Rust Programing Language

Why Rust

- High-level language features without performance penalties.
- Program behaviors can be enforced at compile time
  - Built-in dependency management, similar to npm
- Quickly growling ecosystem of libraries
- Friendly & welcoming developer community

Technical Rust Goodies

- First-class multi-threading
  - Compiler error to improperly access shared data
- Type system:
  - can uncover bugs at compile time
  - Makes refactoring simple
  - Reduces the number of tests needed
- Module system makes code separation simple
- Adding a dependency is 1 line in a config file
- Tooling:
  - Generate docs, line code, auto format

Why rust is useful

- Rust is different by design
- "clean state" approach
  - Important because of suitable language difference between Rust and others
  - Learn Rust versus learn to fight the compiler

Data types in Rust

- Memory only stores binary data
  - Anything can be represented in binary
- Program determines what the binary represents
- Basic types that are universally useful are provided by the language.

Basic Data Types

- Boolean
  - true, false
- Integer
  - 1,2,50,99,-2
- Double / Float
  - 1.1,5.5,200.0001,2.0
- Charecter
  - 'A', 'B', 'C', '6','$'
- String
  - "Hello","string","this is a string","its 42"

What is a variable?

- Assign data to a temporary memory location
  - Allows programmer to easily work with memory
- Can be set to any value & type
- Immutable by default, but can be mutable
  - Immutable: cannot be changed
  - Mutable: can be changed

Sample Program

```
let two = 2;
let hello = "hello";
let j = 'j';
let my_half = 0.5;
let mut my_name = "bill"; // Mutable charecter
let quit_program = false;
let your_half = my_half;
```

What are functions?

- A way to encapsulate program functionality
- Optionally accept data
- Optionally return data
- Utilized for code organization
    - Also makes code easier to read

```
fn add(a: i32,b: i32)->i32{
    a+b
}

fn NameOfFunction(variable1:<Datatype>,variable2:<Datatype>)-><ReturnValueType>
{
    // function code or function body
}

let x = add(1,1);
let y = add(3,0);
let z = add(x,1);
```

Summary of functions

- Functions encapsulate functionality
- Usefull to organize code
- Can be executed by "calling" the function
- Parameters determine what data a function can work with
- Optionally "returns" data
    - Data sent back from the function
    -

Fundemental println macro

- "Prints" (displays) information to the terminal
- Useful for debugging

- Macros use an exclamation point to call
- Generate additional Rust code.

```
let life = 42;
println!("hello");
println!("{:?}",life);
println!("{:?} {:?}",life,life);
```

Fundamental Control flow using "if" statement

- code executed line-by-line
- Actions ae performed and control flow may change
    - specific conditions can change flow
        - "if"
        - "else"
        - "else if"
- Code executing line-by-line
    - This can be changed using "if"
- Try to always include "else", unless there truly is no alternative case.

```
// linear flow
let a =1;
let b =2;
let c =3;
// if...else
let a = 99;
if a>99{
    println!("Big Number");
}else{
    println!("Small Number");
}
// Nested if...else
let a = 99;
if a > 99 {
    if a>200{
        println!("Huge Number");
    }else
    {
        println!("Big number");
    }
}else{
    println!("small number")
}
// if...else if..else
let a = 99;
if a > 200 {
        println!("Huge Number");
else if a>99{
    println!("Big number");
    }else{
    println!("small number");
    }
}
```

Repetition using loops

- Called "looping" or "iteration"
- Multiple types of loops
    - "loop" - infinite loop
    - "while" - conditional loop
- Both types of loops can exit using "break"

```
// loop
let mut a = 0;
loop{
    if(a==5){
        break;
    }
    println!("{:?}",a);
    a = a+1;
}

//while
let mut a = 0;
```

Creating and running a rust code

- After installation of rust open command line or terminal.
- Enter **cargo**
- Move into the directory
- Move inside src foder and run open main.rs
- Write your code and save it
- Open terminal and move to rust program directory
- Type **cargo run**

Creating and running a rust file

- Inside src folder create a folder bin
- create your file <filename.rs>
- cargo run -q --bin filename

Comments in run program

```rust
// this is the entry point of the application
fn main() {
    // Display a message to the user.
    println!("Hello, world!");
    let my_fav_color = "red";
}
```

Sample Programs-1

```rust
// Topic: Functions
//
//Program requirements:
// * Display your first and last name
//
// Notes:
//* Use a function to display your first name
//* Use a function to display your last name
//* Use the println macro to display message to the terminal

//* Use a function to display your first name
fn first_name(){
    println!("Hari prasad");
}


//* Use a function to display your last name
fn last_name(){
    println!("Anand");
}

fn main(){
    first_name();
    last_name();
}
```

Sample program-2 basic arthametic

```
fn sub(a: i32,b:i32)->i32{
    a-b;
}

fn main(){
    let sum = 2+2;
    let value = 10-5;
    let division = 10/2;
    let mult = 5*5;
    let five  = sub(8,3);
    let rem = 6%3;
    let rem2 = 6%4;
}
```

Sample program-3-basic math

```
// Topic: Basic arithmetic
//
// Program requirements:
// * Displays the result of the sum of two numbers
//
// Notes:
// * Use a function to add two numbers together
// * Use a function to display the result
// * Use the "{:?}" token in the println macro to display the result

fn sum(a:i32,b:i32)->i32{
    a+b
}

fn display_result(result:i32){
    println!("{:?}",result);
}

fn main() {
    let result = sum(2,2);
    display_result(result);
}
```

Sample program-4-if else code block

```
// Topic: Flow control using if..else
//
// Program requirements:
// * Displays a message based on the value of a boolean variable
// * When the variable is set to true, display "hello"
// * When the variable is set to false, display "goodbye"
//
// Notes:
// * Use a variable set to either true or false
// * Use an if..else block to determine which message to display
// * Use the println macro to display messages to the terminal

fn main() {
    let my_bool = true;
    if my_bool == true{
        println!("hello");
    }else{
        println!("goodbye");
    }
}
```

Sample program-5-if else if else

```
// Topic: Flow control using if..else if..else
//
// Program requirements:
// * Display ">5", "<5", or "=5" based on the value of a variable
//   is > 5, < 5, or == 5, respectively
//
// Notes:
// * Use a variable set to any integer value
// * Use an if..else if..else block to determine which message to display
// * Use the println macro to display messages to the terminal

fn main() {
    let n = 7;
    if n>5{
        println!(">5");
    }else if n<5{
        println!("<5");
    }else{
        println!("=5");
    }
}
```

Match Expressions

- Add logic to program
- Similar to if...else
- Exhaustive
    - All options must be accounted for

Sample code

```
fn main(){
    let some_bool = true;
    match some_bool{
        true => println!("its true"),
        false => println!("its false"),
    }
}
```

Sample code 2

```
fn main(){
    let some_int = 3;
    match some_bool{
        1=> println!("its 1");
        2=> println!("its 2");
        3=> println!("its 3");
        _=> println!("its something else"); // default case
    }
}
```

Match vs else...if

- match will be checked by the compiler
    - If a new possibility is added, you will be notified when this occurs.
- else...if is not checked by the compiler
    - If a new possibility is added, your code may contain a bug.
- Prefer match over else...if when working with a single variable.
- match considers all possibilities

    - More robust code

- Use underscore (_) to match "anything else"

Sample code 3 match

```
fn main(){
    let my_name = "Hari";
    match my_name {
       "Prasad" => println!("That is my last name"),
       "Hari" => println!("That is my first name"),
        _ => println!("Nice to meet you"),
    }
}
```

Sample code 4 match

```
// Topic: Decision making with match
//
// Program requirements:
// * Display "it's true" or "it's false" based on the value of a variable
//
// Notes:
// * Use a variable set to either true or false
// * Use a match expression to determine which message to display


fn main(){
    let my_bool = true;
    match my_bool {
        true => println!("it's true"),
        false => println!("it's false")
    }

}
```

Sample code 4 match

```
// Topic: Decision making with match
//
// Program requirements:
// * Display "one", "two", "three", or "other" based on whether
//   the value of a variable is 1, 2, 3, or some other number,
//   respectively
//
// Notes:
// * Use a variable set to any integer
// * Use a match expression to determine which message to display
// * Use an underscore (_) to match on any value

fn main() {
    let my_num = 2;
    match my_num {
        1 => println!("ONE"),
        2 => println!("TWO"),
        3 => println!("THREE"),
        _ => println!("SOME OTHER NUMBER")
    }
}
```

Repetition using loop

```
fn main(){
    let mut i = 3; // initiation
    loop {
        println!("{:?}",i);
        i = i-1;// propogation
        if i==0{
            break; // termination
        }
    }
    println!("done!");
}
```

Repetition using while loop

```rust
// Topic: Looping using the while statement
//
// Program requirements:
// * Counts down from 5 to 1, displays the countdown
//   in the terminal, then prints "done!" when complete.
//
// Notes:
// * Use a mutable integer variable
// * Use a while statement
// * Print the variable within the while loop
// * Do not use break to exit the loop

fn main(){
    let mut i =1; // initialization
    while i<=5{ // termination
        println!("{:?}",i);
        i = i+1; // propogation
    }
}
```

Enumeration data type

- Data that can be one of multiple different possibilities
    - Each possibility is called a "varient"
- Provides information abount your program to the compiler
    - More robust programs
- Enums can only be one varient at a time
- More robust programs when paired with match
- Make program code easier to read.

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}

fn which_way(go: Direction){
    match go {
        Direction::Up => "up",
        Direction::Down => "down",
        Direction::Left => "left",
        Direction::Right => "right"
    }
}
```

Structure in Rust

- A type that contains multiple pieces of data
    - All or nothing - cannot have some pieces of data and not others
- Each piece of data is called a "field"
- Makes working with data easier
    - Similar data can be grouped together.
- Structs deal with multiple pieces of data
- All fields must be present to create a struct
- Fields can be accessed using a dot (.)

```
struct ShippingBox {
    depth:i32,
    width:i32,
    height: i32,
}

fun main(){
    let my_box = ShippingBox {
        depth:3,
        width:2,
        height:5,
    }
let tall = my_box.height;
println!("the box is {:?} units tall",tall);
}
```

Tuples in Rust

- A type of "record"
- Store data anonumously
    - No need to name fields
- Useful to return pairs of data from functions
- Can be "destructured" easily into variables.
- Allow for anonymous data access
- Useful when destructing
- Can contain any number of fields
    - Use struct when more than 2 or 3 fields.

```
enum Access{
    Full,
}
fn one_two_three()->(i32,i32,i32){
    (1,2,3)
}
let numbers = one_two_three();
let (x,y,z) = one_two_three();
println!("{:?} {:?}",x,numbers.0);
println!("{:?} {:?}",y,numbers.1);
println!("{:?} {:?}",z,numbers.2);


let (employee,access) = ("Jake",Access:Full);
```

Sample program Tuples

```
fn main(){
    let coord = (2,3);
    println!("{:?}, {:?}",coord.0,coord.1);

    let (x,y) = (2,3);
    println!("{:?}, {:?}",x,y);

    let (name, age) = ("Emma",20);

    let favorite = ("red",14,"MAA","pizza","TV SHOW","home");

    let state = favorite.2;
    let place = favorite.5;

}
```

Expressions in Rust

- Rust is an expression-baseed language
    - Most things are evaluated and return some value
- Expression values coalesce to a single point
    - Can be used for nesting logic
- Expression allow nested logic

- if and match expressions can be nested
    - Best to not use more than two or three levels

```
let my_num = 3;
let is_let_5 = if my_num < 5{
    true
} else{
    false
};
let is_let_5 = my_num < 5; // expression

// sample 2 using match
let my_num = 3;
let message = match my_num {
    1 => "hello",
    _ => "goodbye"
}
```

Testing expressions using enums

```
enum Menu {
    Burger,
    Fries,
    Drink
}

let paid = true;
let item = Menu::Drink;
let drink_type = "water";
let order_placed = match item {
    Menu::Drink =>{
    if drink_type == "water"{
            true;
    }else {
        false;
    }
    _ => true,
};
```

Sample program for Expression

```
enum Access {
    Admin,
    Manager,
    User,
    Guest
}
fn main(){
    // secret file : admins only
    let access_level = Access::Guest;
    let can_access_file = match access_level{
        Access::Admin => true,
        _ => false
    };
    println!("can access: {:?}",can_access_file);
}
```

Sample program for Expression-2

```
// Topic: Working with expressions
//
// Requirements:
// * Print "its big" if a variable is > 100
// * Print "its small" if a variable is <= 100
//
// Notes:
// * Use a boolean variable set to the result of
//   an if..else expression to store whether the value
//   is > 100 or <= 100
// * Use a function to print the messages
// * Use a match expression to determine which message
//   to print

fn print_message(gt_100:bool){
    match gt_100{
        true => println!("its big"),
        false => println!("its small")
    }
    println!();
}

fn main() {
    let value = 101;
    let is_get_100 = value >100;
    print_message(is_get_100);
}
```

Memory in Rust

- Memory uses addresses and offsets
- Addresses are permanent, data differs
- Offsets can be used to "index" into same data

Basic memory refresh

- Memory is stored using binary
    - Bits: 0 or 1
- Computer optimized for bytes
    - 1 byte == 8 contiguous bits
- Fully contiguous

Addresses

- All data in memory has a address
    - used to locate data
    - Always the same - only data changes
- Usually dont utilize addresses directly
    - Variables handle most of the work

Offsets

- Items can be located at an address using an "offset"
- Offsets begin at 0
- Represent the number of bytes away from the original address
    - Normally deal with indexes insted

Ownership in rust

Memory management in rust

- Programs must track memory

    - If they fail to do so, a "leak" occurs
- Rust utilizes an "ownsership" model to manage memory
    - The "owner" of memory is responsible for cleaning up the memory
- Memory can either be "moved" or "borrowed"

The concept of moving

```
enum Light{
    Bright,
    Dull,
}
fn display_light(light:Light){
    match light {
        Light::Bright => println!("bright"),
        Light::Dull => println!("dull"),
    }
}
fn main(){
    let dull = Light::Dull;
    display_light(dull);
    display_light(dull);
}
```

The concept of borrowing

```
enum Light{
    Bright,
    Dull,
}
fn display_light(light:&Light){
    match light {
        Light::Bright => println!("bright"),
        Light::Dull => println!("dull"),
    }
}
fn main(){
    let dull = Light::Dull;
    display_light(&dull);
    display_light(&dull);
}
```

- Memory must be managed in some way to prevent leaks

- Rust uses "ownership" to accomplish memory management
    - The "owner" of data must clean up the momory
    - The occurs automatically at the end of the scope
- Default behavior is to "move" memory to a new owner
    - Use an ampersand (&) to allow code to "borrow" memory

Sample program for borrowing

```
struct Book {
    pages:i32,
    rating: i32,
}

fn display_page_count(book:&Book){
    println!("pages = {:?}",book.pages);
}

fn display_rating(book:&Book){
    println!("ratting = {:?}",book.rating);
}

fn main(){
    let book = Book {
        pages:5,
        rating:9,
    };
    display_page_count(&book);
    display_rating(&book)
}
```

Sample program for borrowing

```rust
// Topic: Ownership
//
// Requirements:
// * Print out the quantity and id number of a grocery item
//
// Notes:
// * Use a struct for the grocery item
// * Use two i32 fields for the quantity and id number
// * Create a function to display the quantity, with the struct as a parameter
// * Create a function to display the id number, with the struct as a parameter

struct GroceryItem {
    quantity:i32,
    id:i32
}

fn display_quantity(item: &GroceryItem){
    println!("quantity: {:?}", item.quantity);
}

fn display_id(item: &GroceryItem){
    println!("id: {:?}", item.id);
}

fn main() {
    let my_item = GroceryItem {
        quantity:3,
        id:99
    };
    display_quantity(&my_item);
    display_id(&my_item);
}
```

Sample program for `impl`

```
// object
struct Temperature {
    degree_f:f64
}

//class
impl Temperature {
    // Self is used for creating a new object
    fn freezing()->Self{
        Self{
            degree_f:32.0
        }
    }
    fn boiling() ->Self{
        Self {degree_f:212.0}
    }
    // self is used for using existing object
    fn show_temp(&self){
        println!("{:?} degree F",self.degree_f);
    }
}

fn main(){
    let hot = Temperature {degree_f:99.9}; // new object with parameters
    hot.show_temp();

    let cold = Temperature::freezing();
    cold.show_temp();

    let boiling = Temperature::boiling();
    boiling.show_temp()
}
```

Sample program on implimentation

```rust
// Topic: Implementing functionality with the impl keyword
//
// Requirements:
// * Print the characteristics of a shipping box
// * Must include dimensions, weight, and color
//
// Notes:
// * Use a struct to encapsulate the box characteristics
// * Use an enum for the box color
// * Implement functionality on the box struct to create a new box
// * Implement functionality on the box struct to print the characteristics

enum Color{
    Brown,
    Red,
}

impl Color{
    fn print(&self){
        match self{
            Color::Brown => println!("brown"),
            Color::Red => println!("red")
        }
    }
}

struct Dimensions {
    width:f64,
    height:f64,
    depth:f64
}

impl Dimensions{
    fn print(&self){
        println!("Width {:?}",self.width);
        println!("Height {:?}",self.height);
        println!("Depth {:?}",self.depth);
    }
}

struct ShippingBox {
    color:Color,
    weight:f64,
    dimensions:Dimensions
}

impl ShippingBox{
```

```
    fn new(weight:f64,color:Color,dimensions:Dimensions)->Self{
        Self {
            weight,
            color,
            dimensions
        }
    }
    fn print(&self){
        self.color.print();
        self.dimensions.print();
        println!("Weight {:?}",self.weight);
    }
}

fn main() {
    let small_dimensions = Dimensions {
        width:1.0,
        height:2.0,
        depth:3.0
    };
    let small_box = ShippingBox::new(5.0,Color::Red,small_dimensions);
    small_box.print();
}
```

Vector data structure in rust

- Multiple pieces of data
    - Must be the same type
- Used for lists of information
- Can add, remove, and traverse the entries

```
let my_numbers = vec![1,2,3]

let mut my_numbers = Vec::new();
my_numbers.push(1);
my_numbers.push(2);
my_numbers.push(3);
my_numbers.pop();
my_numbers.len(); // this is 2


let two = my_numbers[1];

for num in my_numbers {
    println!("{:?}",num);
}
```

- Vectors contain multiple pieces of similar data
- Data can be added or removed
- The vec! macro can be used to make vectors
- Use for...in to iterate through items of a vector

```
struct Test {
    score:i32
}

fn main(){
    let my_scores = vec![
        Test {score:90},
        Test {score:88},
        Test {score:77},
        Test {score:93},
    ];
    for test in my_scores {
        println!("score = {:?}",test.score);
    }
}
```

Strings in Rust

- Two commonly used types of strings
    - String - owned

    - &str - borrowed String slice
- Must use a owned String to store in a struct
- Use &str when passing to a function

Finished till strings start with Deriving functions