Introductory Rust Notes

Introduction to Rust

Core Features

- Memory safety without garbage collection
- Concurrency without data races
- Abstraction without overhead

Variables

Variable bindings are immutable by default, but this can be overridden using the `mut` modifier

```language-rust
let x = 1;
let mut y = 1;
```

Types

[Data Types -Rust book]

The Rust compiler can infer the types that you are using, given the information you already gave it.

[Scalar Types]

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

- integers
- floating-point numbers
- booleans
- characters

Integers

For example

```language-rust
u8, i32, u64
```

Floating point

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.
The default type is `f64` because on modern CPUs it's roughly the same speed as `f32` but is capable of more precision. All floating-point types are signed.

boolean

The boolean type or bool is a primitive data type that can take on one of two values, called true and false . (size of 1 byte)

char

`char` in Rust is a unique integral value representing a Unicode Scalar value
Note that unlike C, C++ this cannot be treated as a numeric type.

Other scalar types

usize

`usize` is pointer-sized, thus its actual size depends on the architecture your are compiling your program for
As an example, on a 32 bit x86 computer, `usize = u32`, while on x86_64 computers, `usize = u64`.
`usize` gives you the guarantee to be always big enough to hold any pointer or any offset in a data structure, while `u32` can be too small on some architectures.
Rust states the size of a type is not stable in cross compilations except for primitive types.

Compound Types

Compound types can group multiple values into one type.

- tuples
- arrays
- struct

Tuples

Example

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Struct

```rust
struct User {
        name : String,
        age: u32,
        email: String,
}
```

Collections

- Vectors

```rust
let names = vec!["Bob", "Frank", "Ferris"];
```

We will cover these in more detail later

Strings

Based on UTF-8 - Unicode Transformation Format

Two string types:

- `&str` a view of a sequence of UTF8 encoded dynamic bytes, stored in binary, stack or heap. Size is unknown and it points to the first byte of the string
- `String`: growable, mutable, owned, UTF-8 encoded string. Always allocated on the heap. Includes capacity i.e. memory allocated for this string.

A String literal is a string slice stored in the application binary (i.e. there at compile time).

String vs str

`String` - heap allocated, growable UTF-8
`&str` - reference to UTF-8 string slice (could be heap, stack ...)

String vs &str - StackOverflow
Rust overview - presentation
Let's Get Rusty - Strings

Arrays

Rust book definition of an array:

"An array is a collection of objects of the same type `T`, stored in contiguous memory. Arrays are created using brackets `[]`, and their length, which is known at compile time, is part of their type signature `[T; length]`."

Array features:

- An array declaration allocates sequential memory blocks.
- Arrays are static. This means that an array once initialized cannot be resized.
- Each memory block represents an array element.
- Array elements are identified by a unique integer called the subscript/ index of the element.
- Populating the array elements is known as array initialization.
- Array element values can be updated or modified but cannot be deleted.

Array declarations

```language-rust
//Syntax1: No type definition
let variable_name = [value1,value2,value3];

let arr = [1,2,3,4,5];

//Syntax2: Data type and size specified
let variable_name:[dataType;size] =
[value1,value2,value3];

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

//Syntax3: Default valued array
let variable_name:[dataType;size] =
[default_value_for_elements,size];
```

```
let arr:[i32;3] = [0;3];

// Mutable array
let mut arr_mut:[i32;5] = [1,2,3,4,5];

// Immutable array
let arr_immut:[i32;5] = [1,2,3,4,5];
```

Rust book definition of a slice:

Slices are similar to arrays, but their length is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice. The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64.

Arrays - TutorialsPoint
Arrays and Slices - RustBook

Numeric Literals

The compiler can usually infer the type of an integer literal, but you can add a suffix to specify it, e.g.
`42u8`
It usually defaults to `i32` if there is a choice of the type.

Hexadecimal, octal and binary literals are denoted by prefixes `0x` , `0o` , and `0b` respectively

To make your code more readable you can use underscores with numeric literals
e.g.

```language-rust
1_234_567_890
```

ASCII code literals

Byte literals can be used to specify ASCII codes
e.g.
`b'C'`

Conversion between types

Rust is unlike many languages in that it rarely performs implicit conversion between numeric types, if you need to do that, it has to be done explicitly.
To perform casts between types you use the `as` keyword
For example

```language-rust
let a = 12;
let b = a as usize;
```

Enums

See docs

Use the keyword `enum`

```language-rust
enum Fruit {
    Apple,
    Orange,
    Grape,
}
```

You can then reference the enum with for example

```language-rust
Fruit::Orange
```

Functions

Functions are declared with the `fn` keyword, and follow familiar syntax for the parameters and function body.

```language-rust
fn my_func(a: u32) -> bool {
    if a == 0 {
        return false;
    }
    a == 7
}
```

As you can see the final line in the function acts as a return from the function
Typically the return keyword is used where we are leaving the function before the end.

Loops

Range:

- inclusive start, exclusive end
  `for n in 1..101 {}`
- inclusive end, inclusive end
  `for n in 1..=101 {}`
  - inclusive end, inclusive end, every 2nd value

```language-rust
for n in (1..=101).step_by(2){}
```

We have already seen for loops to loop over a range, other ways to loop include
`loop` - to loop until we hit a `break`
`while` which allows an ending condition to be specified
See Rust book for examples.

Control Flow

If expressions

See Docs

The `if` keyword is followed by a condition, which *must evaluate to bool*, note that Rust does not automatically convert numerics to bool.

```language-rust
if x < 4 {
        println!("lower");
} else {
        println!("higher");
}
```

Note that 'if' is an expression rather than a statement, and as such can return a value to a 'let' statement, such as

```language-rust
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {}", number);
}
```

Note that the possible values of `number` here need to be of the same type.

We also have `else if` and `else` as we do in other languages.

Printing

```language-rust
 println!("Hello, world!");

 println!("{:?} tokens", 19);
```

Option

We may need to handle situations where a statement or function doesn't return us the value we are expecting, for this we can use Option.

Option is an enum defined in the standard library.
The `Option<T>` enum has two variants:

- `None`, to indicate failure or lack of value, and
- `Some(value)`, a tuple struct that wraps a `value` with type `T`.

It is useful in avoiding inadvertently handling null values.

Another useful enum is `Result`

```language-rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Matching

A powerful and flexible way to handle different conditions is via the `match` keyword
This is more flexible than an `if` expression in that the condition does not have to be a boolean, and pattern matching is possible.

Match Syntax

```language-rust
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Match Example

```language-rust
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,
    }
}
```

The keyword `match` is followed by an expression, in this case `coin`
The value of this is matched against the 'arms' in the expression.
Each `arm` is made of a pattern and some code
If the value matches the pattern, then the code is executed, each arm is an

expression, so the return value of the whole match expression, is the value of the code in the arm that matched.

Matching with `Option`

```language-rust
fn main() {
    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);
}
```

Installing Rust

The easiest way is via rustup

See Docs

Mac / Linux

```bash
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf
| sh
```

Windows

See details here

download and run `rustup-init.exe`.

Other methods

Cargo

See the docs

Cargo is the rust package manager, it will

- download and manage your dependencies,
- compile and build your code
- make distributable packages and upload them to public registries.

Some common cargo commands are (see all commands with --list):

```
 build, b      Compile the current package
check, c      Analyse the current package and report errors,
but don't build
                          object files
clean         Remove the target directory
doc, d        Build this package's and its dependencies'
documentation
new           Create a new cargo package
init          Create a new cargo package in an existing
directory
add           Add dependencies to a manifest file
run, r        Run a binary or example of the local package
test, t       Run the tests
bench         Run the benchmarks
update        Update dependencies listed in Cargo.lock
search        Search registry for crates
publish       Package and upload this package to the
registry
install       Install a Rust binary. Default location is
$HOME/.cargo/bin
uninstall     Uninstall a Rust binary
```

See `cargo help` for more information on a specific command.

Useful Resources

Rustlings

Rust by example

Rust Lang Docs

Rust Playground

Rust Forum

Rust Discord

Rust Continued

Idiomatic Rust

The way we design our Rust code and the patterns we will use differ from say what would be used in Python or JavaScript.
As you become more experienced with the language you will be able to follow the patterns

Memory - Heap and Stack

The simplest place to store data is on the stack, all data stored on the stack must have a known, fixed size.
Copying items on the stack is relatively cheap.
For more complex items, such as those that have a variable size, we store them on the heap, typically we want to avoid copying these if possible.

Clearing up memory

The compiler has a simple job keeping track of and removing items from the stack, the heap is much more of a challenge.
Older languages require you to keep track of the memory you have been allocated, and it is your responsibility to return it when it is no longer being used.
This can lead to memory leaks and corruption.
Newer languages use garbage collectors to automate the process of making available areas of memory that are no longer being used. This is done at runtime and can have an impact on performance.

Rust takes a novel approach of enforcing rules at compile time that allow it to know when variables are no longer needed.

Ownership

Problem

We want to be able to control the lifetime of objects efficiently , but without getting into some unsafe memory state such as having 'dangling pointers'

Rust places restrictions on our code to solve this problem, that gives us control over the lifetime of objects while guaranteeing memory safety.

In Rust when we speak of ownership, we mean that a variable owns a value, for example

```language-rust
let mut my_vec = vec![1,2,3];
```

here `my_vec` owns the vector.
(The ownership could be many layers deep, and become a tree structure.)

We want to avoid,

- the vector going out of scope, and `my_vec` ends up pointing to nothing, or (worse) to a different item on the heap
- `my_vec` going out of scope and the vector being left, and not cleaned up.

Rust 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.

```language-rust
fn main() {
    {
        let s = String::from("hello"); // s is valid from this point forward
        // do stuff with s
    }   // this scope is now over, and s is no longer valid
}
```

Copying

For simple datatypes that can exist on the stack, when we make an assignment we can just copy the value.

```language-rust
fn main() {
    let a = 2;
    let b = a;
}
```

The types that can be copied in this way are

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain these types. For example, `(i32, i32)`, but not `(i32, String)`.

For more complex datatypes such as `String` we need memory allocated on the heap, and then the ownership rules apply

Move

For none copy types, assigning a variable or setting a parameter is a `move` The source gives up ownership to the destination, and then the source is uninitialised.

```language-rust
let a = vec![1,2,3];
let b = a;
let c = a;   // <= PROBLEM HERE
```

Passing arguments to functions transfers ownership to the function parameter

```language-rust
let a = ...

loop {

g(a) // <= after the first iteratation, a has lost ownership

}



fn g(x : ...) {

// ...

}
```

Fortunately collections give us functions to help us deal with moves and iteration

```language-rust
let v = vec![1,2,3];
for mut a in v {
```

```
        v.push(4);
}
```

References

References are a flexible means to help us with ownership and variable lifetimes.
They are written
`&a`
If `a` has type `T` , then `&a` has type `&T`

These ampersands represent *references*, and they allow you to refer to some value without taking ownership of it.

Because it does not own it, the value it points to will not be dropped when the reference stops being used.

References have no effect on their referent's lifetime , so a referent will not get dropped as it would if it were owned by the reference.
Using a reference to a value is called 'borrowing' the value.
References must not outlive their referent.

There are 2 types of references

1. A *shared* reference
   You can read but not mutate the referent.
   You can have as many shared references to a value as you like.
   Shared references are copies
2. A *mutable* reference, mean you can read and modify the referent.
   If you have a mutable ref to a value, you can't have any other ref to that value active at the same time.
   This is following. the 'multiple reader or single writer principle'.
   Mutable references are denoted by `&mut`
   for example this works

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

    let s1 = &mut s;
```

```
            // let s2 =  &mut s;

             s1.push_str(" bootcamp");

             println!("{}", s1);
            }
```

but this fails

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

             let s1 = &mut s;
             let s2 =  &mut s;
             s1.push_str(" bootcamp");

             println!("{}", s1);
            }
```

De referencing

Use the `*` operator

```language-rust
let a = 20;
let b = &a;
assert!(*b == 20);
```

String Data types

See Docs

Strings are stored on the heap
A `String` is made up of three components: a pointer to some bytes, a length, and a capacity. The pointer points to an internal buffer `String` uses to store its data. The length is the number of bytes currently stored in the buffer, and the capacity is the size of the buffer in bytes. As such, the length will always be less than or equal to the capacity.

This buffer is always stored on the heap.

```language-rust
let len = story.len();
let capacity = story.capacity();
```

We can create a `String` from a literal, and append to it

```language-rust
let mut title = String::from("Solana ");
title.push_str("Bootcamp"); // push_str() appends a literal to a String
println!("{}", title);
```

Traits

These bear some similarity to interfaces in other languages, they are a way to define the behaviour that a type has and can share with other types, where behaviour is the methods we can call on that type.
Trait definitions are a way to group method signatures together to define a set of behaviors necessary for a particular purpose.

Defining a trait

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

Implementing a trait

```language-rust
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)
    }
}
```

Default Implementations

```language-rust
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

Using traits (polymorphism)

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

Introduction to Generics

Generics are a way to parameterise across datatypes, such as we do below with `Option<T>` where `T` is the parameter that can be replaced by a datatype such as `i32`.
The purpose of generics is to abstract away the datatype, and by doing that avoid duplication.

For example we could have a struct, in this example `T` could be an `i32`, or a `u8`, or .... depending how you create the Point `struct` in the main function.
In this case we are enforcing `x` and `y` to be of the same type.

```language-rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let my_point = Point { x: 5, y: 6 };
}
```

The handling of generics is done at compile time, so there is no run time cost for including generics in your code.

Introduction to Vectors

Vectors are one of the most used types of collections.

Creating the Vector

We can use `Vec::new()` To create a new empty vector

```rust
let v: Vec<i32> = Vec::new();
```

We can also use a macro to create a vector from literals, in which case the compiler can determine the type.

```rust
let v = vec![41, 42, 7];
```

Adding to the vector

We use `push` to add to the vector, for example

```rust
v.push(19);
```

Retrieving items from the vector

2 ways to get say the 5th item

- using `get`
  e.g. `v.get(4);`
- using an index
  e.g. `v[4];`

We can also iterate over a vector

```rust
let v = vec![41, 42, 7];
for ii in &v {
        println!("{}", ii);
}
```

You can get an iterator over the vector with the `iter` method

```rust
let x = &[41, 42, 7];
let mut iterator = x.iter();
```

There are also methods to `insert` and `remove`
For further details see Docs

Iterators

The iterator in Rust is optimised in that it has no effect until it is needed

```language-rust
let names = vec!["Bob", "Frank", "Ferris"];
let names_iter = names.iter();
```

This creates an iterator for us that we can then use to iterate through the collection using `.next()`

```language-rust
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];
        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
```

Shadowing

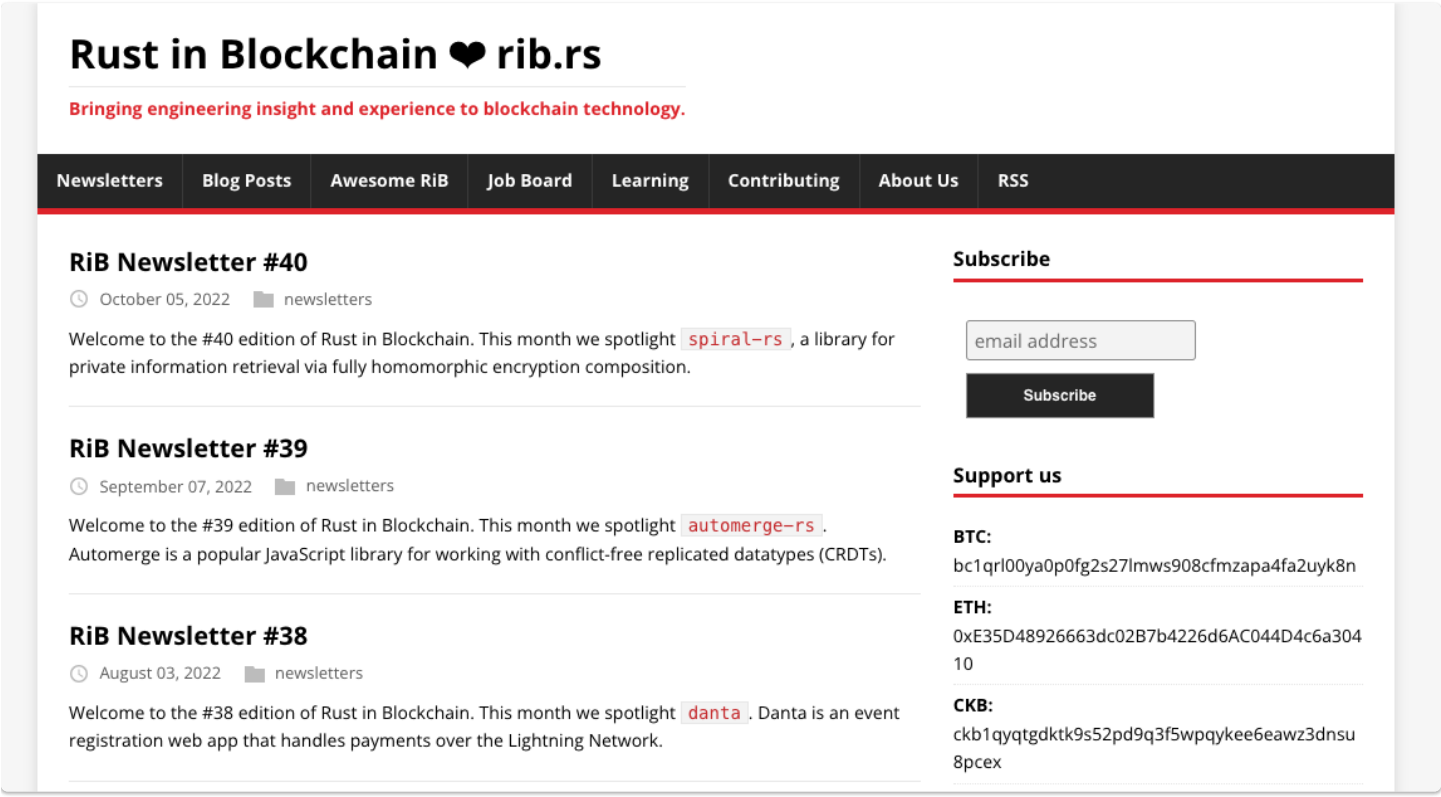
It is possible to declare a new variable with the same name as a previous variable.
The first variable is said to be shadowed by the second,
For example

```rust
fn main() {
    let y = 2;
    let y = y * 3;

    {
        let y = y * 2;
        println!("Here y is: {y}");
    }

    println!("Here y is: {y}");
}
```

We can use this approach to change the value of otherwise immutable variables

Resources

- Rust cheat sheet
- Rustlings
- Rust by example
- Rust Lang Docs
- Rust Playground
- Rust Forum
- Rust Discord
- Rust in Blockchain



Rust continued

Lifetime constraints on references

```
 let x;
{
        let a = 2;
        ...
        x = &a;
        ...
}
assert_eq!(*x,2);
```

This is a problem because

1. For a variable `a` , any reference to `a` must not outlive `a` itself.
   The variable's lifetime must enclose that of the reference.
   => how *large* a reference's lifetime can be
   The lifetime of &a must not be outside the dots
2. If you store a reference in a variable `x` the reference's type must be good for the lifetime of the variable.
   The reference's lifetime must enclose that of the variable.
   => how *small* a reference's lifetime can be.

We should have

```
 let a = 2;
{
        let x = &a;
        assert_eq!(*x,2);
}
```

Slices

Slices are similar to arrays, but their length is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice.
The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64.
Slices can be used to borrow a section of an array, and have the type signature `&[T]`."

```
 use std::mem;

// This function borrows a slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Fixed-size array (type signature is superfluous)
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // All elements can be initialized to the same value
    let ys: [i32; 500] = [0; 500];

    // Indexing starts at 0
    println!("first element of the array: {}", xs[0]);
    println!("second element of the array: {}", xs[1]);

    // `len` returns the count of elements in the array
    println!("number of elements in array: {}", xs.len());

    // Arrays are stack allocated
    println!("array occupies {} bytes",
```

```
mem::size_of_val(&xs));

    // Arrays can be automatically borrowed as slices
    println!("borrow the whole array as a slice");
    analyze_slice(&xs);

    // Slices can point to a section of an array
    // They are of the form [starting_index..ending_index]
    // starting_index is the first position in the slice
    // ending_index is one more than the last position in
the slice
    println!("borrow a section of the array as a slice");
    analyze_slice(&ys[1 .. 4]);

    // Out of bound indexing causes compile error
    println!("{}", xs[5]);
}
```

See

[Arrays - TutorialsPoint]

[Arrays and Slices - RustBook]

Printing / Outputting

See Docs

There are many options

- `format!`: write formatted text to `String`
- `print!`: same as `format!` but the text is printed to the console (io::stdout).
- `println!`: same as `print!` but a newline is appended.
- `eprint!`: same as `print!` but the text is printed to the standard error (io::stderr).
- `eprintln!`: same as `eprint!` but a newline is appended.

We commonly use `println!`

The braces `{}` are used to format the output and will be replaced by the arguments
For example

```
println!("{} days", 31);
```

You can have positional or named arguments
For *positional* specify an index
For example

```
println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");
```

For *named* add the name of the argument within the braces

```
println!("{subject} {verb} {object}",
                 object="the lazy dog",
                 subject="the quick brown fox",
                 verb="jumps over");
```

Packages, crates and modules

A `crate` is a binary or library, a. number of these (plus other resources) can form. a `package` , which will contain a *Cargo.toml* file that describes how to build those crates.
A crate is meant to group together some functionality in a way that can be easily shared with other projects.
A package can contain at most one library crate, but. any number of binary crates.
There can be further refinement with the use of `modules` which organise code within a crate and can specify the privacy (public or private) of the code.
Module code is private by default, but you can make definitions public by adding the `pub` keyword.

Macros

see Docs

Macros allow us to avoid code duplication, or define syntax for DSLs.
Examples of Macros we have seen are

`vec!` to create a Vector
```
let names = vec!["Bob", "Frank", "Ferris"];
```

`println!` to output a line
```
println!("The value of number is: {}", number);
```

Hashmaps

See Docs
and examples

Where vectors store values by an integer index, `HashMap`s store values by key. `HashMap` keys can be booleans, integers, strings, or any other type that implements the `Eq` and `Hash` traits.

Like vectors, `HashMap`s are growable, but HashMaps can also shrink themselves when they have excess space.
You can create a HashMap with a certain starting capacity

using `HashMap::with_capacity(uint)`, or use `HashMap::new()` to get a HashMap with a default initial capacity (recommended).

An example inserting values

```
 use std::collections::HashMap;
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

You can use an iterator to get values from the hashmap

```
 let mut balances = HashMap::new();

balances.insert("132681", 12);
balances.insert("234987", 9);


for (address, balance) in balances.iter() {
        ...
}
```

Using the get method

The get method on a hashmap returns an `Option<&V>` where V is the type of the value

```
 fn main() {
    use std::collections::HashMap;

    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).copied().unwrap_or(0);
}
```

This program handles the `Option` by calling `copied` to get an `Option<i32>` rather than an `Option<&i32>`, then `unwrap_or` to set `score` to zero if `scores` doesn't have an entry for the key.

Rust - (more) pattern matching

See Docs

Ignoring values and matching literals

We can use `_` to show we are ignoring a value, or to show a default match, where we don't care about the actual value.

language-rust

```
    let x = ...;

    match x {
        1 => println!("one"),
        3 => println!("three"),
        5 => println!("five"),
```

```
        _ => println!("some other value"),
    }
```

Ranges

Use `..=`

For example

```
 let x = 5;
match x {
        1..=5 => println!("one through five"),
        _ => println!("something else"),
}
```

Variables

Be aware the match starts a new scope, so if you use a variable it may shadow an existing variable.

```
 let x = Some(5);
let y = 10;
match x {
        Some(50) => println!("Got 50"),
        Some(y) => println!("Matched, y = {y}"),
        _ => println!("Default case, x = {:?}", x),
}
println!("at the end: x = {:?}, y = {y}", x);
```

What we get printed out is

`Matched, y = 5`.

`x = Some(5), y = 10`.

Destructuring

We can destructure structs and match on their constituent parts

For example

```
 fn main() {
let p = Point { x: 0, y: 7 };
```

```
match p {
        Point { x, y: 0 } => println!("On the x axis at {}",
x),
        Point { x: 0, y } => println!("On the y axis at {}",
y),
        Point { x, y } => println!("On neither axis: ({},
{})", x, y), }
}
```

Adding further expressions

```language-rust
let num = Some(4);
match num {
        Some(x) if x % 2 == 0 => println!("The number {}
is even", x),
        Some(x) => println!("The number {} is odd", x),
        None => (),
}
```

Writing tests in Rust

See Docs
See examples

A test in Rust is a function that's annotated with the `test` attribute

To change a function into a test function, add `#[test]` on the line before `fn`.
When you run your tests with the `cargo test` command, Rust builds a test runner binary that runs the annotated functions and reports on whether each test function passes or fails.

```
 #[cfg(test)] mod tests {
#[test]
fn simple_example() {
        let result = 3 + 5;
        assert_eq!(result, 8);
        }
}
```

The `#[cfg(test)]` annotation on the tests module tells Rust to compile and run the test code only when you run `cargo test`, not when you run `cargo build`.