Contents

Programming Concepts in Rust

Variables and Mutability

- default is immutable

  ```
  let x = 5;
  ```

- is safer and simpler to work with
- designating a variable as mutable makes it changeable

  ```
  let mut x = 5;
  ```

- the `mut` makes it clear that the variable is supposed to change at some point in the future

Immutables vs Constants

- constants are not the same as variables without `mut`
- you can never change a constant
- to declare a constant you say

  ```
  const x: u32 = 123;
  ```
- `const` declares the constant and the data type must be annotated
- constants cant be set to results of functions or thing only computed at runtime

Shadowing

- we can declare a new variable with the same name as a previous variable
- the first variable is *shadowed* by the second one, its data is accessed with the identifier
- shadowing can be used to change the value of a variable without making it `mut`:

  ```
  let x = 5;
  let x = x + 1;
  let x = x * 2;
  ```
- it can also be used to convert between data types but keep the name:

  ```
  let spaces: String = "   ";
  let spaces: u32    = spaces.len();
  ```

Data Types

- every value in Rust is of a specific data type
- Rust is *statically typed*, it must know the data types at compile time
- when more than one data type is possible, the programmer must specify which one should be used:

  ```
  let guess: u32 = "42".parse()
      .expect("Not a number!");
  ```

Scalar Types

- single value
- four primary types: integers, floating-point numbers, booleans, characters

Integer Types

- whole number without fractional component, standard is `i32`
- signed numbers are stored using *two's complement*
- all integers except for the byte literal excepts a type suffix such as

  `57u8`

  and underscore as a visual separator like

  `1_000`
- list of integer sizes

Length	Signed	Unsigned
`8-bit`	`i8`	`u8`
`16-bit`	`i16`	`u16`
`32-bit`	`i32`	`u32`
`64-bit`	`i64`	`u64`
`128-bit`	`i128`	`u128`
`arch`	`isize`	`usize`

- list of integer literals

Number Literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

- integer overflow is still a thing

Floating-Point Types

- Rust has `f32` and `f64` floating-point types
- the standard is `f64`

Arithmetic Operations

Operation	Example
Addition	`let sum  = 5 + 10;`
Subtraction	`let diff = 95.5 - 4.3;`
Multiplication	`let prod = 4 * 30;`
Division	`let quot = 56.7 / 32.2;`
Remainder	`let rem  = 43 % 5;`

Boolean Type

- `true` or `false`, takes up one byte in rust

  ```
  let t = true;
  let f: bool = false;
  ```

Character Type

- `char` is the most basic type
- chars are 4 bytes in size and represent unicode values, are specified with single quotes

  ```
  let c = 'z';
  let d: char = 'H';
  ```

- unicode has a lot more than just simple characters so it might be somewhat confusing as to what `char` can store

Compound Types

- combine multiple values into one type
- Rust has two primitive compound types

Tuple Type

- groups together a variety of types into one compound type
- once declared, their size is fixed
- create tuples by writing comma separated values in parenthesis

  ```
  let tup: (i32, f64, u8) = (500, 6.4, 1);
  let tup = (32, 64.6, 3);
  ```

- to access the members of a tuple, *destructuring* pattern matching can be used

  ```
  let tup = (500, 6.4, 1);
  let (x, y, z) = tup;
  ```

- indeces can can also be used to access elements of tuples

  ```
  let tup: (i32, f64, u8) = (500, 6.4, 1);
  let five_hundred = tup.0;
  let one = tup.2;
  ```

Array Type

- compound type that holds multiples of the same type of value
- arrays in Rust have a fixed length

  ```
  let a = [1, 2, 3, 4, 5];
  ```

- data here will be allocated on the stack
- because of the fixed length they are useful for values that do not change in number, e.g. months in a year
- declaring length and type of an array works like this:

  ```
  let a: [i32; 5] = [1, 2, 3, 4, 5];
  ```

- alternatively one can declare an array with e.g. 5 elements and all of them are 15

  ```
  let a [15; 5];
  ```

Accessing Array Elements

- access elements using indexes in square brakets

  ```
  let a = [1, 2, 3, 4, 5];

  let first = a[0];
  ```

Invalid Array Element Access

- if the index is out of bounds, a runtime error will occur
- the access is stopped to make the program safer and more stable

Functions

- pervasive in Rust code
- `fn main()` is the most important one, it's the entry point for many programs
- other functions are declared at any point in the file

  ```
  fn another_function() {
      println!("Another function!");
  }
  ```

- calling a function is simple too

  ```
  fn main() {
      another_function();
  }
  ```

Function Parameters

- the are part of the function definition

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

- defining multiple parameters works with commas

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

Function Bodies, Statements, Expressions

- *Statements* are instructions that perform an action and don't return a value

  ```
  let y = 6;
  ```

- *Expressions* evaluate to a resulting value
- assignments are not expressions in Rust, so this **won't** work

  ```
  let y = (let x = 6);
  ```

- math operations, numbers, macros, functions, scopes are expressions

  ```
  let y = {
      let x = 3;
      x + 1
  }
  ```

- expressions **do not** end in semicolons

Functions with Return Values

- the type of return values is declared after `->` after the function signature
- the return value is the same as the last expression in a code block
- `return` can be used to return explicitely or early, most returns are implicit and on the last line

  ```
  fn five() -> i32 {
      5
  }
  fn plus_one(x: i32) -> i32 {
      x + 1
  }
  ```

Comments

- simple comment

  ```
  // hello world
  ```

- comments are generally above the line of code they are commenting on

  ```
  // minimum age to buy alcohol
  let drinking_age = 21;
  ```

Control Flow

- things that make programming easier by conditionally or repeatedly running code

`if` Expressions

- branches the code depending on certain boolean conditions, elements of the statement are sometimes called arms

  ```
  let number = 3;

  if number < 5 {
      println!("condition is true");
  } else {
      println!("condition is false");
  }
  ```

Multiple conditions with `else if`

```
let number = 6;

if number % 4 == 0 {
    println!("divisible by 4");
} else if number % 3 == 0 {
    println!("divisible by 3");
}
```

Using `if` in a `let` statement

- `if` is an expression, so it can be used in assignments

  ```
  let condition = true;
  let number = if condition {
      5
  } else {
      6
  };
  ```

- the types of all arms need to be the same

Repetition with Loops

- `loop`, `while`, `for` can execute blocks of code more than once

Repeating code with `loop`

- repeat something forever until explicit stop

```
loop {
    println!("again!");
}
```

- use `break` in a `loop` to break out of it normally

Returning values from Loops

- `loop` is an expression that can return values "' let mut counter = 0;

  let result = loop { counter += 1;

  ```
  if counter == 10 {
      break counter * 2;
  }
  ```

  };

Conditional Loops with `while`

- loop with built-in test and break statements

  ```
  let mut number = 3;

  while number != 0 {
      println!("{}!", number);

      number -= 1;
  }
  ```

- this eliminates a lot of nesting

Looping through a Collection with `for`

- `while` can loop through a collection of elements

  ```
  let a = [10, 20, 30, 40, 50];
  let mut index = 0;

  while index < 5 {
      println!("the value is {}!", a[index]);

      index += 1;
  }
  ```

- a more concise and safe way is to use a `for` loop, indices will always work

  ```
  let a = [10, 20, 30, 40, 50];

  for element in a.iter() {
      println!("the value is: {}", element);
  }
  ```

- to use a `for` loop a specified number of times, including the first and excluding the last, use

```
// (1..4) gives [1, 2, 3]
// rev() reverses the order of the numbers
for number in (1..4).rev() {
    // code
}
```

Understanding Ownership

- ownership is meant to make memory safe without having a garbage collector
- this chapter will cover ownership, borrowing, slices, data in memory layouts

What is Ownership

- *ownership* is central to the way Rust works and it's simple to explain
- all programs have to manage a computer's memory for running
- some use garbage collectors that constantly check for unused memory, some need the programmer to manually allocate memory
- rust uses a system that checks rules at compile time and thus does not slow down the program when it is running
- this chapter will cover strings as an example

The Stack and the Heap

- in many programming scenarios the stack and heap are not that important, but for systems programming and rust they are very important
- where data is stored influences the behavior of the language as well as its speed
- stack: memory that stores data in order and returns them in the opposite order, last int, first out
- data stored on the stack must have a known size at compile time, unknown or changing sizes must be stored on the heap
- heap: less organized, a certain amount of space is requested to store data, OS finds the space and returns a pointer (address of its location) to it
- pushing to the stack is faster than allocating on the heap because for the stack no location large enough has to be found and then kept in order
- accessing data on the heap is slower and jumping between data is also slower than working on one piece of data at a time
- when a function is called, the values passed to the function are all pushed onto the stack – to return the values they are popped off the stack
- ownership addresses what code is using data on the heap, cleaning up unused data on the heap etc

Ownership Rules

- each value in Rust has a variable that's called its *owner*
- there can only be one owner at a time
- when the owner goes out of scope, the value will be dropped

Variable Scope

- range in a program for which an item is valid
- when a variable comes *into scope* it is valid, when is goes *out of scope* it becomes invalid
- scopes are generally encapsulated by or related to curly brackets

  ```
  {                       // s comes into scope
      let s = "hello";

                          // s is valid

  }                       // s goes out of scope
  ```

The `String` Type

- simple data types are stored on the stack and popped off when they go out of scope
- more complex data types are stored on the heap and must be cleaned up after use
- `String` will be the example used here insofar as it relates to ownership
- string literals are not always convenient because they are immutable and hard coded
- `String` is allocated on the heap and can change at runtime, they can be created from string literals

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

- the resulting type can be modified:

  ```
  let mut s = String::from("hello");
  s.push_str(", world!");        // appends to s
  ```

- the difference between `String` and string literals is the way they deal with memory

Memory and Allocation

- string literals are hardcoded into the program because they are known at compile time – they are fast efficient
- it is not possible to reserve blobs of memory at compile time for each string that might change

- `String` is growable, so: its memory must be requested from the OS at runtime; the memory must be returned to the OS when the `String` is done
- the programmer does the allocation manually

  `String::from`
- normally memory is either freed by a garbage collector or manually by the programmer, in Rust it is freed when the variable goes out of scope
- when `s` goes out of scope the `drop` function associated with it is automatically called by Rust to free the memory
- this seems simple now, but it can be more complicated in more complicated code

Ways Variables and Data Interact: Move

- if two primitive data types are set equal, the data is copied and then there are two variables with two copies of the same data, both are on the stack

  ```
  let x = 5;
  let y = x;
  ```
- for `String` this is different

  ```
  let s1 = String::from("hello");
  let s2 = s1;
  ```
- `s1` is made up of a `ptr`, `len`, and `capacity`, the pointer points to the first element of the string in memory, `len` is the amount of bytes of memory that the string is currently using and `capacity` is the total amount of memory allocated by the OS
- when `s1` is assigned to `s2`, the three pieces of data are copied, but the data on the stack remains the same, it is not copied and the two pointers point to the same place in memory
- in the example above Rust moves the data from `s1` to `s2` and invalidates `s1` so it is no longer valid
- invalidating `s1` will mean that when `s2` goes out of scope the memory is only freed once and thus does not generate a double free error
- additionally, Rust will never automatically make deep and expensive copies of anything – it will be fast by default

Ways Variables and Data Interact: Clone

- if we do want a deep copy of the data on the heap we use `clone`

  ```
  let s1 = String::from("hello");
  let s2 = s1.clone();
  ```
- `clone` is something that is expensive to call

Stack-Only Data: Copy

- if a type has the `copy` trait, an older version of the variable is still valid after copying, like with integers

  ```
  let x = 5;
  let y = x;
  ```

- a type can't have the `copy` trait if any of its parts implement `drop`
- all simple or primitive types are copy

Ownership and Functions

- passing a variable to a function is similar to assigning values to variables, thus the same rules apply

  ```
  fn main() {
      let s = String::from("hello");      // s comes into scope
      takes_ownership(s);                 // value of s moves into
                                          // function
                                          // it's no longer valid

      let x = 5;                          // x comes into scope
      makes_copy(x);                      // x is Copy and is thus
                                          // still valid
  }   // x and then s go out of scope
      // nothing special happes to s because it is already invalid

  fn takes_ownership(s: String) {         // s comes into scope
      println!("{}", s);
  }   // s goes out of scope and drop is called, memory is freed

  fn makes_copy(i: i32) {                 // i comes into scope
      println!("{}", i);
  }   // i goes out of scope, not affecting x
  ```

- if `s` were to be used after the `takes_ownership(s)` was called, a compile time error would happen

Return Values and Scope

- returning values can also transfer ownership

  ```
  fn main() {
      let s1 = give_ownership();          // fn moves its return
                                          // value to s1

      let s2 = String::from("hello");     // s2 comes into scope
  ```

```
    let s3 = takes_and_gives_back(s2);  // s2 moved into fn
                                        // return value moved to s3
}   // s3 goes out of scope and is dropped, so does s1.
    // s2 is already out of scope, so nothing happens

fn gives_ownership() -> String {        // will move return value
                                        // into calling fn
    let s = String::from("hello");      // s comes into scope
    s                                   // s is returned and moves
                                        // to the calling function
}   // nothing goes out of scope

fn takes_and_gives_back(s: String) -> String {
                                        // s comes into scope
    s                                   // s is returned and moves
                                        // to the calling fn
}   // nothing goes out of scope
```

- assigning the value of a variable to another moves it
- when an active variable goes out of scope, it is dropped
- one option for returning ownership of the argument plus a result is to return a tuple from a function – a better way to do it is to use *references*

References and Borrowing

- if one uses a function that takes ownership and then has to return ownership so the argument can be used afterwards
- passing references to functions instead of taking ownership is the solution to that

```
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()
}
```

- ampersands '`&`' are *references* and enable referring to values without taking ownership
- above, `s` points to `s1` which points to the actual value
- dereferencing is done with `*`
- `&s1` refers to the value of `s1` but does now own it – the value will not be dropped when `s` goes out of scope
- when functions have references as parameters it is called *borrowing*

- references are immutable by default

Mutable References

- creating a mutable string and then passing a mutable reference to a function allows variables to be modified using their references

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

  fn change(s: &mut String) {
      s.push_str(", world");
  }
  ```

- big restriction: there can only be one mutable reference to a particular piece of data in a particular scope
- this only allows restricted mutation – less than most other languages
- Rust can thus prevent *data races* at compile time – these three things need to be true: two or more pointers access data at the same time, at least one of the pointers is used to write to the data, there are no mechanisms to synchronize the access to the data
- data races are undefined and difficult to diagnose
- new scopes allow for more mutable references, just not simultaneous ones
- we also cannot borrow data as mutable if it is also borrowed as immutable
- if an immutable reference is being used it is not expected that the value changes at the same time
- multiple immutable references are ok because nobody can change any of the data
- some intricacies are: if immutable references are no longer used a mutable one can be created even if the other ones are technically still in scope
- borrowing errors are annoying, but they prevent bugs at compile time

Dangling References

- these are created when a reference to a non-existent memory exists, producing undefined behavior

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

  fn dangle() -> &String {                // returns ref to str
      let s = String::from("hello");      // s is new string
  ```

```
    &s                                  // return ref to str
}   // s goes out of scope here and is dropped
    // the reference now refers to nothing
```

- the simple solution here is to return the string with ownership instead

The Rules of References

- at any given time, you can have *either* one mutable reference *or* any number of immutable references
- references must always be valid

The Slice Type

- slices do not have ownership
- slices reference a contiguous sequence of elements in a collection rather than the whole collection