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
  }
  ```