Part 1

Common Programming Concepts

1.1 Variables and Mutability

Variables

- Variables refer to memory locations in system that hold a value for that variable. This value can be changed during the execution of the program.

- Variables are only valid in the scope in which they are declared.

VARIABLE

Value data

Container variable

Types of Variable

- **Mutable Variables:**

  Variables whose value can be modified.

- **Immutable Variables:**

  Variables whose value cannot be modified.

- In Rust, variables are immutable by default for safety. However we can change the mutability of variables.

Example:
Changing immutable variable

```
fn main()  {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Result

This program will give a compile time error because we are trying to change the value of an immutable variable.

- **Compilation / Compile time Error:**

  When a program fails to compile.

Example:
Changing mutable variable

```
fn main()  {
        let mut x = 5;
        println!("The value of x is: {}", x);
        x = 6;
        println!("The value of x is: {}", x);
}
```

Result:

The value of x is 5

The value of x is 6

Constants

- Constants are the values which are bound to a name and typically hold those values which we don't want to change.
- Constants are declared by using "const" keyword and the type of the value must be annotated.

Syntax:

```
const MAX_POINTS: u32 = 100_000;
```

- Rust’s naming convention for constants is to use all uppercase with underscores between words.
- Constants are used as hardcoded values which you want to use throughout your program.

- Constants can be declared in any scope, including the global scope, which makes them useful for values that many parts of the code need to know about.

<u>Example:</u>

Speed of light

pi(π)

Shadowing

- Shadowing is the declaration of a new variable with the same name as the previous variable, and the new variable shadows the previous variable.
- A variable can be shadowed with a new type. E.g. String can be shadowed as an integer.

- We can shadow a variable by using the same variable's name and repeating the use of "let" keyword.

Example: Shadowing Variable

```
fn main()  {
        let x = 5;
        let x = x + 1;
        let x = x * 2;
        println!("The value of x
is: {}", x);
 }
```

Result:

The value of x is 12

Note: Variable remains immutable.

Difference between Shadowing and Mutability

- The difference between mut and shadowing is that because we're effectively creating a new variable when we use the let keyword again, we can change the ***type*** of the value but reuse the same name.

- We will get a compile-time error if we accidentally try to reassign value to an immutable variable without using the let keyword.

- In shadowing, we can change the type of the value but reuse the same name.

Example:
Changing data type by shadowing

```
let spaces = "       ";
let spaces = spaces.len();
```

Note:

This will change the data type from string to integer.

Example

```
let mut spaces = " ";
spaces = spaces.len();
```

Note:
By using ‘mut’ we are not allowed to change the data type of the variable.

1.2 Data Types

Data types:

- Every value in Rust is of a certain data type.
- Rust automatically infers primitive data type without explaining it explicitly.
- Rust is a statically typed language, which means that it must knows the types of all variables at compile time.

Data types in RUST:

- **Scalar type**

  A scalar type represents a single value. Rust has four primary scalar types.

  - Integer.
  - Floating-point.
  - Boolean.
  - Character.

- **Compound type**

  Compound type can group multiple values into one type. Rust has two primitive compound types.

  - Tuples.
  - Arrays.

Scalar Types

1) **Integer Type**:

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
arch	`isize`	`usize`

- In Rust each integer can be either signed or unsigned and has an explicit size.
- The isize and usize types depend on the kind of computer your program is running on: 64 bits if you’re on a 64-bit architecture and 32 bits if you’re on a 32-bit architecture.
- If we do not declare type of an integer then Rust will by default take i32 for an integer.

- Each signed variant can store numbers from $-(2^{n-1})$ to $2^{n-1} - 1$ inclusive.
- Unsigned variants can store numbers from 0 to $2^{n} - 1$.

```
let variable: u32 = 100;
```

Data type

You can write integer literals in any of the forms given below

Number literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b’A’

2) **Floating-Point Types:**

- Rust also has two primitive types for floating-point numbers, which are numbers with decimal points.
- Rust’s floating-point types are f32 and f64, which are 32 bits and 64 bits in size, respectively.

- If we do not declare type of floating point then Rust will by default take f64 for that floating point type.
- The f32 type is a single-precision float, and f64 has double precision.

```
let variable: f32 = 1.23;
```

3) **Boolean Type:**

- A Boolean type in Rust has two possible values: true and false.
- Booleans are one byte in size.

- The Boolean type in Rust is specified using bool.

```
let variable: bool = true;
```

4) **Character Type:**

- Rust supports letters too.
- Rust’s char type is the language’s most primitive alphabetic type.

- Characters are specified with single quotes, as opposed to string literals, which use double quotes.
- Rust's char type is four bytes in size and represents a Unicode Scalar Value.

```
let c = 'z';
```

Numeric Operators:

Operators	Meaning	Example	Result
+	Addition	4+2	6
-	Subtraction	4-2	2
*	Multiplication	4*2	8
/	Division	4/2	2
%	Modulus operator to get remainder in integer division	5%2	1

Compound Type

1) **Tuple Type:**

- A tuple is a general way of grouping together some number of other values with a variety of types.
- Tuples have a fixed length, once declared, they cannot grow or shrink in size.
- Contain heterogeneous data.

- We create a tuple by writing a comma-separated list of values inside parentheses.

- We can access a tuple element directly by using a period (.) followed by the index of the value we want to access.

Syntax:

declaration: let tup = (500, 6.4, 1);

Or

Let tup: (i32 , f64 , u8) = (500 , 6.4 , 1) ;

Accessing a tuple element: let value1 = tup.0;

Example

```rust
fn main() {
 let tup = (500, 6.4, 1);
 let (x, y, z) = tup;
  println!("The value of y is: {}", y);
}
```

Result:

The value of y is: 6.4

Note:
This way of assigning value is called destructuring

2) **Array Type:**

- Another way to have a collection of multiple values is with an array.
- Contains homogenous data.
- Every element of an array must have the same type not like a Tuple.

- Writing an array's type is done with square brackets containing the type of each element in the array followed by a semicolon and the number of elements in the array.
- You can access elements of an array using indexing in square bracket.

- An array isn’t flexible and not allowed to grow or shrink in size.

  declaration: let array = [1, 2, 3, 4, 5];

  Accessing an array element: let value1 = array[0];

Example

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let element = 3;
            println!("The value of
    element is: {}", a[element]);
}
```

Result:

The value of element is: 4

1.3 FUNCTION

FUNCTION

- “fn” keyword allows you to declare new functions.
- Rust code uses *snake case* as the conventional style for function and variable names.
- In snake case, all letters are lowercase and underscores separate the words.

Example

```rust
fn main()  {
    println!("Hello, world!");
    another_function();
}
fn another_function()  {
    println!("Another function.");
}
```

Result:

Hello, world
Another function.

Note: Rust doesn’t care where you define your functions after or before the main function

Function Parameters

- These are the special variables that are part of a function’s signature.
- When a function has parameters, you can provide it with concrete values for those parameters

- Technically, the concrete values are called *arguments*
- But in casual conversation, you can use both i.e parameter or argument
- While calling a function, function’s parameter value has to be set.

Example

```
fn main()  {
 another_function(5);
}
fn another_function(x: i32) {
 println!("The value of x is: {}",
x);
}
```

Result:

The value of x is: 5

In function's signature, you *must* declare the type of each parameter.

Example

```
fn main() {
    another_function(5, 6);
}
fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y); }
```

Result:

```
The value of x is: 5
The value of y is: 6
```

Note

- The function's parameters don't all need to be the same type.
- Multiple parameters can be separated by commas.

Statements and Expressions

- Rust is an expression-based language
- Function bodies are made up of series of statements optionally ending in an expression.

- Statements are instructions that perform some action and do not return a value.
- Expressions evaluate to a resulting value.

Creating a variable and assigning a value to it with the let keyword is an statement.

```
fn main() {
        let y = 6;
 }
```

The let y = 6 statement does not return a value

Note: Statement contains (;) at its end.

Consider a simple math operation, such as 5 + 6, which is an expression that evaluates to the value 11.

```
fn five() -> i32 {
  5
}
```

Expressions can be part of statements

Note: Expression does not contain (;) at its end.

Example

```
fn main() {
    let x = 5;
    let y = {
        let x = 3;
        x + 1
        };
    println!("The value of y is: {}", y);
}
```

Result:
The value of y is: 4

Note: x+1 is an expression without semicolon. If you add a semicolon to the end of an expression, you turn it into an statement, which will then not return a value.

FUNCTION WITH RETURN VALUE

- Functions can return values to the code that calls them. We don't name return values, but we do declare their type after an arrow (`->`).

Example

```
fn five() -> i32 {
  5
}
fn main() {
  let x = five();
  println!("The value of x is: {}",
x);
}
```

Result:
The value of x is: 5

Note: We can't use semicolon(;) after 5 because it is an expression.

1.4 Comments

Comments

- Comment makes code easy to understand when some time extra explanation is warranted.
- Compiler will ignore to compile the comments in code but people reading the source code may find it useful.

- Comments must start with two slashes and continue until the end of the line.
- For comments that extend beyond a single line, you'll need to include // on each line.

Examples: Comments

```rust
fn main() {
// hello
// hi
// hey
}
```

```rust
fn main() {
// hello, world
}
```

1.5 Control Flow

Control Flow

- A control flow is the order in which the piece of program is executed.

In control flow we have following expressions:

1. If expression
2. Else expression
3. Else if expression

If Expression:

- If expression allows to branch your code to any condition.
- If the condition is true then the program in if block will be executed.

- When if expression is not true then the program will be passed to further blocks of else if expressions or else expression.
- The curly brackets defining the program blocks are called arms.

Else Expression:

- Else expression gives the program an alternative block of code to execute when the condition introduced in if block evaluates to false.
- Else expression comes at last.

Example

```
fn main() {
   let number = 3;
    if number < 5 {
        println!("condition was
true");
    } else {
        println!("condition was
false");
    }
}
```

Result:

Condition was true

Example

```
fn main() {
   let number = 7;
    if number < 5 {
       println!("condition was true");
    } else {
       println!("condition was
false");
    }
}
```

Result:

Condition was false

Note:

- The condition introduced in if expression and else if expression should be bool type (true or false) . Otherwise the program will not run.
- Rust will not automatically try to convert non-Boolean types to a Boolean.

Example

```
fn main() {
    let number = 3;
    if number {
        println!("number was
something other than zero");
    }
}
```

Result:

Compile time error occur

Note: Here Rust will give error because instead of giving a bool condition we have placed an integer in condition.

Else If Expression:

- Else if expression is used to introduce multiple conditions.
- This expression is always situated between if expression and else expression

Example

```
fn main() {
    let number = 6;
    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by
4, 3, or 2");
    } }
```

Result:

Number is divisible by 4

Note:

- In the above example there are two true conditions but Rust print the block of first true condition.
- Which means Rust only executes the block of first true condition.

Using if in let statement

- We can use if expression to store different values depending upon the condition in variable using let statement , as mentioned in next example.

Example

```
fn main() {
    let condition = true;
    let number = if condition {
        5        // (no semicolon means
expression)
    } else {
        6        // (no semicolon means
expression)
    };
    println!("The value of number is:
{}", number);
}
```

Note:

Value of 5 will be stored in the number variable.

Example

```
fn main() {
    let condition = true;
    let number = if condition {
        5
    } else {
        "six"
    };
    println!("The value of number is: {}", number);
}
```

Result:
Compile error will occur

Note: Expression types are mismatched in each arm, hence an error will be occured.

Loops:

Loops execute a block of code more than once.

Types of loops

1. loop
2. while
3. for

1. loop

- The loop keyword tells Rust to execute a block of code over and over again forever or until you explicitly tell it to stop.
- ‘break’ expression can be use to stop the loop and the value is placed after the break expression that we want in return.

Example

```rust
fn main() {
    loop {
        println!("again!");
    }
}
```

Result:

again!

again!

again!

again!

(and so on .....)

Example

```
fn main() {
    let mut counter = 0;
    let result = loop {
                counter += 1;
        if counter == 10 {
            break counter * 2;
        }
    };
  println!("The result is {}",
result); }
```

Result:
Here 20 will be save in the result variable

2. While loop

- In this type of loop While the condition is true, the loop runs. When the condition ceases to be true, the program calls break, stopping the loop.

Example

```rust
fn main() {
    let mut number = 3;
        while number != 0 {
            println!("{}!", number);
            number = number - 1;
        }
    println!("LIFTOFF!!!");
}
```

Result:

3!

2!

1!

LIFTOFF!!!

3. For loop

- loops through a block of code a number of times(iteration)

Example

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    for element in a.iter() {
        println!("the value is: {}",
element);
    }
}
```

Result:

the value is: 10
the value is: 20
the value is: 30
the value is: 40
the value is: 50

Note:

- iter() is used for iteration.

Example

```rust
fn main() {
    for number in (1..4).rev() {
        println!("{}!", number);
    }
    println!("LIFTOFF!!!");
}
```

Result:

3!

2!

1!

LIFTOFF!!!

Note:

- (1..4) is for declaring range.
- rev() is used for reverse order

Section Break

Part 2

Understanding Ownership

As a programming language

C/C++

Haskell/Python

more control,
less safety

less control,
more safety

Rust

more control,
more safety

No need for runtime

Memory safety

Data race freedom

Section 2.1

Ownership

What you will learn ...

- Ownership Concept
- Concept of Stack
- Ownership Rules
- String Type
- Memory Allocation
- Ownership and Functions

Memory and Allocation

In Rust, data can be stored either in stack or heap memory.

Stack and Heap

Both are parts of memory to be used at runtime, but they are structured in different ways.

Stack

It stores values in the order it gets them and removes the values in the opposite order.

Referred to as Last In, First Out (LIFO).

Stack: Think of a stack of plates

When you add more plates, you put them on top of the pile, and when you need a plate, you take one off the top.

Adding or removing plates from the middle or bottom wouldn't work as well!

Stack

- **Adding** data is called ***pushing*** *onto the stack*,
- **Removing** data is called ***popping*** off *the stack*.
- Stack memory is an **organized** memory.
- It is **faster** than the heap memory because of the way it **accesses** the memory.

All data stored on the stack must have a **known, fixed size**

Stack Push and Pop Example:

Stack: in action

Stack: How it works

- When code calls a function, the values passed into the function (including, potentially, pointers to data on the heap) and the function's local variables get pushed onto the stack.
- When the function is over, those values get popped off the stack.

Heap

The Heap is Less Organized

Data with a **size** that is **unknown** at compile time or a **size** that might **change** must be stored on the **heap**.

Heap: Allocation

- When you **put** data on the **heap**, you ask for ***some amount*** *of* ***space*** from ***OS***.

- The **operating system** finds an **empty spot** in the heap that is ***big enough***, *marks* ***it as being in use***, and ***returns a pointer***, which is the **address** of that **location**.

Heap

Data Element
Data Element
Reference Pointer
Data Element
Reference Pointer
Hello World!
PIAIC
The quick brown fox jumps over the lazy dog.
The name of my country is Pakistan.
Rust Example
Stack (Organised)
Heap (Haphazard)

Heap: Restaurant as Example

Heap

- Less organized
- Requested from OS
- Slower
- Follow pointer
- large amount... take time to manage data in the heap

Why pushing to the stack is faster than allocating on the heap ?

Because the operating system never has to search for a place to store new data; that location is always at the top of the stack.

Why accessing data in the heap is slower than on the stack ?

Allocating space on the heap requires more work, because you ask for some amount of space to operating system every time.

OS has to follow a pointer every time.

Ownership

All programs have to manage the way they use a computer's memory while running.

Ownership

- Some languages have garbage collection that constantly looks for no longer used memory as the program runs.

- In other languages, the programmer must explicitly allocate and free the memory.

Ownership

- Rust uses a third approach wherein memory is managed through a system of ownership
- Ownership is managed with a set of rules that the compiler checks at compile time.
- None of the ownership features slow down your program while it’s running.

Ownership

In simple words for understanding purpose……

Ownership is the transfer of currently possessed entity to another party which causes the previous owner to no longer have access to the object that is being transferred.

Ownership

- In Rust, there are very clear rules about which piece of code *owns* a resource.
- In the simplest case, it's the block of code that created the object representing the resource.
- At the end of the block the object is destroyed and the resource is released.

Ownership

```
fn main()
{// s is not valid here, it's not yet declared
   let s = "hello";
   // s is valid from this point forward
   // do stuff with s
   println!("{}", s);
}// this scope is now over, and s is no longer valid
```

In the above example the letter or <u>variable</u> "s" is the owner of the word "hello" and is valid from the point of declaration after the start of the parenthesis "{" and remains valid until the end of the parenthesis "}"

Why Ownership

- Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector.

Why Ownership

- Keeping track of <u>what parts of code are using what data</u> on the heap, minimizing the amount of <u>duplicate data</u> on the heap, and <u>cleaning up unused data</u> on the heap so you don't run out of space are all problems that ownership addresses.

Why Ownership

All primitive data types (integers, booleans, string literals) and pointers (address of heap data) are stored on stack whereas for more complicated data types we have heap.

Ownership Rules

Rule # 1

Each value in Rust has a variable that's called its Owner.

Rule # 1 Example

Each value in Rust has a variable that's called its owner.

```
let a = "Hello world!";
```

Variable

Value

In the above example the variable i.e. "a" is also the owner of the value "Hello world!".

Rule # 2

There can only be

one owner at a time.

Rule # 2 Example

Each value in Rust has a variable that's called its owner.

```
let a = String::from("Hello");
```
–> here variable "**a**" is the owner

```
let b = a;
```
–> here the value of "**a**" is moved to variable "**b**" which now becomes the owner of "**Hello**"

Considering both variables "a" and "b" are within the same scope.

Rule # 3

When the owner goes out of scope, so does the value.

Rule # 3 Example

When the owner goes out of scope, so does the value.

```
fn main() {
    {                          –> "a" is not valid here, it's not yet declared
    let a = "Hello";           –> "a" is valid from this point forward
                   –> do stuff with "a"
    }              –> this scope is now over and "a" is no longer valid
}
```

Variable Scope

A **scope** is the range within a program for which an item is valid.

When **variable** comes into **scope**, it is valid. It remains valid until it goes out of scope.

Variable Scope

```
fn main()

{// s is not valid here, it's not yet declared

   let s = "hello";
   // s is valid from this point forward
   // do stuff with s
   println!("{}", s);

}// this scope is now over, and s is no longer valid
```

Variable Scope

- The variable “s” refers to a string literal, where the value of the string is hardcoded into the text of our program. The variable is valid from the point at which it’s declared until the end of the current scope.

- When “s” comes into scope, it is valid.
- It remains valid until it goes out of scope

The String Type

Rust has a second string type, String.

This type is allocated on the heap and is able to store an amount of text that is **unknown** to us at **compile time**.

The String Type

You can create a String from a string literal using the from function, like:

```
let s = String::from("hello");
```

The double colon (::) is an operator that allows us to namespace this particular from function under the String type rather than using some sort of name like string_from.

The String Type

This kind of string can be mutated:

```
let mut s = String::from("hello");
s.push_str(", world!"); // appends a literal to a String
println!("{}", s); // This will print `hello, world!`
```

The difference between String Literal and String type is how these two types deal with memory.

String: Memory and Allocation

The memory must be requested from the OS* at runtime.

Rust automatically calls **drop** function to return memory when variable goes out of scope.

OS = Operating System

String: Memory and Allocation

1

Call
String::from

2

It **requests** the memory it needs

3

Drop returns memory automatically

Ways Variables And Data Interact:

Move

```
fn main() {
    let x = 5;
    let y = x;
}
```

This is ***binding the value 5 to x;*** then make a ***copy*** of the value in **x** and binding it to ***y.***”

We now have two variables, ***x*** and ***y***, and both equal ***5*** and valid.

Move

```
fn main() {
   let s1 = String::from("hello");
   let s2 = s1;
}
```

Looks familiar ? But not quite ...

In this value of s1 is moved to s2

Move

A **String** consists of ***three parts***, shown on the left:

a ***pointer*** to the memory on the heap, a ***length***, and a ***capacity***.

These are stored on the stack.

On the right is the memory on the heap that holds the ***contents***.

Fig: Representation in memory

Move

When we assign s1 to s2, the String data on stack is copied i.e. the ***pointer***, the ***length***, and the ***capacity***

Data on the **heap** is ***not*** copied

Fig: Representation in memory

Move

Operation
s2 = s1 could be very expensive in terms of runtime performance if the data on the heap were **large**.

Fig: Representation in memory

Move

- Drop function is called when a variable goes out of scope
- This is a problem: when `s2` and `s1` go out of scope, they will both try to free the same memory.
- This is known as a *double free error*.
- Freeing memory twice can lead to memory corruption.

Move

Instead of trying to copy the allocated memory, Rust considers s1 to no longer be valid and, therefore, Rust doesn't need to free anything when s1 goes out of scope.

Fig: Representation in memory

Move

You'll get an **error** because Rust prevents you from using the **invalidated reference**

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
    println!("{}, world!", s1);
}
```

Move

```
error[E0382]: use of moved value: `s1`
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |
  = note: move occurs because `s1` has type
`std::string::String`, which does
  not implement the `Copy` trait
```

Move

This differs from shallow copy as Rust also invalidates the first variable, instead it's known as a ***move***.

In this example, we would say that s1 was moved into s2.

Fig: Representation in memory

Clone

```rust
fn main() {
  let s1 = String::from("hello");
  let s2 = s1.clone();
  println!("s1 = {}, s2 = {}", s1, s2);
}
```

- If we do want to deeply copy the heap data of the String, not just the stack data, we can use a common method called clone.
- This works just fine and explicitly produces the behavior, the heap data does get copied.

Stack Only Data: Copy

- We don't have a call to clone, but x is still valid and wasn't moved into y.
- Reason such types like integers
- have **known size** at compile time
- are stored entirely on the **stack,** so copies are quick to make.

```
fn main()
{
  let x = 5;
  let y = x;
  println!("x = {}, y = {}", x, y);}
```

Stack Only Data: Copy

So what types are Copy?

- All integer
- Boolean
- All floating point
- Character
- Tuples, if they contain types that are also Copy. i.e
  - (i32, i32) is Copy,
  - (i32, String) is not.

Ownership & Function

Ownership & Function

- The concept of ownership in a function is as same as the assigning a value to a variable.

- Assigning a value to a function will take it ownership and you would not be able to reuse it

Ownership & Function

```rust
fn main() {
    let s = String::from("hello");
    takes_ownership(s);
    let x = 5;
    makes_copy(x);
}
fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}
fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}
```

Ownership & Function

```
fn main() {
    let s = String::from("hello"
    takes_ownership(s);
    let x = 5;
    makes_copy(x);
}
```

"s" co

"s" value moves into the function "takes_ownership()"

"x" would move into the function

Here, x goes out of scope, then s. But because s's value was moved, nothing special happens.

used afterwards

Ownership & Function

“some_string” comes into scope

```
fn takes_ownership(some_
    println!("{}", some_
}
fn makes_copy(some_integer: i32) {
    println!("{}", some_i
}
```

Drop is called and “some_string” goes out of scope.

“some_integer” goes out of scope. Nothing special happens.

Returning values and scope

- **Returning a value in a function also transfer the ownership**
- *Whenever a function **return a value** the **ownership** is also **returned**.*

Returning values and scope

```rust
fn main() {
    let s1 = gives_ownership();
    let s2 = String::from("hello");
    let s3 = takes_and_gives_back(s2);
}
fn gives_ownership() -> String {
    let some_string = String::from("hello");
    some_string
}
fn takes_and_gives_back(a_string: String) -> String {
    a_string
}
```

Return Values and Scope

```
fn main() {

    let s1 = gives_ownership();

    let s2 = String::from("hello");

    let s3 = takes_and_gives_back(s2);

}
```

gives_ownership **moves** its return value into **s1**

s2 comes into **scope**

Here, s3 goes out of scope and is **dropped**. s2 goes out of scope but was **moved**, so nothing happens. s1 goes out of scope and is **dropped**.

Return Values and Scope

```
fn gives_ownership() -> String {

    let some_string = String::from

    some_string

}
```

gives_ownership will **move** its return value into the function that calls it

some_string is returned and **moves** out to the calling function

Here, some_string goes out of scope but was **moved**, so nothing happens.

Return Values and Scope

takes_and_gives_back will take a String and return one

```
fn takes_and_gives_back(a_string: String) -> String {

    a_string

}
```

a_string come into **scope** nd **moves** out to the calling function

Here, a_string goes out of scope but was **moved**, so nothing happens.

Return Multiple Values

```
fn main()
  let s1
  let (s2, len) = calculate_length(s1);
  println!("The length of'{}'is{}.",s2,len);
}
fn calculate_length(s: String)->(String,usize)
{    let length = s.len();
     (s, length)
}
```

s1 is **moved** into calculate_length, which also **moves** its return value into tuple (s2,len)

(s, length) is returned and **moves** out to the calling function

len() re

Section 2.2

Borrowing and Referencing

What you will learn

- Borrowing
- References
- Mutable References
- Data Race
- Rules of References
- Dangling Reference

What is Borrowing?

Borrowing

We call having references as function parameters *borrowing*. As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back.

What is Referencing?

Referencing

- Reference is the act of consulting someone or something in order to get information
- In terms of Rust programing we can define reference as if we are taking a copy of it without damaging or taking its ownership

Referencing

‘&’ symbol is used to pass the reference

Referencing

Let’s have a look how ‘&’ symbol is used in code.

Referencing

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
println!("The length of {} is {}.", s1, len);
}
fn calculate_length(s: &String) -> usize {
    s.len()}
```

Mutable References

Mutable References

- The concept of mutable reference is same as we use mutable variable

- Mutable reference is used when we have to modify the value we make reference to

Mutable References

- There should be “mut” keyword with the reference as well as the variable or type we are making reference.

```rust
fn main() {
    let s = String::from("hello");
    change(&s);}
change(some_string: &String) {
some_string.push_str(", world");
}
```

If we try to run this code, we will get an error because references are immutable by default in Rust.

We can fix that error by adding 'mut' keyword

```
fn main() {
    let mut s = String::from("hello");
    change(&mut s); }

fn change(some_string: &mut String) {

    some_string.push_str(", world");

}
```

Restriction!

But Mutable References have one big restriction

You can have only one mutable reference to a particular piece of data in a particular scope.

For Example:

```rust
let mut s = String::from("hello");

let r1 = &mut s;

let r2 = &mut s;

println!("{}, {}", r1, r2);
```

Duplicate

This is not allowed.

Restriction Benefit

The benefit of having this restriction is that Rust can prevent ***Data Race*** *at compile time.*

Data Race

Occurs when,

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There’s no mechanism being used to synchronize access to the data.

Data Race

```
fn main() {
    let mut s = String::from("hello");
{
    let r1 = &mut s;
}

    let r2 = &mut s;
}
```

r1 goes out of **scope** here, so we can make a new **reference** with no problems.

Data Race

```
fn main() {
   let mut s = String::from("hello");

   let r1 = &s;
   let r2 = &s;
   let r3 = &mut s;

   println!("{}, {}, and {}", r1, r2, r3);
}
```

no problem

no problem

BIG PROBLEM

Data Race

```
error[E0502]: cannot borrow `s` as mutable because it is
also borrowed as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s;
  |              -- immutable borrow occurs here
5 |     let r2 = &s;
6 |     let r3 = &mut s;
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                    -- immutable borrow later used here
```

Data Race

Data races cause undefined behavior and can be difficult to diagnose and fix when you're trying to track them down at ***runtime****; Rust* ***prevents*** *this problem from happening because it* ***won't even compile code*** *with* ***data races****!*

Dangling References

Dangling pointer that references a location in memory that may have been given to someone else

In Rust the compiler will ensure that the data will not go out of scope before the reference to the data does.

Dangling References

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Dangling References

```
error[E0106]: missing lifetime specifier
 --> main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected lifetime parameter
  |
  = help: this function's return type contains a
borrowed value, but there is
  no value for it to be borrowed from
  = help: consider giving it a 'static lifetime
```

Dangling References

dangle returns a **reference** to a String

```
fn dangle() -> &String {

    let s = String::from("hello");

    &s
}
```

s is a new String

we return a **reference** to the String, s

Here, s goes out of scope, and is dropped. Its memory goes away. Danger!

Dangling References

```rust
fn dangle() -> String {

    let s = String::from("hello");

    s
}
```

Rules of References

- There can either be one mutable reference or any number of immutable references.
- References must always be valid.

Summary

- Ownership and borrowing ensure memory safety at compile time.

- Control over your memory usage in the same way as other systems programming languages

Summary

- Automatically clean up that data when the owner goes out of scope.
- No extra code to get this control.

Section Break

Part 3

Structs for Structure Related Data

Contents:

- *To be able to describe STRUCT in Rust Language*
- *To be able to read and write syntax*
- *To be able to define STRUCTS and manipulate in Rust*
- *Methods syntax and its application*
- *To be able to describe the concept of TRAITS in Rust*
- *To be able to implement TRAITS in Rust Language*

Introduction to Struct

- *A struct, or structure, is a custom data type that lets you name and package together multiple related values that make up a meaningful group.*

- *Under the light of OOP concept , a struct is like an object's data attributes.*

- *Structs are the building blocks for creating new types in your program's domain to take full advantage of Rust's compile time type checking.*

Defining Struct

- *Structs are similar to tuples.*
- *Like tuples, the pieces of a struct can be different types.*
- *Thing which distinguish struct from tuple is that, we can name each piece of data so it's clear what the values mean.*

Defining Struct

keyword

Fields: define the names and types of the pieces of data

```
struct User {
 username: String,
 email: String,
sign_in_count:u64,
active: bool,
}
```

Name: should describe the significance of the pieces of data being grouped together

structs are more flexible than tuples:

you don’t have to rely on the order of the data to specify or access the values of an instance

Instantiating Struct

Instance: struct by specifying concrete values for each of the fields.

```
let user1 = User {

email:
String::from("someone@example.com"),

username:
String::from("someusername123"),

active: true,

sign_in_count: 1,

};
```

- We create an instance by stating the name of the struct and then add curly brackets containing key: value pairs, where the keys are the names of the fields and the values are the data we want to store in those fields.

To get a specific value from the struct, we can use dot notation.

```
user1.email = String::from("anotheremail@example.com");
```

Field Init Shorthand

For variable with the same name as struct field, you can use "***field init shorthand***"

Consider the same User Struct

```
struct User {

username: String,
email: String,
sign_in_count:u64,
active: bool,

}
```

Field Init Shorthand

```rust
fn build_user(email: String, username: String) -> User {

User {

email: email,
username: username,
active: true,
sign_in_count: 1,

}

}
```

field init shorthand

```rust
fn build_user(email: String, username:
String) -> User {

User {
email,
username,
active: true,
sign_in_count: 1,

}

}
```

Creating Instances From Other Instances With Struct Update Syntax

It's often useful to create a new instance of a struct that uses most of an old instance's values but changes some.

```
# struct User {
# username: String,
# email: String,
# sign_in_count: u64,
# active: bool,
# }

let user1 = User {
# email:
String::from("someone@example.com"),
# username:
String::from("someusername123"),
# active: true,
# sign_in_count: 1,
# };

let user2 = User {
email:
String::from("another@example.com"),
username:
String::from("anotherusername567"),
active: user1.active,
sign_in_count: user1.sign_in_count,
};
```

```
# struct User {
# username: String,
# email: String,
# sign_in_count: u64,
# active: bool,
# }

let user1 = User {
# email:
String::from("someone@example.com"),
# username:
String::from("someusername123"),
# active: true,
# sign_in_count: 1,
# };

let user2 = User {
email:
String::from("another@example.com"),
username:
String::from("anotherusername567"),
..user1,
};
```

Tuple Structs without Named Fields to Create Different Types

Tuple structs have the added meaning the struct name provides but don't have names associated with their fields; rather, they just have the types of the fields.

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);


let black = Color(0, 0, 0);

let origin = Point(0, 0, 0);
```

Note: the "black" and "origin" values are different types, because they're instances of different tuple structs. Each struct we define is its own type, even though the fields within the struct have the same types.

For example, a function that takes a parameter of type Color cannot take a Point as an argument, even though both types are made up of three i32 values. Otherwise, tuple struct instances behave like tuples.

Ownership of Struct Data

- In the User struct definition, we used the **owned String type** rather than **the &str string slice type.**
- This is a deliberate choice because we want instances of this struct to own all of its data and for that data to be valid for as long as the entire struct is valid.
- It's possible for structs to store references to data owned by something else, but to do so requires the use of lifetimes.

Let's say you try to store a reference in the struct without specifying lifetimes, like this, it won't work:

Filename: src/main.rs

```rust
struct User {
    username: &str,
    email: &str,
    sign_in_count: u64,
    active: bool,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

The compiler will complain that it needs lifetime specifiers:

```
error[E0106]: missing lifetime specifier
 -->
  |
2 |     username: &str,
  |               ^ expected lifetime parameter

error[E0106]: missing lifetime specifier
 -->
  |
3 |     email: &str,
  |            ^ expected lifetime parameter
```

An Example Program Using Structs

Let’s write a program that calculates the area of a rectangle.

1) We’ll start with single variables, and then refactor the program until we’re using structs instead.
2) Those structs will take the width and height of a rectangle specified in pixels and calculate the area of the rectangle.
3) Listing 5-8 shows a short program with one way of doing exactly that in our project’s *src/main.rs*.

```rust
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Listing 5-8: Calculating the area of a rectangle specified by separate width and height variables

```
The area of the rectangle is 1500 square pixels.
```

The issue with this code is evident in the signature of area :

```
fn area(width: u32, height: u32) -> u32 {
```

- The area function is supposed to calculate the area of one rectangle, but the function we wrote has two parameters.
- The parameters are related, but that’s not expressed anywhere in our program.
- It would be more readable and more manageable to group width and height together.

Refactoring with Tuples

1) In one way, this program is better. **Tuples let us add a bit of structure, and we're now passing just one argument.**

2) **Tuples don't name their elements,** so our calculation has become more confusing because **we have to index into the parts of the tuple**.

3) It doesn't matter if we mix up width and height for the area calculation, but if we want to draw the rectangle on the screen, it would matter! **We would have to keep in mind that width is the tuple index 0 and height is the tuple index 1.**

Refactoring with Structs: Adding More Meaning

1) We use structs to add meaning by labeling the data. **We can transform the tuple we're using into a data type with a name for the whole as well as names for the parts**, as shown in Listing 5-10.

2) **Our area function is now defined with one parameter**, which we've named rectangle , whose type is **an immutable borrow** of a struct Rectangle instance.

3) We want to **borrow the struct rather than take ownership of it**.

Defining a Rectangle struct

4) This way, main retains **its ownership and can continue** using rect1 , which is the reason **we use the & in the function signature** and where we call the function.
5) The area function accesses the width and height fields of the Rectangle instance.
6) This conveys that the **width and height are related to each other**, and it gives descriptive names to the values rather than using the tuple index values of 0 and 1 .
7) This is a win for clarity.

Adding Useful Functionality with Derived Traits

1) It'd be nice to be able to print an instance of Rectangle while we're debugging our program and see the values for all its fields. **Listing 5-11 tries using the println! macro** as we have used in previous chapters. This won't work, however.

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!("rect1 is {}", rect1);
}
```

2) When we run this code, **we get an error** with this core message:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

3) The println! macro can do many kinds of formatting, and by default, the curly brackets tell println! to use formatting known as **Display**: output intended for direct end user consumption.

If we continue reading the errors, we'll find this helpful

```
= help: the trait `std::fmt::Display` is not implemented for `Rectangle`
= note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print)
instead
```

Let's try it! The println! macro call will now look like println!("rect1 is {:?}", rect1); . Putting the specifier :? inside the curly brackets tells println! we want to use an output format called Debug . The Debug trait enables us to print our struct in a way that is useful for developers so we can see its value while we're debugging our code.

Run the code with this change. Drat! We still get an error:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Debug`
```

But again, the compiler gives us a helpful note:

```
= help: the trait `std::fmt::Debug` is not implemented for `Rectangle`
= note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`
```

Rust *does* include functionality to print out debugging information.

- But we have to explicitly opt in to make that functionality available for our struct. To do that, we add the annotation **#[derive(Debug)]** just before the struct definition, as shown in **Listing 5-12**.

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!("rect1 is {:?}", rect1);
}
```

- Now when we run the program, **we won't get any errors**, and we'll see the following output:

```
rect1 is Rectangle { width: 30, height: 50 }
```

Method Syntax

Methods are similar to functions

- They're declared with the **fn keyword** and their name.
- They can have **parameters** and **a return value.**
- They contain some code that is run when **they're called from somewhere else**.
- However, **methods are different from functions** in that they're defined within the context of a struct.
- Their first parameter is always **self**, which represents the instance of the struct the method is being called on.

Defining Methods

- Let's change the area function that has a **Rectangle instance as a parameter** and instead make an area method defined on **the Rectangle struct, as shown in Listing 5-13**.

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Listing 5-13: Defining an area **method on the** Rectangle **struct**

- To define **the function** within the context of Rectangle , we start an impl (implementation) block.

- Then we move the area function within the impl curly brackets and change the first (and in this case, only) parameter to be **self in the signature** and everywhere within the body.

- In main, where we called the area function and passed rect1 as an argument, we can instead use *method syntax* to call the area method on our Rectangle instance. The method syntax goes after an instance: we add a dot followed by the method name, parentheses, and any arguments.

Where's the -> Operator?

1) In **C and C++,** two different operators are used for calling methods: you use . if you're calling a method on the object directly and -> if you're calling the method on a pointer to the object and need to dereference the pointer first. In other words, if object is a pointer, **object->something()** is similar to **(*object).something()** .

2) Rust doesn't have an equivalent to **the -> operator**; instead, Rust has a feature called ***automatic referencing and dereferencing***. Calling methods is one of the few places in Rust that has this behavior.

Here's how it works:

Here's how it works: when you call a method with `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` so `object` matches the signature of the method. In other words, the following are the same:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

The first one looks much cleaner. This automatic referencing behavior works because methods have a clear receiver—the type of `self`. Given the receiver and name of a method, Rust can figure out definitively whether the method is reading (`&self`), mutating (`&mut self`), or consuming (`self`). The fact that Rust makes borrowing implicit for method receivers is a big part of making ownership ergonomic in practice.

Methods with More Parameters

1) Let's practice using methods by implementing a second method on the **Rectangle struct**.
2) This time, we want an instance of Rectangle to take another instance of Rectangle and return true if the second Rectangle can fit completely within self; otherwise it should return false .
3) That is, we want to be able to write the program shown **in Listing 5-14**, once **we've defined the can_hold method**.

Using the as-yet-unwritten can_hold method

```rust
fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    let rect2 = Rectangle { width: 10, height: 40 };
    let rect3 = Rectangle { width: 60, height: 45 };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Listing 5-14: Using the as-yet-unwritten `can_hold` method

And the expected output would look like the following, because both dimensions of `rect2` are smaller than the dimensions of `rect1` but `rect3` is wider than `rect1`:

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

1) We know we want to define a method, so it will be within the **impl Rectangle block.** The method name will be **can_hold** , and it will take an immutable borrow of another Rectangle as a parameter.

2) The return value of **can_hold will be a Boolean**, and the implementation will check whether the width and height of self are both greater than the width and height of the other Rectangle, respectively.

3) Add the new can_hold method to the **impl block from Listing 5-13**, shown in **Listing 5-15.**

Implementing the can_hold method on Rectangle

- When we run this code with the main function in **Listing 5-14**, we'll get our **desired output**.

- Methods can take **multiple parameters** that we add to the signature after the self parameter, and those parameters work just like parameters in functions.

Listing 5-15: Implementing the can_hold method on Rectangle that takes another Rectangle instance as a parameter

```
#![allow(unused_variables)]
fn main() {
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
}
```

Associated Functions

1) Another useful feature of **impl blocks** is that we're allowed to define functions within **impl blocks** that *don't* take self as a parameter.
2) These are called ***associated functions*** because they're associated with the struct. They're still functions, not methods, because they don't have an instance of the struct to work with.
3) You've already used the **String::from associated** function.

Associated functions are often used for constructors.

- Associated functions are often used for constructors **that will return a new instance of the struct**.

- For example, we could provide an associated function that would have one dimension parameter and use that as both **width and height**, thus making it easier to **create a square Rectangle** rather than having to specify the same value twice:

- To call this associated function, we use the :: syntax with the **struct name; et sq = Rectangle::square(3);** is an example. This function is namespaced by the struct: the **::** syntax is used for both associated functions and namespaces created by modules.

Multiple impl Blocks

- Each struct is allowed to have **multiple impl blocks.**
- For example, **Listing 5-15** is equivalent to the code shown in **Listing 5-16**, which **has each method in its own impl block**.

```rust
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Outline

1. Generic Types, Traits, and Lifetimes
    1. 10.1. Generic Data Types
    2. 10.2. **Traits: Defining Shared Behavior**
    3. 10.3. Validating References with Lifetime
2. Functional Language Features: Iterators and Closures
    1. 13.1. Closures: Anonymous Functions
3. Fearless Concurrency
    1. 16.1. Using Threads to Run Code Simultaneously

Generic Data Types

We can use generics to create definitions for items like function signatures or structs, which we can then use with many different concrete data types.

Ways to provide the generic code

- There are two ways to provide the generic code:
- Option<T>
- Result<T, E>

Option<T>

Rust standard library provides Option where 'T' is the generic data type. It provides the generic over one type.

```
enum Option<T>
{
        Some(T),
        None,
}
```

Continue...

In the last case, enum is the custom type where <T> is the generic data type. We can substitute the 'T' with any data type. Let's look at this:

```
let x : Option<i32> = Some(10);  // 'T' is of type i32.

let x : Option<bool> = Some(true);  // 'T' is of type bool.

let x : Option<f64> = Some(10.5); // 'T' is of type f64.

let x : Option<char> = Some('b'); // 'T' is of type char.
```

Continue...

In the previous case, we observe that 'T' can be of any type, i.e., i32, bool, f64 or char. But, if the type on the left-hand side and the value on the right hand side doesn't match, then the error occurs. Let's look at this:

```
let x : Option<i32> = Some(10.8);
```

Continue...

In the above case, type on the left-hand side is i32, and the value on the right-hand side is of type f64. Therefore, the error occurs "type mismatched".

Result<T,E>

Rust standard library provides another data type Result<T,E> which is generic over two type, i.e., T & E:

```
enum Result<T,E>

  {   OK(T),

      Err(E),  }
```

Note: It is not a convention that we have to use 'T' and 'E'. We can use any capital letter.

Generic Functions

Generics can be used in the functions, and we place the generics in the signature of the function, where the data type of the parameters and the return value is specified.

Continue...

When the function contains a single argument of type 'T'.

Syntax:

```
fn function_name<T>(x:T) // body of the function.
```

The above syntax has two parts:

<T> : The given function is a generic over one type.

(x : T) : x is of type T.

Example

```
fn main()  {
  let a = vec![1,2,3,4,5];
  let b = vec![2.3,3.3,4.3,5.3];
  let result = add(&a);
  let result1 = add(&b);


  println!("The value of result is {}",result);
  println!("The value of result1 is {}",result1);
}
```

Example

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

Struct Definitions

Structs can also use the generic type parameter in one or more fields using <> operator.

Syntax:

```
struct structure_name<T>

// Body of the structure.
```

we declare the generic type within the angular brackets just after the structure_name, and then we can use the generic inside the struct definition.

Example

```
fn main()  {
  let integer = Value{a:2,b:3};
  let float = Value{a:7.8,b:12.3};
  println!("Integer values : {},{}",integer.a,integer.b);
  println!("Float values :{},{}",float.a,float.b);
}
```

Example

```
fn main()  {
  let integer = Value{a:2,b:3};
  let float = Value{a:7.8,b:12.3};
  println!("Integer values : {},{}",integer.a,integer.b);
  println!("Float values :{},{}",float.a,float.b);
}
```

Output:

Integer values : 2,3

Float values : 7.8,12.3

In the above example, Value<T> struct is generic over one type and a and b are of the same type. We create two instances integer and float. Integer contains the values of type i32 and float contains the values of type f64.

Let's see another simple example.

```
struct Value<T>  {
  a:T,
  b:T,
}
fn main()  {
  let c = Value{a:2,b:3.6};
  println!("c values : {},{}",c.a,c.b);
```

Continue…

- In the above example, Value<T> struct is generic over one type, and a and b are of the same type. We create an instance of 'c'. The 'c' contains the value of different types, i.e., i32 and f64.
- Therefore, the Rust compiler throws the "mismatched error".

Enum Definitions

An enum can also use the generic data types. Rust standard library provides the Option<T> enum which holds the generic data type. The Option<T> is an enum where 'T' is a generic data type.

Option<T>

- It consists of two variants, i.e., Some(T) and None.
- Where Some(T) holds the value of type T and None does not contain any value.

Option<T>
Some(T)
None

Case

```
enum Option<T>
{
    Some(T),
    None,
}
```

In the above case, Option is an enum which is generic over one type 'T'. It consists of two variants Some(T) and None.

Result<T, E>

We can create the generic of multiple types. This can be achieved through Result<T, E>.

```
enum Result<T,E>
{
    OK(T),
    Err(E)
```

Continue…

- In the above case, Result<T, E> is an enum which is generic over two types, and it consists of two variants, i.e., OK(T) and Err(E).

- OK(T) holds the value of type 'T' while Err(E) holds the value of type 'E'.

Method Definitions

We can implement the methods on structs and enums.

Let's see a simple example:

```
struct Program<T> {
    a: T,
    b: T,
}
```

Example

```
impl<T> Program<T>   {
    fn a(&self) -> &T   {
        &self.a
    }
}
fn main() {
let p = Program{ a: 5, b: 10 };
    println!("p.a() is {}", p.a());
```

Resolving Ambiguities

Rust compiler automatically infers the generic parameters. Let's understand this through a simple scenario:

```
Let mut v = Vec::new();   // creating a vector.

v.push(10);.

println!("{:?}", v); // prints the value of v.
```

In the above case, we insert the integer value into the vector. Therefore, the Rust compiler got to

Continue…

If we delete the second last line, then it looks like;

```
Let mut v = Vec::new();   // creating a vector.
println!("{:?}", v); // prints the value of v.
```

The above case will throw an error that "it cannot infer the type for T".

We can solve the above case in two ways:

1. We can use the following annotation:

```
let v : Vec<bool> = Vec::new();
println!("{:?}",v) ;
```

Continue…

2. We can bind the generic parameter 'T' by using the 'turbofish' ::<> operator:

```
let v = Vec :: <bool> :: new();

println!("{:?}",v) ;
```

Topic 10.2

Traits Defining Shared Behavior

Traits: Defining Shared Behavior

- A trait tells the Rust compiler about functionality a particular **type has and can share with other types**.
- We can use traits to **define shared behavior** in an abstract way.
- We can **use trait bounds** to specify that **a generic** can be any type that has certain behavior.
- Note: Traits are similar to a feature often called ***interfaces*** in other languages, although with some differences.

Defining a Trait

- A type behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types.
- Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

A Summary trait that consists of the behavior provided by a summarize method

- We want to make a **media aggregator library** that can display summaries of data that might be stored in a NewsArticle or Tweet instance.

- To do this, we need a summary from each type, and we need to request that **summary by calling a summarize method on an instance**. **Listing 10-12** shows the definition of a Summary trait that expresses this behavior.

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

- Here, we declare a trait using the **trait keyword** and then the **trait's name**, which is Summary in this case. Inside the curly brackets, we declare the method signatures that describe the behaviors of the types that implement this trait, which in this case is **fn summarize(&self) -> String**.

Implementing a Trait on a Type

- Now that we've **defined the desired behavior using the Summary trait**, we can implement it on the types in our **media aggregator**.
- **Listing 10-13** shows an implementation of the **Summary trait on the NewsArticle struct** that uses the headline, the author, and the location to create the return value of summarize .
- For the Tweet struct, we define summarize as the username followed by the entire text of the tweet, assuming that tweet content is already limited to 280 characters.

Listing 10-13: Implementing the Summary **trait on the** NewsArticle **and** Tweet **types**

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Implementing a trait on a type is similar to implementing regular methods.

- The difference is that after **impl**, we put the trait name that we want to implement, then use the **for** keyword, and then specify the name of the type we want to implement the trait for.

- Within the **impl block**, we put the method signatures that the trait definition has defined. Instead of adding a semicolon after each signature, we use curly brackets and fill in the method body with the specific behavior that we want the methods of the trait to have for the particular type.

After implementing the trait, we can call the methods on instances of NewsArticle and Tweet in the same way we call regular methods, like this

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

```
1 new tweet: horse_ebooks: of course, as you probably already know, people.
```

Let's say this *lib.rs* is for a crate we've called aggregator and someone else wants to use our crate's functionality to implement the Summary trait on a struct defined within their library's scope.

- They would need to bring the trait into their scope first. They would do so by specifying use **aggregator::Summary;** which then would enable them to implement Summary for their type.
- One restriction to note with trait implementations is that we can implement a trait on a type only if either the trait or the type is local to our crate. For example, we can implement standard library traits like **Display** on a custom type like **Tweet** as part of our **aggregator crate** functionality, because the type **Tweet** is local to our **aggregator crate.**

Default Implementations

- Sometimes it's useful to have default behavior for some or all of the methods in a trait instead of requiring implementations for all methods on every type.

Continue...

Listing 10-14 shows how to specify a default string for the `summarize` method of the `Summary` trait instead of only defining the method signature, as we did in Listing 10-12.

Filename: src/lib.rs

```rust
#![allow(unused_variables)]
fn main() {
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
}
```

Listing 10-14: Definition of a `Summary` trait with a default implementation of the `summarize` method

To use a default implementation to summarize instances of **NewsArticle** instead of defining a custom implementation, we specify an empty `impl` block with `impl Summary for NewsArticle {}`.

- Even though we're no longer defining the **summarize** method on **NewsArticle** directly, we've provided a default implementation and specified that **NewsArticle** implements the **Summary** trait.
- As a result, we can still call the **summarize** method on an instance of **NewsArticle** , like this:

```rust
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from("The Pittsburgh Penguins once again are the best
    hockey team in the NHL."),
};

println!("New article available! {}", article.summarize());
```

This code prints `New article available! (Read more...)`.

Default implementations can call other methods in the same trait, even if those other methods don't have a default implementation.

```rust
#![allow(unused_variables)]
fn main() {
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
}
```

To use this version of `Summary`, we only need to define `summarize_author` when we implement the trait on a type:

```rust
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

After we define summarize_author , we can call summarize on instances of the Tweet

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from("of course, as you probably already know, people"),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

This code prints `1 new tweet: (Read more from @horse_ebooks...)`.

Note that it isn't possible to call the default implementation from an overriding implementation of that same method.

Traits as Parameters

- Now that you know how to define and implement traits, we can explore how to use traits to define functions that accept many different types.

```
pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

Instead of a concrete type for the item parameter, we specify the impl keyword and the trait name.

Trait Bound Syntax

- The **impl Trait** syntax works for straightforward cases but is actually syntax sugar for a longer form, which is called ***a trait bound***; it looks like this:

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

The impl Trait syntax is convenient and makes for more concise code in simple cases.

```
pub fn notify(item1: impl Summary, item2: impl Summary) {
```

If we wanted this function to allow `item1` and `item2` to have different types, using `impl Trait` would be appropriate (as long as both types implement `Summary`). If we wanted to force both parameters to have the same type, that's only possible to express using a trait bound, like this:

```
pub fn notify<T: Summary>(item1: T, item2: T) {
```

The generic type `T` specified as the type of the `item1` and `item2` parameters constrains the function such that the concrete type of the value passed as an argument for `item1` and `item2` must be the same.

Specifying Multiple Trait Bounds with the + Syntax

```
pub fn notify(item: impl Summary + Display) {
```

The `+` syntax is also valid with trait bounds on generic types:

```
pub fn notify<T: Summary + Display>(item: T) {
```

With the two trait bounds specified, the body of `notify` can call `summarize` and use `{}` to format `item`.

Clearer Trait Bounds with where Clauses

```rust
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

we can use a `where` clause, like this:

```rust
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

This function's signature is less cluttered: the function name, parameter list, and return type are close together, similar to a function without lots of trait bounds.

Returning Types that Implement Traits

```rust
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know, people"),
        reply: false,
        retweet: false,
    }
}
```

However, you can only use impl Trait if you're returning a single type. For example, this code that returns either a NewsArticle or a Tweet with the return type specified as impl Summary wouldn't work:

```rust
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from("Penguins win the Stanley Cup Championship!"),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from("The Pittsburgh Penguins once again are the best
            hockey team in the NHL."),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course, as you probably already know, people"),
            reply: false,
            retweet: false,
        }
    }
}
```

Fixing the largest Function with Trait Bounds

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^^
  |
  = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

This time when we compile the code, we get a different set of errors:

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
```

This time when we compile the code, we get a different set of errors:

```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 --> src/main.rs:2:23
  |
2 |     let mut largest = list[0];
  |                       ^^^^^^^
  |                       |
  |                       cannot move out of here
  |                       help: consider using a reference instead: `&list[0]`

error[E0507]: cannot move out of borrowed content
 --> src/main.rs:4:9
  |
4 |     for &item in list.iter() {
  |         ^----
  |         ||
  |         |hint: to prevent move, use `ref item` or `ref mut item`
  |         cannot move out of borrowed content
```

A working definition of the largest function that works on any generic type that implements the PartialOrd and Copy traits

- To call this code with only those types that implement the Copy trait, we can add Copy to the trait bounds **of T ! Listing 10-15** shows the complete code of a generic largest function that will compile as long as the types of the values in the slice that we pass into the function implement the **PartialOrd *and* Copy traits, like i32** and char do.

```rust
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Using Trait Bounds to Conditionally Implement Methods

Conditionally implement methods on a generic type depending on trait bounds

```rust
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

Dungeons and Dragons Example

Yes... We have simplified D&D rules for this presentation

The focus of this talk is traits with D&D used as a metaphor.

D&D Races

Dwarf

Elf

Half-Orc

Human

Let’s create some structs!

```
struct Dwarf {

name: String

}
```

```
struct Elf {

name: String

}
```

Let's create some structs!

```
struct Dwarf {
    name: String
}
```

```
struct Elf {
    name: String
}
```

```
struct HalfOrc {
    name: String
}
```

```
struct Human {
    name: String
}
```

Let’s make a character!

```
let my_dwarf = Dwarf {

name: String::from(“NellDwarf”)

};
```

Character Traits

- Strength
- Dexterity
- **Constitution**
- Intelligence
- Wisdom
- Charisma

Let’s make a trait!

```
pub trait Constitution {

}
```

Let’s make a trait!

```
pub trait Constitution {

fn constitution_bonus(&self) -> u8;

}
```

Let's implement that trait!

Let’s implement that trait!

```
impl Constitution for Dwarf {

}
```

Let's implement that trait!

Constitution

constitution_bonus

Let’s make a trait!

```
pub trait Constitution {

fn constitution_bonus(&self) -> u8;

}
```

The constitution bonus for a dwarf is 2

Let’s implement that trait!

```
impl Constitution for Dwarf {

fn constitution_bonus(&self) -> u8 {

}

}
```

Let’s implement that trait!

```
impl Constitution for Dwarf {

fn constitution_bonus(&self) -> u8 {

  2

}

}
```

Let’s make a character!

```
let my_dwarf = Dwarf {

name: String::from(“NellDwarf”)

};
```

Let’s make a character!

```
let my_dwarf = Dwarf {

name: String::from(“NellDwarf”)

};

my_dwarf.constitution_bonus();


// Returns 2
```

Let's implement that trait!

```
struct Dwarf {
    name: String
}
```

```
struct Elf {
    name: String
}
```

```
struct HalfOrc {
    name: String
}
```

```
struct Human {
    name: String
}
```

Let's implement that trait!

Constitution

Let’s implement that trait!

```
impl Constitution for HalfOrc {

}
```

Constitution

Let's implement that trait!

Constitution

constitution_bonus

The constitution bonus for a half-orc is 1

Let’s implement that trait!

```
impl Constitution for HalfOrc {

fn constitution_bonus(&self) -> u8 {

  1

}

}
```

Let’s implement that trait!

```
let my_half_orc = HalfOrc {

name: String::from(“NellOrc”)

};
```

Let’s implement that trait!

```
let my_half_orc = HalfOrc {

name: String::from(“NellOrc”)

};

my_half_orc.constitution_bonus();

// Returns 1
```

Let's implement that trait!

```
struct Dwarf {
    name: String
}
```

```
struct Elf {
    name: String
}
```

```
struct HalfOrc {
    name: String
}
```

```
struct Human {
    name: String
}
```

The constitution bonus for both a human and a half-elf is 0

We could implement it like this...

```
impl Constitution for Elf {

fn constitution_bonus(&self) -> u8 {

0

}
```

Repetitive!

```
}

impl Constitution for Human {
```

Most races have a constitution bonus of 0...

Let’s make 0 the default

Let’s add a default!

```
pub trait Constitution {

fn constitution_bonus(&self) -> u8;

}
```

Let’s add a default!

```
pub trait Constitution {

fn constitution_bonus(&self) -> u8 {

0

}

}
```

Let's implement that trait!

Constitution

Constitution

Let’s implement that trait!

```
impl Constitution for Elf {

}
```

```
impl Constitution for Human {

}
```

Let’s implement that trait!

```
let my_elf = Elf {

name: String::from(“NellElf”)

};
```

Let’s implement that trait!

```
let my_elf = Elf {

name: String::from(“NellElf”)

};



my_elf.constitution_bonus();

// Returns 0
```

Let’s implement that trait!

```
let my_human = Human {

name: String::from(“Nell”)

};
```

Let’s implement that trait!

```
let my_human = Human {

name: String::from(“Nell”)

};


my_human.constitution_bonus();

// Returns 0
```

Yay! We have a trait!

Section Break

Presentation for Rust Quiz # 4

Prepared by: (RUSTLING KNIGHTS)

Anas Baig (Team Leader)
Arsalan Nawaz
Muhammad Danial Siddiqui
Sheikh Hassaan Bin Nadeem

CHAPTER # 6

ENUMS & PATTERN MATCHING

6.1 Enums

Defining Enums

Enumerations or Enums are a custom data type that allow you to define a type by enumerating its possible values.

Example:

- IP Addresses:
  Version 4 or Version 6
- The property of being V4 or V6 at a time make enum structures appropriate, because enum values can only be one of the variants.
- But both versions are still IP Addresses and should be treated as the same type.

Code it:

```
enum IpAddrKind {
    V4,
    V6,
}
```

Variants

Creating Instances:

```
#![allow(unused_variables)]

fn main() {

enum IpAddrKind {

    V4,

    V6, }

let four = IpAddrKind::V4;

let six = IpAddrKind::V6; }
```

Defining & Calling a Function:

We can define a function that takes any IpAddrkind and can call it with either of its variant.

```
#![allow(unused_variables)]

fn main() {

enum IpAddrKind {

    V4,

    V6,

}
```

Where’s the Data?

- We’ve just learnt how to make Enums and to create instances and defining and calling functions.
- But we have not stored any data in last example.
- So we can add data using structs and by some other ways..

Adding Data Using Structs:

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

A Concise Way Using an Enum:

Hurray!

We've finally used structs to somehow associate values with the variants.

BUT

We can do the same thing in a more concise way using just an enum.

Adding Data Using Enums:

This new definition of the IpAddr enum says that both V4 and V6 variants will have associated String values.

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

An Advantage Against Struct:

Each variant can have different types and amounts of associated data in enums.

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

Standard Library Definition for IpAddr:

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

As we wanted to store IP addresses and encode which kind they are is so common that the standard library has a definition we can use! Let's look at how the standard library defines IpAddr.

Is This A Win For Any Data Type!

Yes! The previous codes shown us that we can put any kind of data inside an Enum Variant.
We can even include Enums!

Another Example:

A message enum which has a wide variety of types embedded in its variants.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Same Thing Using Structs:

Defining an enum with variants is almost similar to defining different kinds of struct definitions.

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); //
tuple struct
```

So Enums are Easy!

Because if we use different structs which have different types than we couldn’t as easily define a ***function*** to take any of these kinds of messages as we could with the Message enum which itself is only single type.

One More Similarity: “Impl Block”

Just as we’re able to define methods on structs using impl, we’re also able to define methods on enums.

```
impl Message {
    fn call(&self) {
        // method body would be defined here
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

6.2 The “Option” Enum

The “Option” Enum:

The “Option” enum is another enum defined by the standard library. The Option type is used in many places because it encodes the very common scenario in which a value could be something or it could be nothing.

Null Feature for Languages:

Null is a value that means there is no value there. In languages with null, variables can always be in one of two states: null or not-null.

Problem?

The problem with null values is that if you try to use a null value as a not-null value, you'll get an error of some kind. Because this null or not-null property is pervasive, it's extremely easy to make this kind of error.

Does Rust have Nulls?

No! However, the concept that null is trying to express is still a useful one: a null is a value that is currently invalid or absent for some reason.

So, What does Rust have?

Enum! that can encode the concept of a value being present or absent. This enum is “Option<T>”, and it is defined in the standard library.

The “Option<T>” Enum Defined by Std. Lib:

The Option<T> enum is so useful that it’s even included in the prelude; you don’t need to bring it into scope explicitly. In addition, so are its variants: you can use Some and None directly without the Option:: prefix. The Option<T> enum is still just a regular enum, and Some(T) and None are still variants of type Option<T>.

```
enum Option<T>
{
    Some(T),
    None,
}
}
```

Example:

Here we've an example of using Option values to hold number types and string types.If we use None rather than Some, we need to tell Rust what type of Option<T> we have, because the compiler can't infer the type that the Some variant will hold by looking only at a None value.

```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

So Why Option<T> is Better than having Null?

Because Option<T> and T (where T can be any type) are different types, the compiler won't let us use an Option<T> value as if it were definitely a valid value. Like a code which is trying to add an i8 to an Option<i8> won't compile and will give an error because Rust doesn't understand how to add an i8 and an Option<i8>, because they're different types.

```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

That’s How You Code Safely and Confidently!

In order to have a value that can possibly be null, you must explicitly opt in by making the type of that value Option<T>. Everywhere that a value has a type that isn’t an Option<T>, you can safely assume that the value isn’t null. That’s how Rust increases the safety of code.

6.3 “Match” Control Flow Operator

The “Match” Control Flow Operator:

- “Match” is an extremely powerful control flow operator which allows you to compare a value against a series of patterns and then execute code based on which pattern matches.

- Patterns can be made up of literal values, variable names, wildcards, and many other things.

Let’s Talk About Coin-Sorting Machine!

Cardboard Coin-Sorting Machine

Coins slide down a track with variously sized holes along it, and each coin falls through the first hole it encounters that it fits into.

Code It:

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

In the same way as the coin sorting machine works, values go through each pattern in a match, and at the first pattern the value “fits,” the value falls into the associated code block to be used during execution.

Let’s Break Down “Match”

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        },
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Why didn't we use
" If " ?

Because the expression with " If " needs to return a boolean value but with "Match", it can be any type...

Curly Brackets in Match Arm:

```
fn value_in_cents(coin: Coin) -> u8
{
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter
from {:?}!", state);
            25
        },
    }
}
```

Curly brackets typically aren't used if the match arm code is short, as it was in the last example where each arm just returns a value. If you want to run multiple lines of code in a match arm, you can use curly brackets.

Patterns That Bind to Values:

Another useful feature of match arms is that they can bind to the parts of the values that match the pattern. This is how we can extract values out of enum variants.

Example:

```
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

From last example, let's change one of our enum variants to hold data inside it. From 1999 through 2008, the United States minted quarters with different designs for each of the 50 states on one side. No other coins got state designs, so only quarters have this extra value. We can add this information to our enum by changing the Quarter variant to include a UsState value stored inside it.

Continued..

Let’s imagine that a friend of ours is trying to collect all 50 state quarters. While we sort our loose change by coin type, we’ll also call out the name of the state associated with each quarter so if it’s one our friend doesn’t have, they can add it to their collection.

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter
from {:?}!", state);
            25
        },
    }
}
```

What if we Call a Particular State!

If we were to call
value_in_cents(Coin::Quarter(UsState::Alaska)),
coin would be Coin::Quarter(UsState::Alaska).

When we compare that value with each of the match arms, none of them match until we reach Coin::Quarter(state). At that point, the binding for state will be the value UsState::Alaska. We can then use that binding in the println! expression, thus getting the inner state value out of the Coin enum variant for Quarter.

6.4 Matching With Option<T>

Matching With Option<T>:

When we talked about the Option<T>, we wanted to get the inner T value out of the “Some” case when using Option<T>; we can also handle Option<T> using match as we did with the Coin enum! Instead of comparing coins, we’ll compare the variants of Option<T>, but the way that the match expression works remains the same.

Example:

Let’s write a function that takes an Option<i32> and, if there’s a value inside, adds 1 to that value. If there isn’t a value inside, the function should return the None value and not attempt to perform any operations.

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Matches are Exhaustive!

Consider this version of our plus_one function that has a bug and won't compile because we didn't handle the None case.

```
fn plus_one(x: Option<i32>) ->
Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

The _ Placeholder Pattern:

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

Rust also has a pattern we can use when we don't want to list all possible values. For example, a u8 can have valid values of 0 through 255. If we only care about the values 1, 3, 5, and 7, we don't want to have to list out 0, 2, 4, 6, 8, 9 all the way up to 255. Fortunately, we don't have to: we can use the special pattern _ instead:

CHAPTER # 7

PACKAGES,CRATES & MODULES

INTRODUCTION :

What names does the compiler know about at this location in the code?

What functions am I allowed to call? What does this variable refer to?

The Module System encompasses

- Packages are a Cargo feature that let you build, test, and share crates.
- Crates are a tree of modules that produce a library or executable.
- Modules and the use keyword let you control the scope and privacy of paths.
- A path is a way of naming an item such as a struct, function, or modules

PACKAGES AND CRATES FOR MAKING LIBRARIES AND EXECUTABLES :

- A crate is a binary or library.
- The crate root is a source file that is used to know how to build a crate.
- A package has a Cargo.toml that describes how to build one or more crates. At most one crate in a package can be a library.

(CONTINUED):

Example of a Package:

```
$ cargo new my-project
     Created binary (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
main.rs
```

THE MODULE SYSTEM TO CONTROL SCOPE AND PRIVACY :

Features:

1. Modules, a way to organize code and control the privacy of paths
2. Paths, a way to name items
3. use, a keyword to bring a path into scope
4. pub, a keyword to make items public
5. Renaming items when bringing them into scope with the as keyword

THE MODULE SYSTEM TO CONTROL SCOPE AND PRIVACY :

Features (continued):

6. Using external packages
7. Nested paths to clean up large use lists
8. Using the glob operator to bring everything in a module into scope
9. How to split modules into individual files

MODULES :

Modules let us organize code into groups.

Example:

```
mod sound {
    fn guitar() {
        // Function body code goes here
    }
}
fn main() {
}
```

MODULES (NESTED) :

```
mod sound {
    mod instrument {
        mod woodwind {
            fn clarinet() {
                // Function body code goes here
            }
        }
    }
    mod voice {
    }
}
fn main() {
}
```

MODULES (NESTED) :

Hierarchy

```
crate
 └── sound
    ├── instrument
    |    └── woodwind
    └── voice
```

MODULE TREE VS FILESYSTEM :

- This tree might remind you of the directory tree of the filesystem you have on your computer
- Just like directories in a filesystem, you place code inside whichever module will create the organization you'd like.
- Another similarity is that to refer to an item in a filesystem or a module tree, you use its **path**

PATHS :

If we want to call a function, we need to know its path

A path can take two forms:

An absolute path starts from a crate root by using a crate name or a literal “crate”.

A relative path starts from the current module and uses “self”, “super”, or an identifier in the current module.

Both absolute and relative paths are followed by one or more identifiers separated by double colons (::) .

PATHS (EXAMPLE) :

```
mod sound {
    mod instrument {
        fn clarinet() {
            // Function body code goes here
        }
    }
}
fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();
    // Relative path
    sound::instrument::clarinet();
}
```

MODULES AS PRIVACY BOUNDARY :

- Modules are used for organization
- Modules are also used for privacy boundary in Rust.
- Privacy rules:
  - All items (functions, methods, structs, enums, modules, and constants) are private by default.
  - You can use the pub keyword to make an item public.
  - You aren’t allowed to use private code defined in modules that are children of the current module.
  - You are allowed to use any code defined in ancestor modules or the current module.

PATHS (EXAMPLE WITH PUB) :

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}
fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();

    // Relative path
    sound::instrument::clarinet();
}
```

STARTING RELATIVE PATHS WITH SUPER:

```
fn main() {}
```

```
mod instrument {
    fn clarinet() {
        super::breathe_in();
    }
}

fn breathe_in() {
    // Function body code goes here
}
```

USING PUB WITH STRUCTS :

```
mod plant {
    pub struct Vegetable {
        pub name: String,
        id: i32,
    }
    impl Vegetable {
        pub fn new(name: &str) -> Vegetable
{
            Vegetable {
                name: String::from(name),
                id: 1,
            }
        }
    }
}
```

```
fn main() {
    let mut v =
plant::Vegetable::new("squash");
    v.name = String::from("butternut squash")
    println!("{} are delicious", v.name);
    // println!("The ID is {}", v.id);
}
```

USING PUB WITH ENUMS :

```
mod menu {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

fn main() {
    let order1 = menu::Appetizer::Soup;
    let order2 = menu::Appetizer::Salad;
}
```

THE “USE” KEYWORD TO BRING ABSOLUTE PATHS INTO A SCOPE :

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}
use crate::sound::instrument;
fn main() {
    instrument::clarinet();
    instrument::clarinet();
    instrument::clarinet();
}
```

THE “USE” KEYWORD TO BRING RELATIVE PATHS INTO A SCOPE :

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}
use self::sound::instrument;
fn main() {
    instrument::clarinet();
    instrument::clarinet();
    instrument::clarinet();
}
```

ABSOLUTE VS RELATIVE PATHS WITH "USE" :

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }}}
mod performance_group {
    use crate::sound::instrument;
    pub fn clarinet_trio() {
        instrument::clarinet();
        instrument::clarinet();
        instrument::clarinet();
    }}
fn main() {
    performance_group::clarinet_trio();
}
```

Idiomatic use Paths for Functions

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        }
    }
}
use crate::sound::instrument::clarinet;
fn main() {
    clarinet();
    clarinet();
    clarinet();
}
```

Idiomatic use Paths for Structs/Enums & Other Items

```
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert(1, 2);
}
```

Idiomatic use Paths for Structs/Enums & Other Items

- Exception to this idiom is if the use statements would bring two items with the same name into scope, which isn't allowed.
- use std::fmt;
  use std::io;
  fn function1() -> fmt::Result {}
  fn function2() -> io::Result<()> {}

- We would have two Result types in the same scope and Rust wouldn't know which one we meant when we used Result.

Renaming Types Brought Into Scope with the as Keyword

- We can bring two types of the same name into the same scope
- We can specify a new local name for the type by adding “as” and a new name after the “use”

- Example:

```
use std::fmt::Result;
use std::io::Result as IoResult;
fn function1() -> Result {  }
fn function2() -> IoResult<()> {  }
```

Re-exporting Names with pub use

- When you bring a name into scope with the use keyword, the name being available in the new scope is private.
- If you want to enable code calling your code to be able to refer to the type as if it was defined in that scope just as your code does, you can combine pub and use.
- This technique is called re-exporting because you’re bringing an item into scope but also making that item available for others to bring into their scope.

Re-exporting Names with pub use (Example)

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            // Function body code goes here
        } } }
mod performance_group {
    pub use crate::sound::instrument;
    pub fn clarinet_trio() {
        instrument::clarinet();
        instrument::clarinet();
        instrument::clarinet();
    } }
fn main() { performance_group::clarinet_trio();
    performance_group::instrument::clarinet(); }
```

Using External Packages

- There are many packages that members of the community have published on https://crates.io.
- Pulling any of them into your package involves following steps:
  - listing them in your package's Cargo.toml
  - bringing items defined in them into a scope in your package with use.
- Example: A package "rand" can be pulled with following code in Cargo.toml file:

```
[dependencies]
rand = "0.5.5"
```

Nested Paths for Cleaning Up Large use Lists

- When you use many items defined by the same package or in the same module, listing each item on its own line can take up a lot of vertical space in your files.

- For example, these two use statements bring items from std into scope:
  use std::cmp::Ordering;
  use std::io;

- We can use nested paths to bring the same items into scope in one line instead of two as:
  use std::{cmp::Ordering, io};

Bringing All Public Definitions into Scope with the Glob Operator

- To bring all public items defined in a path into scope, you can specify that path followed by *, the glob operator
  use std::collections::*;

- The glob operator is often used when testing to bring everything under test into the tests module

- The glob operator is also sometimes used as part of the prelude pattern

Separating Modules into Different Files

- So far we have defined multiple modules in one file.
- When modules get large, you may want to move their definitions to a separate file to make the code easier to navigate.

```
mod sound;
fn main() {
    // Absolute path
    crate::sound::instrument::clarinet();
    // Relative path
    sound::instrument::clarinet();
}
```

Separating Modules into Different Files (continued)

src/sound.rs file:

```
pub mod instrument {
    pub fn clarinet() {
        // Function body code goes here
    }
}
```

- The module tree remains the same and the function calls in main continue to work without any modification, even though the definitions live in different files. This lets you move modules to new files as they grow in size.

CHAPTER # 8

COMMON COLLECTION

Common Collections

Rust's standard library includes a number of very useful data structures called *collections*. Most other * **data types** represent one specific value, but collections can contain multiple values. We'll discuss three collections that are used very often in Rust programs:

1. A Vector
2. A String
3. A Hash Map

1. <u>Vector:</u>

Allows you to store a variable number of values next to each other.

2. <u>String:</u>

Is a collection of characters.

3. <u>Hash Map:</u>

Allows you to associate a value with a particular key. It's a particular implementation of the more general data structure called a *map*.

STORING LISTS OF VALUES WITH VECTORS

The first collection type we'll look at is **Vec<T>**, also known as a **vector**. Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory. Vectors can only store values of the same type. They are useful when you have a list of items.

Creating The Empty Vector:

```
fn main() {
    let v: Vec<i32> = Vec::new():
}
```

Creating The Vector Containing Values:

```
fn main() {
    let v = vec![1,2,3];
    println!("The First Value is = {},
    Second Value is = {},Third Value is
    = {} ",v[0], v[1], v[2]);
}
```

Updating the Vector

```
fn main() {
   // Creating The Empty Vector and Pushing The Value
   let mut v1 = Vec::new();
   v1.push(5);
   v1.push(6);
   v1.push(7);
   v1.push(8);
   println!("Empty Vector v1 having no value after
   Pushing Value in Vector v1 {}, {} , {}, {}",
   v1[0], v1[1], v1[2], v1[3]);
}
```

UPDATING THE VECTOR:

```
fn main() {
   // Creating Vector Having Some Value
   let mut v = vec![1, 2, 3];
   v.push(4);
   v.push(5);
   v.push(6);
   v.push(7);
   println!("The Vector having Value is {}, {}, {}
   After Pushing Value The Vector v Value is
   {}, {}, {}, {} ",
   v[0], v[1], v[2], v[3], v[4], v[5], v[6]);
}
```

READING ELEMENTS OF VECTOR:

```
fn main() {
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {}", third);

match v.get(2) {
    Some(third) => println!("The third element is {}",
third),
    None => println!("There is no third element."),
}
}
```

READING ELEMENTS (PANICKED):

```
fn main() {
  let v = vec![1, 2, 3, 4, 5];
  let does_not_exist = &v[100];
  let does_not_exist = v.get(100);
}
```

READING ELEMENTS (Error):

```
fn main() {
  let v = vec![1, 2, 3, 4, 5];
  Let first = &v[0];
  v.push(6);
  println!(“The First Element is: {}, First);
}
```

ITERATING OVER THE VALUE IN A VECTOR:

```
fn main() {
  let v = vec![100, 32, 57];
  for i in &v5 {
  println!("{}",i);
  }
}
```

ITERATING OVER MUTABLE REFERENCE IN A VECTOR:

```
fn main() {
  let v = vec![10, 20, 30];
  for i in &mut v6 {
  *i += 50;
  println!("{}",i);
  }
}
```

USING ENUM TO STORE MULTIPLE TYPE

The vectors can only store values that are the same type. This can be inconvenient; there are definitely use cases for needing to store a list of items of different types.

Fortunately, the variants of an enum are defined under the same enum type, so when we need to store elements of a different type in a vector, we can define and use an enum!

STRING

STORING UTF-8 ENCODED TEXT WITH STRING

What are Strings?

Strings. There are two types of **strings in Rust**: **String** and &str. **String** is heap allocated, growable and not null terminated. &str is a slice (&[u8]) that always points to a valid UTF-8 sequence, and can be used to view into a **String** , just like &[T] is a view into Vec<T> .

Rust has only one string type in the core language, which is the string slice str that is usually seen in its borrowed form &str. We talked about *string slices*, which are references to some UTF-8 encoded string data stored elsewhere. String literals, for example, are stored in the program's binary and are therefore string slices.

CREATING THE STRING:

```
fn main() {
    let s = String::new();
    println!("Creating a Empty String: {}", s);
    let data = "initial contents";
    let s = data.to_string();
    println!("The Value of s: {}", s);
    {
        // the method also works on a literal directly:
        let d = String::from(“initial contents);
        println!("The Value of d: {}", d);
    }
}
```

UPDATING THE STRING:

```
#![allow(unused_variables)]
fn main() {
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {}", s2);
}
```

CONCATENATION WITH THE OPERATORS:

```
fn main() {
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2;
    println!("{}",s3);
}
```

CONCATENATION WITH THE OPERATORS:

```
fn main (){
    let s1 = String::from("tic");
    let s2 = String::from("tac");
    let s3 = String::from("toe");
    let s = s1 + "-" + &s2 + "-" + &s3;
    println!("{}",s);
}
```

CONCATENATION WITH FORMAT MACRO

```
fn main() {
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
let s = format!("{}-{}-{}", s1, s2, s3);
}
```

INDEXING INTO STRINGS:

```
fn main (){
    let s1 = String::from("hello");
    let h = s1[0];
}
```

This code will result error:

Rust strings don’t support indexing. But why not? To answer that question, we need to discuss how Rust stores strings in memory.

STRING BYTES STORAGE:

```
fn main (){
    let len = String::from("Hola").len();
    println!("{}",len);
{
    let len = String::from("Здравствуйте").len();
    println!("{}",len);
}
}
```

BYTES, SCALAR VALUES & GRAPHEME CLUSTER:

In UTF-8 there are three ways to look at strings from Rust's perspective:

1. **By Bytes**
2. **By Scalar**
3. **By Grapheme Cluster**

SLICING STRING:

```
fn main (){
    let hello = "Здравствуйте";
    let s = &hello[0..4];
    println!("{}",s)
}
```

ITERATING OVER STRING:

```
fn main (){
    for c in "नमस्ते".chars(){
    println!("{}", c);}
{
    for b in "नमस्ते".bytes() {
    println!("{}", b);}
}
}
```

HASH MAP

Hash Map

The type HashMap<K, V> stores a mapping of **keys** of type K to **values** of type V. It does this via a ***hashing function***, which determines how it places these keys and values into memory. Hash maps are useful when you want to look up data not by using an index, as you can with vectors, but by using a key that can be of any type.

CREATING NEW HASH MAP

```
use std::collections::HashMap;
fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);
    for (key, value) in &scores {
        println!("{}: {}", key, value);
    }
    println!("{:?}", scores);
}
```

ANOTHER WAY TO CREATING NEW HASH MAP

```
use std::collections::HashMap;
fn main() {
  let teams = vec![String::from("Blue"), String::from("Yellow")];
  let initial_scores = vec![10, 50];
  let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
  for (key, value) in &scores {
      println!("{}: {}", key, value);
  }
  println!("{:?}", scores);
}
```

HASH MAP AND OWNERSHIP

```
use std::collections::HashMap;
fn main() {
  let field_name = String::from("Favorite color");
  let field_value = String::from("Blue");
  let mut map = HashMap::new();
  map.insert(field_name, field_value);
  println!("{:?}", map);
  }
```

ACCESSING VALUE IN A HASH MAP

```
use std::collections::HashMap;
fn main(){
  let mut scores = HashMap::new();
  scores.insert(String::from("Blue"), 10);
  scores.insert(String::from("Yellow"), 50);
  let team_name = String::from("Blue");
  let score = scores.get(&team_name);
  for (key, value) in &scores {
      println!("{}: {}", key, value);
}
       println!("{:?}", score);
   }
```

UPDATING A HASH MAP

OVERWRITING THE VALUE:

```
use std::collections::HashMap;
fn main()
{
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    println!("{:?}", scores);
    scores.insert(String::from("Blue"), 25);
    println!("{:?}", scores);
}
```

UPDATING A HASH MAP

INSERTING THE VALUE:

```
use std::collections::HashMap;
fn main(){
  let mut scores = HashMap::new();
  scores.insert(String::from("Blue"), 10);
  scores.entry(String::from("Yellow")).or_insert(50);
  scores.entry(String::from("Blue")).or_insert(50);
  println!("{:?}", scores);
}
```

UPDATING A HASH MAP

UPDATING THE VALUE:

```
use std::collections::HashMap;
fn main(){
  let text = "hello world wonderful world";
  let mut map = HashMap::new();
  for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}
println!("{:?}", map);
}
```

HASHING FUNCTION:

By default, HashMap uses a “cryptographically strong”[1] hashing function that can provide resistance to Denial of Service (DoS) attacks. This is not the fastest hashing algorithm available, but the trade-off for better security that comes with the drop in performance is worth it. If you profile your code and find that the default hash function is too slow for your purposes, you can switch to another function by specifying a different hasher. A hasher is a type that implements the BuildHasher trait.

CHAPTER # 9

ERROR HANDLING

CHAPTER # 9

ERROR HANDLING

ERROR :

In computer **programming**, a logic **error** is a bug in a **program** that causes it to operate incorrectly, but not to terminate abnormally (or crash). ... Unlike a **program** with a syntax **error**, a **program** with a logic **error** is a valid **program** in the language, though it does not behave as intended.Error resulting from bad code in some program involved in producing the erroneous result.

TYPES OF ERRORS :

There are basically three types of errors that you must contend with when writing programs:

- Syntax errors - Syntax errors represent ***grammar errors*** in the use of the programming language
- Runtime errors - Runtime errors occur when a program with ***no syntax errors*** asks the computer to do something that the computer is ***unable to reliably do***.
- Logic errors - Logic errors occur when there is a ***design flaw*** in your program.

SMART COMPILER :

In Rust - Compiler does the most significant job to prevent errors in Rust programs. It ***analyzes the code at compile-time*** and issues warnings, if the code does not follow memory management rules or lifetime annotations correctly.
Rust compiler checks not only issues related with lifetimes or memory management and also common coding mistakes.

EXPLAIN ERROR CODE :

Error messages are very descriptive and we can easily see where is the error. But while we can not identify the issue via the error message, ***rustc --explain*** commands help us **to identify the error type and how to solve** it, by showing **simple code samples** which express the same problem and the solution we have to use.

```
// ------ Compile-time error ------ //
error[E0382]: use of moved value: `a`
 --> src/main.rs:6:22
  |
3 |     let b = a;
  |         - value moved here
4 |
5 |     println!("{:?}", a);
  |                      ^ value used here after move
  |
  = note: move occurs because `a` has type `std::vec::Vec<i32>`,
which does not implement the `Copy` trait

error: aborting due to previous error
For more information about this error, try `rustc --explain E0382`.

// instead using #[allow(unused_variables)], consider using "let _ b
= a;" in line 4.
// Also you can use "let _ =" to completely ignore return values
```

PANICKING :

- In some cases, while an error happens we can not do anything to handle it, **if the error is something, which should not have happened**. In other words, if it's an **unrecoverable error**.
- Also **when we are not using a feature-rich debugger or proper logs**, sometimes we need to **debug the code by quitting the program from a specific line of code** by printing out a specific message or a value of a variable binding to understand the current flow of the program.
  For above cases, we can use panic! macro.

PANIC! MACRO :

Rust has the **`panic`**! macro. When the **`panic`**! macro executes, your program will print a failure message, unwind and clean up the stack, and then quit. This most commonly occurs when a bug of some kind has been detected and it’s not clear to the programmer how to handle the error.

01. Quit From A Specific Line.

```
fn main() {
    // some code

    // if we need to debug in here
    panic!();
}

// -------------- Compile-time error --------------
thread 'main' panicked at 'explicit panic', src/main.rs:5:5
```

02. Quit With A Custom Error Message.

```
#[allow(unused_mut)] //A lint attribute used to suppress the warning;
username variable does not need to be mutable
fn main() {
    let mut username = String::new();

    // some code to get the name

    if username.is_empty() {
        panic!("Username is empty!");
    }

    println!("{}", username);
}

// -------------- Compile-time error --------------
thread 'main' panicked at 'Username is empty!', src/main.rs:8:9
```

USING A PANIC! BACKTRACE :

Let’s look at another example to see what it’s like when a panic! call comes from a library because of a bug in our code instead of from our code calling the macro directly. Listing 9-1 has some code that attempts to access an element by index in a vector.

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

BACKTRACE :

To protect your program from this sort of vulnerability, if you try to read an element at an index that doesn’t exist, Rust will stop execution and refuse to continue. Let’s try it and see:

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3
but the index is 99', libcore/slice/mod.rs:2448:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

RECOVERABLE ERRORS WITH RESULTS :

- Most errors aren’t serious enough to require the program to stop entirely. Sometimes, when a function fails, it’s for a reason that you can easily interpret and respond to. For example, if you try to open a file and that operation fails because the file doesn’t exist, you might want to create the file instead of terminating the process.

Recall from “Handling Potential Failure with the Result Type” , that the Result enum is defined as having two variants, Ok and Err, as follows:

```
#![allow(unused_variables)]
fn main() {
enum Result<T, E> {
    Ok(T),
    Err(E),
}
}
```

How do we know File::open returns a Result?

Let’s try it! We know that the return type of File::open isn’t of type u32, so let’s change the let f statement to this:

```
let f: u32 = File::open("hello.txt");
```

Using a match expression to handle the Result variants that might be returned

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening
the file: {:?}", error)
        },
    };
}
```

Output from the panic! macro:

```
thread 'main' panicked at 'There was a problem
opening the file: Error { repr:
Os { code: 2, message: "No such file or
directory" } }', src/main.rs:9:12
```

MATCHING ON DIFFERENT ERRORS

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Tried to create file but there was
a problem: {:?}", e),
            },
            other_error => panic!("There was a problem opening the
file: {:?}", other_error),
        },
    };
}
```

The next code has the same behavior as the last code but doesn't contain any match expressions and look up the unwrap_or_else method in the standard library documentation. There’s many more of these methods that can clean up huge nested match expressions when dealing with errors.

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let f = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error|
{
                panic!("Tried to create file but there was a
problem: {:?}", error);
            })
        } else {
            panic!("There was a problem opening the file:
{:?}", error);
        }
    });
}
```

Shortcut for PANIC on ERROR:

- Unwrap
- Expect

UNWRAP

If the Result value is the Ok variant, unwrap will return the value inside the Ok. If the Result is the Err variant, unwrap will call the panic! macro for us. Here is an example of unwrap in action:

```rust
use std::fs::File;

fn main() {

    let f = File::open("hello.txt").unwrap();

}
```

Output

```
thread 'main' panicked at 'called `Result::unwrap()`
on an `Err` value: Error {

repr: Os { code: 2, message: "No such file or
directory" } }',
```

EXPECT

Another method, expect, which is similar to unwrap, lets us also choose the panic! error message. Using expect instead of unwrap and providing good error messages can convey your intent and make tracking down the source of a panic easier. The syntax of expect looks like this:

```
use std::fs::File;

fn main() {

    let f =
File::open("hello.txt").expect("Failed to open
hello.txt");

}
```

Unwrap_err() for Result types

```
// 01. unwrap_err error message for Ok
fn main() {
    let o: Result<i8, &str> = Ok(8);

    o.unwrap_err();
}
```

Expect_err() for Result types ;

```
// 02. expect_err error message for Ok
fn main() {
    let o: Result<i8, &str> = Ok(8);

    o.expect_err("Should not get Ok value");
}
```

unwrap_or();

```
fn main() {
    let v1 = 8;
    let v2 = 16;

    let s_v1 = Some(8);
    let n = None;

    assert_eq!(s_v1.unwrap_or(v2), v1); // Some(v1) unwrap_or v2
= v1
    assert_eq!(n.unwrap_or(v2), v2);    // None unwrap_or v2 =
v2

    let o_v1: Result<i8, &str> = Ok(8);
    let e: Result<i8, &str> = Err("error");

    assert_eq!(o_v1.unwrap_or(v2), v1); // Ok(v1) unwrap_or v2 =
v1
    assert_eq!(e.unwrap_or(v2), v2);    // Err unwrap_or v2 = v2
}
```

unwrap_or_default();

```
fn main() {
    let v = 8;
    let v_default = 0;

    let s_v: Option<i8> = Some(8);
    let n: Option<i8> = None;

    assert_eq!(s_v.unwrap_or_default(), v);       // Some(v)
unwrap_or_default = v
    assert_eq!(n.unwrap_or_default(), v_default); // None
unwrap_or_default = default value of v

    let o_v: Result<i8, &str> = Ok(8);
    let e: Result<i8, &str> = Err("error");

    assert_eq!(o_v.unwrap_or_default(), v);       // Ok(v)
unwrap_or_default = v
    assert_eq!(e.unwrap_or_default(), v_default); // Err
unwrap_or_default = default value of v
}
```

UNWRAP_OR_ELSE();

```
fn main() {
    let v1 = 8;
    let v2 = 16;

    let s_v1 = Some(8);
    let n = None;
    let fn_v2_for_option = || 16;

    assert_eq!(s_v1.unwrap_or_else(fn_v2_for_option), v1); // Some(v1)
unwrap_or_else fn_v2 = v1
    assert_eq!(n.unwrap_or_else(fn_v2_for_option), v2);    // None
unwrap_or_else fn_v2 = v2

    let o_v1: Result<i8, &str> = Ok(8);
    let e: Result<i8, &str> = Err("error");
    let fn_v2_for_result = |_| 16;

    assert_eq!(o_v1.unwrap_or_else(fn_v2_for_result), v1); // Ok(v1)
unwrap_or_else fn_v2 = v1
    assert_eq!(e.unwrap_or_else(fn_v2_for_result), v2);    // Err
unwrap_or_else fn_v2 = v2
}
```

PROPAGATING ERRORS :

When you're writing a function whose implementation calls something that might fail, instead of handling the error within this function, you can return the error to the calling code so that it can decide what to do.

- we can handle them inside the same function. Or,
- we can return ***None*** and ***Err*** types immediately to the caller. So the caller can decide how to handle them.

A function that returns errors to the calling code using match:

```rust
use std::io;
use std::io::Read;
use std::fs::File;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```

OPERATORS

- If an Option type has **Some** value or a Result type has a **Ok** value, **the value inside them** passes to the next step.
- If the Option type has **None** value or the Result type has **Err** value, it **returns them immediately** to the caller of the function.

Example with Option type,

```
fn main() {
    if complex_function().is_none() {
        println!("X not exists!");
    } }
fn complex_function() -> Option<&'static str> {
    let x = get_an_optional_value()?; // if None, returns immidiately; if
Some("abc"), set x to "abc"

    // some more code example
    println!("{}", x); // "abc" ; if you change line 19 `false` to `true`

    Some("")
}
fn get_an_optional_value() -> Option<&'static str> {

    //if the optional value is not empty
    if false {
        return Some("abc");
    }
    //else
    None
}
```

Example with Result Type,

```
fn main() {
    // `main` function is the caller of `complex_function` function
    // So we handle errors of complex_function(), inside main()
    if complex_function().is_err() {
        println!("Can not calculate X!");
    }
}
fn complex_function() -> Result<u64, String> {
    let x = function_with_error()?; // if Err, returns immediately; if Ok(255),
set x to 255

    // some more code example
    println!("{}", x); // 255 ; if you change line 20 `true` to `false`
    Ok(0)
}
fn function_with_error() -> Result<u64, String> {
    //if error happens
    if true {
        return Err("some message".to_string());
    }
    // else, return valid output
    Ok(255)
}
```

The ? Operator Can Only Be Used in Functions That Return Result :

- The ? operator can only be used in functions that have a return type of Result

- Let’s look at what happens if we use the **?** operator in the main function ;

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").unwrap();
}
```

- When we compile this code, we get the following error message:

```
thread 'main' panicked at 'called
`Result::unwrap()` on an `Err` value: Error {
repr: Os { code: 2, message: "No such file or
directory" } }',
src/libcore/result.rs:906:4
```

- we write the main function so that it does return a Result<T, E> :

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt").expect("Failed to
open hello.txt");
}
```

TIP !

If you want to know about the all kind of errors *std::fs::File::open*() can produce, check the error list on *std::fs::OpenOptions*.

TO PANIC! OR NOT TO PANIC! :

To Panic!

So how do we decide when we should call **panic!**
When code **panics**, there's no way to recover. We could call **panic!** for any error situation, whether there's a possible way to recover or not, but then we're making the decision on behalf of the code calling our code that a situation is **unrecoverable.**

TO PANIC! OR NOT TO PANIC!:

Not To Panic!

So how do we decide when we should return **Result?**

When we choose to return a **Result** value, we give the calling code options rather than making the decision for it. The calling code could choose to attempt to recover in a way that's appropriate for its situation.

EXAMPLES, PROTOTYPE CODE, AND TEST:

EXAMPLES:

When you're writing an example to illustrate some concept, having robust error-handling code in the example as well can make the example less clear. In examples, it's understood that a call to a method like unwrap that could panic is meant as a placeholder for the way you'd want your application to handle errors, which can differ based on what the rest of your code is doing.

PROTOTYPE CODE

Prototype:

Similarly, the unwrap and expect methods are very handy when prototyping, before you're ready to decide how to handle errors. They leave clear markers in your code for when you're ready to make your program more robust.

EXAMPLES, PROTOTYPE CODE, AND TEST:

Test:

If a method call fails in a test, you'd want the whole test to fail, even if that method isn't the functionality under test. Because panic! is how a test is marked as a failure, calling unwrap or expect is exactly what should happen.

GUIDELINES FOR ERROR HANDLING

- The bad state is not something that's expected to happen occasionally.
- Our code after this point needs to rely on not being in this bad state.
- There's not a good way to encode this information in the types we use.

CREATING CUSTOM TYPES FOR VALIDATION

The idea of using Rust's type system to ensure we have a valid value one step further and look at creating a custom type for validation. The guessing game in which our code asked the user to guess a number between 1 and 100. We never validated that the user's guess was between those numbers before checking it against our secret number; we only validated that the guess was positive.

CREATING CUSTOM TYPES FOR VALIDATION

```
loop {
    // --snip--

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("The secret number will be between 1 and 100.");
        continue;
    }

    match guess.cmp(&secret_number) {
    // --snip--
}
```

CREATING CUSTOM TYPES FOR VALIDATION

```
#![allow(unused_variables)]
fn main() {
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {
            value
        }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
}
```

Summary

Rust's error handling features are designed to help us write more robust code. The **panic!** macro signals that our program is in a state it can't handle and lets us tell the process to stop instead of trying to proceed with invalid or incorrect values.

Continue …

The Result **enum** uses Rust's type system to indicate that operations might fail in a way that our code could recover from.

We can use **Result** to tell code that calls our code that it needs to handle potential success or failure as well. Using **panic!** and **Result** in the appropriate situations will make our code more reliable in the face of inevitable problems.

Section Break

Rust Lifetime

Rust Lifetime

- Lifetime defines the scope for which reference is valid.
- Lifetimes are implicit and inferred.
- Rust uses the generic lifetime parameters to ensure that actual references are used which are valid.

Preventing Dangling references with Lifetimes

- When a program tries to access the invalid reference, it is known as a **Dangling reference**.
- The pointer which is pointing to the invalid resource is known as a **Dangling pointer**.

Example

```
 fn main()  {
  let a;
{
    let b = 10;
     a = &b;
  }
  println!("a : {}",a);
}
```

Output

```
C:\Windows\system32\cmd.exe - cmd

D:\>rustc lifetime.rs
error[E0597]: `b` does not live long enough
 --> lifetime.rs:6:11
  |
6 |         a = &b;
  |             ^ borrowed value does not live long enough
7 |     }
  |     - `b` dropped here while still borrowed
8 |     println!("a : {}",a);
9 | }
  | - borrowed value needs to live until here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.

D:\>_
```

Continue

- In the above example, the outer scope contains the variable whose named as 'a' and it does not contain any value. An inner scope contains the variable 'b' and it stores the value 10. The reference of 'b' variable is stored in the variable 'a'. When the inner scope ends, and we try to access the value of 'a'.

Compile Error

- The Rust compiler will throw a compilation error as 'a' variable is referring to the location of the variable which is gone out of the scope. Rust will determine that the code is invalid by using the borrow checker.

Borrow checker

- The borrow checker is used to resolve the problem of dangling references. The borrow checker is used to compare the scopes to determine whether they are valid or not.

```
{
 let a;
 {
 let b = 5;
 a = &b;
 }
print!("{}",a);
}
```

- In the above example, we have annotated the lifetime of 'a' variable with the 'a and the lifetime of 'b' variable with the 'b. At the compile time, Rust will reject this program as the lifetime of 'a' variable is greater than the lifetime of 'b' variable. The above code can be fixed so that no compiler error occurs.

```
{
    let b = 5;
    let a = &b;
    print!("{}",a);
}
```

- In the above example, the lifetime of 'a' variable is shorter than the lifetime of 'b' variable. Therefore, the above code runs without any compilation error.

Lifetime annotation syntax

- Lifetime annotation does not change how long any of the references live.
- Functions can also accept the references of any lifetime by using the generic lifetime parameter.
- Lifetime annotation describes the relationship among the lifetimes of multiple parameters.

Steps to be followed for the lifetime annotation syntax:

- The names of the lifetime parameters should start with (') apostrophe.
- They are mainly lowercase and short. For example: 'a.
- Lifetime parameter annotation is placed after the '&' of a reference and then space to separate annotation from the reference type.

Some examples of lifetime annotation syntax are given below:

- &i32 // reference
- & 'a i32 // reference with **a** given lifetime.
- & 'a mut i32 // mutable reference with **a** given lifetime.

Lifetime Annotations in Function Signatures

- The 'a represents the lifetime of a reference. Every reference has a lifetime associated with it. We can use the lifetime annotations in function signatures as well.

- The generic lifetime parameters are used between angular brackets <> , and the angular brackets are placed between the function name and the parameter list. Let's have a look:
- fn fun<'a>(...);

In the above case, fun is the function name which has one lifetime, i.e., 'a. If a function contains two reference parameters with two different lifetimes, then it can be represented as:

```
fn fun<'a,'b>(...);
```

- Both & 'a i32 and & 'a mut i32 are similar. The only difference is that 'a is placed between the & and mut.
- & mut i32 means "mutable reference to an i32" .
- & 'a mut i32 means "mutable reference to an i32 with a lifetime 'a".

Lifetime Annotations in struct

We can also use the explicit lifetimes in the struct as we have used in functions.

Let's look:

struct Example

x : & 'a i32, // x is a variable of type i32 that has the lifetime 'a.

Example

```
fn main() {
    let y = &9;
    let b = Example{ x: y };
    println!("{}", b.x);
}
```

Output

9

impl blocks

We can implement the struct type having a lifetime 'a using impl block.

Let's see a simple example:

```
struct Example<'a> {
    x: &'a i32,
}
```

Continue

```
impl<'a> Example<'a>
{
fn display(&self)
{
  print!("Value of x is : {}",self.x);
}
}
```

Continue

```
fn main() {
    let y = &90;
    let b = Example{ x: y };
    b.display();
}
```

Output:

Value of x is : 90

Multiple Lifetimes

- There are two possibilities that we can have:

- Multiple references have the same lifetime.
- Multiple references have different lifetimes.

When references have the same lifetime.

```
fn fun <'a>(x: & 'a i32 , y: & 'a i32) -> & 'a i32
   //block of code.
```

In the above case, both the references x and y have the same lifetime, i.e., 'a.

'static

The lifetime named as 'static is a special lifetime. It signifies that something has the lifetime 'static will have the lifetime over the entire program. Mainly 'static lifetime is used with the strings. The references which have the 'static lifetime are valid for the entire program.

Let's look:

```
let s : & 'static str = "javaTpoint tutorial" ;
```

- In the above example, the lifetime of 'a' variable is shorter than the lifetime of 'b' variable. Therefore, the above code runs without any compilation error.

Lifetime annotation syntax

- Lifetime annotation does not change how long any of the references live.
- Functions can also accept the references of any lifetime by using the generic lifetime parameter.
- Lifetime annotation describes the relationship among the lifetimes of multiple parameters.

Steps to be followed for the lifetime annotation syntax:

- The names of the lifetime parameters should start with (') apostrophe.
- They are mainly lowercase and short. For example: 'a.
- Lifetime parameter annotation is placed after the '&' of a reference and then space to separate annotation from the reference type.

Some examples of lifetime annotation syntax are given below:

- &i32 // reference
- & 'a i32 // reference with **a** given lifetime.
- & 'a mut i32 // mutable reference with **a** given lifetime.

Lifetime Annotations in Function Signatures

- The 'a represents the lifetime of a reference. Every reference has a lifetime associated with it. We can use the lifetime annotations in function signatures as well.

- The generic lifetime parameters are used between angular brackets <> , and the angular brackets are placed between the function name and the parameter list. Let's have a look:
- fn fun<'a>(...);

In the above case, fun is the function name which has one lifetime, i.e., 'a. If a function contains two reference parameters with two different lifetimes, then it can be represented as:

```
fn fun<'a,'b>(...);
```

- Both & 'a i32 and & 'a mut i32 are similar. The only difference is that 'a is placed between the & and mut.
- & mut i32 means "mutable reference to an i32" .
- & 'a mut i32 means "mutable reference to an i32 with a lifetime 'a".

Lifetime Annotations in struct

We can also use the explicit lifetimes in the struct as we have used in functions.

Let's look:

struct Example

x : & 'a i32, // x is a variable of type i32 that has the lifetime 'a.

Example

```
fn main() {
    let y = &9;
    let b = Example{ x: y };
    println!("{}", b.x);
}
```

Output

9

impl blocks

We can implement the struct type having a lifetime 'a using impl block.

Let's see a simple example:

```
struct Example<'a> {
    x: &'a i32,
}
```

Continue

```
impl<'a> Example<'a>
{
fn display(&self)
{
  print!("Value of x is : {}",self.x);
}
}
```

Continue

```
fn main() {
    let y = &90;
    let b = Example{ x: y };
    b.display();
}
```

Output:

Value of x is : 90

Multiple Lifetimes

- There are two possibilities that we can have:

- Multiple references have the same lifetime.
- Multiple references have different lifetimes.

When references have the same lifetime.

```
fn fun <'a>(x: & 'a i32 , y: & 'a i32) -> & 'a i32
   //block of code.
```

In the above case, both the references x and y have the same lifetime, i.e., 'a.

STATIC

The lifetime named as 'static is a special lifetime. It signifies that something has the lifetime 'static will have the lifetime over the entire program. Mainly 'static lifetime is used with the strings. The references which have the 'static lifetime are valid for the entire program.

Let's look:

```
let s : & 'static str = "javaTpoint tutorial" ;
```

Lifetime Elision

Lifetime Elision is an inference algorithm which makes the common patterns more ergonomic. Lifetime Elision makes a program to be ellided.

Lifetime Elision can be used anywhere:

& 'a T

& 'a mut T

T<'a>

Lifetime Elision can appear in two ways:

Input lifetime: An input lifetime is a lifetime associated with the parameter of a function.

Output lifetime: An output lifetime is a lifetime associated with the return type of the function.

Let's look:

```
fn fun<'a>( x : & 'a i32);                    // input lifetime
fn fun<'a>() -> & 'a i32;                     // output lifetime
fn fun<'a>(x : & 'a i32)-> & 'a i32;   // Both input
                                              and output lifetime.
```

Rules of Lifetime Elision:

- Each parameter passed by the reference has got a distinct lifetime annotation.

```
fn fun( x : &i32, y : &i32)
{
}
```

If the single parameter is passed by reference, then the lifetime of that parameter is assigned to all the elided output lifetimes.

```
fn fun(x : i32, y : &i32) -> &i32
{
}
```

If multiple parameters passed by reference and one of them is &self or &mut self, then the lifetime of self is assigned to all the elided output lifetimes.

```
fn fun(&self, x : &str)
{
}
```

Example

- fn fun(x : &str); // Elided form.

- fn fun<'a>(x : & 'a str) -> & 'a str; // Expanded form.

Rust Closures

They are anonymous functions

Can be saved in a variable, or pass as arguments to other functions.

Unlike functions, closures can capture values from the scope in which they are defined.

Functional Language Features: Iterators and Closures

Closure Definition

- To define a closure, we start with a pair of vertical pipes (|), inside which we specify the parameters to the closure.
- Example:

```
let closure = |num| {
    num
};
```

Here…
'num' is a parameter

Closure with multiple parameters

- We can have more than one parameters in closure.

- Example:

```
let closure = |num1, num2| {
    num1 + num2
};
```

Remember…

- The ***'let'*** statement contains the definition of an anonymous function, not the resulting value.
- Hence, the variable 'closure' in the previous example does not contain the resulting value of *num1* + *num2*.

Calling Closure…

- We call a closure like we do for a function.

- Example:

```
fn main() {
    closure(5);
};
```

Closure Type Inference & Annotations

- Closures don’t require you to annotate the types of the parameters or the return value like ***fn*** functions do.
- They are usually short and relevant only within a narrow context, and hence, the compiler is reliably able to infer the types of the parameters and return types.

Closure Type Inference & Annotations Continued…

- But we can explicitly add type annotations in closure definitions.
- Example:

```
let closure = |num: u32| ->
u32 {
    num
};
```

Vertical Comparison of Closure and Function

With type annotations, the syntax of both the function and closure looks more similar.

```
fn add_one_v1 (x: u32) -> u32 { x + 1 } // function
let add_one_v2 = |x: u32| -> u32 { x + 1 }; // closure
let add_one_v3 = |x| { x + 1 }; // closure without annotation
let add_one_v4 = |x| x + 1 ; // closure without parenthesis
```

(for single expression only)

Type Inference Error

- Closure definitions will have one concrete type inferred for each of their parameters and for their return value.
- The example below will give error, as the provided types are different each time.

```
let closure = |num| num;

let s = closure(String::from("Hello"));

let s = closure(5);
```

More on Closures...

- Each closure instance has its own unique anonymous type.
- For two closures having same signatures, we will find their types are different.
- All closures implement at least one of the traits: ***Fn***, ***FnMut***, or ***FnOnce***.

Struct with Closure

- We can create a struct that will hold the closure and the resulting value of the calling closure.
- The struct will call the closure only if we need the resulting value, which will be cached by the struct.

Struct with Closure, Cont…

- To make a struct that holds a closure, we need to specify the type of the closure, as the struct definition needs to know the types of each of its fields.
- To define structs that use closures, we use generics and trait bounds.

Generics and the *Fn* Traits

- The *cacher* struct has a *calculation* field of generic type **T**.
- The trait bounds on **T** specify that it is a closure by using the **Fn** trait.

```
struct Cacher<T> where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>
}
```

Generics and the *Fn* Traits, Cond…

- Any closure we want to store in the calculation field must have one *u32* parameter and must return a *u32*.
- The *value* field is of type Option<u32>.
- Before executing the closure, the *value* will be **None**.

The logic around the 'value' field

- When code using a ***Cacher*** asks for the result of the closure, the ***Cacher*** will execute the closure and store the result within a *Some* variant in the *value* field.
- Then if code asks for the result of the closure again, instead of executing the closure again, the ***Cacher*** will return the result held in the *Some* variant.

Explanation…

- We can call the *value* method as many times as we want, but the closure will execute only once.

```
let closure = Cacher::new(|num| num);
closure.value(3);
closure.value(5);
closure.value(8);
```

First Problem…

- As we passed different values in the *value* function, we should always get ‘3’ as the **Cacher** instance saved **Some(3)** in *self.value*.
- This is the panic situation in Rust and will fail.

Second Problem...

- It only accepts closures that take one parameter of type u32 and return a u32.
- We might want to cache the results of closures that take a string slice and return *usize* values. To cater this, we need to introduce more generic parameters.

Capturing the environment with closures

- One of the main difference between the closures and the functions is that the closures can capture their environment and access variables from the scope in which they're defined.

Example Function…

```
fn main() {
    let x = 4;
    fn myFunc(param:u32) -> bool {
        param == x
    };
    let y = 4;
    assert!(closure(y));
}
```

We cannot do the same with functions. If we try with the following example, we will get an error.

Memory Overhead

- When a closure captures a value from its environment, it uses memory to store the values for use in the closure body. This use of memory is overhead.
- Functions never allowed to capture their environments, defining and using functions will never incur overhead.

Capturing Environment…

- Closure capture values from its environment in three ways:
- FnOnce
- FnMute
- Fn

FnOnce

- Consumes the variable it captures from its enclosing scope, known as the closure's environment.
- Must take ownership of the captured variables.
- Can't take ownership of the same variable more than once.

FnMut & Fn

- Can change the environment because it mutably borrows values.
- Borrows values from the environment immutably.

More on traits…

- All closures implement ***FnOnce*** because they can all be called at least once.
- Closures that don’t move the captured variables also implement ***FnMut***.
- Closures that don’t need mutable access to the captured variables also implement ***Fn***.

Move Ownership…

- The ***move*** keyword is used to force the closure to take ownership of the values in the environment.
- The technique is useful when passing a closure to a new thread to move the data so it’s owned by the new thread.

Fearless Concurrency

Concurrency

- It is the ability of a program to be decomposed into parts that can run independently of each other.
- In contrast, the parallel program is where different parts of the program execute at the same time.

Thread

- A thread is the smallest unit of a process.
- A single process may contain multiple threads.

Multitasking vs Multithreading

- Multitasking allows CPU to perform multiple tasks (program, process, task, threads) simultaneously.
- Multithreading allows multiple threads of the same process to execute simultaneously.

Threads – Benefits

- Splitting the computation in your program into multiple threads can improve performance, as the program does multiple tasks at the same time.
- Other benefits include resource sharing, responsiveness, and utilization of multiprocessor architecture.

Threads - Drawbacks

- Despite its benefits, it also adds complexity.
- There is no guarantee about the order in which parts of your code on different threads will run.
- This can lead to problems such as race conditions, deadlocks and irreproducible bugs.

Creating Threads in Rust

- To create a new thread, we call the **thread::spawn** function.
- We need to import **std::thread** in order to utilize threading functionality in Rust.
- Let’s look at an example.

Threading Example

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Result

- The new thread will be stopped when the main thread ends, whether or not it has finished running.

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

Explanation…

- The calls to **thread::sleep** force a thread to stop its execution for a short duration, allowing a different thread to run.
- The thread will probably take turns, but that isn’t guaranteed: it depends on how your OS schedules the threads.
- The program finishes its execution when the main thread is finished and thus, doesn’t wait for the inner thread to complete.

Join Handles

- To overcome the premature stopping of the inner thread, Rust gives us a feature called ***JoinHandle***.
- By implementing this feature, Rust prevents the program from being stopped prematurely and waits till the particular thread finishes its execution.

JoinHandle Example

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

Result

```
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 1 from the spawned thread!
hi number 3 from the main thread!
hi number 2 from the spawned thread!
hi number 4 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

Explanation

- The two threads continue alternating, but the main thread waits because of the call to **handle.join()** and does not end until the spawned thread is finished.
- The result will be different if we move **handle.join()** before the **for** loop in **main**.

Moving handle.join()

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Result

- The main thread will wait for the spawned thread to finish and then run its **for** loop.

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

Using *move* Closures with Threads

- The move closure is often used alongside thread::spawn because it allows you to use data from one thread in another thread.
- The technique is useful when creating new threads in order to transfer ownership of values from one thread to another.

Without *move*

- The code will produce error as the closure is trying to borrow '*v*'.
- Rust cannot tell how long the spawned thread will run, so it doesn't know if the reference to '*v*' will always be valid.

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

With *'move'*

- The 'move' keyword overrides Rust's conservative default of borrowing; it doesn't let us violate the ownership rules.

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

PRESIDENTIAL INITIATIVE

ARTIFICIAL INTELLIGENCE AND COMPUTING