Rust notes

The basic stuff.

Cargo

You create a new project with the command "cargo new". Then if you want to run it you can either use the command "cargo run" which will build your project and then run it, or the command "rustc src/file.rs" where file.rs is name of your rust file (src/ doesn't need to be there if you aren't using a cargo project).

If you just want to build the project without running it, you can use "cargo build". Using either "cargo build" or "cargo run" creates an executable in the folder target/debug which can be run directly in the command line.

The command "cargo check" will check if your code compiles but won't actually run or build it.

The command "cargo build --release" will build your project for the intent of you distributing it. This command should probably only be used for completed projects.

If you want to see documentation for the crates you are using in your project, you can either go to the specific crate's page on crates.io or run the command "cargo do --open".

Crates

Crates are open-source rust projects that can be added to your cargo project found on the crates.io website. You can add a crate to your cargo project by going to the cargo.toml file, going to dependencies, adding the line mentioned on the crates.io page for the package, and then running "cargo build" to fully add the crate.

Importing library functions

The keyword "use" is used to include functions from libraries in your rust code. When they've been included, the function is named the same as it was named in the library. So, the line "use ferris_says::say" imports the library "say" from the "ferris_says" namespace, and the "say" library is used as just "say" instead of "ferris_says::say" in your code. This means that any function you create yourself that is called "say" will create an error if you include "ferris_says::say" as well.

Variables and data types

Rust provides 2 types of variable types, which are scalar and compound. There are only 4 scalar types, which are integers (from 8bit to 128bit), Booleans, floats (64bit by default, can be 32bit with proper assignment), and characters (denoted by ''). Compound data types are when you can group multiple

values into one type in things like tuples, arrays, strings, structs, etc.

There are several differences between a string and a string literal which are important to remember.

- They use different type declaration ("String" and "&str" respectively)
- They are declared differently, which strings using the "String" namespace and string literals using quotations.
- String literals can only be immutable.
- String literals won't work for things not known at compile time, such as user input.
- They are also stored differently in memory, which can be read about in the memory management chapter.

Important methods to keep in mind for strings and string literals are "String::from(str)", which turns a string literal into a string, and "String.as_str()", which does the opposite.

Tuples can contain any number of values of any types that can be unique to each other. Values of a tuple can be assigned to other variables either individually or all at once (see variables project in practice folder). Arrays act like arrays in C++ in that they can only have a static size and all of the values can only be of one type or null. Both are immutable by default but can be made mutable like normal variables.

You can create a variable by using the "let" keyword, and the variable will be immutable by default, which means that its value cannot be changed. Adding the keyword "mut" after "let" makes whatever variable you're creating mutable, meaning its value can change.

Rust also can create constants with the "const" keyword. This is similar to just using "let" without "mut" but there are some differences:

- Constants cannot be made mutable since they are always immutable by design.
- Constants can be declared in any scope, including the global scope, which makes them useful for storing data that many parts of code need to use.
- Constants must be given a "type annotation", whereas immutable variables declared with "let" can be assigned a type through type inference.
- Constants can only be set to a constant value, and not a value that can only be computed at runtime, such as the result of a function. Think of it like constant values in mathematics.

- The naming convention for constants is usually that they are all uppercase.

Rust supports type inference, meaning that a type will be assigned to a variable based on what the compiler thinks makes sense. If you want to create a variable of a certain type but don't want to assign a value yet, you can either just define a mutable variable with no type for scalar types or use type "Type::new()" for compound types that interact with the heap (like strings and lists) where "Type" is replaced with whatever variable type you're using.

String interpolation in println!() can be done with {}. So, if you have a variable "decimal" that contains a float and want to print it you'd use println!("The variable decimal equals {decimal}") or println!("The variable decimal equals {}", decimal). Most people choose to use the second one, as that allows you to much easily display things like the length of an array without having to store that data in a new variable for the specific purpose of displaying it.

You can "shadow" variables by creating a new scope and then declaring a variable with an existing name. This is valid and you can do things to it in scope that do not affect the variable declared out of scope. However, if you define a mutable variable, create a new scope, and then change that variable without declaring it in that scope, it will affect the variable that was originally declared. Variables that go out of scope cannot be affected out of scope as they no longer exist.

Functions

Functions are defined with the "fn" keyword. The parameters of functions must be explicitly typed when the function is defined.

Function bodies are made up of a series of statements and expressions. Statements are instructions that perform some action and do not return a value, and expressions evaluate to a resultant value. The line "let y = 6;" is a statement because it does not evaluate a resultant value.

You can use scopes as methods of returning a value. You just have to not have a semi-colon on the returned value and the scope has to have a semi-colon at the end. An example can be found in the functions project in the practice folder.

Returning a value in a rust function works like most other languages, in that you just place return before the value that you wish to return. It's even simpler if the return statement is the final expression in the function, in which you can just remove the semi-colon from the value. If you want to return multiple values, then you can return a tuple by encasing your

return values in brackets and also encasing the return types in the function declaration in brackets in the same order as the output.

Sine Rust functions are void by default, you must contain a "->" and then a data type which is the data type the function is trying to output.

If statements

Unlike in something like C++, you don't have to put parentheses around the Boolean expression in an if statement, which is syntactically much closer to something like Python. You can for longer Boolean expressions with multiple parts, for the sake of simplicity when reading, but it's not a necessity.

If statements in Rust must have a block at the end even if they are single line, unlike in C++ where blocks containing a single line don't need the curly brackets. The same goes for a lot of other things, like loops.

You can place if statements at the end of variable declarations in order to assign values to that variable based on Boolean expressions, as long as the values are returned in the blocks by not having a semi-colon. Very important to remember that every return value in the if statement has to be of the same variable type.

If statements do not need an else at the end and can just simply be one if branch, or an if branch with any number of else if branches. So "else pass" would just be empty space.

Loops

Using "loop" starts an infinite loop in your code that can be broken by using the keyword "break". Using the "continue" keyword will skip to the very next iteration of the loop.

Variables declaration can be assigned values using loops, similar to if statements. The difference is that in order to return a value in the loop, you must use the "break" keyword followed by the value you wish to return and then a semi-colon (a semi-colon must be used in this situation unlike in others, like in if statement value assignment). This does not apply the same exact way to while or for loops.

Loops can be doubled inside each other and can be labelled by using one ' character at the start of the name. This means that you can break the first loop inside the second one, terminating both loops at the same time. Full example in control flow project in practice folder.

Rust supports regular for loops which resemble Python more than C++. You can easily loop through an array like in python and can use range by typing the start and end integers and separating them by 2 full stops. If you want to add an

increment between the iterations and use .rev() if you want the for loop to run in reverse (these can be combined in any order)

Rust doesn't just support regular while loops that terminate when a condition is false, but also supports a "while let loop" which are used with enums.

Memory management and how Rust ensures memory safety.

An overview on ownership

Ownership is the set of rules that Rust uses to manage memory. These rules do not slow down your program and if they are broken in your program, then your code won't compile. The concept of ownership can be hard to learn but will allow you to make efficient and memory safe code when you understand it better.

There are 3 basic rules for ownership:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

There are a number of problems that are inherent to allocating data to the heap, such as:

- Keeping track of what parts of your code are using what data on the heap.
- Trying to minimise the amount of duplicate data on the heap.
- Cleaning up unused data on the heap to preserve space.

Rust's ownership system seeks to address these issues and aims to alleviate the need for a programmer to think about the stack and the heap as much once they truly understand the concept.

A small note on scope. When a variable is declared on the stack and then goes out of scope, the value is dropped from the stack. This can be when new scope that's been started ends or when a function is called and then finishes. So, in other words, a variable is valid when it comes into scope and then becomes in valid when it goes out of scope.

The stack and the heap

Whether a value is placed on the stack or in the heap is extremely important for systems languages like Rust, C, C++, Go, and others, since it affects how the language behaves with allocating variables and memory.

The stack and the heap are both parts of the memory that are available for your code to use at runtime, but they are distinctly different. The stack stores values in a first-in, last-out, and everything stored on the stack has to have a known, fixed size at compile time. This can be things like variables with scalar data types like 32bit integers, but also the primitive compound data types like basic arrays and structs since they both have a defined type and size that cannot be changed. as well as structs.

On the other hand, data with an unknown size at compile time, or changeable size, must be stored in the heap. The heap is much less organised and when data is sent to the heap it has to request an amount of space. Then the memory allocator will find an empty place on the heap with the right size for the data, mark it as used, and return a pointer to that memory location. Pointers can be stored on both the stack and the heap.

Pushing to the stack is faster than allocating to the heap because data is always placed at the very end of the stack, and its less work because the allocator has to find an empty space that's big enough to store the data you're trying to allocate, and then to perform the bookkeeping to prepare for the next allocation. Accessing the data on the stack is also faster because you don't have to follow a pointer and data on the stack is very close to each other by default.

When you call a function, the parameters and the function's local variables get pushed onto the stack and are popped when the function finishes.

How does the string data type work?

Strings and string literals seem very similar and can interact in some ways, but they are also different in some important ways, which are discussed in the "variables and data types" subchapter. They are also stored differently in the computer's memory. String literals are stored entirely on the stack when they are declared (and cannot change since they are immutable), whereas strings manage data that gets allocated to the heap and is able to store text that is unknown at compile time.

Despite this, when both variables go out of scope, their value gets dropped where normally, given that strings are allocated space on the heap, strings would create memory leaks because it would drop the pointer variable and not the heap data. Rust

manages to drop all of the heaped memory by calling the function “drop()” automatically at the end of a closing curly bracket.

Shallow copying, moving, and deep copying data.

If you define a variable x with a type that has a defined size, then define y which takes the value of x, the value of y will be a copy of the value of x at that exact moment, meaning that if you mutate x then y is not affected. This is because the value of x is stored on the stack, even if it’s a compound data type, and not a pointer to a value on the heap. The data types where this can be done are:

- All integer types, unsigned or signed.
- All floating point types.
- Characters
- Booleans
- Tuples and arrays, as long as they only contain any of the previously.

However, if you try the same thing with a data type that interacts with the heap, like a string, you will attempt a “shallow copy” which mean you have 2 variables that point to the same list, meaning any edit you make to one you also make to the other. It’s the same issue as with Python’s lists.

Rust attempts to avoid this issue by implementing a “move”. If you attempt to shallow copy the value of x to y, then x will be marked as invalid by the compiler, and you can only use y to access the string. This is referred to as “moving” since you are moving the value from x to y and it’s done to avoid the “double free” error where the code attempts to free data that has already been freed, which can lead to memory corruption (which may lead to security vulnerabilities).

If you want to “deep copy” the string, then you need to use the “.clone()” method. This takes longer and requires more space on the heap because it goes through the entire string and copies every single character onto the heap, just under a different name, but it solves the issues you get with shallow copying.

Rust has a trait called “Copy” which indicates that any variable attempting to copy it should not move and invalidate the variable its copying from. This is automatically placed on variables that are stored on the stack and their values can be copied extremely fast. Another trait is “Drop”. This trait is used when a special process has to be performed to drop a data type, like with strings or lists. This trait also cannot coexist in a data type with the “Copy” trait because if something interacts with the heap, then it needs a special

process in order for its values to be dropped. **Both of these traits can be applied to your own defined data structures as well. [CHECK THIS AT CHAPTER 8]**

Ownerships and functions

When a function is declared that takes another variable type as an argument, the compiler will act differently depending on the type of that argument. If the argument passed is of a type that is stored on the stack, like the ones mentioned in the above subchapter, then passing them into a function will copy the value directly and the parameter will be passed to the stack as a new variable and dropped when the function ends. But if it's a type that does not have the Copy trait, usually because it interacts with the heap, then passing that variable as an argument moves ownership to the parameter in the function and then when the function ends, the value gets dropped and the original variable no longer has the value.

A similar principle is applied to returning values from a function (or even a new scope). If a function returns a stack value, the value is just simply copied to the calling function. If a string is being output, or some other data type that interacts with the heap, the function passes ownership of that value onto whatever variable it's returning to. It can pass ownership back to the original variable, or a different variable of the same type, or as part of a variable declaration.

Referencing and dereferencing

In rust, you can use the "&" character to reference a value and the "*" value to dereference a value. Referencing a value is where you grab the pointer to where the value is stored, and dereferencing is where you find the value at the end of a pointer. Printing a reference will return the value in that address and not the address. If you want to print to address, then type "{:p}" and add the reference you want to print at the end of the print macro.

Using references in a function declaration allows you to access the data value without taking ownership of it. References are immutable by default to stop you from accidently editing the value but can be made mutable if you wish to edit that variable.

One problem that can arise from the use of references is something known as a "data race", which is a "race condition" in which:

- 2 or more pointers attempt to access the same data at the same time.
- At least one pointer is being used to write to that data.

- And there is no mechanism to synchronise access to the data.

Data races can cause unpredictable behaviour and can be difficult to fix. Rust prevents these issues by restricting how may references there can be in one scope. These restrictions say that in regard to references a single piece of data:

- You can have as many immutable references as you want.
- If a mutable reference is created, it is the only allowed reference from that point on.
- There are allowed to be immutable references before the declaration of a mutable reference, but only if they are not used after the mutable reference is declare.

The restrictions around the number of references to a value only exist in that scope, meaning that both functions that take references as parameters and also inner and outer scopes are unaffected.

One error that exist in languages with less oversight on your memory fuckery is creating a "dangling pointer", which is a pointer that references a location in memory that may have been given to something else. This can be done by freeing some memory while also retaining a pointer to the freed memory. Rust prevents this by throwing an error whenever something tries to return a reference to a piece of data that will be freed immediately afterwards.

So, in short, Rust's compiler ensures memory safety when it comes to memory management and references by not compiling unless 2 rules are followed. At any given time, you can only have one mutable reference or any immutable references, and every reference must be valid (which means it points to a memory location that contains data).

Slices

A slice is a type of reference, so it does not have ownership, that lets you reference a contiguous sequence of elements in a collection rather than the entire collection itself. For example, since Rust does not support indexing an array in the same way as other languages, if you wanted to return the first word of a string you could have a function return a slice of that string that represents the word. This reference is immutable and has to follow the other rules of references.

String literals are stored in the binary. And a variable of type "&str" is a slice pointing to that specific point in the binary. This is why string literals are immutable; "&str" is an immutable reference.

You can create functions that not only have a slice as their output by using the "&str& type signature for the output, but also can take either a string, a string literal, or a slice of either as an input by using the "&str" type signature for the input.

Slice don't just work on strings. They also very easily work on arrays as well in the same way as they work for strings. In addition, if you want to pass an array as an input or if you want to return an array as an output, you cannot use "[t]", where t represents what data type is inside the array, because the compiler will throw an error about not knowing the size of the array. Instead, use the type signature "&[t]".

Structures and creating your own objects.

What is a struct?

A struct is a custom data type that lets you package multiple different data values of different types together under one name. Tuples and structs are similar in that they are both containers that can contain multiple values of different data types. But the 2 containers are quite different:

- The data in structs can be accessed with custom labels instead of just indices, making it easier to access and to affect.
- Different structs have to be given unique names and cannot all be defined under the type "struct", which is the opposite of how tuples work.
- You can create methods for the structs that can be attached to the structs, which you can't do for tuples.

Structs can inherit he values of other struct Structs that contain heaped data, like strings and lists, will move ownership if they are used as inputs to a function because the compiler does not know how to move them like it does with integers, floats, characters, and booleans. So, when they are passed as inputs in functions, they should be passed as references.

Structs are defined usually in the global scope. They can be defined in other scopes, but the definition will become invalid once the scope ends. Variables can be defined with the struct type can be assigned all of its values immediately, or you can interact with all of the fields of the struct and assign the values one by one if the struct variable is made mutable.

You can define structs without labels, which are called “tuple structs”, in order to simply contain small amounts of simple data, like a point in 3D space. The syntax for declaring the tuple struct definition is more simple than normal structs because you don’t add label names to the data values. These kinds of structs are used when it’s easier to identify what the values are, like with a point in 3D space, or the RBG values of a colour.

You can also define a completely empty struct, which is called a “unit struct”. These structs hold no data at all and are defined by using the struct keyword, naming the struct, and then typing a semi-colon to end the statement. They are kind of similar to the unit tuples and can be useful when you want to implement a trait on some type of data but don’t have any data that you want to store in the type itself (traits are explained later).

Since structs that contain heaped data need a specific defined method in order to be correctly deep copied (among other memory allocation things), it may seem like a better idea to use a reference for that data type. But when this is done, the compiler throws an error saying that there is no lifetime parameter on the reference (lifetime parameters are explained in the same chapter as traits).

Displaying struct data.

The normal methods of printing struct information is to define your own function or struct method (explained in the next subchapter) that will print the data in the struct in the way that you specifically want. This is the preferable way if the struct you are working with contains data that gets a bit complicated when printing with the print macros.

Attempting to print a struct using a print macro will usually lead to an error telling you that you have no implemented the “Display” trait on the struct. To use the print macro without engaging with the trait, derive debugging information by using “#[derive(Debug)]”, and use the string interpolation key “{:?}” which you also used for printing arrays and tuples. Now you can print a struct in a very clinical and uncontrollable way, which displays the type of struct and then every label name and the values assigned to them. Using “{:#?}” instead prints each label on a new line.

If you have derived debugging information, then you can use the debug macro “dbg!()” to display the struct data in a way that is similar to using “{:#?}”. It’s important to remember that you will need to t reference the struct you are using because this macro takes ownership, unlike the print macros.

Adding methods

Applying a method to a struct can be done by opening a block with the implementation keyword "impl", and then declaring your method functions in this block. Any functions that you want to interact with the struct itself, should have their first parameter called "&self" which is shorthand for "self:&Self" and provides the name self to the struct inside the method's code (this parameter is placed automatically when the function is called and does not need to be added to the function call parenthesis). If you want the method to mutate the struct, then you will need to include "&mut self" instead.

A struct having a method that takes ownership of the instance by using just self as the first parameter is rare; this technique is usually used when the method transforms self into something else and you want to prevent the caller from using the original instance after the transformation.

Methods can be given the same names as the attributes of the struct. This does not cause an issue when trying to differentiate between them because the attribute does not include the parentheses, whereas the function requires the parentheses even if it does not take any arguments.

An "associated function" is a method for a struct that doesn't take any version of the "self" argument. This is commonly used for constructors that either create an empty form of the struct, or to generate a version of the struct based on the input data (like a rectangle struct having a square constructor that only takes one size input because it's a square). Associated functions must be called by using the "::" syntax instead of a dot.

You can use multiple implementation blocks to define the methods for a single struct type. This is valid syntax even if it is inadvisable for simple structs, however it is a good idea to do this when you implement generic types and traits (explained later)

A small note on the -> operator

In Rust, this operator is used to signify the output data type of a function, but in C and C++ it is used to call methods where, while dot syntax (".") is used to call a method on the object directly, this operator is used to call the method on a pointer of the object and you need to dereference the pointer first. So, if you have an object named "object" and it's a pointer, then "object->method()" is similar to "(*object).method()".

In Rust, this operator does not work in the same way because Rust has a system of automatic dereferencing where, when you call a method on a reference of an object, Rust will automatically add the "&", "&mut", or "*" operators where they

are needed. So even if object is a pointer, “object.something” is perfectly fine in Rust.

Enumerations

What are enumerations?

Enumerations, also called “enums” are custom data types that you can define that have enumerated values. Where structs give you a way to group related fields and data, enums allow you to say that a value is one of a possible set of values. For example, structs can allow you to group the data related to shapes like rectangles and circles, enums would let you define a “shape” type which would include both of them and more.

When you declare an instance of an enum, you have to use the “::” syntax to assign which of the variants you want to use and for calling methods that don’t include the “&self” parameter, and the “.” Syntax for calling methods that do.

Initially the variants, what the things inside an enum are called, do not have any set data applied to them, but you can still create a variable equal to a variant of that enum and you can still do things with it, like basic pattern matching.

You can make your variants take data as well, which allows you to define variables that not only take a variant of an enum but can contain some data as well. You can choose no data, one or more pieces of data of the same type, or two or more piece of data with unique types (like a tuple), and some of the data types can be your own structs or enums, or even references **(but this requires implementing lifetimes which are explained in chapter 10).** You can even give the values labels similar to the fields of a struct, but this requires the curly bracket syntax similar to defining a struct as opposed to the normal regular bracket syntax of defining an enums types.

They may seem very similar, but there are situations where using an enumeration is better than a struct. For example, if you want to define a type called “message” that has 4 variations that each represent a different type of message, one of which has no data associated with it. If you tried to use a struct called “message” you’d have to give empty values to the fields that you didn’t want to use and also include an extra field that indicates which of the other fields to use, or you could define 4 different structs for each variant that aren’t grouped together in any way. Enums don’t need the indication variable, the variants are all grouped together, and you don’t need to include the redundant data for the other variants when you declare an instance of the enum.

You can also define methods for enums by using an implementation block and the syntax for this is the same as

the syntax for defining methods for a struct, such as the "&self" parameter to represent a reference of the enum itself.

The Option enum; Rust's version of null

In a language like C, there exists a null reference which can be attached to a variable in order to give it an empty value. This causes a lot of issues in programming and many languages have found their own ways of handling null references. Rust's method is to define an enum called "Option" that has 2 variants, one empty variant called "None" and one variant that takes one piece of data with a custom type called "Some". This method is the same as Haskell's "Maybe" data type and its variants "Just" and "Nothing".

This type can be applied to a variable that may or may not contain a piece of data. This normally causes issues because you will have to handle what happens if the variable does not get that piece of data for whatever reason (like when using user input to find a number and the user inputs a letter instead). If the code receives a "Some" value, then you can pattern match to it and can proceed to process the output that is attached to the enum. If the code receives a "None" value, then you can pattern match to it and run some code to handle the presence of a null value. The enum "Result" with its 2 variants "Ok" and "Err" is a similar enumeration used in general error handling.

When a "Some" is outputted, you cannot directly use the value itself in other code and instead have to extract the data inside the variant. The easiest way to do this is with pattern matching the enum because this allows you to grab the value from inside a "Some" value and also allows you to handle the possibility of a "None" value. You can also use the in-built method "unwrap()" which takes no inputs and outputs the value inside, but this will throw an error if done on a "None" type.

A quick overview of the match statement

The match statement is Rust's version of a switch statement and provides a form of pattern matching. Match statements are technically faster than if statements. However, as Rust's compiler is built out of (to quote someone on stack overflow here) "eldritch black magic" and will find ways to optimize your code so much that the performance difference between the two is negligible.

Match statements have advantages over switch statements, like being able to use ranges for integers by using the double dot syntax, whereas switch statements require a very large number of follow through cases. The same issue exists for OR statements, which in match statements is handled with a "|" between each value. Match statements can also handle string literals and strings, which switch statements in C and C++ cannot do.

The syntax of the match statements starts with using the "match" keyword, specifying the variable you're pattern matching, and starting a block. Then each case is included, with a "=>" operator pointing to the code that runs as in that case. Unlike with if statements, you don't have to include the block around code that is only one line as long as the line ends with a comma. OR Boolean is handled with the "|" syntax, ranges are handled with 2 dots, and the default case is handled with an underscore.

When handling enums, you take the values inside of an enum by labelling them and then using those labels in the code for that case. If you don't want to use those values, then you use an underscore as a label. You can also use a "()" to signify a "pass" since match statements throw a compiler error if they don't handle every variant of an enum and you may not want to do something with every variant.

If let statements

An if let statement is a smaller and more concise form of enum pattern matching that is best used when you only need to check for one case. Unlike match statements, if let statements do not need to include each different variant of an enum, or even an else case. If you want, you only need to include one variant and process it. The syntax is a bit specific, requiring you to provide the enum variant you are patterning with, followed by an equal sign, and then variable you are matching the enum variant to.

Managing Growing Projects with Packages, Crates, and Modules

When projects get larger it's important to organize your code by grouping by functionality and separating code with distinct features. While you can do this with one module, like the projects in the practice folder that were created for chapters 1 to 6, more complex project can contain using multiple modules and then different files. A single cargo project can contain multiple binary crates and optionally one library crate. As the project, usually called a package, grows , you can extract parts into separate creates that become external dependencies.

This section talks about how to do this, as well as encapsulating implementation details, which let you reuse code at a higher level so that, once an operation is implemented, other code can call it via its public interface without having to know how the implementation works. The way you write the code defines what is public and private which allows you to limit the amount of detail you have to remember.

Scope, the nested context that code is written in, is also a related concept. When reading, writing, and compiling code,

programmers and compilers need to know whether a particular name at a particular spot refers to a variable, function, struct, enum, module, constant, or another item and what that item means. You can create scopes and change which names are in or out of scope and you can't have 2 items with the same name in the same scope.

Rust has a number of features that allow you to manage your code's organization, including what details are exposed, what is private, and what names are in each scope. These features, collectively referred to as the "module system", include:

- Packages: A cargo feature that lets you build, test, and share crates.
- Crates: A tree of modules that produces a library or executable.
- Modules and Use: Let you control the organization, scope, and privacy of paths.
- Paths: A way of naming an item, such as a struct, function, or module

Packages and Cargo

A crate is the smallest amount of code that the Rust compiler considers at a time. Even if you run "rustc" on a single code file, the compiler considers that a crate. A crate can either be a binary crate or library crate. "Binary crates" are programs you can compiler to an executable that you can run, and they must have a main function. "Library crates" don't have a main function and instead, they define the functionality intended to be shared with multiple projects, such as random generation or vectors. The "crate root" is a source file that the compiler starts from and is the root module of your crate.

You create a cargo project that had a library

A package is a bundle of one or more crates that provides a set of functionalities. It contains a cargo.toml file that describes how to build those crates. Cargo is actually a package that contains the binary create for the command line tool and a library crate that contains the modules the binary crate uses. A package can contain as many binary crates as you like but only one library crate or none at all, and a package must have at least one crate.

When you run the command "cargo new", one of the files that are generated is the cargo.toml file. This file does not contain any mention of the "src" folder or the "main.rs" file because cargo follows a convention that src/main.rs is the crate root of a binary crate with the same name as the package. Likewise, cargo also knows that if there is a file called "src/lib.rs", then that is a library crate that is the

crate root for the package. Cargo won't throw an error if both files are in the src folder but will if there are more than one binary there since they should be placed in the src/bin folder instead.

Defining modules to control scope and privacy.
When using modules and the module system, the "use" keyword brings a path into scope and the "pub" keyword declare a module as public. All of them have certain rules on how they work.

- Start from the crate root because this is where the compiler goes to first.
- You can declare new modules in the crate root file by using the "mod" keyword and giving it a name. If you define a module called "garden", the compiler will look for the module's code in either:
  - A code block after the name declaration.
  - In the file src/name.rs
  - In the file src/name/mod.rs
- You can declare submodules in any file than the crate root, and you can do the same thing to submodules to give them their own sub-submodules, and so on. So, if you have a module named "garden" and you want to define a submodule called "flower", then that submodule's code can be defined in either:
  - A code block after the name declaration.
  - In the file src/garden/flower.rs
  - In the file src/garden/flower/mod.rs
- Once a module is part of a crate, you can refer to that module elsewhere in the same crate, as long as the privacy rules, using the path to the code. So a "Petal" type defines in the "flower" submodule would be found at "crate::graden::flower::Petal"
- Modules and their code are private to its parent modules by default, unless the "pub" keyword is used before the "mod" keyword.
- The "use" keyword creates shortcuts to items to reduce repetition of long paths. So using "use" before "crate::graden::flower::Petal" would mean that you only have to use "Petal" to use the type.

See the chapter 7 projects for specific examples.

Declaring structs and enums work different when being declared as private or not. Structs can be declared as public but by default all of the fields will be private and must be made individually public if you want people to be able to interact with them directly. In contrast, when declaring a public enum, all of the variants become public as well.

Paths to items in the module tree

Paths to modules can take 2 forms. One is an "absolute path" which is the full path starting from the crate root which starts with "crate", and the other is a "relative path" which starts from the current module and uses "self", "super" or an identifier in the current module. Both use the "::" syntax.

When you are trying to use a crate, you do not have to specifically use the "use" keyword and can simply call the function directly from the absolute or relative path. This can be optimal if you only want to use the function once or twice or if you want to declare your own function with the same name as the function you're using.

Knowing what path to use when using a module requires you to consider where the module is in the module tree and where in the module tree you are using it. If you are using a function that has been declared I the same context, then you are allowed to use that function even if it is not declared as public. This also applies to modules in the same context as a function that is attempting to use parts of the module. While the parts being used have to be declared public, the module itself does not have to be. This is because the outer module and the function using parts of it are siblings in the module tree.

A crate path staring with the prefix "super" indicates that it is trying to access modules and/or crates from the parent of where you want to use them. This is the equivalent of "cd .." in the command line but in the module tree. You can also stack them if you want to go back by a few generations, and anything you access this way to not have to be declared public.

Bringing paths into scope with the use keyword

The use keyword can be used to bring both functions and modules into whatever scope the programmer needs them in. The use keyword should be used when there is some code that is needed in a program and is going to be used many times. When you use the keyword on a function, you cannot have a different function with the same name because the function outside being used has been brought into scope with that name.

You can also "use" a module name instead of having to grab just a function instead. This can be very useful if you want to access many children of the module and you don't want to "use" them all individually, but still want a shorthand to call them. It may also be a good idea to "use" the parent module instead of the function so that it s clear in your code that you are using a function that isn't locally defined.

Anything that you "use" will create a shorthand that is only usable in that particular scope. So, while using "use" in the main scope may allow the function or module to be accessed in

the main function and other similarly defined functions, it cannot be accessed from any modules that get defined. For them, the same "use" line must be run in the module's code for that module's functions to access the function or module, and this only applied to the functions that are direct children of that module, not the child modules of the first module.

You can use the "as" keyword after specifying the module or function path to rename it. This is good to use if you want to use a function, but you already have a different function with the same name and you don't want to use the longhand path name for the imported function.

You can do something called "re-exporting" by using both the keywords "pub" and "use" on a module path. This allows you to create a path to the module that exist in the scope by bringing the item itself into scope. This changes the path to the module in the tree, which can be useful when the internal structure of your code is different from how programmers calling your code would think about the domain.

If you have many things from a module that you wish to use in your project, instead of having a use statement for every single one of them you can include a block at the end with every item you want to use in that block, each separated by a comma. If you want to include every item from the namespace, then you can simply use an asterisk at the end instead of a block with every single item in the namespace.

Using external packages

After an external package has been added to the project, either with adding it in the dependencies section of the cargo file or by using the "cargo add" command in the project folder, different modules and functions that have been made public in the package. You have to bring in individual parts of the package instead of just the entire thing. Anything from the standard library is already preloaded into a new project and you do not need to add "std" to the dependencies.

Common collections in the standard library

Rust's standard library offers a large number of data structures that are collections of data where you don't not know how much data will be there at compile time. The standard library contains a large number of collections for many different purposes, such as vectors, strings, and HashMaps. The rust doc page with links to all of them can be found here.

Some of these collections do not have to be "used" like other packages, like vector and string, but some of them have to be explicitly "used", like HashMap. However, the rust-analyser extension on VSCode will automatically add a "use" statement in your file if you are attempting to implement a collection like that in your code. The reason that things like HashMaps

are not included in new projects by default is that it, and a lot of other standard library collections, are generally not used as much in projects as vectors and string and so it's easier to just not include them by default in the prelude and have the user add it instead.

Vectors (docs)

Vectors allow you to store any number of values of a single uniform type next to each other in memory. To create a new, empty vector variable, you can use the "Vector::new()" method where you have to give a normal type declaration for the variable because the compiler has to know what type the vector has.

There are many methods for adding values to a vector, such as:

- Append: takes a second vector and moves all of its elements into the first vector, leaving the second empty. The 2 vectors must be mutable and the second must be passed as a mutable reference.
- Insert: places an element into the vector at whatever specified position, pushing any other elements backwards, but won't work if the index is larger than the length of the vector.
- Push: pushes a value to the end of the vector.

You can also use the "vec!" macro followed by an array of values you want to include in the vector to declare a new vector variable. If you use an empty array to create an empty vector, you will have to do the normal type declaration like with "new()".

When you want to read an element from the vector, you can either use something like "&v[i]" or "v.get(i)" where v is the vector and i is the index. The first will return a reference to the item but will panic if the index is invalid, and the second will return an Option enum with a Some if the index is valid and a None if it isn't.

Bear in mind that using the push method to add to the vector will create a mutable reference to the vector, meaning that any references you've created before hand cannot be used after the push by Rust's memory safety laws. This is because the items in a vector have to be placed next to each other in memory, which is why the time complexity of indexing the vector is constant, which is achieved by finding a space in memory where all of the elements can fit together and then dropping all of the previous elements. This means that the reference you created before the push will become invalid and cannot be used.

Since every element of an array is placed together in memory, like in an array, you can easily iterate over every single

item without needing to use indexing or the get method. You can instead create a for loop like "for item in &v" or "for item in &mut v" if you want to mutate the items in the vector (although you must dereference whatever you're trying to mutate).

Since a vector can only hold a single data type for its items, this can seem a bit restricting, but the easy way around this is to create an enumeration with a variant for each different data type you'll want to use. Then the vector will take the enum as the data type and you can include whatever variants you want whenever you want, and then when you want to use them you just put them into a match statement.

When a vector goes out of scope, every value of the vector also gets dropped along with it. This is because the drop trait has been declared for the data structure which deallocates all of the items in the vector, and this trait gets invoked when the vector goes out of scope, so the user of the vector does not have to worry about deleting all of the values themselves.

Strings (docs)

The only string-like data type in the core language is the string slice that is usually borrowed in the form "&str". String slices are references to some UTF-8 encoded string data stored elsewhere. String literals are stored in the program's binary and are therefore string slices, which is why string literals are immutable by default. In contrast, the string type is mutable and growable.

Both strings and string literals use UTF-8 encoding, which encodes 1,112,064 different characters, which means that you can store a string containing characters of a large number of different languages.

Strings can be updated by either pushing an individual character, pushing a string literal, concatenating with the + operator and another string (though not using a reference of the second string will cause it to move ownership and become invalid). If you want to create a string variable that isn't empty, then you can use the from method, string concatenation, or the format macro which lets you format multiple strings and/or string literals together with whatever characters you want in between without taking ownership of the strings.

You cannot index a string like you can in other languages because of how the characters are encoded and how the string data structure works. A string is a wrapper for a vector containing unsigned, 8-bit integers that represent the individual letters English letters as well as numbers and punctuation are stored in one byte, or 8 bits. But a string containing Hindi, which is written in Devanagari script, is

encoded differently. For instance, the word "नमस्ते" is 4 characters letters long but in a string it is encoded as 18 bytes since each letter makes up 4 bytes to be properly encoded and after the 3rd and 4th characters there is an extra byte of data encoding a "diacritic", which are symbols that provide pronunciation context for different symbols, that don't make sense on their own, but have to be included to properly represent the string that the user is trying to create.

As a result, where indexing a vector is constant because each item is equally spaced in memory, indexing a string would only return the byte at that specific index. If there was a method to return the specific character, then it would have a linear time complexity because it would have to scrub through each character individually. As a result, it is impossible in Rust to index as string because the index trait has not been implemented for the data structure.

You can slice a string, but it is usually a bad idea unless you know very specifically where the bytes of the characters in the string start and end. So, if you take a string that has characters represented by 4 bytes (like the Hindi above) and then try to slice the string from index 0 to 7, the code will panic because 7 will be in the middle of a character. And since Hindi needs the diacritics which are single bytes instead of 4, you can't always split the string in powers of 4, making string slicing even harder (the "len()" method also only measures the number of bytes in the string as opposed to the number of characters).

If you want to grab individual characters, then you iterate through the string in a for loop where the string variable ends with the ".chars()" method. And if you want to iterate through the bytes of a string, then you use the ".bytes()" method instead.

HashMap (docs)

HashMaps are collections of data that are each given a key, and from this key it uses a "hashing function" that determines where it places these keys and values in memory. Many languages support this kind of data structure, but it's given a number of different names such as hash, map, object, hash table, dictionary, associative array, and more.

HashMaps function the same way as the unordered map in C++. However, because of the use of the hashing function, getting and adding elements has constant time complexity. The hashing function used comes from the hashbrown crate, which is a rust port of Google's "SwissTable" and is also used in the "HashSet" standard library data structure.

HashMaps have to be included by the user instead of being included by default in a new project like vectors and strings. They also generally have less support from the standard library, which is why it has no built-in macro to construct it.

You can index a HashMap like you can with vectors, but instead you use a key instead of an index, and like with vectors, the code will panic if the key is invalid (you can also iterate through a HashMap like a vector). There is also a ".get()" method which will return a Some containing a reference to the data at that key or a None if the key is invalid. In order to take away the reference, you can use the ".cloned()" or the ".copied()" method if the data type of the item has one of the 2 traits implemented. Then you can follow that with the method ".unwrap_or()" which allows you to handle the possibility of an invalid key in a much more elegant way. Here, if the returned value is a Some, then the method will return whatever value is in the enum. But if the returned value is a None, then the method will return whatever value is contained inside the brackets.

If a HashMap is initialised with a string type as either the key or value, and a string variable is created that is then used as a key/value then the HashMap will take ownership of tat string data and the variable will be rendered unusable. The solution for this is to instead use a string reference as the HashMap key/value type and reference the string variable instead. This however means that you cannot mutate the variable while the HashMap still exists because it has been borrowed. This also means that the HashMap and the string variables must exist in the same scope and the HashMap must be dropped either at the same time as the string variables or beforehand.

There are a couple of ways to update a HashMap. You can use the ".insert()" method to insert a key and value, which will overwrite any values that have the same key you're attempting to add. Alternatively, you can use the ".entry()" method which will add attempt to add the value and return an enum called "Entry" that represents a value that may or may not exist. If the key doesn't exist then the value given in the method parameter, and if the key exists then nothing happens.

The method ".entry()" also returns a reference to the value in memory that exists so that you can store the value without taking ownership of it. This also means that you can affect the value itself in memory. So, if you want to create new entries in a HashMap while also incrementing new values, then you use the entry method inside a loop.

The hashing function used by the container, along with HashSet, is called SipHash. SipHash was developed to provide

resistance against DDoS attacks involving hash tables, which reduced its speed at the expense of the security, but if you would prefer to use a different hashing algorithm instead then you can specify that in your code (not sure how to, should look this up). A hasher is a type that implements the "BuildHasher" trait, and instead of needing to implement your own hasher from scratch, crates.io has libraries shared by other Rust users that provide hashers implementing many common hashing algorithms.