**What is RUST?**

* Open-Source Language similar to c++
* Designed for performance and reliability
* It is compiled language means uses a compiler that source code into machine code that can run into your system.
* **Statically Typed:** In Rust, the type of a variable is checked at compile time, meaning it must be known before the program runs. This helps catch errors early.
* **Strongly Typed:** Rust ensures that once a variable has a type, it can only hold values of that type. You can't mix different types (like adding a number to text) without explicitly converting them.

**Memory Management in Rust:**

* Rust ensures all memory access is valid, preventing issues like null pointers or invalid references.
* It’s memory-safe by default, without needing a garbage collector.
* These safety features don’t slow down performance.

**Cargo:**

* Manages dependencies for repeatable builds.
* Downloads and build external libraries.
* Calls rustc with correct parameters.

**Variable:**

* Name associated with a value stored in the computer memory.
* Declare in RUST by using **let** keyword.

**Mutability:**

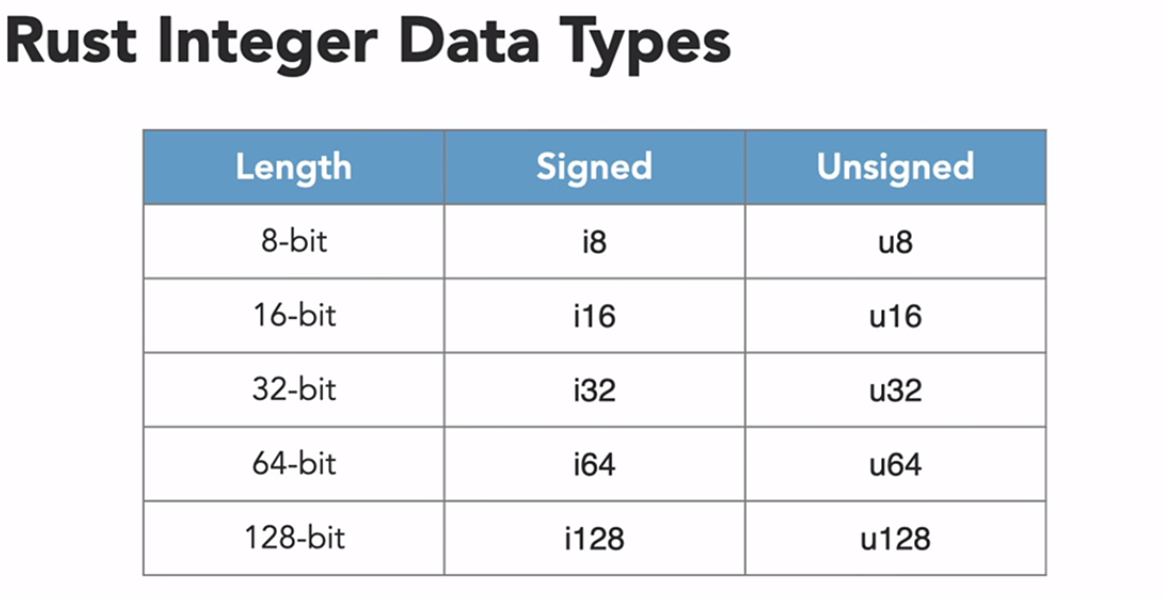
* RUST variables are immutable by default.
* Variable must be explicitly declared as mutable using **mut** keyword.

**Rust Variable Naming Rules:**

* Can contain letters, numeric digits, and the underscore character.
* Must begin with a letter or an underscore.
* Case-sensitive: uppercase and lowercase letters are distinct.
* Variable named cannot be keywords such as **let** or **mut.**

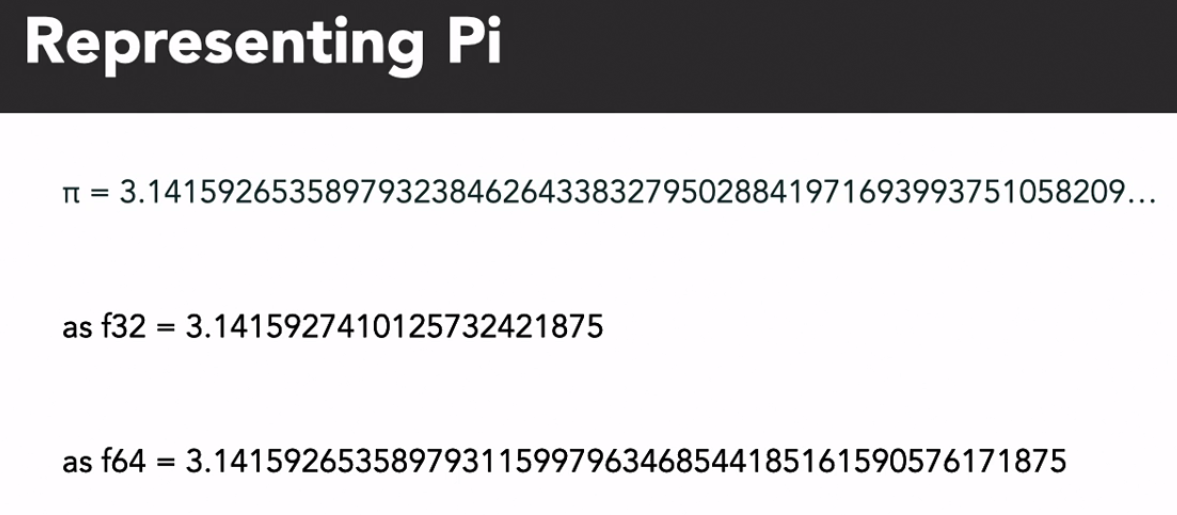
**Data Type:**

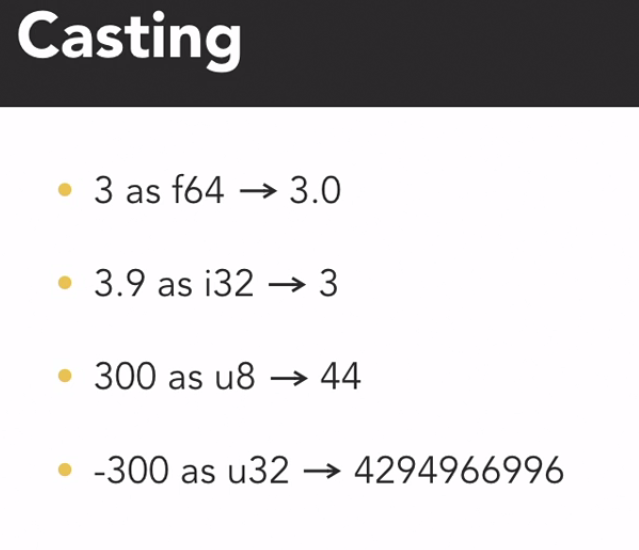
* Defines how data should be stored and interpreted.
* Defines which operation can be used to interpret data.



**Floating-Point Data Types**

* Represent numbers with decimal points using IEEE 754 standard
* Rust has two floating-point types: f32 and f64
* Value stored as fractional and exponential components
* Max f64 value≈ 1.7976931348623157 × 10308.





**Bitwise Operators:**

Logical operations on patterns of bits at the individual bit level

* **NOT**: Flips the bits of a number (1 becomes 0, 0 becomes 1).
* **AND**: Compares two numbers bit by bit, returning 1 only if both bits are 1.
* **OR**: Compares two numbers bit by bit, returning 1 if at least one bit is 1.
* **XOR**: Compares two numbers bit by bit, returning 1 if the bits are different.
* **SHIFT**: Moves the bits of a number left (adds 0s on the right) or right (adds 0s on the left).
  + let result = 0b1010 << 2; // Left shift 1010 (binary) by 2 places becomes 101000

**Boolean Data Types and its Operations:**

* Values can be either true or false.
* Logical Operations: NOT, AND, OR and XOR.
* 1 🡪 True
* 0 🡪 False

**Short Circuiting Logical Operations:**

* **&& (AND):** If the first condition is false, the second condition is not checked, because the result will always be false.
* **|| (OR):** If the first condition is true, the second condition is not checked, because the result will always be true.

**Note:** panic!() macro is used to stop the program immediately when something goes wrong, displaying an error message.

**Char Data Type:**

* Represents single character.
* **Unicode scalar:** Can represent any Unicode character.
* Stored 4 bytes.

**Scalar Types:**

Scalar types represent a single value. Rust has four primary scalar types:

* **Integer:** Whole numbers (e.g., i32 for 32-bit signed integers, u8 for 8-bit unsigned integers).
* **Floating-point:** Numbers with decimals (e.g., f32 for 32-bit, f64 for 64-bit precision).
* **Boolean:** Represents true or false values (true, false).
* **Char:** Represents a single Unicode character (e.g., 'a', '🦀').

**Compound Types:**

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

* **Array:** 
  + A fixed-size collection of values of the same type (e.g., [1, 2, 3] of type [i32; 3]).
  + Stored in contiguous memory locations.
* **Tuple:** 
  + A fixed-size collection of values of different types (e.g., ("hello", 5, true) of type (str, i32, bool)).

**usize Data Type:**

* Size is based on number of bytes needed to reference memory
* Compiling for 32-bit processor→ usize is 4 bytes
* Compiling for 64-bit processor→ usize is 8 bytes

**Function:**

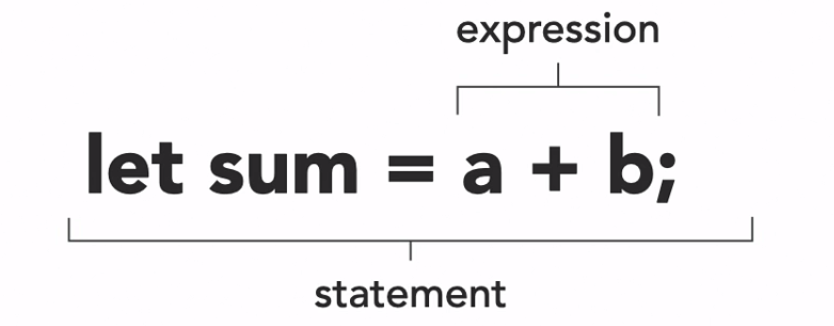
* Organize sections of code into reusable modules.
* **Parameters:** 
  + Variable used to provide input data to a function.
  + Defined in parenthesis of the function signature.

**Statement:**

* Performs an action without returning a value
* Ends with semicolon

**Expression:**

* Evaluates to a resulting value.
* Does not end with semicolon.



**Unit Data Type:**

* Used when there is no other meaningful value that can be returned.
* Represent with ()

**Loops**

**Loop:** Infinite loop that repeats a block of code until explicitly stopped.

**Break Keyword:** Immediately exit the loop and continue execution afterward.

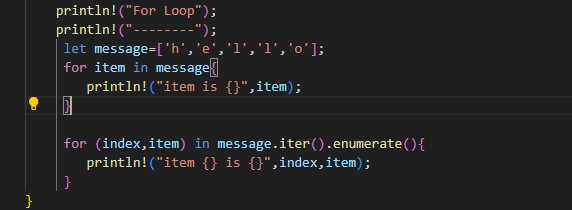
**While Loop:** Loop that repeats a block of code while condition is true

**Loop vs While Loop:**

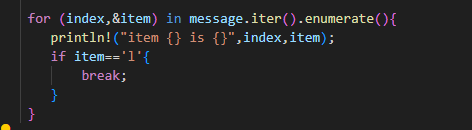


Break keyword in loop can be used to return value while in while loop break keyword does not return value.

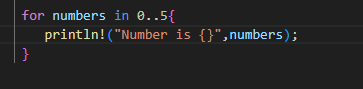
**For Loop:**



* iter() allows you to loop through each item in the collection (like a list or array) one by one.
* iter() gives references to elements.
* enumerate() gives each item an index, so you can access both the index and the item in the loop.



* &item is used because iter() gives references to elements. Writing &item lets you access the value directly instead of a reference.



**Summary of Rust Loops**

**Loop:**

* Repeat a block of code forever.
* Need the loop to return a value.

**While Loop:**

* Continue repeating the block of code as long as condition is true.

**For Loop:**

* Iterate over each item in a collection.
* Repeat a block of code N times -> iterate over range 0..N.

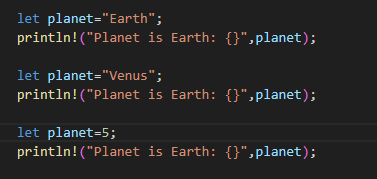
**Scope:**

* Scope defines where a variable can be accessed in a program.
* A variable becomes usable when it enters scope.
* The variable remains valid until it exits scope.
* Variable bindings are limited to a block of code, which is defined by curly braces {}, such as in functions, loops, and conditional expressions.

**Variable Mutability:** Variable in RUST are immutable by default.

**Shadowing:**

* Declaring a new variable with same name as an existing variable.
* New variable **shadows** the previous variable.



Shadowing Example



Shadowing Example Output

**Program Memory:**

In Rust, program memory is divided into stack and heap:

**Stack:**

* Stores values in sequential order.
* Works as Last In, First Out (LIFO), meaning the last added data is removed first.
* Data on the stack is processed quickly, but it must have a known and fixed size.

**Heap**:

* Used for dynamically adding and removing data.
* Accessing data from the heap is slower compared to the stack.
* In Rust, pointers hold memory addresses, and smart pointers like Box, Rc, and RefCell are used to safely manage heap memory.

**Note:**

* Integer, Floating point, Boolean, Char, Array and Tuple stored on Stack.
* String store on Heap.

**String Literal:**

* A string literal is a sequence of characters that is hard-coded directly into the source code of a program.
* **Immutable:** It cannot be changed once created.
* **Compile Time:** Its value must be known before the program runs.

**String Type:**

* A string stored on the heap.
* **Mutable**: It can be changed after creation.
* **Dynamic**: Created and modified during the program's execution, not at compile time.

**Note:** The heap has plenty of space but not infinite space.

**Explicit Allocation and Deallocation**

* Programmer is the responsible for memory management e.g. C/C++ malloc() and free() function
* **Advantage:** Programmer has lots of control
* **Disadvantage:** Memory leaks (A **memory leak** is a situation in a computer program where memory that is no longer needed is not released back to the system.)

**Garbage Collection**

* Garbage collector automatically cleans up memory Examples: Java, Python, C#, Ruby, Go
* **Advantage:** It's easy
* **Disadvantage:** 
  + **Wasteful of memory**: Memory isn't freed immediately when it's no longer needed, leading to higher memory usage until the garbage collector runs.
  + **Can run at inconvenient times**: The garbage collector might run unexpectedly, causing brief pauses or slowdowns during program execution.

**Ownership Variables**

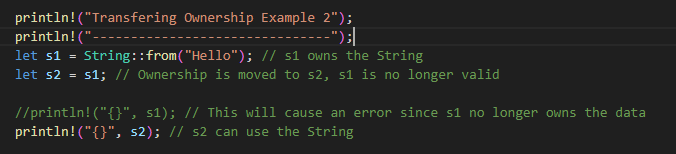
* Ownership Variables are responsible for freeing their own resources
* **Rules:**

1. Every value is "owned" by one, and only one, variable at a time.
2. When the owning variable goes out of scope, the value is dropped.

* **Advantage:** Safe and Efficient ·
* **Disadvantage:** Requires understanding of ownership.

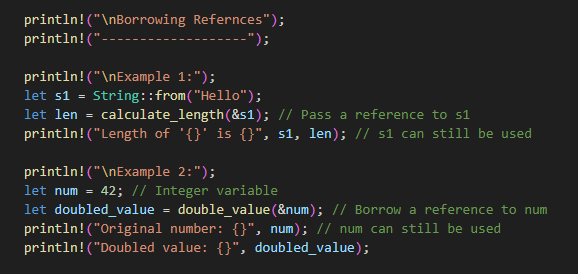
**Ownership:**

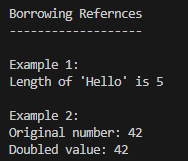
* Resources can only have one owner at a time.
* **Move:** Transfers ownership. The original variable becomes invalid.
* **Copy:** Creates a duplicate, both variables remain valid.



**Borrowing References**

* Instead of transferring ownership, you can **borrow** a reference to the data. This allows multiple parts of your code to read it without taking ownership.





**Mutable References**

* Mutable references allow you to borrow a value and modify it.
* You use **&mut** to create a mutable reference.
* **Note:**
  + **One mutable reference at a time**: Rust allows only one mutable reference to avoid data races and ensure safe concurrent access.
  + **Multiple immutable references at the same time**: Multiple immutable references can coexist because they don't allow modification, so there's no risk of data inconsistency.
  + **No mixing mutable and immutable references**: Rust prevents having both a mutable and immutable reference to the same variable to avoid unsafe access.

**Dangling References**

* A **dangling reference** happens when a reference points to data that no longer exists.
* Rust prevents this at compile time to ensure safety.

**Slice:**

* Reference to contiguous section of a collection.
* Commonly encountered as the string slice data type: &str
* String literals are slices.

**as\_bytes**

* Converts a string (&str) to a byte slice (&[u8]), showing the UTF-8 byte representation.

**b' ' (Byte Literal)**

* A single-byte representation of a character, like space (b' '), as a u8 (ASCII value 32).

**&str:**

* A borrowed string slice, representing a part of a String or a string literal.
* It is immutable and doesn’t own its data.
* Prefer &str for function parameters when possible, it’s more versatile.

**&String:**

* A reference to an owned String type, allowing read-only access to a String without ownership.

**String::from("...")**

* **Type**: String
* **Properties**: Owned (Value is owned by itself), heap-allocated, growable, can be mutable.
* **Use Case**: When you need to modify or own the string data.

**"..." (string literal)**

* **Type**: &str
* **Properties**: Borrowed (Value is owned by itself), immutable, stored in the program’s binary.
* **Use Case**: For fixed, read-only strings.

**‘Use’ Keyword:**

* Bring the module path into the scope.
* Usually included at the top.

**RUST Library Standard:**

* Available to all RUST programs by default.

**The Prelude:**

* List of things that automatically imported into every RUST program.
* Do not include the entire RUST Standard Library.

**Standard Input:**

* Read command line inputs from the user.
* Part of the **std::io** module.

**Note:**

* **trim():** This function removes any leading and trailing whitespace (like spaces, newlines, etc.) from the buffer string.
* **parse():** This function converts the trimmed string into a specified data type.
* **unwrap()**: This function is used to get the actual value from the result of parse(). If parsing fails, unwrap() will cause the program to panic (stop running with an error).

**Crates:**

* In Rust, **crates** are packages of code.
* They can be libraries or executable programs that you can use or share with others.
* Crates make it easy to manage and reuse code in projects.
* The cargo tool is used to create, manage, and add crates to your Rust projects.

**Command Line Argument:**

* Arguments passed to the program when it is invoked.
* Common Uses:
  + File paths
  + Configuration Setting

**std::env::args:**

* It returns a list (iterator) of all the arguments given.
* The first thing in the list is usually the path to the program itself.

**env:**

env in Rust is a tool that helps your program interact with its environment. It lets you:

* Access command-line arguments
* Read and set environment variables
* Find or change the current folder