // methods for String

  // c.replace_range(0..1,"And ");  // no supported method as str isn't meant to grow
  d.replace_range(0..0,"And ");     // grow
}
```

# Ownership and Data Structures

- `vec_owner.rs` provides a demo of several uses for how ownership changes wrt to the Vector data structure
- In some cases, ownership of data transfers to the Vector: memory block will be de-allocated when the Vector is de-allocated
- In other cases, Vector only contains a reference to data owned elsewhere
- Some versions and lines are commented out as they are rejected by the compiler: **being able to explain why these won't compile is good practice for. . .**
  - Writing your own code
  - Exam Questions which might ask for an explanation

# Exercise: Ownership within Data Structures

Data structures like Vectors can be composed of

- Owned data: de-allocated when DS is dropped
- Borrowed data: data persists when DS is dropped

```
// vec_ownership.rs:
  //////// A ////////
  let s = String::from("abc");  // string is born
  let mut v = vec![];           // vec is born
  v.push(&s);                   // string ref from vec
  println!("s: {s}");           // okay as a ref is passed
  println!("v[0]: {}",v[0]);    // okay as s is in scope

  //////// B ////////
  let s = String::from("abc");  // string is born
  let mut v = vec![];           // vec is born
  v.push(s);                    // vec now owns string
  // println!("s: {s}");        // ERROR: s lost ownership
  println!("v[0]: {}",v[0]);    // okay as v owns string
```

What is the inferred type of `v` in A / B examples?

# **Answers**: Ownership within Data Structures

Data structures like Vectors can be composed of

- Owned data: de-allocated when DS is dropped
- Borrowed data: data persists when DS is dropped

What is the inferred type of v in A / B examples?

```
// vec_ownership.rs:
  //////// A ////////
  let s = String::from("abc");  // string is born
  let mut v = vec![];           // vec is born
  v.push(&s);                   // string ref from vec
  println!("s: {s}");           // okay as a ref is passed
  println!("v[0]: {}",v[0]);    // okay as s is in scope

  // v: Vec<&String> "Vector of String Refs"

  //////// B ////////
  let s = String::from("abc");  // string is born
  let mut v = vec![];           // vec is born
  v.push(s);                    // vec now owns string
  // println!("s: {s}");        // ERROR: s lost ownership
  println!("v[0]: {}",v[0]);    // okay as v owns string

  // v: Vec<String> "Vector of Strings", owns strings
```

# Exercise: Issues in DS Ownership

- Examples C-F of code involve a Vector owning a String
- Will each example compile or will `rustc` find ownership issues?

*This is tricky but is good practice for exam questions*

```
// vec_ownership.rs:
//////// C ////////
fn make_vec() -> Vec<String>{
  let s = String::from("abc");
  let mut v = vec![];
  v.push(s);
  return v;
}
fn use_make_vec() {
  let v = make_vec();
  println!("v[0]: {}",v[0]);
}

//////// D ////////
fn make_vec() -> Vec<&String>{
  let s = String::from("abc");
  let mut v = vec![];
  v.push(&s);
  return v;
}
fn use_make_vec() {
  let v = make_vec();
  println!("v[0]: {}",v[0]);
}
```

```
// vec_ownership.rs:
//////// E ////////
fn make_vec() -> Vec<&'static String>{
  let s = String::from("abc");
  let mut v = vec![];
  v.push(&s);
  return v;
}
fn use_make_vec() {
  let v = make_vec();
  println!("v[0]: {}",v[0]);
}

//////// F ////////
fn vec_ref(s: &String) -> Vec<&String>{
  let mut v = vec![];
  v.push(s);
  return v;
}
fn use_vec_ref(){
  let s = String::from("abc");
  let v = vec_ref(&s);
  println!("v[0]: {}",v[0]);
}
```

# **Answers:** Issues in DS Ownership 1 / 2

```
// vec_ownership.rs:
//////// C //////// OKAY
fn make_vec() -> Vec<String>{   // CORRECT: return vec of String
  let s = String::from("abc");  // string is born
  let mut v = vec![];           // vec is born
  v.push(s);                    // string moves into vec, vec owns it
  return v;                     // safe to return
}
fn use_make_vec() {
  let v = make_vec();
  println!("v[0]: {}",v[0]);
}

//////// D //////// ERROR
fn make_vec() -> Vec<&String>{  // ERROR: unable to resolve types
  let s = String::from("abc");  // string is born within function
  let mut v = vec![];           // vec is born
  v.push(&s);                   // ref to string from vec, string still owned by s
  return v;                     // ERROR: returning from func would de-allocate s
}                               // but refs remain to it in v
fn use_make_vec() {
  let v = make_vec();
  println!("v[0]: {}",v[0]);
}
```

# **Answers:** Issues in DS Ownership 2 / 2

```
// vec_ownership.rs:
//////// E //////// ERROR
fn make_vec() -> Vec<&'static String>{
  let s = String::from("abc");  // for (D), compiler suggests adding &'static
  let mut v = vec![];           // which is a "lifetime" for the string. This
  v.push(&s);                   // does not help in this case as the fundamental
  return v;                     // error is that s's String needs to be owned by
}                               // the escaping vector
fn use_make_vec() {
  let v = make_vec();
  println!("v[0]: {}",v[0]);
}

//////// F //////// OKAY
fn vec_ref(s: &String) -> Vec<&String>{ // string owned from elsewhere
  let mut v = vec![];                   // vec is born
  v.push(s);                            // vec refers to string, doesn't own it
  return v;                             // safe to return as string isn't owned by v
}
fn use_vec_ref(){
  let s = String::from("abc");  // string is born
  let v = vec_ref(&s);
  println!("v[0]: {}",v[0]);
}
```

# Defining New Data Types: `struct` and `enum`

- Basic data type declaration are laid out like in OCaml
  - `struct` : like an OCaml record (or C `struct`)
  - `enum` : like an OCaml Algebraic (Variant) types
- There are few surprises, just slightly different syntax
- Next few slides demo basic aspects
- Then we'll proceed to Rust's version of methods via `impl`-ementations

# struct : Creating New Types with Fields

```
// struct_enum_demo.rs:
struct Omelet {                 // a new data type with fields
  cook_time: f32,               // floating point field
  is_cooked: bool,              // boolean field
  ingredients: String,          // string field
}

fn use_omelet(){                // demonstrates use of Omelet type
  println!("===USE OMELET====");
  let om = Omelet {             // create an immutable Omelet
    cook_time: 2.5,
    is_cooked: false,
    ingredients: String::from("bacon cheddar")
  };                            // print out a field
  println!("om.ingredients: {}",om.ingredients);

  let mut mom = Omelet {        // create a fully mutable Omelet
    cook_time: 0.0,
    is_cooked: false,
    ingredients: String::from("spinach swiss")
  };
  mom.cook_time += 4.999;       // alter fields
  mom.is_cooked = true;
  mom.ingredients.push_str(" mushroom");
  println!("mom.ingredients: {}",mom.ingredients);
  println!("mom.cook_time: {:.2}",mom.cook_time);
}
```

# enum: New Data Type with Variants

```
enum Breakfast {                // like OCaml's algebraic types, 4 variants of type
  None,                         // Variant with no additional data
  Meager(String),               // Variant with String data
  Hearty(Omelet),               // Variant with Omelet data
  Misc(u32,String)              // Variant with pair of u32 and String
}
use Breakfast::*;               // allow bare names of variants like Meager()

fn breakfast_count(br: &Breakfast) -> u32 {
  return match br {             // match on the variants of Breakfast
    None            => 0,
    Misc (count,_) => *count,   // read about * on your own time
    _               => 1        // all other variants
  };
}
fn use_breakfast(){
  let dog_br = Meager(String::from("kibble"));
  let md_br  = Misc(4,String::from("cups oatmeal"));
  let ck_br  = Hearty(Omelet{cook_time: 5.00, is_cooked: true,
                             ingredients: String::from("ham swiss")});
  let br_vec = vec![dog_br, md_br, ck_br, None ]; // all of Breakfast type
  for (i,x) in br_vec.iter().enumerate() {  // iterate over array handling
    let desc = match x {                     // each variant in a match
      None      => "none",   Meager(_) => "meager",
      Hearty(_) => "hearty", Misc(_,_) => "misc",
    };
    let count = breakfast_count(&x);
    println!("{i}: {desc}, count: {count}");
  }
}
```

# Implementing "Methods" with `impl` Blocks

- Rust is NOT object-oriented but can feel that way due to its data syntax and types which feature the "dot" notation used in OO languages
- Rust favors an approach similar to C++: define data type (`struct`) in one place, define associated functions in another place via `impl` construct
- Rust supports various syntatic "sugar" around `impl` such as "method" invocation
- Several `impl` can exist for a `struct` allowing associated functions to be defined across several files (though this is atypical)

```
struct Omelet {      // a new data type
  ...;
}

impl Omelet {   // "methods" for Omelets
  // construct an Omelet
  fn new(ingr: &str) -> Omelet {
    ...
  }
  // cook an omelet
  fn cook(&mut self, time: f32){
    ...
  }
}
...
impl Omelet{   // MORE "methods" for Omelets
  // add an ingredient
  fn add_ingredient(&mut self, ingr: &str){
    ...
  }
  // check for overcooked
  fn is_overcooked(&self) -> bool {
    return self.cook_time > 8.0;
  }
  // non-"method" function
  fn denver_ingredients() -> &'static str {
    return "ham cheese peppers";
  }
}
```

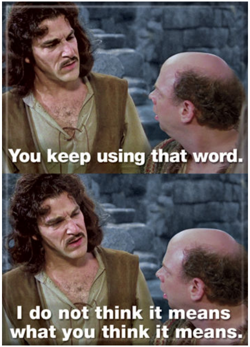
# impl_demo.rs Highlights 1 / 2

```
// impl_demo.rs:
struct Omelet { cook_time: f32, is_cooked: bool, ingredients: String}

impl Omelet {                          // "methods" for Omelet struct
  fn new(ingr: &str) -> Omelet {       // construct an Omelet
    return Omelet{cook_time: 0.0,
                  is_cooked: false,
                  ingredients: String::from(ingr)};
  }
  fn cook(&mut self, time: f32){       // cook an omelet
    self.cook_time += time;            // note the first param: &mut self
    if self.cook_time >= 5.0 {         // which is assigned by the compiler
      self.is_cooked = true;           // to &mut Omelet based on the context
    }
  }
}
fn main(){
  let ingr = Omelet::denver_ingredients(); // invoke function in Omelet namespace
  let mut denver = Omelet::new(ingr);      // typical "constructor" invocation

  Omelet::cook(&mut denver, 0.25);     // direct invocation of cook() function
  denver.cook(0.25);                   // "dot" invocation of cook() function
  denver.add_ingredient(" mushrooms"); // "dot" invocation of function

  while !denver.is_overcooked() {      // literally overdo it
    denver.cook(0.5);
  }

...
```

# Aside: "methods" and Methods

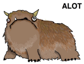
> *Methods are similar to functions: we declare them with the fn keyword and a name, they can have parameters and a return value, and they contain some code thats run when the method is called from somewhere else. Unlike functions, methods are defined within the context of a struct (or an enum or a trait object, which we cover in Chapter 6 and Chapter 17, respectively), and their first parameter is always self, which represents the instance of the struct the method is being called on.*
>
> – *The Rust Programming Language, Ch 5.3*

- Rust "methods" are mostly just functions
- Proper Methods are a family of functions with overriding usually through sub-classing and dynamic dispatch to select one of several possible functions at runtime based on the specific type of the data being used though alternatives exist
- Rust does not have sub-classing / overriding
- It does have a dynamic dispatch facility that is still evolving and requires use of smart pointers like `Box` and Trait Objects



# Traits: Data Supporting Common Operations

- Common idea: data with operations XYZ can be used here (in this function, in this algorithm, in this data structure, etc)
- Common XYZ examples are (1) ability to compare data to sort sorting it and (2) ability to write/read a representation of data to files for saving/loading
- Pure OO-Solution accomplishes this with a class hierarchy, abstract parent classes with concrete child classes extending parents and overriding necessary methods, has downsides acknowledged by folks who know OO well
- Alternative: denote XYZ is present without inheritance
  - `interface` in Java, Type Class in Haskell, Protocol in Clojure
  - **Trait** in Rust
- A Trait is a list of functions that must be present that data can implement to be compatible with other functionalities
- A function or data structure type can be annotated to accept only parameters that implement Traits
- As Java uses Classes a lot, Rust uses Traits alot!



# (A) New Trait / New Datatype Implementations

A reasonably common case: new Datatype and new Trait

```
// trait_impl.rs:
struct Omelet {                    // new datatype Omelet
  cook_time: f32, is_cooked: bool, ingredients: String,
}

impl Omelet {                       // "methods" for Omelet
  fn new(ingr: &str) -> Omelet { ... }
  fn cook(&mut self, time: f32){ ... }
}

trait Updateable {                 // a new trait Updateable
  fn update(&mut self);
}

impl Updateable for Omelet {       // (A) new Trait Updateable, new type Omelet
  fn update(&mut self){
    self.cook(0.25);               // update an Omelet by cooking it a bit
  }
}
```

# (B) Existing Trait / New Datatype Implementations

Common allows New datatypes to use existing Rust functionality

```
impl Iterator for Omelet{        // (B) existing Trait Iterator, new datatype Omelet
  type Item = ();                // iterator returns unit, denoted ()
  fn next(&mut self) -> Option<Self::Item> { // required "method" for Iterator
    if self.is_cooked {                      // when cooked, stop iterating
      return None;
    }
    else {
      self.cook(0.5);            // iterate by cooking a little and returning
      return Some(());           // some of unit
    }
  }
}

impl Display for Omelet{         // (B) existing Trait Display, new datatype Omelet
  fn fmt(&self, f: &mut Formatter<'_>) -> Result {
    return write!(f,"Omelet{{ cook_time: {:.2}, is_cooked: {}, ingredients: {}}}",
                  self.cook_time, self.is_cooked, self.ingredients);
  }
}                                // allows Omelet to be println!()'d

fn main() {
...
  for _ in &mut omelet {                   // iterate on omelet due to Iterator
    println!("Iterating on omelet");
    // println!("omelet: {}",&omelet);   // ERROR: mut ref to omelet already owned above
  }
  println!("Finished omelet: {}",omelet); // print formatted omelet due to Display
...
```

# (C) New Trait / Existing Datatype Implementations

Somewhat common: teach an old data type a new trick

```
trait Updateable {               // a new trait Updateable
  fn update(&mut self);
}

impl Updateable for i32 {        // (C) new Trait Updateable, existing type i32
  fn update(&mut self){
    *self = *self + 1;
  }
}

impl Updateable for String {     // (C) new Trait Updateable, existing type String
  fn update(&mut self){
    self.push('_');
  }
}
```

# (D) Existing Trait / Existing Datatype Implementations

ERROR: prevented by Rust for reasons that are not explained

- Suggested work around is to define a new type that mirrors existing type and use it instead
- Somewhat dissatisfying but so it goes

```
impl Iterator for String {      // (D) exiting Trait Iterator, existing type String
  type Item = char;             // ERROR: this case is not allowed by rust
  fn next(&mut self) -> Option<Self::Item> {
    return self.pop();          // return Some(last_char) or None
  }
}
```

```
>> rustc trait_impl.rs && trait_impl
error[E0117]: only traits defined in the current crate can be implemented for types defined
  --> trait_impl.rs:83:1
   |
83 | impl Iterator for String {      // (D) exiting Trait Iterator, existing type i32
   | ^^^^^^^^^^^^^^^^^^------
   | |                 |
   | |                 `String` is not defined in the current crate
   | impl doesn't use only types from inside the current crate
   |
   = note: define and implement a trait or new type instead

error: aborting due to previous error
```

# Rust Trait Implementation Summary

## Rust Traits

| | Trait | Type | Ok? | Notes |
|---|---|---|---|---|
| A | New | New | Yes | Can implement `MyTrait` for `MyData` |
| B | Exists | New | Yes | Can implement `Iterator` for `MyData` |
| C | New | Exists | Yes | Can implement `MyTrait` for `i32` |
| D | Exists | Exists | Sub | Cannot implement `Iterator` for `i32`<br>Can create new subtype for `i32`<br>to implement `Iterator` |

May return to discussion of extending data types and available functions later time permitting, hopefully discuss the **Expression Problem**

## Java Interfaces Are Similar

| | Interf | Class | Ok? | Notes |
|---|---|---|---|---|
| A | New | New | Yes | Can implement `MyInterf` for `MyClass` |
| B | Exists | New | Yes | Can implement `Iterator` for `MyClass` |
| C | New | Exists | Sub | Must subclass `Object` implement `MyInterf` |
| D | Exists | Exists | Sub | Must subclass `Object` implement `Iterator` |

# Trait Usage

Rust Type system allows for the use of Traits to constrain Generic (polymorphic) functions to accept only data with certain methods

```
fn show_it<T:Display>(thing: T){ // accept any type with Display trait
  println!("The thing is {}",thing);
}
...
fn main(){                      // show use of generic function
  show_it("Hello");
  show_it(1.234);
  show_it(Omelet{cook_time:1.0, is_cooked: false,
                 ingredients: "ham cheese".to_string()});
}

// Alternative syntaxes for constraining types
fn show_it2(thing: impl Display){
  println!("The thing is {}",thing);
}

fn show_it3<T>(thing: T)
where T: Display
{
  println!("The thing is {}",thing);
}
```

NOTE: 3 separate syntaxes available to dictate data must have a certain trait

# Ref in Returns create Lifetime Problems

- A common example of ownership difficulties is the following function from Ch 10.3 of The Rust Programming Language

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

- Because a `&str` (string slice / ref) is returned, the borrow checker is nervous that its underlying owned data could be dropped prior to use.
- `rustc` needs a little re-assurance and is kind enough to ask:

```
>> rustc generic_lifetimes.rs
  --> generic_lifetimes.rs:18:33
   |
18 | fn longest(x: &str, y: &str) -> &str {
   |               ----     ----     ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value,
     but the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
   |
18 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
   |           ++++     ++          ++          ++
```

# Lifetimes as Generic Types

```
// generic on datatype T: constrains v / z to have a type relation
fn add_front<T>(v: &mut Vec<T>, z: T){
  v.push(z);
  v.rotate_right(1);
}

// generic on lifetime 'a: constrains x / y / return to have a lifetime relation
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  if x.len() > y.len() {
    x
  } else {
    y
  }
}
```

- `’a` is a lifetime, `<’a>` indicates generic func on lifetimes
- `x:  &’a str` indicates `x`’s ref’d data has a lifetime at least as long as `’a`
- `y` and the returned `&str` do as well
- Borrow checker assuaged in that returned type will have lifetime at least as long as parameters
  - Obvious in this simple case to smart humans
  - One day the compiler may get that smart too

# ’a Notation as Minimum Lifetimes

- Note that `'a` lifetimes are best interpreted as a “minimum” lifetime, NOT “equal” lifetimes[2]
- Example below shows shorter + longer lifetime data passed to `longest()`

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

// a static string: program-length lifetime
static STAT_STRING : &'static str = "Who wants to live forever?";

fn different_lifetimes() {
  // a string with function-length lifetime
  let string1 = String::from("abcd");

  // compiles and runs fine
  let result = longest(string1.as_str(), STAT_STRING);
  println!("The longest string is {}", result);
}
```

[2] Rust made an unfortunate notation choice as folks coming from OCaml have seen `'a` and will instinctively assume two things that are both `'a` are equal in type

# Linking Structures

Challenge: Define a Singly Linked List with the following "methods"

- `new()` to create an empty list
- `push(data)` to add a new node with given `data` at the head
- `pop()` to remove the head node and return its data
- Works with any data type (generic / polymorphic)

You should feel confident of doing so in

- Java, Python, OCaml

How about in Rust?

# Rust Singly Linked Lists

- `box_linked_list.rs` shows one possibility for a Rust singly-linked list which leans towards mutability
- Uses combination of `enum` for Nodes (a la OCaml) and `struct` for list itself
- Requires use of `Box`, a "smart pointer" which allows heap allocation of arbitrary data
- Requires use of the `mem::replace(dest,src)` function to deal with tricky ownership issues
- Singly linked lists which add/remove from the head are as simple as it gets and require "Chapter 15" functionality
- Doubly Linked Lists, Trees, and more intense data structures require more advanced techniques beyond our scope here

Difficulty with creating Linked Data Structures is the price of the Safety that Rust guarantees

# Highlights of box_linked_list.rs

```
enum Node {                                    // type for nodes in list
  Cons(i32, Box<Node>),
  Nil,
}

struct List {                                  // type for list
  head: Node,
  len:  u32,
}

impl List {
  fn push(&mut self, data: i32){               // add element at head of list
    let stolen =                               // adjust ownership of head to stolen,
      mem::replace(&mut self.head, Nil);       // head becomes Nil momentarily
    self.head = Cons(data, Box::new(stolen)); // assign head to newly allocated node
    self.len += 1;                             // update length
  }

  fn pop(&mut self) -> Option<i32> {           // remove head element and return it or None
    let stolen =                               // steal the head making it Nil for a moment
      mem::replace(&mut self.head, Nil);       // so that it can be owned in match
    match stolen  {                            // match head node
      Nil => return None,                      // head was Nil so remains so
      Cons(data, box_next) => {                // head was Cons
        self.head = *box_next;                 // set head to next node, deref needed
        self.len -= 1;                         // decrement length
        return Some(data);                     // return stolen data
      }
    }
  }
}
```

# Sundries

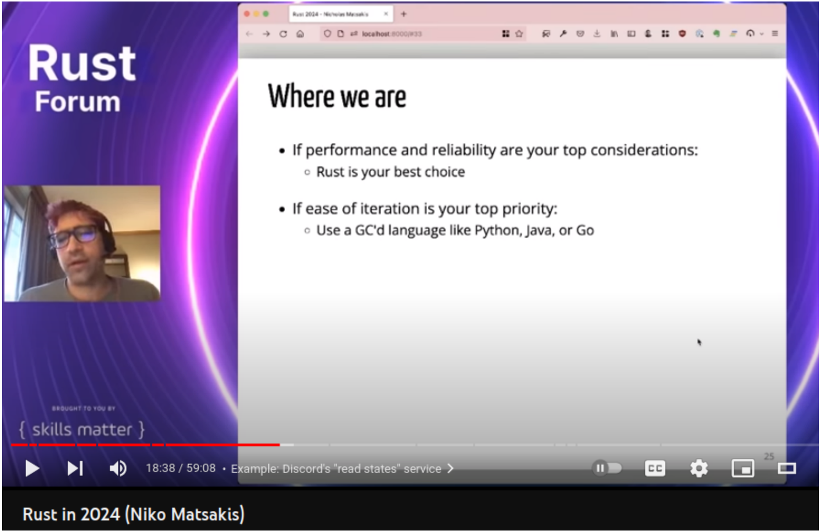
## Arithmetic Overflow Checks `overflow.rs`

- Default `rustc` options augment every `+ - *` operation to check for overflow
- Can be disabled with `rustc -C overflow-checks=off`, increases speed of arithmetic by about 2X on simple benchmarks
- Opposite of C which defaults to no checks but can be enabled with compiler options `gcc -ftrapv`

## Sorted Print `sorted_print.rs`

- Attempted to provide a more complex example of Trait usage in `sorted_print.rs`
- Attempts to create a generic function which accepts an iterable data structure with elements that implements several traits
- Sort the data and print it
- Lots of strange syntax, borrowing problems abound
- Gave up after seeing a 3-year old Stack Overflow post with the same errors and no answers
- Student contributions welcome

# Rustlings on Rust




https://www.youtube.com/watch?v=OuSiuySr6_Q

# Further Study

- Rust is a BIG language and environment
- Immediate next steps in study would be to examine
  - `cargo` the build tool and package manager
  - Functional language features and Closures
- More advanced features of note include
  - "Fearless" Concurrency with Threads and Smart Pointers
  - Macro implementation like with `vec![]` and `println!()`
  - Unsafe layer which is used to implement much of the standard library, which is very safe. We promise.

Or you could wait 10 years to see if Rust stabilizes, perhaps adds a full garbage collector, or collapses under its own complexity giving rise to a new language that Gen-Next rabidly admires and believes is the one true way… at which point you'll be old enough to snort derisively at it and talk about what you had to go through to learn programming

```
(quote "Prediction is very difficult, especially about the future!"
       "-- attributed to Niels Bohr, physicist and Nobel laureate")

(quote "Fashion changes, but style endures." 'Coco 'Chanel)

(quote "God had a deadline. So he wrote it all in Lisp." 'Eternal 'Flame)
```