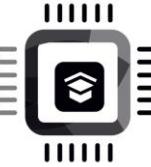


Master The Rust Programming Language : Beginner To Advanced



This presentation is related to the Udemy course,
Master The Rust Programming Language : Beginner To Advanced
by Fastbit Embedded Brain Academy

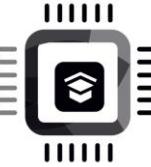
Course link:

<https://courses.fastbitembedded.com/courses/rust>

Check all our other courses:

<https://fastbitembedded.com/pages/courses>

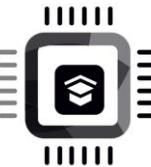
Contact : contact@fastbitlab.com



Part-I

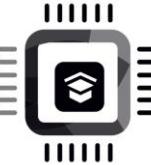
This presentation contains slides for the below sections

- 1) Introduction
- 2) Prints
- 3) Variables and Data types
- 4) Testing
- 5) References
- 6) Decision making
- 7) Strings
- 8) Ownership
- 9) Loops
- 10) Tuples
- 11) Structure
- 12) Enums



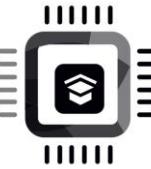
Rust programming language

1. Rust is a systems programming language designed for writing system software. It was developed to be fast, efficient, and reliable.
2. Rust is statically typed, which means that variable types are known at compile time rather than runtime. This can lead to more efficient and predictable performance.
3. Rust is a compiled language, which means that the source code must be translated into machine code before it can be executed.
4. Rust emphasizes safety, concurrency, and memory management. It provides memory safety through a system of ownership and borrowing, which helps prevent common programming errors like null pointer dereferences and buffer overflows.



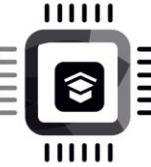
Rust is compiled language like C/C++

Rust is a compiled language, meaning the source code must be translated into machine code before it can be executed. This is different from interpreted languages like Python, which are executed directly without the need for compilation. The compilation process results in faster and more efficient performance compared to interpreted languages at the cost of a longer development cycle.



Support for OOP

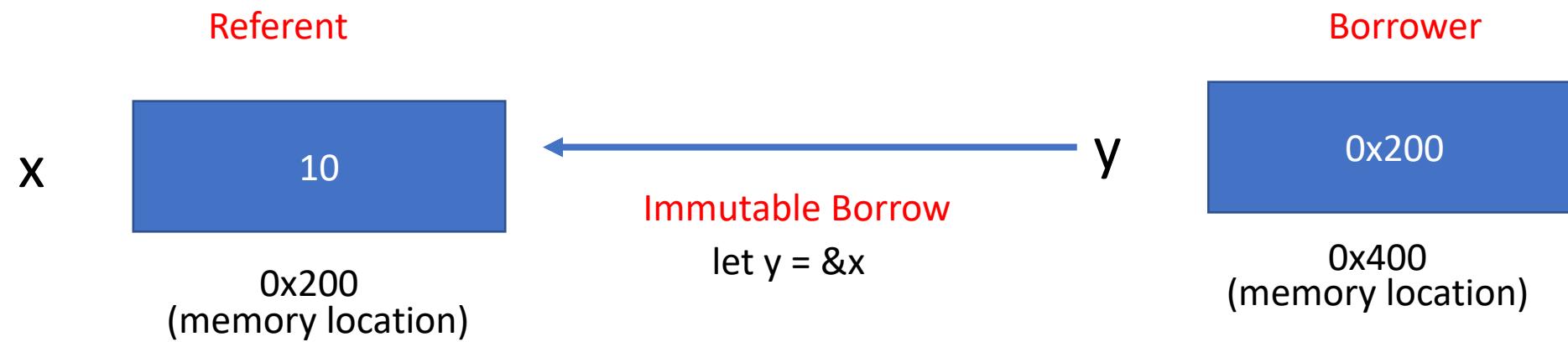
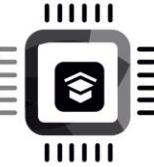
- Rust supports object-oriented programming (OOP) through the use of structures (structs) and traits. While Rust's approach to OOP is not as fully developed as traditional OOP languages like Java or C++, it provides enough features to implement common OOP design patterns. The ownership and borrowing model in Rust also helps ensure safe and predictable object lifetimes, making it a good choice for OOP programming.

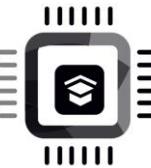


How is Rust Different from Other Programming Languages?

Memory safety:

Memory safety refers to the guarantee that a program will not cause undefined behavior or crashes due to invalid access to memory. Rust achieves this through a combination of features such as a strict ownership model, automatic reference counting, and has a borrow checker that ensures that a piece of memory can only be accessed by one part of the program at a time, preventing common programming errors such as null, data races and dangling pointer references.



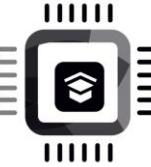


Comparing memory cleanup in C and Rust when variable goes out of scope

```
fn example_function() {  
    let s = String::from("Hello");  
    send_data(s);  
    /* when 's' goes out of scope  
       clean up happens automatically */  
}  
  
fn main() {  
    example_function();  
}
```

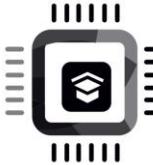
```
void example_function() {  
    char* s = (char*)malloc(20 * sizeof(char));  
    strcpy(s, "Hello");  
    send_data(s);  
    /*Manual clean up : Not freeing leads to memory leak*/  
    free(s);  
}
```

In both the cases the string “Hello” is stored in the heap
Rust cleans up the memory using the drop method



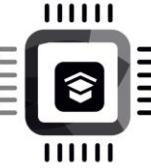
Rust automatically deallocates heap memory

- In Rust, you typically do not manually manage heap memory using functions like **new** and **free** as you would in C++.
- Rust automates memory management through its ownership system, where each piece of data has a clear, compile-time defined owner, and the memory is automatically deallocated when the owner goes out of scope.



Why Rust variables are immutable by default in its design ?

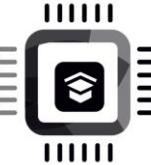
1. Improves code safety and predictability: When a variable is declared as immutable, it cannot be changed once it has been assigned a value. This helps write safer and more predictable code.
2. Facilitates better code reasoning: Immutable variables make it easier to reason about the code and reduce the chance of introducing bugs.
3. Catches potential problems at compile time: By allowing mutability only when explicitly requested with the `mut` keyword, Rust helps catch potential problems at compile time before they cause runtime errors.
4. Prevents accidental mutations: By requiring explicit declaration of mutability, Rust helps prevent accidental mutations.
5. Improves code readability, maintainability, and testability: Encouraging functional-style programming and preventing accidental mutations makes the code easier to understand, maintain, and test.



How is Rust different from other programming languages?

Ownership model:

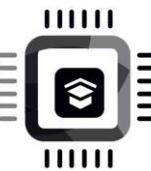
- ✓ In Rust, ownership refers to the way that the Rust compiler manages the lifetime of values in memory. Ownership rules state that a value can have only one owner, and when the owner goes out of scope, the value is dropped automatically.
- ✓ By enforcing these rules, Rust helps to prevent data race conditions and other common programming errors, making it a safe and efficient language for systems programming



```
fn main() {  
    let s = String::from("Hello");  
    let t = s;  
    println!("t is {}", t);  
    // println!("s is {}", s); //Error : s has moved to t  
}
```

A simple example of Rust's ownership feature

- 1) Avoids double free problem , which is an undefined behaviour in C/C++
- 2) Avoids use after free
- 3) Avoids data races
- 4) Efficient usage of heap memory. No two copies of the data “Hello” . Copies of data requested explicitly using clone() method
- 5) Makes the code more predictable and less prone to memory errors



Equivalent C++ code

```
int main() {
    std::string s = "Hello";
    std::string t = std::move(s);
    std::cout << "t is " << t << std::endl;

    //OK. but s is empty string
    std::cout << "s is " << s << std::endl;

    return 0;
}
```

Using ‘move’

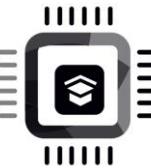
```
int main() {
    std::string s = "Hello";
    /*
    there are two separate copies of the string
    "Hello" on the heap,
    one owned by 's' and the other owned by 't'
    */
    std::string t = s;

    std::cout << "t is " << t << std::endl;
    std::cout << "s is " << s << std::endl;
}

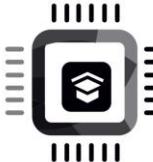
return 0;
}
```

Without using ‘move’

- ✓ In Rust, the transfer of ownership of a String from one variable to another happens automatically and by default, without the need for an explicit ***std::move***.
- ✓ In Rust, after the transfer of ownership, the original ***String*** is no longer valid, and attempting to use it will result in a compile-time error.



```
fn fun1() {  
  
    let mut s = String::from("Hello");  
    let t = String::from("World");  
    /*  
     't' ownership moved to 's'.  
     memory pointed by 's' is cleaned up  
     and 't' becomes uninitialized'  
    */  
    s = t;  
  
    /* prints "s is world" */  
    println!("s is {}",s);  
  
    /* Error : cannot use 't' unless re-initialization */  
    //println!("t is {}",t);  
}  
  
fn main()  
{  
    fun1();  
}
```



Rust avoids dangling pointer

```
int *get_pointer()
{
    int x = 5;
    /* returns dangling pointer */
    return &x;
}

int main()
{
    int *p = get_pointer();
    printf("Value pointed to by p: %d\n", *p);
    return 0;
}
```

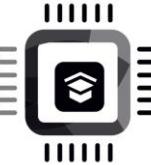
Dangling pointer issue in C

- ✓ Code compiles with a warning -Wreturn-local-addr
- ✓ ‘p’ holds an address which is not valid
- ✓ Dereferencing ‘p’ leads to undesired behavior or crashes

```
fn get_pointer() -> &'static i32 {
    let value = 5;
    &value
}

fn main() {
    let p = get_pointer();
    println!("Value pointed to by p: {}", *p);
}
```

Doesn't even compile



How is Rust different from other programming languages?

Type inference:

Rust has a ***type inference*** feature, which means that the compiler can deduce the type of a variable or expression automatically, without the need to explicitly specify it. This makes the code more readable and less verbose.

```
let x = 42; // The type of `x` is inferred to be `i32`
```

```
int x = 42; // The type of `x` is explicitly set to `int`
```



How is Rust different from other programming languages?

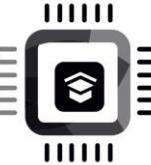
- ✓ **Concurrency:** Rust has built-in support for concurrency, allowing for the safe execution of code on multiple threads. This is achieved with lightweight "tasks" and message-passing concurrency, as opposed to shared-state concurrency used in other languages.
- ✓ **Package manager:** Rust has a built-in package manager called "Cargo" that makes it easy to find, use, and share libraries and packages. This allows for efficient development and code reuse, and it also promotes a strong community-driven ecosystem.
- ✓ **Strong typing:** Meaning, variables have a specific type, and that type must be respected. This helps to catch errors early, as the compiler will flag any type mismatches, and it also makes the code more predictable.
- ✓ **No runtime:** Rust does not have a runtime and it doesn't have a garbage collector. This allows for more control over the program and eliminates the need for a runtime, which could reduce performance, and the added complexity of a garbage collector.



How is Rust different from other programming languages?

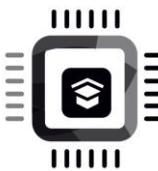
Error handling: Rust has a unique approach to error handling, which allows for the safe and explicit handling of errors. In Rust, errors are represented as specific types, which can be handled by the programmer with the ***Result*** type and the **? operator**.

Cross-platform compatibility: Rust can be used to write code that runs on a variety of platforms, including Windows, Linux, macOS, and even embedded systems. This makes it a versatile language for various types of projects and systems, including web development, Linux kernel development, game development, and IoT.



Error handling

- ✓ Rust uses return value-based error handling instead of exceptions like C++.
- ✓ In Rust, errors are typically represented with the ***Result*** and ***Option*** types to return success or error or no-value conditions from functions.
- ✓ The Rust programming language uses explicit error handling and compile-time checks to prevent undefined behavior, while C++ relies on exceptions.



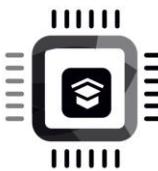
Rust uses return value based error handling

```
use std::fs::File;  
  
fn main() {  
    match File::open("file.txt") {  
        Ok(_file) => {  
            //File open success  
            //Do something with the file  
        },  
        Err(error) => {  
            //Handle the error  
            println!("Error opening file:{:?}", error);  
        },  
    }  
}
```

In Rust, the enum **Result** type is used to handle errors. Here is an example of how to open a file in Rust and handle any errors that may occur.

In this example, the **File::open** function is used to open a file. This function returns a **Result<File, Error>**, which can have two possible values: **Ok** if the file was successfully opened and contains the file handle or **Err** if an error occurred and contains the error value.

The **match** statement is used to check the result of the **File::open** function and handle the error accordingly



Exception handling C++ : try and catch block

```
#include <fstream>
#include <iostream>

int main() {
    std::ifstream file("file.txt");
    try{
        if(file.is_open()){
            //File open was successful
            //Do something with the file...

        }else{
            throw std::ios_base::failure("File not available");
        }
    }catch (std::ios_base::failure& e) {

        //Handle the error
        std::cout << "Error opening file:" << e.what() << std::endl;
    }

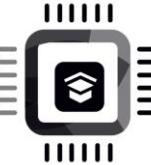
    return 0;
}
```

In C++, exceptions are implemented using try-catch blocks, where the code that might throw an exception is put in a try block and the exception handler is put in the catch block

The exception-based approach can make code more readable and concise, as it allows for error handling to be separated from the normal flow of the code.

This can make the code easier to understand and maintain, as the error-handling logic is not mixed with the normal logic.

.



Is error handling based on return values in Rust better for embedded software use cases than C++?

There are several issues with using C++ exception handling in embedded systems:

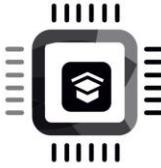
Overhead: C++ exceptions add extra overhead to the memory and processing resources of embedded systems. This can be a problem for systems with limited resources, as it can cause the system to run out of memory or slow down.

Unexpected behavior: C++ exceptions can cause unexpected behavior on embedded systems if not handled properly. This can lead to crashes or unexpected system behavior, which can be difficult to track and fix.

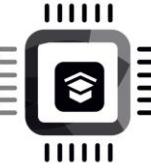
Non-deterministic behavior: C++ exceptions can cause non-deterministic behavior, which is not suitable for real-time embedded systems. Exceptions can interrupt the normal flow of the program and make it difficult to predict how the program will execute.

Reduced performance: C++ exceptions add a level of indirection, which can make it less efficient than Rust's return value-based approach.

create, build and test simple hello world Rust program



1. Install the Rust programming language by downloading the installer from the official website (<https://www.rust-lang.org/tools/install>) and running it on your computer.

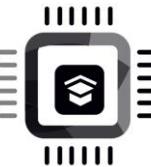


create, build and test simple hello world Rust program

2. Open your terminal or command prompt and create a new work-space

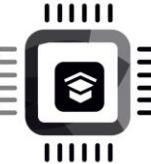
```
mkdir rust_projects
```

```
cd rust_projects
```



Cargo: The package manager for Rust

- ✓ Cargo is the package manager for the Rust programming language.
- ✓ There are similar package managers exist for other programming languages. For example, npm (Node Package Manager) for JavaScript, pip for Python, gem for Ruby, etc. These package managers allow developers to easily manage dependencies and install packages and libraries for their projects.
- ✓ It allows you to manage dependencies, compile and build your code, and manage the distribution and publishing of your software.
- ✓ It also provides a convenient way to manage your project's metadata, such as versioning, author information, and dependencies. Additionally, Cargo provides features such as optimization, release builds, and automatic dependency resolution, making it easier for you to build high-quality, performant, and well-documented Rust code.

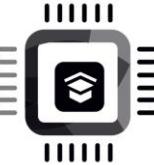


Cargo: The package manager for Rust

There are similar package managers exist for other programming languages.

- **(Node Package Manager)** for JavaScript
- **pip** for Python,
- **gem** for Ruby, etc

Cargo allows you to manage your project's dependencies, compile and build your code, and distribute and publish your software.



create, build and test simple hello world Rust program

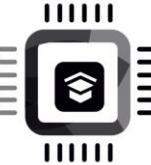
3. Initialize a new Rust project by running the command

```
cargo new hello_world_001
```

This command will create a new cargo project with the name "hello_world_001"

"main.rs" can be considered as the binary crate in a Rust project managed by Cargo.

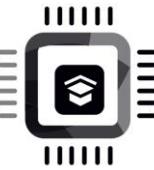
Binary crates are the executable crates, which contains the **main** function that serves as the entry point for the program.



Crate

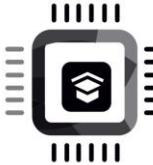
A crate is a collection of code and modules that are compiled into a binary file or library file. Crates provide a way to organize and structure your code, as well as make it reusable and sharable across multiple projects. By building your code into a crate, you can publish it to a registry and make it available for others to use as a dependency in their own projects.

create, build and test simple hello world Rust program



4. Open the newly created "main.rs"(which is the rust source file) file in a text editor.
This file will contain the main function, which is the entry point for your program.
Put the below code in main.rs

```
fn main() {  
    println!("Hello, World!");  
}
```



‘fn’ keyword in Rust is used to declare functions

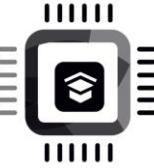
Indentation is not mandatory,
but 4 spaces of indentation
make code easier to read and
it's a common convention

```
fn main() {  
    println!("Hello, World!");  
}
```

Function name
Function arguments list
Function body starts here
Code statement in Rust, must end with semicolon(;)
Function body ends here

- ✓ “main” function definition which is the entry point of the program. This is where the program starts executing.
- ✓ `println!("Hello, World!");` is a Rust macro that prints the string "Hello, World!" to the console. The `println!` macro is similar to the print function in other programming languages, but it also adds a new line at the end.

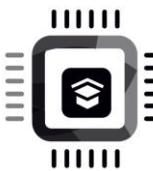
create, build and test simple hello world Rust program



5. Build and run the program

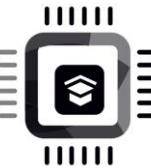
```
cargo build  
cargo run
```

You can also use the command ***cargo build --release*** to create a release build of your program.
This will optimize the program for performance and make it ready for distribution



Other flavours of `println!`!

- `print!`: similar to `println!`, but it doesn't add a new line at the end of the output. This can be useful when you want to print multiple things on the same line.
- `eprintln!`: similar to `println!`, but it prints to the standard error stream instead of the standard output stream. This can be useful for printing error messages.
- `format!`: creates a new string with the given format. This can be useful when you want to create a string that includes multiple values or when you want to use more advanced formatting options.
- `write!` : It's similar to `format!` but it writes to a buffer rather than returning a new string.
- The ! at the end of the macro name indicates that it is a macro, not a function and macros are part of the Rust standard library.
- All the above macros can be used to format output with placeholders. These placeholders are denoted by {}.



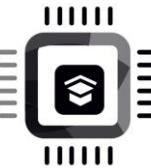
Example

Here's an example of how you can use different print variants in one function

```
fn main() {  
    print!("Hello, ");  
    eprintln!("An error occurred: invalid input");  
    let name = "John";  
    let age = 30;  
    let message = format!("My name is {} and I am {} years old", name, age);  
    println!("{}", message);  
    println!("Hello, World!");  
}
```

Output:

```
An error occurred: invalid input  
Hello, My name is John and I am 30 years old  
Hello, World!
```

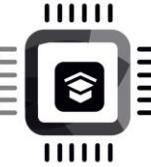


eprintln!

[1]

- Prints to the standard error stream, with a newline.
- Equivalent to the `println!` macro, except that output goes to `io::stderr` instead of `io::stdout`
- Use `eprintln!` only for error and progress messages. Use `println!` instead for the primary output of your program
- By default, the standard output and standard error streams are both displayed in the terminal
- The terminal provides two separate streams where the program can write output: one for regular output (`stdout`) and one for error messages (`stderr`)
- The standard error or standard output stream can be redirected to a separate file using redirection operators such as '`>`' (redirects `stdout` messages) or '`2>`' (redirects `stderr` messages)

[1]: <https://doc.rust-lang.org/std/macro.eprintln.html>



format!()

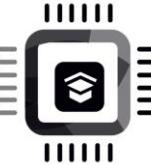
syntax:

```
let output = format("{}", value);
```

- where ‘value’ is the value that you want to insert into the string.
- {} is the placeholder for ‘value’
- You can also use multiple placeholders {} in the format string to insert multiple values.

```
let name = "John";
let age = 30;
let message = format!("My name is {} and I am {} years old", name, age);
println!("{}", message);
```

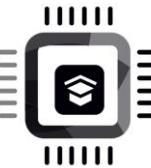
In this example, the **format!** macro is used to create a new string with the format "**My name is {} and I am {} years old**" and the values of the variables **name** and **age** are inserted into the placeholders. The resulting string is stored in the variable **message**.



Writing clean, efficient, and idiomatic Rust code

Recommended tools for Rust developers.

1. **rustfmt** : This tool automatically formats Rust code according to the community style guidelines. It ensures that code across your project has a consistent style, which is particularly helpful in collaborative environments.
2. **Clippy**: is a static analysis tool for Rust, often referred to as a linter or providing lint checks. It analyzes Rust code without executing it to find a wide range of issues that the Rust compiler itself doesn't necessarily catch.
3. **cargo fix**: This tool automatically applies fixes to Rust code based on the diagnostics provided by the compile(rustc) , particularly useful when upgrading your code to comply with newer versions of Rust



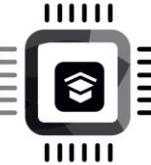
Named placeholders

```
let name = "John";
let age = 30;
let message = format!("My name is {user_name} and \
|           I am {user_age} years old", user_age=age, user_name=name);
println!("{}", message);
```

This place holder is identified by the name 'user_name'



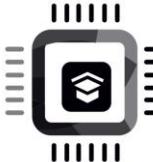
'\` is used to split a single long string into multiple lines in the source code, while the string will still be treated as a single line when it is processed, allowing the code to be written in multiple lines for readability



Print floating point values

```
//print floating point values with custom decimal places
let real_value = 3.14159;
println!("With 2 decimal places value would be {:.2}", real_value);
println!("With 6 decimal places value would be {:.6}", real_value);
println!("int part of the real value {} is {}", real_value, real_value.as_i32);
```

:.2 and :.6 are format specifiers



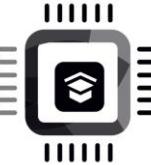
Print decimal numbers in hex

You can use any ***print*** or ***format*** macro along with the **`:#X`** or **`:#x`** or **`:x`** or **`:X`** format specifier to convert and print a decimal number as hexadecimal in Rust.

```
//print in hexa decimal style
let decimal_num = 6789;
let output1 = format!("decimal number {} in hex is {:#X}", decimal_num, decimal_num);
let output2 = format!("decimal number {} in hex is {:#x}", decimal_num, decimal_num);
let output3 = format!("decimal number {} in hex is {:x}", decimal_num, decimal_num);
println!("{}\n{}\n{}", output1, output2, output3);
```

Output:

```
1 decimal number 6789 in hex is 0x1A85
2 decimal number 6789 in hex is 0xa85
3 decimal number 6789 in hex is 1a85
```



Printing a number in binary format

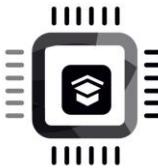
Use **:b** or **:#b** format specifier .

The **#b** format specifier is used to print binary numbers with a **0b** prefix.

```
//print in binary  
println!("decimal number {num} in binary is {num:#b}", num=6789);  
//println!("decimal number {} in binary is {:#b}", 6789, 6789);
```

Output:

```
decimal number 6789 in binary is 0b1101010000101
```

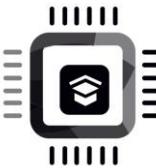


Printing a text which with special characters

```
David says , " Programming is fun !"  
**Conditions apply , "Offers valid until tomorrow"  
C:\My computer\My folder\  
\ \ \ \ Today is holiday \ \ \ \  
This is a triple quoted string """" This month has 30 days """"
```

```
/*  
... print a message which contains special  
... characters like single quotes ' ' and backslash '\'  
*/  
//println!("David says, \"Programming is fun\"");//Error  
println!("David says, \"\"\"Programming is fun\"\"\""); //OK. Note that \" used to help compiler escape \" twice  
  
//println!("C:\My computer\My folder");//Error  
println!("C:\\\\My computer\\\\My folder");//OK. \\ used to help compiler escape \\ twice
```

In Rust, you can use a combination of the **print** macro and the escape character ‘\’ to print a string that contains special characters like double quotes and backward slashes (\)



Raw string

Consider this example : here we want to print below line of text
C:\My computer\My folder

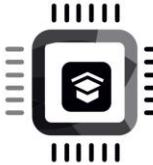
Solution 1 : Use the escape character '\'

```
·println! ("C:\\My\\computer\\\\My\\folder");
```

Solution 2 : tag the string with letter 'r'.

Instead of using an escape character '\' for every backslash we want to print , just tag the whole string with letter 'r' which makes no escape sequences are being recognized in the string

```
/* Works because in 'r' tagged string , escape character '\' is not recognized */
println!(r"C:\\My\\computer\\\\My\\folder");
let message = r"\\\\"Today is holiday\\\\"";
println!("{}" , message);
```



String tagging with r#

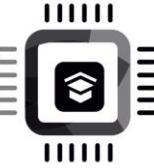
Try to execute this code

```
let message = r"\ \ \"Today is holiday\" \\ \\";
println!("{}",message);
```

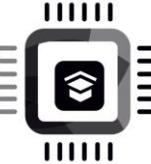
'r' tagged string by default
doesn't recognize double
quotes present in the message

Solution is to tag the string with **r#** and mark the end of tagging with **#**

```
/* string tagging with r#.....# */
/* #### used for readability purpose , you can use as many #s you want */
let message = r####"\ \ \"Today is holiday\" \\ \\ "#;
let message = r#"\\ \"Today is holiday\" \\ \\ "#; /* same as above */
println!("{}",message);
```



Print large passage of text



```
fn main(){
    println!("C:\My computer\My folder");

    let message = "####"
    Keywords and control flow
    In Rust, blocks of code are delimited by curly brackets, and control flow is annotated with keywords

    fn main() {
        let mut values = vec![1, 2, 3, 4];

        for value in &values {
            println!("value = {}", value);
        }

        if values.len() > 5 {
            println!("List is longer than five items");
        }

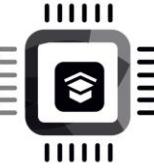
        // Pattern matching
        match values.len() {
            0 => println!("Empty"),
            1 => println!("One value"),
            2..=10 => println!("Between two and ten values"),
            11 => println!("Eleven values"),
            _ => println!("Many values"),
        };
    }

    // while loop with predicate and pattern matching using let
    while let Some(value) = values.pop() {
        println!("value = {}", value); // using curly braces to format a local variable
    }
}

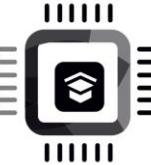
"####;

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

[https://en.wikipedia.org/wiki/Rust_\(programming_language\)](https://en.wikipedia.org/wiki/Rust_(programming_language))

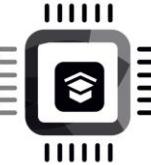


Variables, Data types and Mutability



Variables, Data types and Mutability

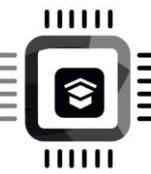
- Data types
- How to declare variables
- Type inference feature of Rust
- Mutable and immutable nature of the variables
- Rules for writing legal variable names
- Variable naming conventions



Data types (Primitive)

Primitive types in Rust:

- Integers: `i8, i16, i32, i64, isize, u8, u16, u32, u64, usize.`
- Floating-point numbers: `f32, f64`
- Boolean: `bool`
- Character: `char`
- Arrays: `[T; N]`, where T is the type of elements and N is the number of elements.
- Slices: `&[T]`
- Tuples: A tuple is a fixed-length collection of elements, where each element can have a different type. They are defined by a set of parentheses enclosing a comma-separated list of types. `(T1, T2, T3, ...)`

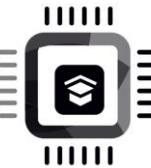


Rust numerical types

Type	Size (bytes)	Range
i8	1	-128 to 127
i16	2	-32768 to 32767
i32	4	-2147483648 to 2147483647
i64	8	-9223372036854775808 to 9223372036854775807
isize	platform dependent (usually 4 or 8)	platform dependent
u8	1	0 to 255
u16	2	0 to 65535
u32	4	0 to 4294967295
u64	8	0 to 18446744073709551615
usize	platform dependent (usually 4 or 8)	platform dependent
f32	4	-3.4×10^{38} to 3.4×10^{38}
f64	8	-1.8×10^{308} to 1.8×10^{308}

The isize and usize types are platform dependent, their size is 4 bytes on 32-bit platforms and 8 bytes on 64-bit platforms.

The i128 and u128 types were introduced in Rust version 1.26, they are intended to represent 128-bit signed and unsigned integers respectively.



C Program

```
#include <stdio.h>

int main()
{
    int num1 = 5; //int32_t num1=5;
    int num2 = 10;
    int sum = num1 + num2;

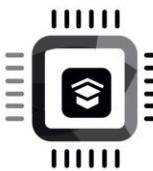
    printf("Sum is %d\n",sum);

    return 0;
}
```

Rust Program

```
fn main() {
    let num1: i32 = 5;
    let num2: i32 = 10;
    let sum: i32 = num1 + num2;
    println!("Sum is {}",sum);
}
```

A program that creates 2 variables, initializes, adds and prints the result



Variable definition(immutable)

Syntax:

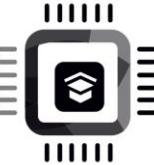
In Rust, a variable is defined using the '**let**' keyword, followed by the variable name and an equal sign, and then the value that you want to assign to the variable. The basic syntax for defining a variable in Rust is as follows:

```
let variable_name = value;          /* No type is mentioned */

let variable_name: type = value;    /* Type is explicitly spelled out */

let variable_name;
variable_name = value;            /* Delayed initialization*/
```

If you want to specify the **type** of a variable, you can use a **colon(:)** followed by the **type** after the variable name.



Example

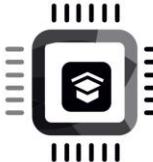
```
let num: i32 = 100;
```

For readability purpose maintain single spaces here

Coding guidelines:

Ensure that operators such as +, =, *, /, -, and others are always surrounded by a single space.

Ensure that the **type** name is preceded by one space

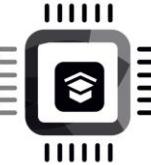


It's worth noting that in Rust, you don't have to explicitly specify the type of a variable when declaring it, the Rust compiler is able to infer the type based on the value that is being assigned

```
fn main() {  
    let num1 = 5;  
    let num2 = 10;  
    let sum = num1 + num2;  
    println!("Sum is {}", sum);  
}
```

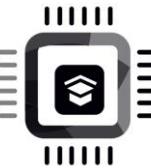
Same program with type inference:

Rust compiler infers the type of these variables as *i32* as they are integers. Since both *num1* and *num2* are of type *i32*, the *sum*'s type is also inferred as *i32*.



Type inference

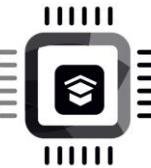
- ✓ It's a feature that allows the compiler to automatically determine the type of a variable or expression based on the context(value assignment, functional call , etc.) in which it is used. .
- ✓ It's worth noting that type inference is not always possible, and in some cases the programmer must specify the type explicitly. For example, when using a generic type or when a variable is mutable.



What's the type of **num1** and **num2** ? i32 or u8?

```
fn main() {  
    let num1 = 10;  
    let num2 = 20;  
    let sum: u8 = num1 + num2;  
    println!("Sum = {} ", sum);  
}
```

Here, the binding of num1 and num2 occurs after comprehending the context in which those variables are being utilized.

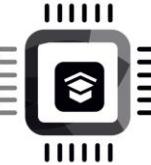


About initialization

- ✓ Variables must be initialized at the time of their definition. This means that you must provide a value for the variable when you define it, and that value cannot change later if the variable is of immutable type.
- ✓ Rust's strict initialization requirement is a safety feature of the language, it ensures that all variables have valid values, which can help prevent bugs and undefined behavior in your code.

```
fn main() {  
    let num1: i32;  
    let num2: i32;  
    //num1 = 30;  
    //num2 = 50;  
    let sum = num1 + num2;  
    println!("Sum is {}", sum);  
}
```

This code will not compile



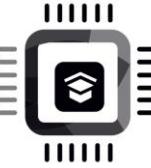
Immutable

When a variable is declared using the "let" keyword, it is considered immutable by default. This means that once a value is assigned to the variable, it cannot be reassigned a new value.

```
let num = 10;  
num = 50; //Error. num is immutable
```

```
let mut sum = 0;  
sum = num + 50; //OK. sum is mutable
```

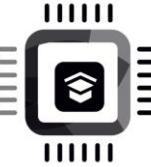
//Equivalent code in 'C'
const int num = 10;



Variable definition(mutable)

In Rust, a variable can be declared as mutable by using the "***mut***" keyword. The syntax for declaring a mutable variable is as follows:

```
let mut variable_name: type = value;
```



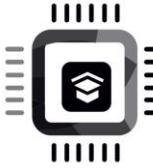
Rust enforces strict type checking.

This code doesn't work in Rust

```
fn main() {  
    let num1 = 20;  
    let num2 = 4.5;  
    let mul = num1 * num2;  
    println!("{}", mul);  
}
```

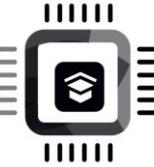
```
error[E0277]: cannot multiply '{integer}' by '{float}'  
--> src/main.rs:6:20  
|  
6 |     let mul = num1 * num2;  
|           ^ no implementation for '{integer} * {float}'  
|  
= help: the trait `Mul<{float}>` is not implemented for '{integer}'  
= help: the following other types implement trait `Mul<Rhs>`:  
    <&'a f32 as Mul<f32>>  
    <&'a f64 as Mul<f64>>  
    <&'a i128 as Mul<i128>>  
    <&'a i16 as Mul<i16>>  
    <&'a i32 as Mul<i32>>  
    <&'a i64 as Mul<i64>>  
    <&'a i8 as Mul<i8>>  
    <&'a isize as Mul<isize>>  
and 49 others
```

Here, the compiler tries to find an implementation of the **Mul trait** for the types i32 and f64. Since no such implementation exists, the compiler reports an error.



Rust enforces strict type checking.

- ✓ The reason the expression `num1 * num2` does not compile in Rust is because the multiplication operator `*` is not defined for the combination of `i32` (the type of the literal `20`) and `f64` (the type of the literal `4.5`).
- ✓ Rust defines the ***Mul*** trait for types that support the multiplication operator `*`. The ***Mul*** trait is implemented for many numeric types, including ***i32*** and ***f64***. However, Rust does not provide an implementation of ***Mul*** for a combination of ***i32*** and ***f64***.
- ✓ The ***Mul*** trait is an example of Rust's strong type checking, where Rust ensures that only types that implement the `Mul` trait can be multiplied using the `*` operator. By doing so, Rust can catch any attempts to multiply values of incompatible types at compile time, which helps prevent type errors



Trait

- They are similar to interfaces in other programming languages.
- A trait is a feature in Rust that allows you to define a set of methods that can be implemented by multiple types. Traits provide a way to abstract over types and define shared behavior
- For example, the **Add** trait is defined in the Rust standard library and provides an implementation for the + operator. Types that implement the **Add** trait can be added together, regardless of their specific type.

```

trait Hello {
    fn say_hello(&self) -> String {
        "Hello!".to_string()
    }

    fn introduce(&self) -> String;
}

struct Person {
    name: String,
}

impl Hello for Person {
    fn introduce(&self) -> String {
        format!("My name is {}.", self.name)
    }
}

struct Animal {
    name: String,
}

impl Hello for Animal {
    fn introduce(&self) -> String {
        format!("I am a {}.", self.name)
    }
}

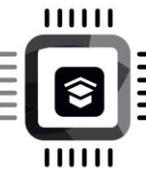
fn main() {
    let person = Person { name: "John".to_string() };
    let animal = Animal { name: "Dog".to_string() };

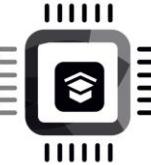
    println!("{}", person.say_hello());
    println!("{}", person.introduce());
    println!("{}", animal.say_hello());
    println!("{}", animal.introduce());
}

```

Output

Hello!
My name is John.
Hello!
I am a Dog.





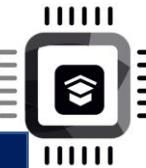
```
fn main() {  
    let num1 = 10;  
    let num2 = 20;  
    let sum: u8 = num1 + num2;  
    println!("Sum = {}", sum);  
}
```

This works

```
fn main() {  
    let num1: i32 = 10;  
    let num2: i32 = 20;  
    let sum: i64 = num1 + num2;  
    println!("Sum = {}", sum);  
}
```

This doesn't work

```
18 |     let sum: i64 = (num1 + num2);  
18 |     |-----^ expected `i64`, found `i32`  
18 |     |  
18 |     |expected due to this  
18 |  
18 |     help: you can convert an `i32` to an `i64`  
18 |  
18 |     let sum: i64 = ((num1 + num2)).into();  
18 |             +++++++
```



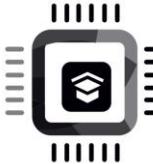
Will this code compile?

```
fn main() {  
    let num1: i32 = 0xffff_ffff;  
    let num2: i32 = 20;  
    let sum: i64 = (num1 + num2) as i64;  
    println!("Sum = {}", sum);  
}
```

Rust compiler detects an unusual coding pattern here , that is an attempt to store positive literal value which doesn't fit in positive range of i32

- ✓ By adding the **`#[allow(overflowing_literals)]`** attribute, you can suppress the lint check for overflowing literals, and the code will compile without any errors
- ✓ A lint check is a static analysis tool that helps identify potential problems in the code before it is compiled

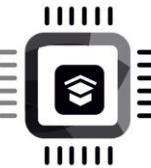
Read more about lint checks : <https://doc.rust-lang.org/reference/attributes/diagnostics.html>



```
let x = 5; //x is i32 type  
let y = 5 + 3.14; // Error: cannot add i32 to f64
```

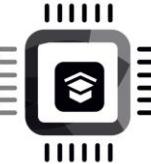
```
let x = 5; //x type is i32  
let y = 5 as f64 + 3.14; //y type is f64
```

- ✓ In this case, the programmer needs to help the compiler by using '**as**' keyword to cast **5** to a **f64** (floating-point number) so the addition can be performed correctly.
- ✓ Without this explicit conversion, the compiler would not be able to infer the correct type for **y** and would raise an error.
- ✓ In general, when the compiler is unable to infer the correct type for a variable or expression based on the context, the programmer may need to provide an explicit type annotation to help the compiler out.



'as' keyword

- ✓ The 'as' keyword in Rust is used for explicit casting, meaning it is used to convert a value of one primitive type into a value of another primitive type.
- ✓ Unlike some other programming languages, Rust does not automatically perform type casting. Instead, you must explicitly use the 'as' keyword to tell the compiler that you are aware of the type conversion and that you understand the potential loss of data that may occur during the casting.
- ✓ 'as' is also used to rename imports in use and extern crate statements:



Suffixes to specify the type of a number

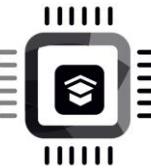
In Rust, you can use suffixes to specify the type of a number when it is being written in the source code

For integers, these suffixes can be used : **u8, i8, u16, i16, u32, i32, u64, i64, u128, i128 , isize, usize**

For floating point numbers : **f32 and f64**

```
let num1 = 100_u8;  
let num2 = 0xffffu32;  
let num3 = 0.5_f32;
```

Type of variable **num1** will be **u8** because it was explicitly spelled out using the **u8** suffix for the value 100.

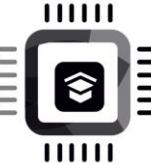


Underscore literal

- ✓ You can use the underscore (`_`) character to separate groups of digits in numeric literals, it's called an "*underscore literal*" in Rust. This can make large or long numbers more readable.
- ✓ For example, instead of writing `1000000` you can write `1_000_000` which makes it more readable.
- ✓ You can also use this feature in combination with type suffixes.

```
let x = 1_000_000; // x = 1000000
let y = 0xff_ff_ff_ff_u32; // y = 0xffffffff
let z = 3_.1423_56f32; // z = 3.142356
```

The underscore characters (`_`)
are used as visual separators
and are ignored by the
compiler.



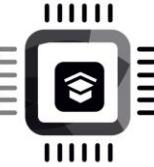
Storing ascii code

- For this use primitive type ***u8***
- ***u8*** is an unsigned 8-bit integer type. It can store values between 0 and 255 (inclusive).
- Use the '***b***' prefix if you want to store the ascii codes using ascii characters in the code
- The type of ***code*** is ***u8*** in the below code

```
let code = b'+';
let code = '+'_u8; //Invalid syntax
```

'b' syntax before a character literal is used to create a ***u8*** type, which represents an 8-bit unsigned integer.

The ASCII value of the '+' character is 43, so the ***code*** variable will have the value 43.



char vs u8

let x = b'+';

x type is **u8**

x consumes 1 byte

Ascii code of '+' is stored in **x** which is 43

Here '+' is byte literal because of prefix 'b'

let y = '+';

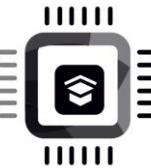
y type is **char**

In rust **char** type consumes 4 bytes

Unicode Scalar Value (USV) of '+' is stored in **y**

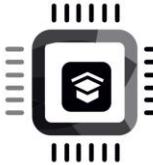
USV of character '+' = U+002B (decimal 43)

Here '+' is character literal



char

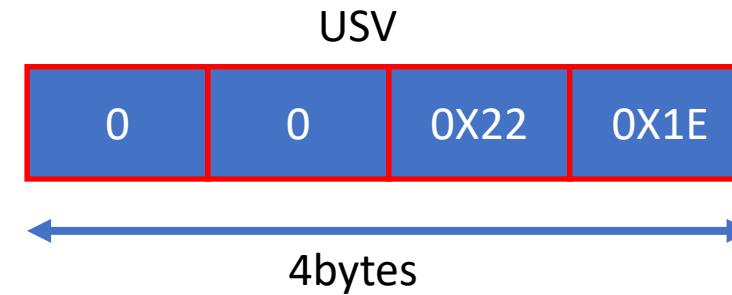
- In Rust, **char** data type uses Unicode Scalar Values (USV) to represent characters.
- Unicode is a standardized character encoding system that assigns unique numbers, called scalar values, to every character in every writing system in the world. These scalar values are in the range of 0 to 0x10FFFF and can be represented as a u32 value
- *Unicode Scalar Value*. Any Unicode *code point* except high-surrogate and low-surrogate code points. In other words, the ranges of integers 0 to D7FF₁₆ and E000₁₆ to 10FFFF₁₆ inclusive
- A Unicode scalar value is a unique number that represents a single character, regardless of the platform or program you are using. For example, the letter "A" has the scalar value 65, the letter "a" has the scalar value 97, and the symbol "∞" (infinity) has the scalar value 8734.



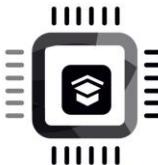
Example of using Unicode Scalar Value to create a **char** type variable in Rust using '\u{}' syntax

```
let infinity_symbol = '\u{221E}';
println!("symbol = {}, usv = {}", infinity_symbol, infinity_symbol.as_u32);
```

symbol = ∞, usv = 8734



The \u is used to specify a Unicode code point in Rust. The number inside the curly braces, {221E}, is the hexadecimal representation of the Unicode code point for ∞



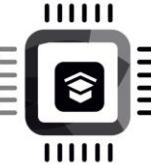
Converting u32 to char

```
let usv_of_inf = 0x221e_u32;
println!("symbol = {}", usv_of_inf as char); //Error
```

To convert a **u32** value to a **char**, you can use the **char::from_u32** function, which returns an **Option<char>**

```
fn main() {
    let usv_of_inf = 0x221e_u32;
    if let Some(inf_symbol) = char::from_u32(usv_of_inf) {
        println!("symbol = {}", inf_symbol);
    } else {
        println!("Not a valid Unicode scalar value");
    }
}
```

The **char** type represents a Unicode scalar value, which can have values ranging from U+0000 to U+10FFFF. However, not all **u32** values are valid **char** values



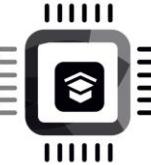
```
fn main() {  
    let ka_in_kannada = '\u{C95}'; //let ka_in_kannada = 'ಕ';  
    let ka_in_hindi = '\u{915}';  
    let ka_in_chinese = '\u{5f00}';  
  
    println!("{}\\n{}\\n{}", ka_in_kannada, ka_in_hindi, ka_in_chinese);  
}
```

Output

ಕ

କ

开

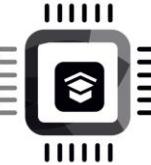


About '\u{}' syntax

In Rust, single quotes are used to specify a character literal and byte literal

A character literal is a single Unicode character, represented by a single code point.

When you use single quotes to enclose the code point '\u{221E}' , you are indicating to the Rust compiler that you want to create a character type variable, which can store a single character.



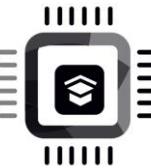
bool

The **bool** type represents a Boolean value, which can be either **true** or **false**. It is a primitive type and is used to represent logical conditions.

bool values can also be used in various other contexts, such as in Boolean operations such as **&&** (and) and **||** (or),

you also can use **bool** with comparison operators like **==**, **!=**, **>**, **<**, **>=**, **<=** etc.

```
fn main() {
    let is_raining = true;
    if is_raining {
        println!("Bring an umbrella!");
    } else {
        println!("Enjoy the sunshine!");
    }
}
```



Array

- ✓ An array is a data structure that holds a fixed number of elements of the same data type.
- ✓ Arrays are fixed-size, meaning the size of an array must be known at compile time and cannot be changed once created

Syntax

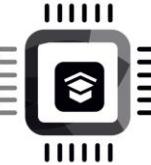
```
let variable_name: [type; size] = [value1, value2, value3, ...];
```

Annotations for the syntax:

- Name of the variable that will hold the array
- Type of the elements in the array
- Number of elements in the array
- Initial values of the array

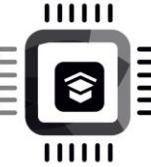
//Array of 3 elements of type i32
let my_array: [i32; 3] = [1, 2, 3];

Type of my_array is
[i32; 3]



Array

- ✓ In Rust, it is not possible to use a variable to specify the size of an array at runtime. The size of an array must be a constant expression known at compile time.
- ✓ Since size of the array cannot be modified at run time, , Rust does have a dynamic alternative to arrays called "Vectors" which can change in size
- ✓ Vectors are implemented as a wrapper around a dynamically allocated array, and provide methods to push and pop elements, as well as other useful functionality. So, in Rust you can use Vector if you want a dynamic array.

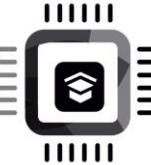


What's the type of this array ?

```
let my_array = [1, 2, 3_u8, 67, 89];
```

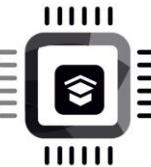
The type of *my_array* would be **[u8; 5]**.

This means that it's an array of unsigned 8-bit integers with a length of 5. The type of the elements in the array is u8, and the size of the array is 5.



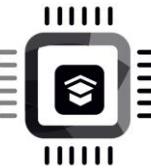
Printing array

```
fn main() {  
    let my_array = [1,2,3];  
  
    /* `[{integer}; 3]` doesn't implement `std::fmt::Display` */  
    println!("{}",my_array); //Error  
  
    /*  
     * { :? } or {:#? } format specifier uses  
     * Debug trait of the arry which is used  
     * to print arrays in a concise and readable format  
     */  
    println!("{:?}",my_array); //OK  
    println!("{:#?}",my_array); //OK  
  
}
```



Why `{:?}` format specifier ?

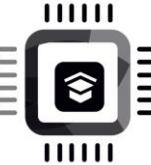
- ✓ The `{}` format specifier is used for formatting display output. However, arrays do not implement the ***Display trait*** by default, which is required for using the `{}` format specifier
- ✓ In Rust, arrays automatically implement the ***Debug trait***. The `:?` format specifier is a shorthand for the ***Debug trait***, so when you use `:?` to print a value, it's equivalent to calling the ***Debug*** implementation for that value.
- ✓ The `{}` format specifier is meant to format values in a way that is suitable for end-user consumption, while the `:?` format specifier is meant to format values in a way that is more suitable for debugging.
- ✓ You can also use the `:?` format specifier to print enums, structs, and vectors, HashMaps in Rust, which allows you to print their values using the `:?` format specifier



Repeat expression

```
/*
 * creating an array and initialize all elements to zero
 * [V, N]. V=value , N= repeat expression
 * V is repeated N times
 */
let array2: [i32;10] = [0;10]; ...

/*
 * Creates an array of type u8 of 4 elements and
 * initialize all emlements to 5
 */
let array3 = [5_u8;4];
```

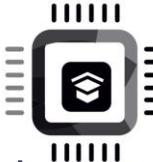


Array indexing

```
let array1 = [5, 4, 3, 2, 1];
let element = array1[3];
println!("{}", element);

/*
 * Cannot modify the array because
 * by default it is immutable
 */
array1[3] = 10; //Error

let mut array2 = [5_u8;5];
array2[4] = 10; //OK
println!("{:?}", array2); // [5, 5, 5, 5, 10]
```

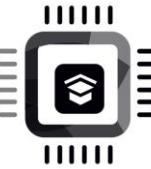


Iterating over an array

```
fn main() {  
    let my_array = [56, 78, -67, 56, 89, 45];  
    let mut sum = 0;  
    for i in my_array {  
        sum = sum + i;  
    }  
    println!("Sum is {}", sum);  
}
```

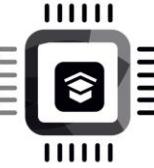
A program which adds all the elements of the given array and prints the result

- ✓ In this code, the **for** loop is used to iterate over the elements in the **my_array** array.
- ✓ The loop variable **i** takes on the value of each element in the **my_array** array in each iteration of the loop.
- ✓ So, in the first iteration, '**i**' will be equal to 56, in the second iteration, '**i**' will be equal to 78, and so on, until all elements have been processed.
- ✓ The type of '**i**' is **i32**, which is the type of the elements in the **my_array** array.
- ✓ The **for** loop automatically infers the type of '**i**' based on the type of the elements in **my_array** or context in which the array being used.



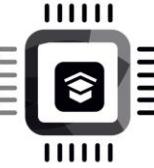
Arithmetic operators in Rust

Operator	Symbol	Description
Addition	+	Adds two numbers.
Subtraction	-	Subtracts the second number from the first.
Multiplication	*	Multiplies two numbers.
Division	/	Divides the first number by the second, truncating towards zero for integers.
Modulus	%	Computes the remainder of division of the first number by the second.

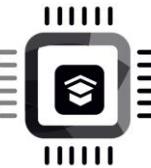


Modulus operator

The modulus operator is frequently used for wrapping around to the start after reaching the end of a range or sequence.



Functions

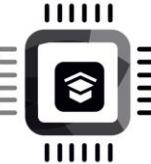


Function definitions

- You can define functions using the **fn** keyword followed by the function name and parameter list. The function body is enclosed in curly braces.

```
fn main() {  
    greet("Ram");  
}  
  
//takes a single parameter name of type &str (a string slice).  
fn greet(name: &str) {  
    println!("Hello, {}!", name);  
}
```

```
fn greet(name: &str) -> String {  
    format!("Hello, {}!", name)  
}  
  
fn main() {  
    let greeting = greet("Ram");  
    println!("{}", greeting);  
}
```

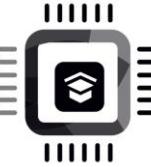


Returning from function

- In Rust, functions can return values using the `return` keyword followed by the value to be returned. However, Rust functions are also expressions, which means that they will return the last expression in the function by default.

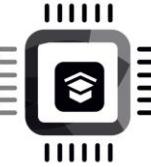
```
fn add(x: i32, y: i32) -> i32 {  
    x + y // this is the last expression in the function, so it will be returned  
}
```

```
fn add(x: i32, y: i32) -> i32 {  
    return x + y; // using return keyword to return value explicitly  
}
```



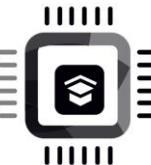
Function prototyping

- You don't need to write a function prototype or forward declaration like you do in C. This is because Rust's type system is designed to handle forward references to functions and other items without needing explicit declarations.
- You can define a function anywhere in your code, and as long as it's in scope where it's called, the compiler will be able to find it and check that its signature matches the call.



Function overloading

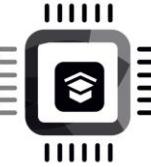
- Rust does not support function overloading. Functions cannot have the same name even if they have different types or a different number of parameters.
- However, functions with the same name can exist in different modules, allowing for some degree of abstraction



```
// Define a function named `add` in module `math`
mod math {
    pub fn add(x: i32, y: i32) -> i32 {
        x + y
    }
}

// Define a function named `add` in module `stats`
mod stats {
    pub fn add(x: f64, y: f64) -> f64 {
        x + y
    }
}

fn main() {
    let x = 5;
    let y = 7;
    let sum = math::add(x, y);
    let a = 3.14;
    let b = 2.71;
    let total = stats::add(a, b);
    println!("Sum of {} and {} is {}", x, y, sum);
    println!("Total of {} and {} is {}", a, b, total);
}
```

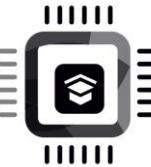


Default parameters

Rust does not support default parameters in function signatures. If a function is defined with certain parameters, those parameters are required when the function is called.

```
fn greet(name: &str, greeting: &str) {
    println!("{} , {}!", greeting, name);
}

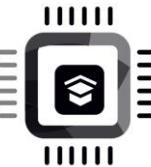
fn main() {
    greet("Alice", "Hello"); // prints "Hello, Alice!"
    greet("Bob", "Hi"); // prints "Hi, Bob!"
    greet("Charlie"); // won't compile, missing argument for parameter 'greeting'
}
```



Type mismatch during function call

- The parameter types of a function must match the argument types that are passed in during a function call.
- If the types do not match, the Rust compiler will give a *type mismatch error*

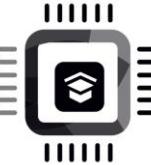
```
fn multiply_by_two(num: i32) -> i32 {  
    num * 2  
}  
  
fn main() {  
    let float_num = 3.14;  
    let result = multiply_by_two(float_num); // error: mismatched types  
    println!("Result: {}", result);  
}
```



```
fn print_array(arr: &[i32]) {  
    for elem in arr.iter() {  
        println!("{} ", elem);  
    }  
}  
  
fn main() {  
    let arr = [1, 2, 3, 4, 5];  
    print_array(&arr); //type of &arr is &[i32; 5]  
}
```

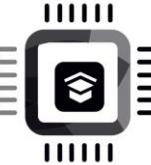
```
fn print_string(string: &str) {  
    println!("{} ", string);  
}  
  
fn main() {  
    let my_string = String::from("Hello, world!");  
    print_string(&my_string);  
}
```

When we pass `&arr` to the `print_array` function in `main`, Rust will automatically coerce the type of `&arr` from `&[i32; 5]` (ref to array of 5, `i32` values) to `&[i32]` (slice of `i32` values), which is a form of implicit type conversion in Rust. So, there won't be a type mismatch error in this case.



'mut' keyword in function parameters

- You can use the ***mut*** keyword in function parameters to indicate that a parameter is mutable. This allows the function to modify the parameter's value.
- It must still adhere to Rust's borrowing rules. Specifically, there can only be one mutable reference to a given piece of data at any given time, and that reference must have exclusive access to the data

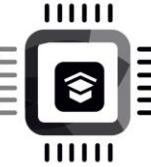


Legal function names

- Function names can only contain alphanumeric characters and underscores and must start with a letter or an underscore.

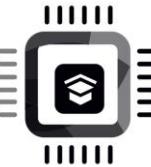
Illegal function names:

- ***1function*** (cannot start with a number)
- ***add-two-numbers*** (cannot use hyphens)
- ***fn*** (cannot use reserved keywords as function names)
- ***my function*** (cannot have spaces)
- ***print_hello_world!*** (cannot use special characters except underscores)



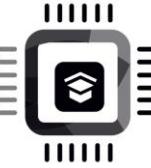
Rust offers built-in testing framework

- Rust includes a built-in testing framework that is part of its standard library.
- This means that you don't need to install any additional packages or libraries to start writing and running tests in a Rust project.
- The testing framework is seamlessly integrated with **Cargo** making it straightforward to use immediately.
- You simply need to write your test cases and run **cargo test**



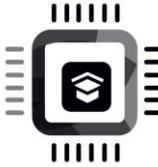
Different types of testing you can perform in Rust

1. **Unit testing:** Tests individual functions, methods, or modules in isolation to ensure that they perform as expected.
2. **Integration Testing:** Tests the integration of multiple components or the entire module as a whole to ensure they work together correctly.
3. **Documentation Testing (Doc-Tests) :** Ensures that code examples in your documentation (doc comments) are accurate and executable.



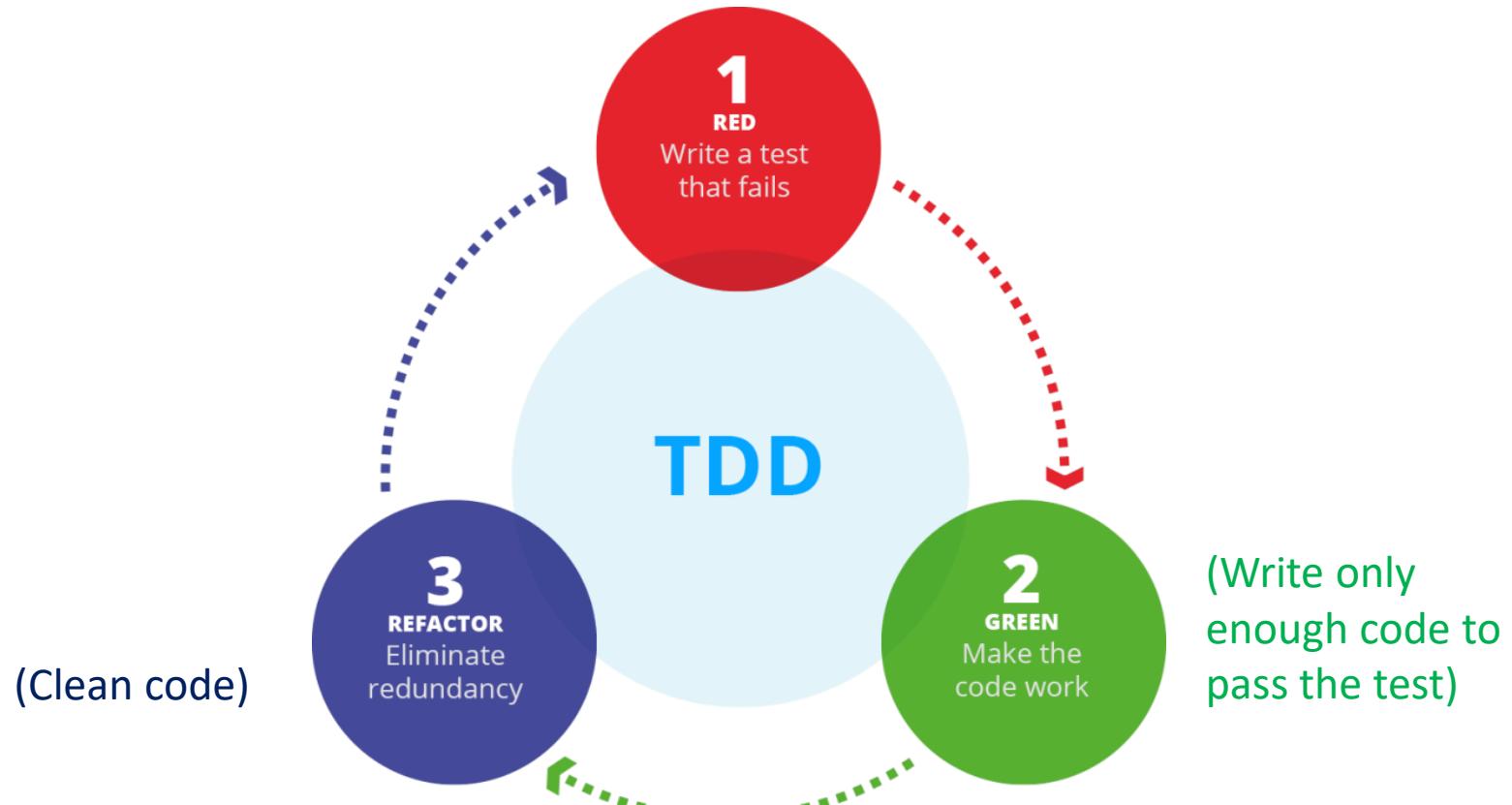
Software development practices

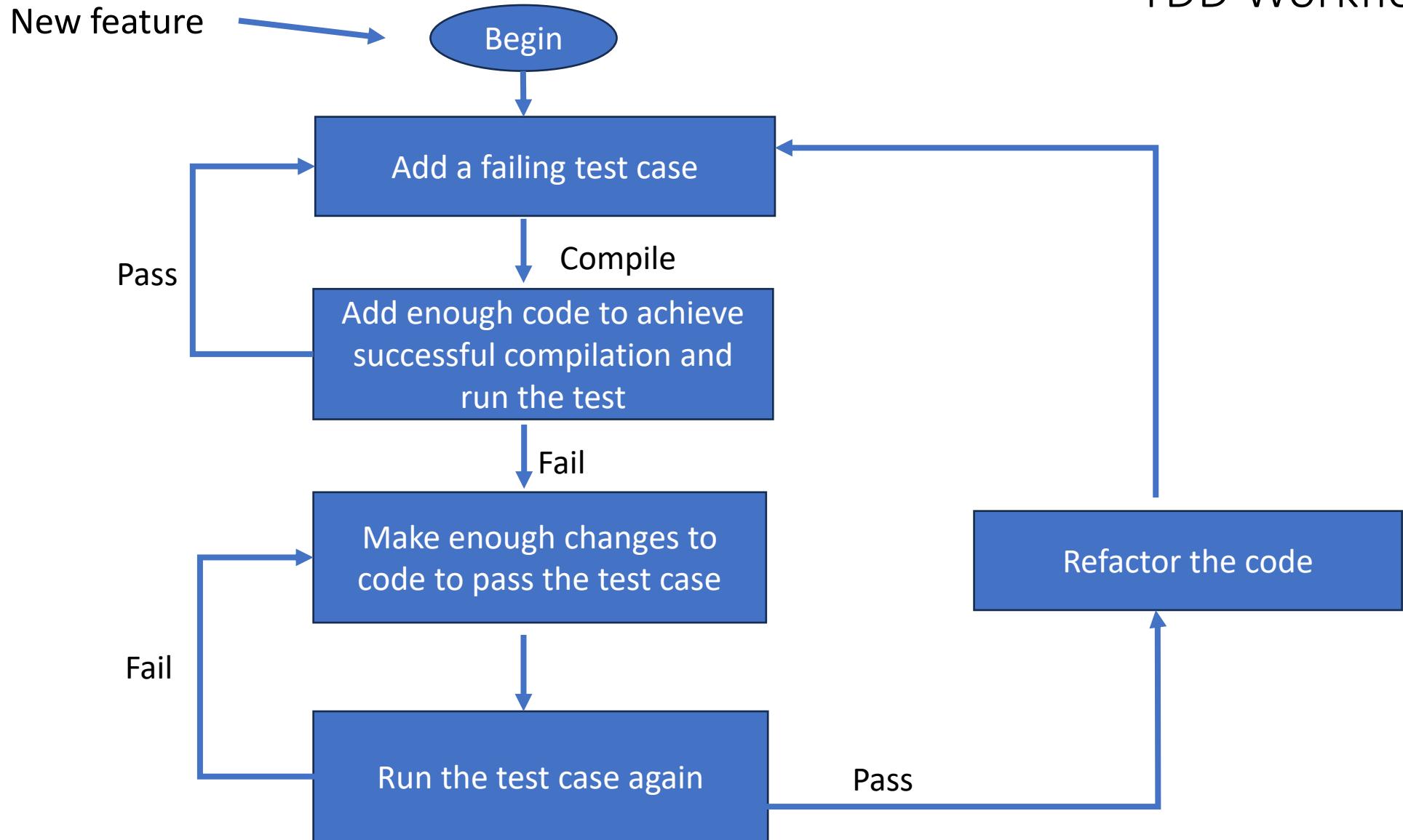
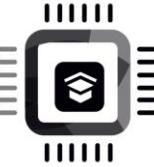
- **Test-Driven Development (TDD)**
- **Test-Last Development (TLD)**

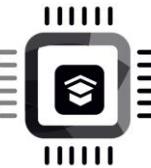


Test-Driven Development (TDD)

Tests drive the development of your software

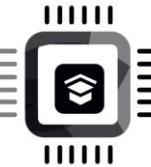






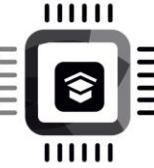
Advantageous of TDD

- **Incremental development**
 - By focusing on one small test at a time, you manage complexity by breaking the software down into manageable and testable pieces.
- **Continuous feedback:**
 - Running tests frequently provides immediate feedback about how the software is developing and highlights any issues early in the process.
- **Refinement and improvement:**
 - Regular refactoring keeps the codebase clean without changing its behavior.



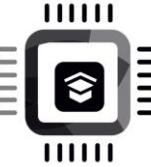
Advantageous of TDD

- **Improved Code Quality:**
 - Tests act as a safety net, catching errors early in the development process.
- **Better Design:**
 - Thinking about tests beforehand encourages a more structured and well-designed codebase.
- **Easier Maintenance:**
 - Existing tests help you understand and modify code without unintended consequences.



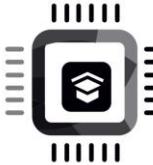
Test-Last Development (TLD)

- Tests are written after the code is developed for a feature



Which one is good?

- The debate on whether TDD (Test Driven Development) is superior to TLD (Test Last Development) is still ongoing and has not been settled yet.
- Many organisations adopt a hybrid approach, using TDD for the most critical parts of their systems where bugs can have severe consequences and using TLD for less critical areas where speed of development is more crucial.

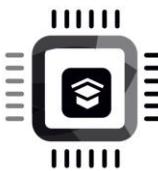


Setting up and organizing test cases

```
① #[cfg(test)]  
② mod tests {  
    //test cases goes here  
}
```

- ① The **#[cfg(test)]** is a conditional compilation attribute that tells the rust compiler to include the annotated module only when compiling the code for testing, not when building for production
- ② **mod tests { }** This declares a module named tests. In rust modules are a way of organizing and encapsulating code.

Writing test cases



```
#[cfg(test)]
mod tests {
    // This function is not a test case;
    // it's a helper function used by the test cases
    fn helper_function() {
    }

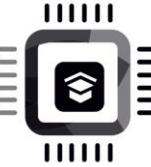
#[test]
fn test_your_test_case_1() {
}

#[test]
fn test_your_test_case_2() {
}

//more test cases
}
```

This attribute tells the Cargo build system to treat this function as a test case.

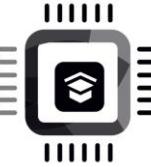
This means that the function will be compiled and executed only when you run the **cargo test** command, but it will not be included in your final application binary.



Rust assert macros

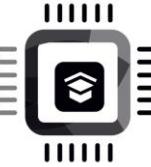
Rust **assert** macros help verify conditions during test execution and ensure that the code behaves as expected

- `assert!(expression);`
- `assert_eq!(left, right);`
- `assert_ne!(left, right);`
- `debug_assert!(expression);`
- `#[should_panic(expected = "panic message")]`



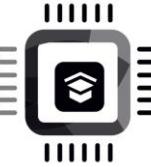
assert!(expression)

- The assert! macro checks a given condition and causes the program to panic if the expression evaluates to **false**.
- It is active in both debug and release builds. This means that no matter how your program is compiled, assert! will always perform the check and will panic if the condition is not met.
- Usage examples : <https://doc.rust-lang.org/std/macro.assert.html>



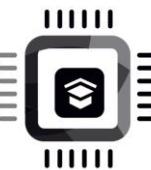
assert_eq!(left, right);

- This macro is used to check whether two expressions evaluate to the same value
- It is commonly used in testing to verify that a function's output matches the expected result
- The macro takes two arguments, typically referred to as **left** and **right**, and it doesn't matter which one is the *expected value* and which is the *actual value*. If the values are not equal, **assert_eq!** will panic



```
#[should_panic(expected = "panic message")]
```

- This attribute is used in testing to assert that a particular piece of code should cause a panic with a specific panic message.
- When annotating a test function with this attribute, you're indicating that the code under test (CUT) is expected to panic and that the test should only pass if the CUT indeed panics with a message that includes the specified text.



```
fn read_from_stdin() -> String {  
    let mut input = String::new();  
  
    println!("Enter the time of the day in seconds");  
    io::stdin()  
        .read_line(&mut input)  
        .expect("Failed to read line");  
  
    return input;  
}  
  
fn parse_string_as_u32(input: String) -> u32 {  
    let total_seconds: u32 = input  
        .trim()  
        .parse()  
        .expect("Input number only without any sign!");  
  
    return total_seconds;  
}
```

Initializes a mutable variable 'input' as an empty *String*

It provides access to the standard input (stdin)

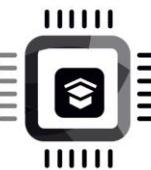
Attempts to read a line from the standard input

If an error occurs (e.g. an I/O error), the program will panic and print the message

Removes any leading or trailing whitespace characters from the string

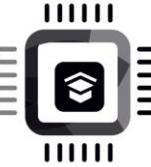
Attempts to parse the string into a *u32*

Cause the program to panic with the provided error message if parsing results in Error



```
1  use std::io;←  
2  
3  /*  
4   Convert seconds to HH:MM:SS format  
5  */  
6  
7  ▶ Run | Debug  
8  fn main() {  
9      let total_seconds: u32 = get_user_input();  
10     println!(  
11         "Time in 24hr format is: {}",  
12         convert_seconds_to_24hr_format(total_seconds)  
13     );  
14 }  
15  
16  
17 fn convert_seconds_to_24hr_format(total_seconds: u32) -> String {  
18     if total_seconds > 86399 {  
19         panic!("Your input should be between 0 to 86,399 ");  
20     }  
21  
22     let hours: u32 = total_seconds / 3600;  
23     let remaining_seconds: u32 = total_seconds % 3600;  
24     let minutes: u32 = remaining_seconds / 60;  
25     let seconds: u32 = remaining_seconds % 60;
```

Brings the *io* module into the scope of your program, allowing you to use *io::stdin()* and other I/O-related functions and types directly.



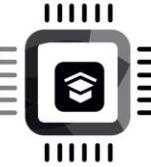
matches!() macro

```
matches!(expression, pattern)
```

pattern against which you want to
check the expression

expression that you want to check

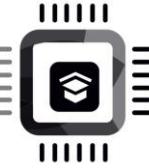
The macro evaluates whether the expression matches the given pattern and returns a boolean value (**true** if it matches, **false** otherwise).



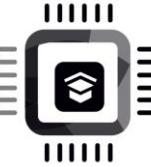
Slice

- ✓ Slice in Rust is used to reference a portion of an array, but they can also be used to reference other types of contiguous data structures, such as a string or a vector. A slice provides a flexible and efficient way to work with subsets of data and is a commonly used data structure in Rust programming.

- ✓ Understanding how references work and how they are used in Rust can give you a better understanding of how slices work and how they can be used to refer to portions of arrays or other data structures



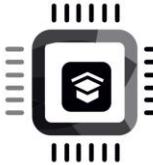
References



Reference in Rust

A reference in Rust can be thought of as a pointer in C, in the sense that it refers to the memory location of a value. However, Rust references are different from C pointers in several important ways.

- ✓ References are immutable by default
- ✓ Cannot be null and will never dangle
- ✓ The borrow checker ensures that the reference will only be valid for the lifetime specified, and will not outlive that lifetime



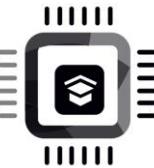
References

```
fn main() {
    let value: i32 = 42;
    let ref_of_value = &value;
    println!("Value is {}", *ref_of_value); //Manual dereferencing
    println!("Value is {}", ref_of_value); //automatic dereferencing
}
```

- Here, the variable ***ref_of_value*** is a reference to the variable ***value***
- Type of ***value*** is ***i32***
- Type of ***ref_of_value*** is ***&i32***

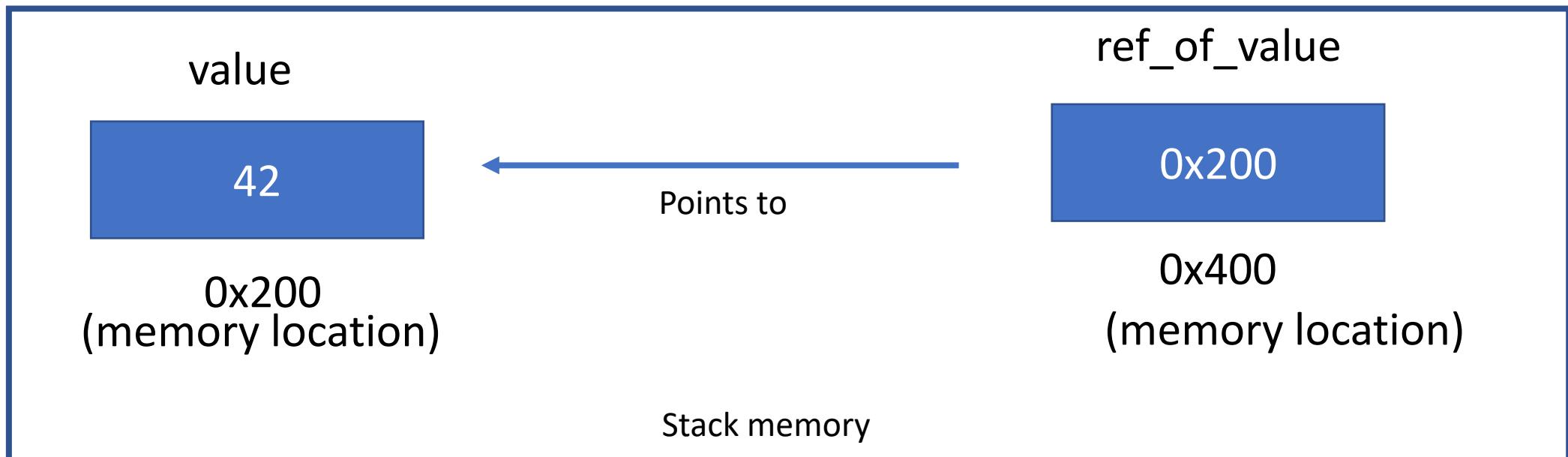
```
int main()
{
    const int value = 42;
    const int *const pointer_to_value = &value;
    printf("value = %d\n", *pointer_to_value);
}
```

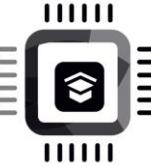
‘C’ equivalent



Memory representation

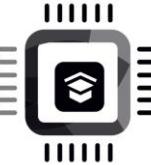
```
fn main() {  
    let value: i32 = 42;  
    let ref_of_value = &value;  
    println!("Value is {}", *ref_of_value); //Manual dereferencing  
    println!("Value is {}", ref_of_value); //automatic dereferencing  
}
```





Why does `println!("{}", ref_of_value)` print 42 instead of the memory address of the variable 'value'?

Display for references is implemented to print the referenced value, not the address. You need `{:p}` instead of `{}` for that.

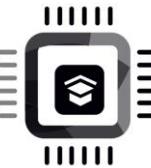


```
fn print_value(val: i32) {
    println!("Value is: {}", val);
}

fn main() {
    let value = 32;
    let ref_of_value = &value;
    ...

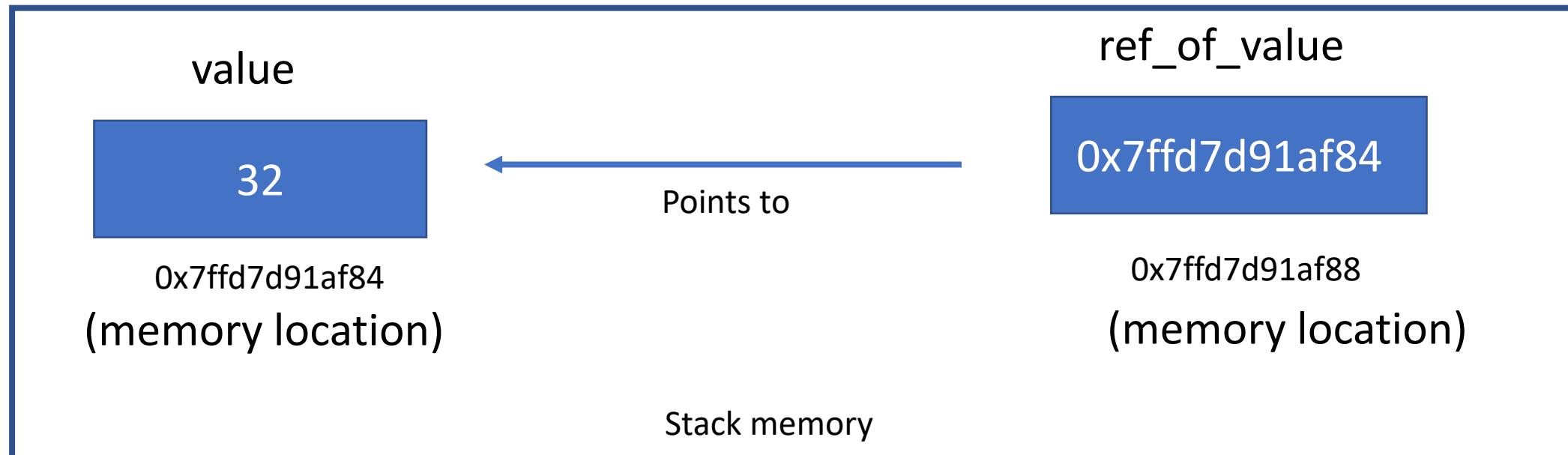
    let tmp_ref: i32 = ref_of_value; //Error cannot assign &i32 to i32
    let tmp_ref: i32 = *ref_of_value; //OK
    ...

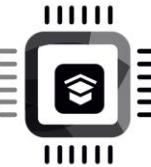
    print_value(ref_value); //Error print_value() expects i32
    print_value(*ref_value); //OK
}
```



Printing address of the variable

```
fn main() {  
    let value = 32;  
    let ref_of_value = &value;  
    println!("{}" , ref_of_value); //32  
    println!("{}" , *ref_of_value); //32  
    println!("{:p}" , ref_of_value); //0x7ffd7d91af84 (address of 'value')  
}
```

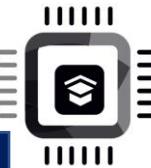




Borrow, Borrower, Referent

```
Referent →  
let value = 32;  
let ref_of_value = &value;  
Borrower → (Also called reference)  
This is borrow operation
```

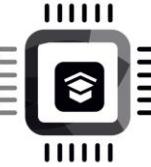
- ✓ When you create a reference to a value using the `&` operator, it is called a **borrow**.
- ✓ The variable that holds the reference is called the borrower. In this case, the borrower is `ref_of_value`
- ✓ The referent is the value or variable that the reference points to. In this case, the referent is variable `value`
- ✓ A borrow allows the borrower to access the value stored in the referent without taking ownership of it.



Immutable reference

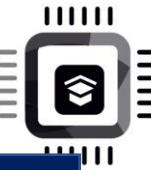
```
fn main() {  
    ...  
    // immutable data  
    let num = 32;  
    ...  
    //  
    // Here, 'ref_of_num' is an immutable reference.  
    // because it holds the reference to immutable referent  
    //  
    let ref_of_num = &num; //immutable borrow  
    ...  
    //Error. cannot dereference immutable reference  
    *ref_of_num = 100;  
    ...  
    println!("{}" ,ref_of_num); //32  
}
```

By using immutable reference you can only read the value of the referent but cannot modify it.



Mutable borrow of the immutable referent

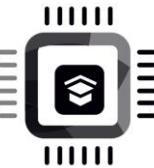
```
fn main() {  
    ...  
    ... //immutable data  
    let num = 32;  
    ...  
    ... //  
    ... // Error.  
    ... // Cannot perform mutable borrow of immutable referent  
    ... //  
    let ref_of_num = &mut num; // 'num' cannot be borrowed as mutable.  
    ...  
    ...  
    println!("{}" ,ref_of_num); //32  
}
```



Multiple immutable borrows

```
//  
//This function too borrows 'num' as readonly.  
  
fn print_value(ref3: &i32) {  
    println!("{} ", ref3);  
}  
  
  
fn main() {  
    let num = 32;  
    let ref1 = &num; //OK.1st borrow  
    let ref2 = &num; //OK.2nd borrow  
    print_value(&num); //OK.3rd borrow  
}
```

This example illustrates that you can have multiple immutable references(borrows) to the same value, allowing multiple parts of the code to read the data simultaneously without the risk of data races or other undefined behavior



Reference to mutable data

Mutable borrow

Immutable borrow



```

fn print_value(value: &i32) {
    println!("{}", value);
}

fn main() {
    //Mutable data
    let mut num = 32;

    //OK. Mutable Borrow
    let ref1 = &mut num;
    type of 'ref1' is &mut i32

    //
    //Error
    //cannot borrow `num` as immutable because
    //it is also borrowed as mutable
    //

    print_value(&num);
    let ref2 = &num;

    //
    //Error.
    //cannot borrow `num` as mutable more than once at a time
    //

    let ref3 = &mut num;

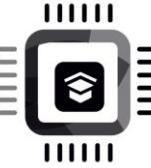
    //

    *ref1 = 50; //OK
    print_value(ref1); //50
}

```

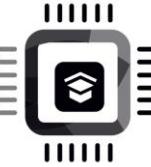
- ✓ When you create a mutable reference to a value using the **&mut** operator, you can both read and modify the data stored in the referent.
- ✓ However, the borrow checker in Rust enforces the "single write or multiple read" policy, which means that you can only have one unique mutable reference to a value at a time.
- ✓ This prevents data races and other undefined behavior that could occur if multiple parts of the code were able to modify the same data simultaneously.

Rust enforces “single write or multiple read” policy



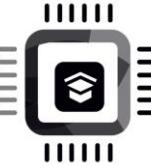
Single write or multiple read

In Rust, you cannot have a mutable borrow and an immutable borrow of the same variable at the same time. This is to ensure that the variable's state is not changed unexpectedly. Once a variable is borrowed mutably, any subsequent attempts to borrow it immutably will cause a compile-time error, and vice versa.



Slices

- ✓ Slices in Rust are used to reference a portion of an array, but they can also be used to reference other types of contiguous data structures, such as a string or a vector. A slice provides a flexible and efficient way to work with subsets of data and is a commonly used data structure in Rust programming.
- ✓ Understanding how references work and how they are used in Rust can give you a better understanding of how slices work and how they can be used to refer to portions of arrays or other data structures



Slice of an Array

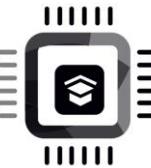
A slice in Rust is a reference to a contiguous region of an array. It is represented by the `&[T]` type, where `T` is the type of elements in the array.

A slice allows you to borrow a portion of an array without taking ownership of the entire array and provides a way to work with subarrays in a safe and efficient manner without having to make copies.

You can create a slice from an array using `&` operator, for example ,

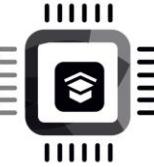
```
let slice = &array[1..3];
```

this will create a slice of the array from index 1 to 2 ([2, 3])



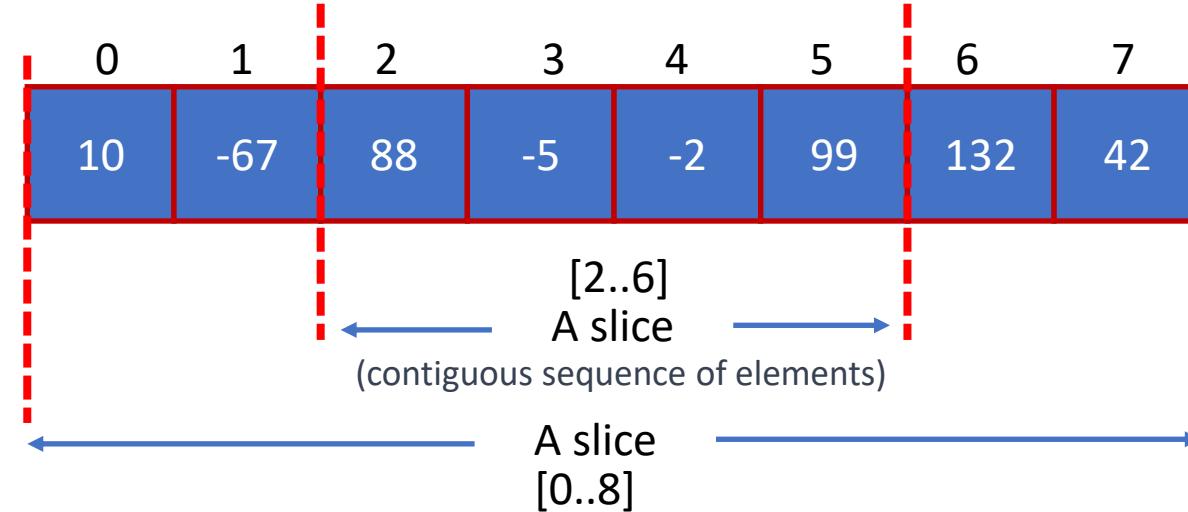
Slice

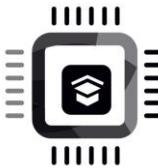
- When you create a slice, you specify a range of indices that determine which elements of the original array are included in the slice. The slice itself is a reference to this sequence of elements, and you can use it to read and modify the elements in the slice.
- Because a slice is just a reference to the original array, modifying the elements in the slice will also modify the corresponding elements in the original array. Similarly, any changes made to the original array will also be reflected in the slice.
- Note that slices can also be created from vectors and other collections in Rust, not just arrays.



```
let array1 = [10,-67,88,-5,-2,99,132,42];
```

Type of array1 → [i32; 8]





Slice example

How to create a slice ?

We can create slices using a range within brackets by specifying **[starting_index..ending_index]**, where **starting_index** is the first position in the slice and **ending_index** is one more than the last position in the slice.

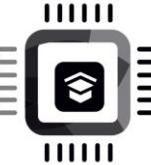
Internally, the slice data structure stores the starting position and the length of the slice, which corresponds to ending_index minus starting_index

Syntax :

```
let slice = &array[start_index..end_index];
```

You can create a slice of an array by using the & operator and specifying the start and end indices of the slice

'..' operator is called the **range** operator or exclusive range operator. It is used to create a range that includes the start value but excludes the end value

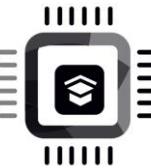


Range expression (..)

'..' is often referred to as the "range expression operator" in Rust.

It creates a range of values, and is used in various contexts to specify a sequence of values, such as for slicing arrays, creating for loops, and match patterns.

Production	Syntax	Type	Range
<i>RangeExpr</i>	start .. end	std::ops::Range	$\text{start} \leq x < \text{end}$
<i>RangeFromExpr</i>	start ..	std::ops::RangeFrom	$\text{start} \leq x$
<i>RangeToExpr</i>	.. end	std::ops::RangeTo	$x < \text{end}$
<i>RangeFullExpr</i>	..	std::ops::RangeFull	-
<i>RangeInclusiveExpr</i>	start ..= end	std::ops::RangeInclusive	$\text{start} \leq x \leq \text{end}$
<i>RangeToInclusiveExpr</i>	..= end	std::ops::RangeToInclusive	$x \leq \text{end}$



Examples of Slice of an array

```
let my_array = [10, -67, 88, -5, -2, 99, 132, 42];

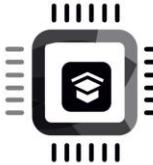
//refers to the array from start_index=1 to end_index = 5(exclusive
//type of slice1 is &[i32]
let slice1 = &my_array[1..5];
println!("{:?}", slice1); //Output : [-67, 88, -5, -2]

let slice2 = &my_array[..]; //refers to the whole array
println!("{:?}", slice2); //Output : [10, -67, 88, -5, -2, 99, 132, 42]

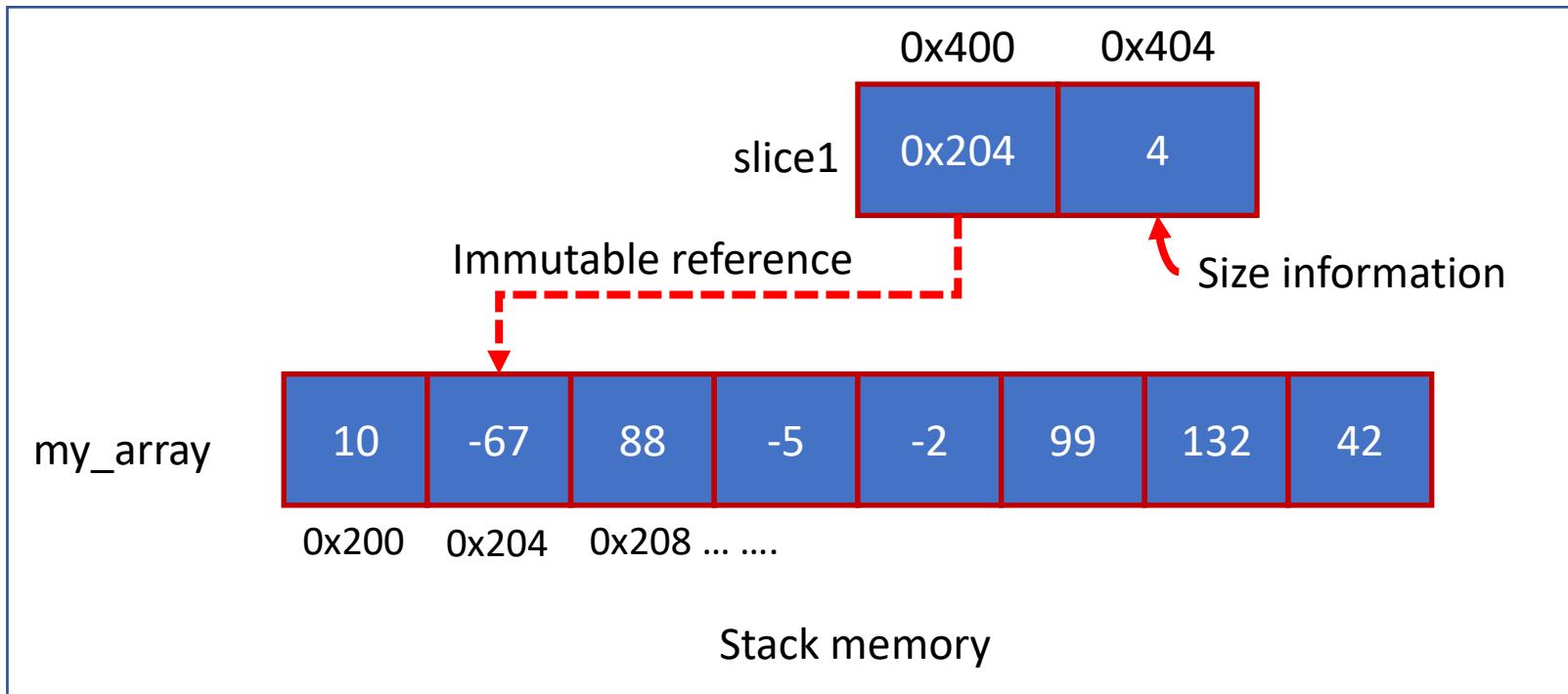
let slice3 = &my_array[0..]; //refers to the whole array
println!("{:?}", slice3); //Output : [10, -67, 88, -5, -2, 99, 132, 42]

//refers to the array from start_index=5 to end of the array
let slice4 = &my_array[5..];
println!("{:?}", slice4); //Output : [99, 132, 42]

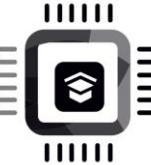
//refers to the array from start_index=0 to end_index = 2(inclusive)
let slice5 = &my_array[..=2];
slice[0] = 77; //Error. my_array is not mutable
println!("{:?}", slice5);
```



```
let my_array = [10,-67,88,-5,-2,99,132,42];
//refers to the array from start_index=1 to end_index = 5(exclusive)
//type of slice1 is &[i32]
let slice1 = &my_array[1..5];
```



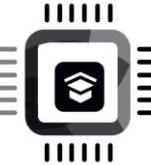
0x200 is memory location(address) where first element of the `my_array` array is found
0x400 is the memory location of the `slice1` where '**reference**' information is stored
0x404 is the memory location of the `slice1` where '**size**' information is stored



```
let mut my_array = [10, -67, 88, -5, -2, 99, 132, 42];
let len = my_array.len();

//type of slice6 is &mut [i32]
let slice6 = &mut my_array[3..len];
slice6[0] = 77;

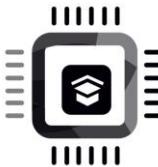
println!("slice6:{:?}", slice6); //slice6:[77, -2, 99, 132, 42]
println!("Array:{:?}", my_array); //Array:[10, -67, 88, 77, -2, 99, 132, 42]
```



Iterating over the slice of an array

Write a program that creates a slice of size 4 from a given array, computes the sum of all the elements in the slice, and prints the result.

```
let my_array = [10,-67,88,-5,-2,99,132,42];
```

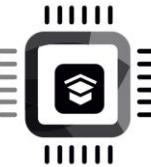


Iterating over the slice

```
fn main() {  
    let my_array = [10, -67, 88, -5, -2, 99, 132, 42];  
    let slice = &my_array[..=3];  
    let mut sum = 0;  
  
    for i in slice {  
        sum = sum + i;  
    }  
  
    println!("sum {}", sum);  
}
```

Note

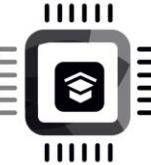
1. The 'slice' variable is a reference to a slice of integers
2. For each iteration of the '**for**' loop, the loop variable '*i*' is assigned a different reference to an element in the slice
3. Therefore, the type of '*i*' is a reference to an integer, written as '**&i32**'
4. When iterating over a slice, the loop variable takes on the value of each reference in the slice, rather than the value of each element in the slice. This is why '*i*' has the type '**&i32**'



In `sum = sum + i;` ‘*i*’ is of type `&i32` .

How did this work to compute `i32` value ?

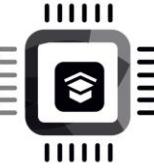
- When you write `sum + i`, Rust checks whether there is an implementation of the **Add trait** for the types of `sum` and `i`. In this case, `sum` is of type `i32` and `i` is of type `&i32`, so Rust checks for an implementation of `Add<i32, &i32>`.
- This trait is implemented for the `(i32, &i32)` pair in the standard library, which allows the addition to be performed by converting the reference to a value of type `i32` and adding the two values together.



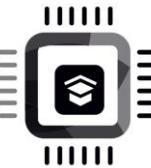
Std. library functions of the slice type

Write a program to ,

- Reverse an array
- Sort an array
- Find the biggest element in an array
- Concatenate two given arrays
- Split a given array at a specified index
- Check if a given value is contained within an array



Decision making in Rust

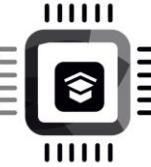


- If ..else
- If ..else if..else
- If..let
- match

In Rust, there are two main decision-making statements:

1.*if* statement: Used for conditional execution of code.

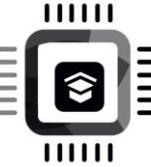
2.*match* statement: Used for pattern matching, often as an alternative to multiple if statement



The 'if and else' statement

```
if · expression · {  
    ... //Code · to · execute · if · expression · i · true  
} · else · {  
    ... //Code · to · execute · if · expression · is · false  
}
```

The 'expression' in an ***if*** statement must evaluate to a boolean value, either ***true*** or ***false***.

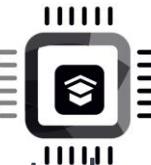


Expression vs Statement

Expressions evaluate to a value, while statements do not.

In Rust, both ***if*** and ***match*** can be used in a way that they return a value, making them expressions.

In Rust, ***if*** and ***match*** can be used as expressions to return values.

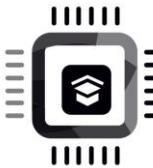


'If else' used as an expression

```
fn main() {  
    let age = 10;  
  
    let message = if age < 18 {  
        "You cannot vote" // Note: no ; here  
    } else {  
        "You can vote" // Note: no ; here  
    }; // Note: ; here  
  
    println!("{}" , message);  
}
```

In Rust, expressions that are used as the final value in a code block do not require semicolons and the value of the block is the value of its last expression.

- The first branch of the ***if*** expression is executed and the string "You cannot vote" is returned.
- If the expression is false, the second branch of the ***if*** expression is executed and the string "You can vote" is returned.
- It's important to note that in Rust, ***if*** and ***else*** block expressions must produce a value of the same type.
- ***if*** block expression without ***else*** block evaluate to `()`

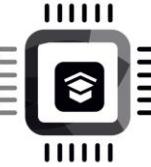


```
fn main() {
    let age = 10;

    let message = if age < 18 {
        "You cannot vote"; // returns ()
    } else {
        "You can vote"; // returns ()
    }; // Note: ; here

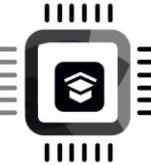
    println!("{}:{}", message);
}
```

- If you add a semicolon after an expression that returns a value, you will turn that expression into a statement. When an expression is used as a statement, it doesn't return a value and the result of the expression is discarded.
- For example, a semicolon after “**You cannot vote**”, the value returned by the **if** block will be discarded, and result will have the type **() (unit)**, which represents the absence of a value. **()** is returned if there is no other meaningful value that could be returned.



The 'unit' type (())

- () is a special value in Rust that represents the absence of a value.
- It's commonly referred to as the "unit type" or "unit value". The unit type () is used in several places in Rust to indicate that a function or expression doesn't return a meaningful value.
- <https://doc.rust-lang.org/std/primitive.unit.html>



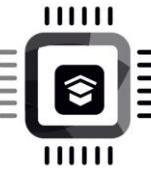
match

Syntax

```
match value {  
    pattern => expression ,  
    pattern => expression ,  
    ...  
    _ => expression,  
}
```

- **Value** is matched against different patterns.
- If the **value** matches the **pattern**, then the code on the right-hand side of => is executed.
- **_** is a catch-all pattern that matches any **value** and is used as a default case.

The **match** statement is used to perform a match against a value and execute different code based on which pattern matches the value. If no patterns match the value, the catch-all pattern (**_**) is executed.

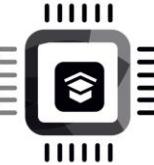


'match' statement is an exhaustive match by default

```
fn main() {  
    let x = 5;  
  
    match x {  
        1 => println!("one"),  
        2 => println!("two"),  
    }  
}
```

```
fn main() {  
    let x = 5;  
  
    match x {  
        1 => println!("one"),  
        2 => println!("two"),  
        _ => println!("other"),  
    }  
}
```

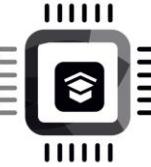
In Rust, the **match** statement is an exhaustive match by default. This means that if you have a **match** expression with multiple cases, you must handle all possible cases explicitly, or else the Rust compiler will produce a compile-time error



Example

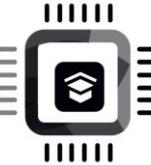
Use a ***match*** statement to check the values of the tuple against several different patterns, each of which corresponds to a specific action to take.

- If the second item in the tuple is negative, the code prints a message indicating that the second item is negative and takes "action 1".
- If both items in the tuple are zero, the code prints a message indicating that both items are zero and takes "action 2".
- If the tuple does not match either of these patterns, the code prints a message indicating that "all fine".



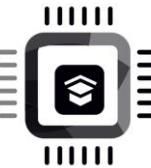
Example

Determine if an array contains negative numbers and update the flag variable 'invalid_array' accordingly."



| symbol in match expression

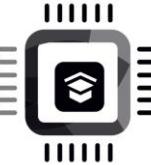
- Commonly referred to as a "pipe" or a "vertical bar"
- The | symbol is used to combine multiple patterns into a single match arm that should execute the same code. This allows you to group together patterns that have the same outcome and write more concise and readable code



if-else if-else

```
if · expression1 · {  
    ... // code block to execute if expression1 is true  
} · else · if · expression2 · {  
    ... // code block to execute if expression1 is false and expression2 is true  
} · else · {  
    ... // code block to execute if all conditions are false  
}
```

- The final else branch in an **if-else if-else** chain is not mandatory.
- It's generally a good practice to include a final **else** block in an **if-else if-else** statement, even if it contains no code. This ensures that all possible cases are covered and makes the code more explicit and easier to understand.

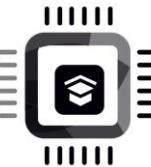


if let

Syntax

```
if let pattern = expression {  
    ... // code to execute if expression matches the pattern  
} else {  
    ... // code to execute if expression does not match the pattern  
}
```

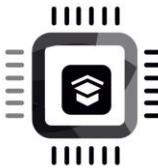
if let statement is a shorthand for a
match statement with just one
pattern



```
fn main() {  
    let point = (0, 0);  
  
    //Using if let  
    if let (0, 0) = point {  
        println!("point is zero");  
        return;  
    }  
  
    //Using match  
    match point {  
        (0, 0) => {  
            println!("Point is zero");  
            return;  
        }  
  
        _ => (),  
    }  
}
```

pattern that is matched against the result of the expression.

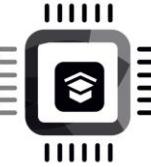
if let statement is a shorthand for a match statement with just one pattern



Comparison operators in Rust

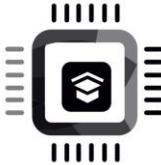
- Comparison operators are binary operators, meaning they typically take two operands
- These operators compare the two operands and return a boolean value based on the result of the comparison.

Operator	Name	Description	Traits
<code>==</code>	Equal to	Checks if values are equal.	PartialEq
<code>!=</code>	Not equal to	Checks if values are not equal.	PartialEq
<code>></code>	Greater than	Checks if left value is greater than the right value.	PartialOrd
<code><</code>	Less than	Checks if left value is less than the right value.	PartialOrd
<code>>=</code>	Greater than or equal to	Checks if left value is greater than or equal to the right value.	PartialOrd
<code><=</code>	Less than or equal to	Checks if left value is less than or equal to the right value.	PartialOrd



Traits that govern comparison operators

1. **PartialEq** trait is what governs the == (equality) and != (not equality) operators.
2. When you implement **PartialEq**, you enable these operators to be used with your custom data types.
3. **PartialOrd** trait governs the less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) comparison operators



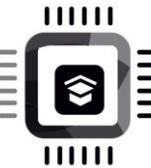
Logical operators in Rust

Operator	Name and description
!	Logical NOT inverts the boolean value.
&&	Logical AND true if both operands are true.
	Logical OR true if at least one operand is true.

A	!A
false	true
true	false

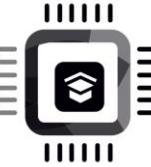
A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

A	B	A B
false	false	false
false	true	true
true	false	true
true	true	true



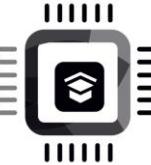
Logical operators in Rust

- Logical operators `&&` and `||` are binary operators, meaning they take two operands
- `!` (Logical Not) is a unary operator, meaning it operates on a single operator
- The operands must strictly be of boolean type. i.e., either `true` or `false`.
- logical operators are not governed by any traits. This is because logical operators operate directly on boolean values (`bool`), and their behaviour is fixed and cannot be overloaded for other types.



Bitwise operators in Rust

Operator	Operation	Associated Trait
!	Bitwise Complement	Not
&	Bitwise AND	BitAnd
	Bitwise OR	BitOr
<<	Left shift	Shl
>>	Right shift	Shr
^	Bitwise exclusive OR	BitXor



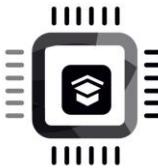
! acts as Bitwise NOT

```
let x = 0b0000_1111_u8;          // In binary: 00001111
let y = !x;                      // Results in 11110000 (inverted bits)
```

! acts as Logical NOT

```
let a = true;
let b = !a; // Results in false
```

Rust does not have the `~` operator for bitwise NOT as in C



Bitwise AND and Bitwise OR

```
let a = 0xFF;  
let b = 0xF;  
let c = a & b;  
println!("{}"); //15
```

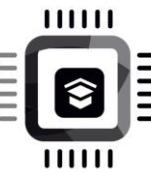
a b0000 0000 0000 0000 0000 1111 1111
 &
b b0000 0000 0000 0000 0000 0000 1111

a & b b0000 0000 0000 0000 0000 0000 1111

```
let a = 0x1A;  
let b = 0xF;  
let c = a | b;  
println!("{}"); //31
```

a b0000 0000 0000 0000 0000 0000 0001 1010
 |
b b0000 0000 0000 0000 0000 0000 0000 1111

a | b b0000 0000 0000 0000 0000 0000 0001 1111



Bitwise Left Shift

```
let c = a << b;
```

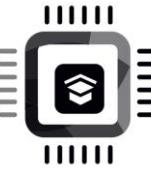
Shifts all bits of the operand(a) to the left by the specified number of positions(b). Zeros are filled in on the right.

0xC0ABCDEF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	1	1	0	0	0	0	0	0	1	0	1	0	1	0	1	1	1	1	0	0	1	1	0	1	1	1	0	1	1	1

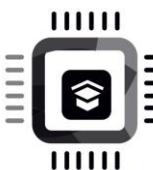
0x02AF37BC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	0	1	0	1	0	1	1	1	1	1	0	0	1	1	0	1	1	1	1	0	1	1	1	0	0



Bitwise Right Shift (>>)

- Shifts all bits of the operand to the right by the specified number of positions. For unsigned numbers, zeros are filled in on the left. For signed numbers, the sign bit (most significant bit) is replicated.
- The right shift operation is an **arithmetic shift** for signed integers, which means it preserves the sign of the integer by filling shifted-in bits on the left with the sign bit.



0x00ABCDEF

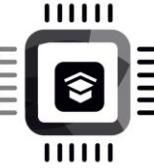
Masking using Bitwise AND

0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0

The diagram shows four groups of eight orange rectangular boxes each, representing memory locations. A red dotted line encloses the first group. Red arrows point from the fourth box of the first group to the fourth box of the second group, and from the fourth box of the second group to the fourth box of the third group. The fourth box of the third group has a red arrow pointing to it from the fourth box of the fourth group. The fourth box of the fourth group is labeled with the text ">> by 4 position".

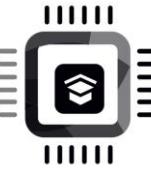
0 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 0 = 0 0 0 0 0 0 0 0 + 1 1 0 1 1 1 1 0

Ans: 0x000000DE



Exercise

- Create an array of integers and use a for loop to iterate over its elements. Inside the loop, use a match statement to check the value of each element in the array: if the value is even, print "even"; if the value is odd, print "odd"; and if the value is negative, print "negative".



Strings in Rust

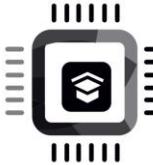
Different Types of Strings in Rust:

1. String literal (&str): A string slice that points to a sequence of UTF-8 encoded characters in the binary and is often created from a string literal in the source code. It has a fixed size.

```
let message = "Good Morning"; //This is a string literal. The text information "Good  
Morning" is hardcoded in the binary of the program.
```

2. String: An owned, heap-allocated string type, which provides methods for mutating its contents and has a dynamic size.

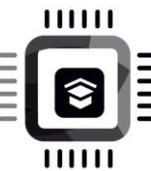
String literal



```
fn main() {  
    ... // type of 'greeting' is &str  
    ... // the string literal "Hello World" cannot be modified  
    let greeting = "Hello World";  
    println!("{}", greeting);  
}
```

String literals are always immutable, meaning they are fixed during compile time.

In the code, the string literal 'Hello World' is non-mutable, which means you cannot modify it, increase or decrease its length using a mutable reference

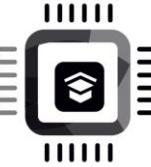


The String data type

If you want to create a mutable string that you can modify during run time of the program, use the “String” type instead of string literal

```
fn main() {  
    //  
    // create a new mutable string  
    // type of the variable 'greeting' is String  
    //  
    let mut greeting = String::from("Good morning");  
  
    // replace "morning" with "evening"  
    greeting.replace_range(5.., "evening");  
  
    // append ", world!" to the string  
    greeting.push_str(", world!");  
  
    println!("{}", greeting); // prints "Good evening, world!"  
}
```

The **String** type is used when the length of the string is not known at compile time or when it needs to be modified during runtime, such as taking and processing user inputs

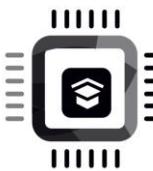


Creating a String variable

Lets discuss 2 important ways by which you can create a ***String*** variable in your program

- 1) Creating an empty ***String*** variable
- 2) Creating a ***String*** variable using string literal

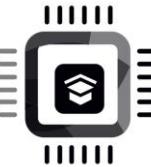
Creating an empty new String



```
let my_string = String::new();
```

- ‘my_string’ is String variable or instance of String
- new() is one of the method of the String type in Rust
- The String type(which is a struct) is defined in the ***std::string::String*** module, which is part of the ***std*** crate
- When you call String::new(), it creates a new instance of the String struct with an empty buffer on the heap.
- The buffer is initially allocated with a capacity of 0 bytes, but it can be dynamically grown as you add characters to the string.
- The String struct also keeps track of the length of the string, which is the number of valid bytes in the buffer, and the capacity of the buffer, which is the total number of bytes that have been allocated for the buffer.

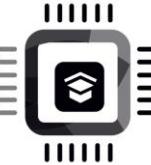
Explore more: <https://doc.rust-lang.org/std/string/struct.String.html>



Creating String using String literal

```
let my_string = String::from("Hello, world!");  
  
//This is equivalent to using the 'from' method.  
let my_string = "Hello, world!".to_string();
```

from is a function defined in the ***From*** trait which allows conversion between types. The ***String::from()*** function is implemented using this trait.



to_string()

Trait std::string::ToString

1.0.0 · [source](#) · [-]

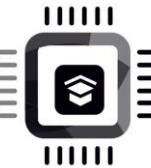
```
pub trait ToString {  
    fn to_string(&self) -> String;  
}
```

[-] A trait for converting a value to a `String`.

This trait is automatically implemented for any type which implements the `Display` trait. As such, `ToString` shouldn't be implemented directly: `Display` should be implemented instead, and you get the `ToString` implementation for free.

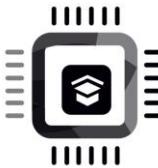
```
fn main() {  
    let mut num_as_string = 3.1416.to_string();  
    num_as_string.insert_str(0, "PI = ");  
    println!("{}" ,num_as_string); //Output: PI = 3.1416  
}
```

Source : https://doc.rust-lang.org/std/string/trait.ToString.html#tymethod.to_string



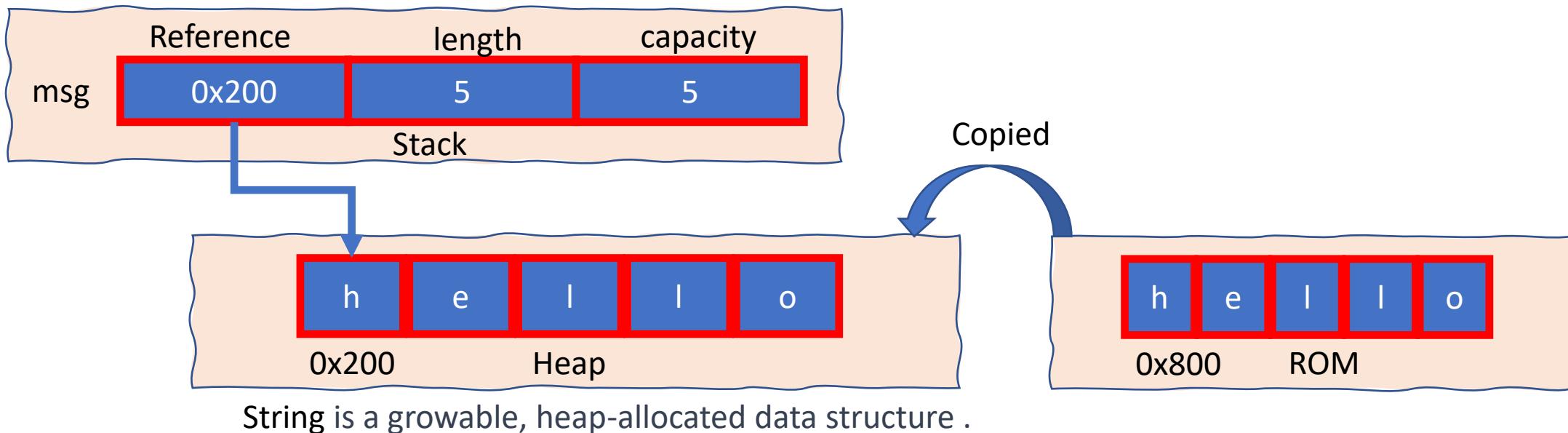
:: operator

- In Rust, the double colon :: is used to access an associated function or a static method of a module, struct, or enum. In the case of ***String::new()***, it means that we are accessing the new() associated function of the ***String*** struct defined in the ***std::string*** module
- The :: syntax is used to call a static method or a function that is associated with a module or type, without having an instance of that module or type.
- In the case of ***String::new()***, ***new()*** is a static method associated with the ***String*** type in the ***std::string*** module. Since we don't have an instance of the ***String*** type yet, we use the :: syntax to call the ***new()*** method directly on the ***String*** type itself. However, to access methods of an instance of the type, you use the dot (.) operator.

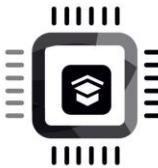


Memory representation

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



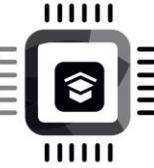
- **Length** refers to the number of bytes in the string's current contents.
- **Capacity** refers to the number of bytes that the string can hold without reallocating memory(total number of bytes received from the OS)



Heap clean up when string goes out of scope

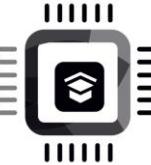
```
fn main() {  
    ...  
    {  
        let msg = String::from("Hello world");  
        // Heap memory allocated to hold the string will be deallocated automatically  
        ...  
        //variable 'msg' is no longer valid here  
        ...  
        println!("{}" ,msg); //Error  
    }  
}
```

- When ‘msg’ goes out of scope, Rust automatically calls the *drop* function, which deallocates the heap memory used by the string.
- Rust does not rely on a garbage collector or a runtime system to manage memory allocation and deallocation. Instead, Rust uses a combination of ownership, borrowing, and lifetimes to ensure memory safety at compile time.
- The Rust compiler inserts code to call the *drop* method automatically for any value that goes out of scope which needs cleaning up, such as String values.
- The drop method is defined in the Rust standard library's Drop trait.



String copy

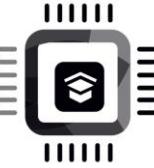
- 1) Shallow copy
- 2) Deep copy



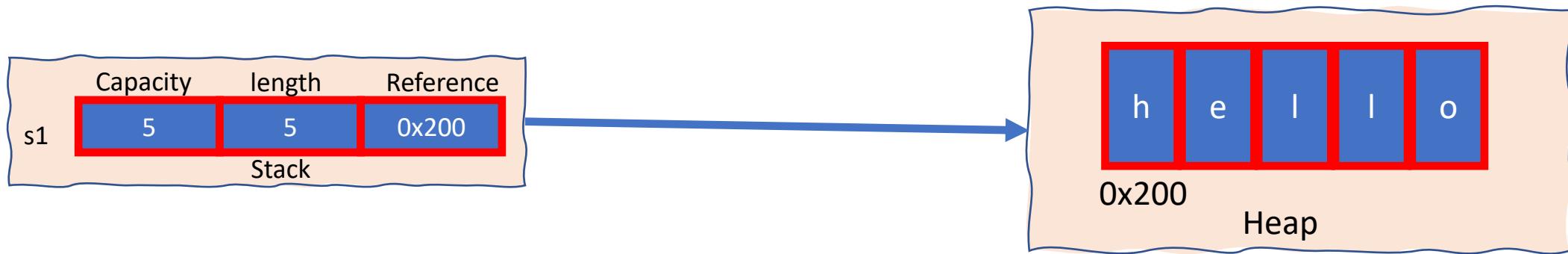
Shallow copy

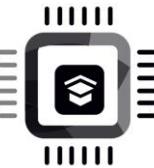
When a String value is assigned to another variable, the variable is assigned a copy of the pointer, length, and capacity of the original String value, but the underlying heap memory is not copied. This is known as a shallow copy or a copy by reference.

```
fn main() {
    let s1 = String::from("hello");
    ...
    let s2 = s1; // s1 is now invalid
    println!("{} world!", s2); // s2 points to the same memory as s1
}
```

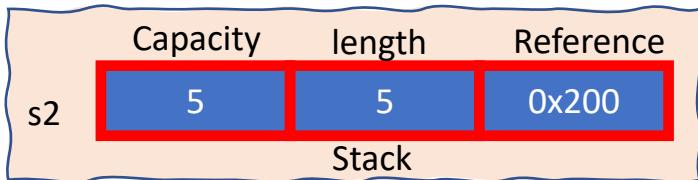
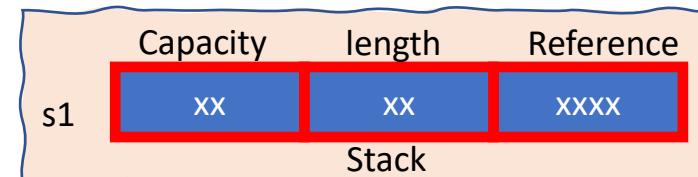


```
let s1 = String::from("hello");
```

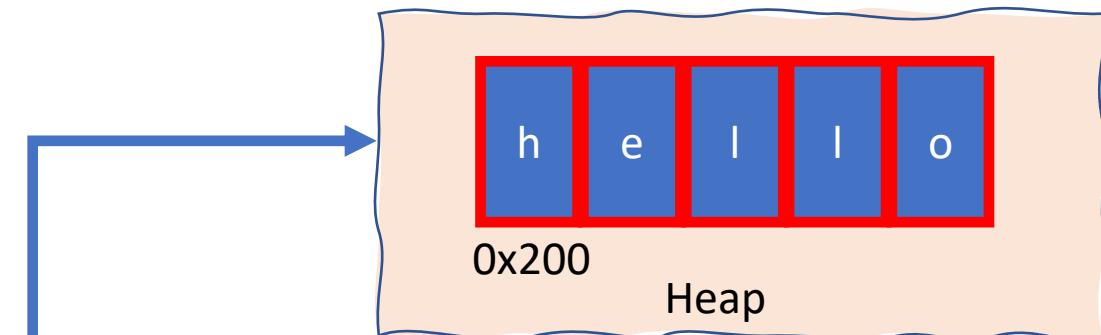




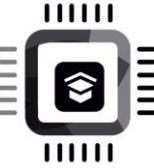
`let s2 = s1; // s1 is now invalid`



Shallow copy
(Move)



This is a fast operation because it only involves copying a small amount of data (memory address, len, capacity) rather than copying the entire contents of the string



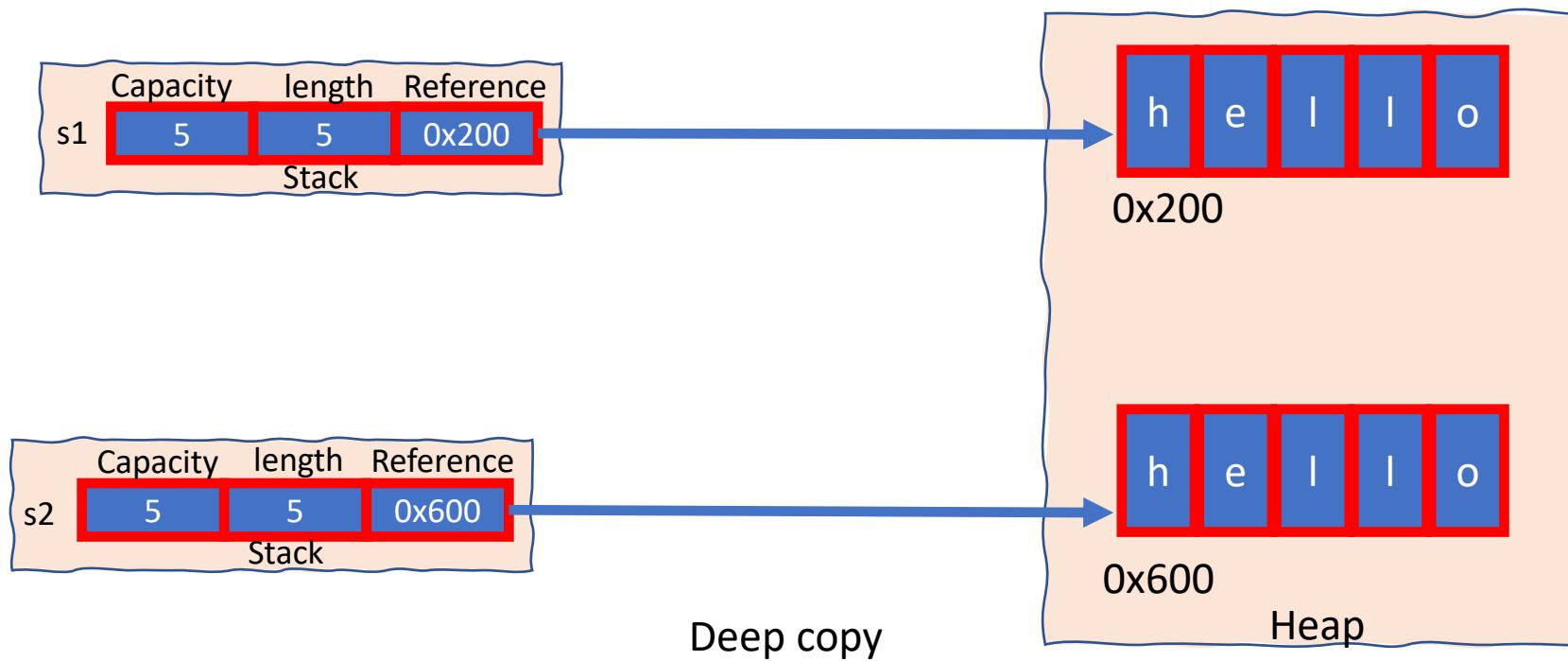
Deep copy

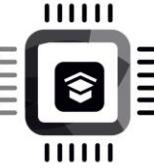
When a **String** value is cloned, a new heap allocation is created with the same contents as the original String. This is known as a deep copy or a copy by value.



```
let s1 = String::from("hello");
let s2 = s1.clone();
```

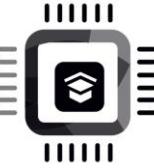
Cloning is a more expensive operation than a shallow copy because it requires allocating new memory and copying the original string's data to the new location



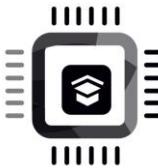


In Rust every piece of data has a single owner at any given time

Note: Reference Counted (RC) type is an exception to the strict single ownership rule.

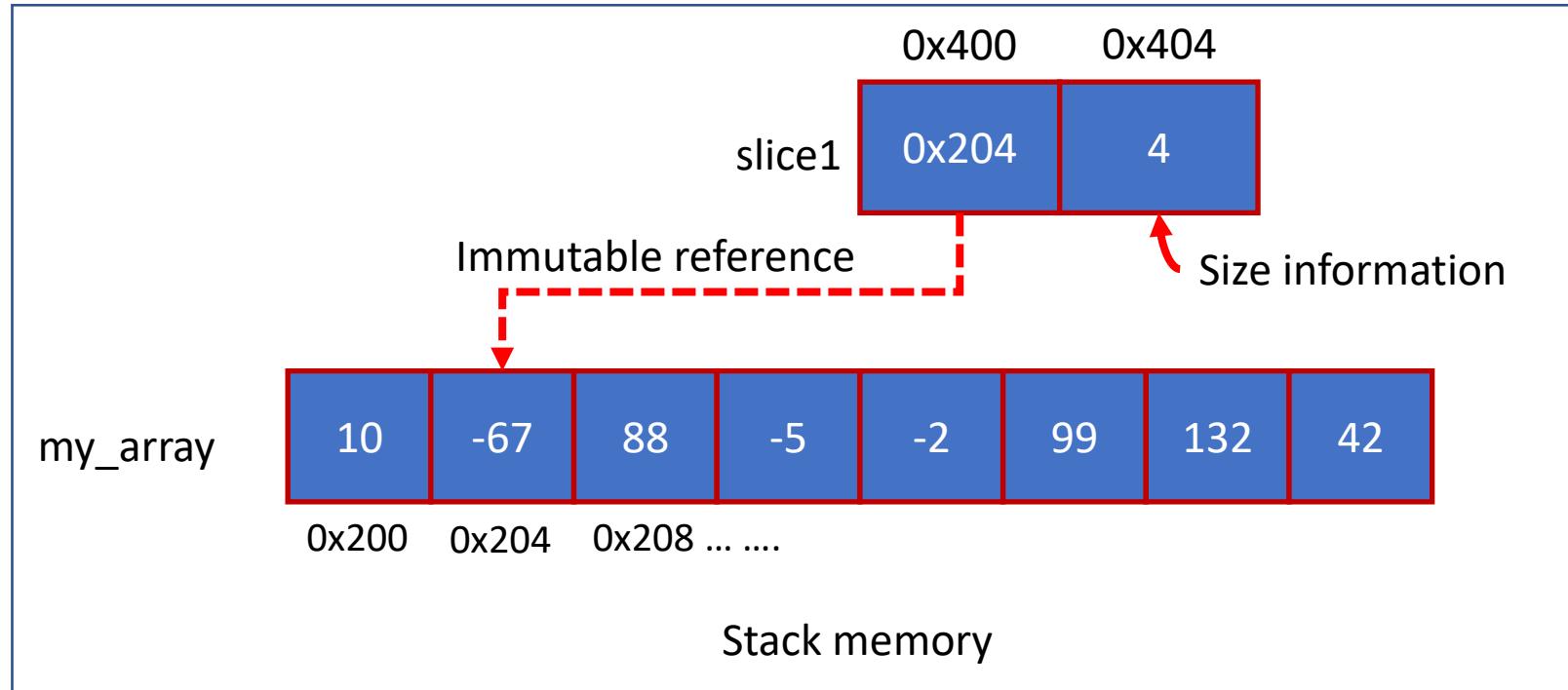


Slice of a String



```
let my_array = [10,-67,88,-5,-2,99,132,42];
```

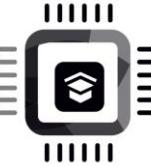
```
let slice1 = &my_array[1..5]; //type of slice1 is &[i32]
```



0x200 is memory location(address) where first element of the `my_array` array is found

0x400 is the memory location of the `slice1` where '**reference**' information is stored

0x404 is the memory location of the `slice1` where '**size**' information is stored



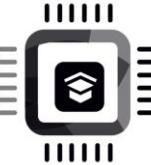
Slice of a String

```
let s1 = String::from("hello");
// 's2' is a slice of type &str
let s2 = &s1[0..=3];
println!("{}", s2); // output: hell
```

String slices are represented in Rust by the `&str` type

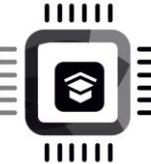
A string literal is actually a string slice

```
// s3 is a string slice. Its type is &str
let s3 = "hello!"
```



Primitive Type 'str' vs Primitive Type 'slice'

- `str` represents a sequence of UTF-8 characters that is immutable. It is accessed using a reference as `&str`, which allows safe, read-only operations.
- A "slice" in Rust is a dynamic view of a contiguous sequence of elements. It can be accessed as `&[T]` for immutable access, or `&mut [T]` for mutable access.



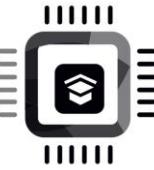
Convert String to slice and vice versa

```
fn main() {
    let s = String::from("hello");
    let slice1 = s.as_str();
    let slice2 = &s;

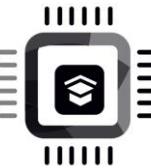
    println!("s = {}", s);
    println!("slice1 = {}", slice1);
    println!("slice2 = {}", slice2);
}
```

```
fn main() {
    let slice = "hello";
    let string1 = slice.to_string();
    let string2 = String::from(slice);

    println!("slice = {}", slice);
    println!("string1 = {}", string1);
    println!("string2 = {}", string2);
}
```

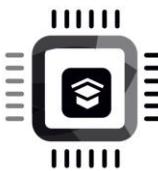


String and UTF-8 encoding

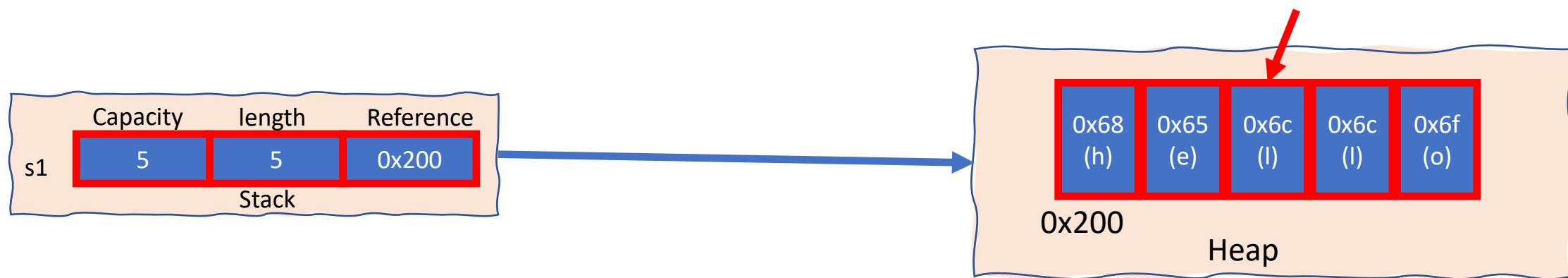


Strings are stored in UTF-8 format in Rust

- Rust's `String` and `String` literal are stored as UTF-8 encoded bytes.
- **UTF-8(Unicode Transformation Format)** is an efficient encoding technique to store Unicode Scalar Value of a character in computer memory.
- UTF-8 can represent some characters in one-byte units, it can also represent characters in two, three, or four-byte units. The "8" in UTF-8 refers to the fact that it is an 8-bit variable-width encoding, meaning that the number of bytes used to represent a character varies from 1 to 4, depending on the Unicode scalar value of the character.

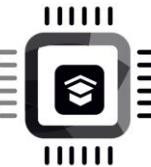


```
let s1 = String::from("hello");
```



Since the ASCII character set contains only 128 characters (7-bit values), and the first 128 Unicode code points are identical to ASCII, UTF-8 encoding of ASCII characters will always use a single byte. Therefore, any ASCII string is a valid UTF-8 string

ASCII characters are encoded in UTF-8 using a single byte



```
let s1 = String::from("hello+∞+😊+♂");
```

UTF-8 encoded bytes for ASCII

68 65 6C 6C 6F 2B E2 88 9E 2B F0 9F 98 8A 2B E0 B2 B0

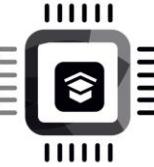
UTF-8 encoded bytes for U+221E (∞)

UTF-8 encoded string

UTF-8 encoded bytes for U+0CB0 (♂)

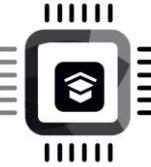
UTF-8 encoded bytes for U+263A (😊)

The ASCII characters are represented using a single byte, while other characters such as the infinity symbol, emojis require multiple bytes in UTF-8 encoding



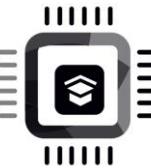
String concatenation

Can we use the '+' operator to concatenate two or more strings and produce a new string?



String Indexing

Rust strings don't support indexing because strings in Rust are encoded as UTF-8, which means that each character in the string can take up a variable number of bytes, depending on its codepoint value. This makes indexing a string a potentially expensive operation, since it would require iterating over the string from the beginning to find the Nth character.



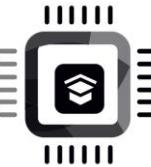
```
use unicode_segmentation::UnicodeSegmentation;
fn main() {
    let computer_in_hindi = "कंप्यूटर";
    println!("Hindi word: {}", computer_in_hindi);

    print!("All characters of the string: ");
    for ch in computer_in_hindi.chars() {
        print!("{} ", ch);
        print!("\n");
    }

    println!();
    println!("Total chars: {}", computer_in_hindi.chars().count());

    // Convert string to byte array
    let byte_array = computer_in_hindi.as_bytes();
    println!("Byte array: {:?}", byte_array);

    // Convert string to grapheme cluster iterator and print
    let graphemes = computer_in_hindi.graphemes(true);
    for grapheme in graphemes {
        println!("Grapheme: {}", grapheme);
    }
}
```



Computer written in hindi: कंप्यूटर

All characters of the string: कं प् यू टर

Total chars : 8

Byte array: [224, 164, 149, 224, 164, 130, 224, 164, 170, 224, 165, 141, 224, 164, 175, 224, 165, 130, 224, 164, 159, 224, 164, 176]

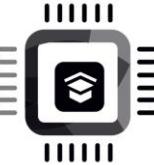
Grapheme: कं

Grapheme: प्

Grapheme: यू

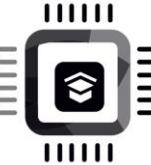
Grapheme: टर

Grapheme: र



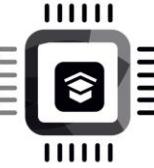
```
let computer_in_hindi = "कंप्यूटर";  
  
for ch in computer_in_hindi.chars() {  
    print!("{} ", ch);  
    print!("\n");  
}
```

कंप्यूटर

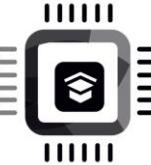


Taking slice can lead to invalid UTF-8 string

Taking slices on non-ASCII string can be a bad idea if the slice operation cuts in the middle of a multi-byte character, as it can lead to invalid UTF-8 encoding and cause a panic at runtime.

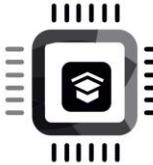


Ownership



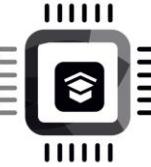
Ownership

- In Rust, every value has a single owner that determines its lifetime
- Rust doesn't have a `free` or `delete` keyword like C++ does. Instead, Rust uses a concept called "ownership" and a mechanism called "drop" to manage memory
- The owner is responsible for deallocated the memory associated with the value when it goes out of scope.
- Ownership of a variable can either be moved or copied.



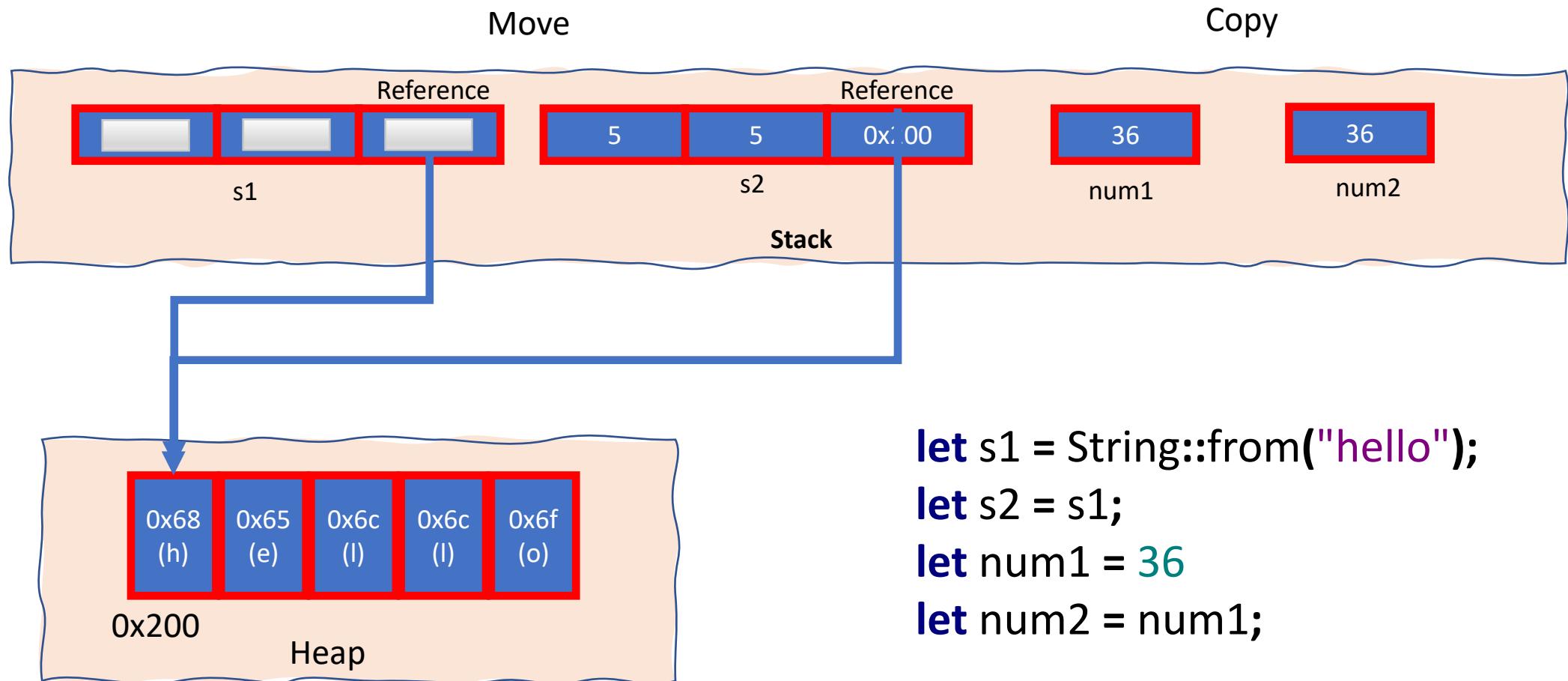
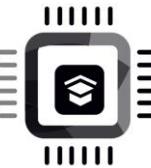
Move

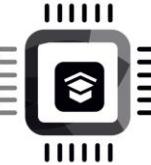
- Ownership of a value can be transferred between variables or functions using the move semantics. When move happens, the previous owner will no longer be able to access the value. This prevents issues such as use-after-free bugs that can occur in other programming languages.
- Move happens with heap-allocated data types like `String`, `Vec`, and other complex types that allocate memory on the heap.
- Move doesn't apply to primitive data types like `integer`, `char`, `float`, `bool` and other simple stack-based data types because they implement the `Copy` trait.
- Types that are composed only of other types that implement `Copy` trait are also automatically considered `Copy`. For example, `i32` and `bool` are both `Copy`, so a struct containing only `i32` and `bool` fields would also be `Copy`.



Copy

- When ownership of a variable is copied, a new variable is created that is a complete copy of the original variable, including its data.
- Types that implement Copy trait are automatically copied when they are assigned to a new variable, passed as a function argument, or returned from a function.





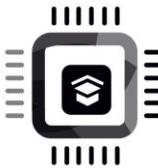
```
fn main() {
    ...
    //S1 is the owner of the string "hello"
    let s1 = String::from("hello");

    ...
    //Ownership of 's1' is transferred to 's'
    //fun1 returns ownership of 's' back to s2
    let s2 = fun1(s1);

    ...
    println!("{}", s2); //OK
    println!("{}", s1); //Error
}

fn fun1(mut s: String) -> String {
    ...
    println!("{}", s);

    ...
    s.push_str(".World");
    ...
    s //ownership returned
}
```

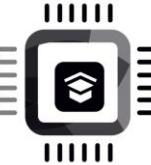


Move while indexing

```
fn main() {  
  
    let array1 = [5,6,7,8];  
  
    let array2: [String; 3] = [  
        String::from("foo"),  
        String::from("bar"),  
        String::from("baz"),  
    ];  
  
    println!("{}" ,array1[0]);  
  
    let first = array2[0];  
  
    println!("{}" ,first);  
  
    //let array3 = array2;  
    //println!("{}" ,array3);  
}
```

The first variable now owns the String value "foo", and the array2 array is no longer in a valid state. The remaining elements of the array still exist, but their contents are undefined. If you were to try to use the array2 array at this point, you would likely encounter a runtime error or other unexpected behavior.

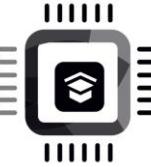
To avoid this situation, Rust disallows moving values out of arrays by default, and requires you to create a copy of the value instead. This ensures that the original value remains in the array, and that the array remains in a valid state.



$\&mut T$ is Move while $\&T$ is Copy

```
fn main() {
    let msg = "Hello".to_string();
    let ref1 = &msg;
    let ref2 = ref1;

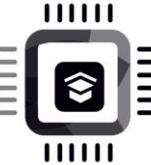
    if ref1 == ref2 {
        println!("Both refs are pointing to the same data");
    }
}
```



&mut T is Move

```
fn main() {
    let mut msg = "Hello".to_string();
    let ref1 = &mut msg;
    let ref2 = ref1; //Move

    ...
    //Error 'ref1' moved to 'ref2'
    if ref1 == ref2 {
        println!("Both refs are pointing to the same data");
    }
}
```



Trait implementations

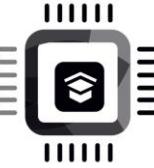
The following traits are implemented for all `&T`, regardless of the type of its referent:

- `Copy`
- `Clone` (Note that this will not defer to `T`'s `Clone` implementation if it exists!)
- `Deref`
- `Borrow`
- `fmt::Pointer`

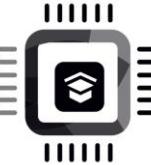
`&mut T` references get all of the above except `Copy` and `Clone` (to prevent creating multiple simultaneous mutable borrows), plus the following, regardless of the type of its referent:

- `DerefMut`
- `BorrowMut`

Source : <https://doc.rust-lang.org/std/primitive.reference.html>

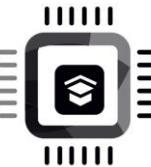


References



Call by value and Call by reference

- *Call by value* is a method of passing function arguments where the value of the argument is copied and passed to the function, leaving the original value unchanged
- *Call by reference* is a method of passing function arguments where a reference to the value is passed to the function instead of the value itself. This allows the function to access and modify the original value directly, without creating a copy of the value
- Call by reference does not claim ownership of the value being passed to the function, but instead passes a reference to the value, which allows the function to borrow the value temporarily without taking ownership of



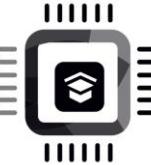
```
fn main() {
    let arr = [10, 20, 30, 40, 50];

    // Call by value
    let max_value = find_greatest_value_by_value(arr);
    println!("Maximum value by value: {}", max_value);

    // Call by reference
    let max_value = find_greatest_value_by_reference(&arr);
    println!("Maximum value by reference: {}", max_value);
}

// Call by value
fn find_greatest_value_by_value(arr: [i32; 5]) -> i32 {
    // TODO
}

// Call by reference
fn find_greatest_value_by_reference(arr: &[i32]) -> i32 {
    // TODO
}
```



References to References

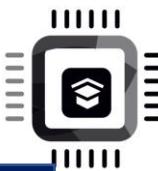
```
fn main() {
    let x = 42;
    let x_ref_1 = &x; //type of 'x_ref_1' : &i32
    let x_ref_2 = &x_ref_1; //type of 'x_ref_2' : &&i32
    let x_ref_3 = &x_ref_2; //type of 'x_ref_3' : &&&i32

    println!("Value of x using 1 level of reference: {}", x_ref_1);
    println!("Value of x using 2 levels of reference: {}", x_ref_2);
    println!("Value of x using 3 levels of reference: {}", x_ref_3)

    //println!("Value of x using 1 level of reference: {}", *x_ref_1);
    //println!("Value of x using 2 levels of reference: {}", **x_ref_2);
    //println!("Value of x using 3 levels of reference: {}", ***x_ref_3);
}

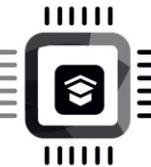
}
```

Comparing references

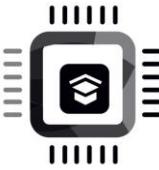


```
fn main() {
    let x = 42;
    let x_ref = &x; //type of 'x_ref' : &i32
    ...
    let y = 42;
    let y_ref = &y; //type of 'y_ref' : &i32
    ...
    //
    // Values are compared by their contents,
    // and when comparing two references (==,>,<,<=,>=, !=, etc), Rust compares
    // the values they are pointing to, not the memory addresses.
    // So in this case, the condition will evaluate to true,
    //
    if x_ref == y_ref {
        println!("values are the same");
    }
    /*
    if *x_ref == *y_ref {
        println!("values are the same");
    }
    */
}
```

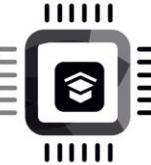
Comparison operators compare the values that the references point to rather than comparing the memory addresses of the references themselves



```
fn main() {
    let x = 42;
    let x_ref_1 = &x; //type of 'x_ref_1' : &i32
    let x_ref_2 = &x_ref_1; //type of 'x_ref_2' : &&i32
    ...
    //This is Error
    if x_ref_1 == x_ref_2
    {
        println!("values are the same");
    }
    ...
    //This is OK
    if x_ref_1 == (*x_ref_2)
    {
        println!("values are the same");
    }
}
```



Loops



Loops

loop:

Used to create an infinite loop that continues indefinitely until a ***break*** statement is encountered within the loop.

while:

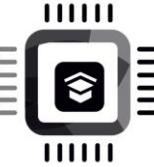
Used to loop over a block of code as long as a specified condition remains true.

while let:

This is a variant of the while loop that allows you to match against a pattern and extract variables from it, continuing to loop as long as the pattern matches.

for in:

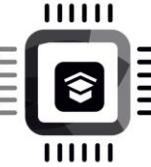
Used to iterate over a collection of items such as an array, vector, or range. It is similar to the for loop in other programming languages



loop

```
'label_name: loop {  
    // loop body  
}
```

You can use a label to identify a loop and then use the *break* or *continue* statement with the label name to break out of or continue to a specific loop.



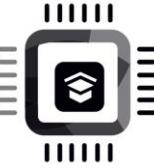
loop with break

```
fn main() {
    let mut i = 0;

    loop {
        if i == 3 {
            break;
        }
        println!("i = {}", i);
        i += 1;
    }

    println!("loop ends");
}
```

When *break* executes, the loop is immediately terminated, and the program continues executing from the statement after the loop.

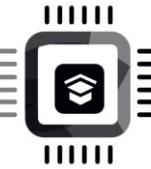


break with return value

```
fn main() {
    let mut i = 0;

    let result = loop {
        if i == 3 {
            break i * 2;
        }
        i += 1;
    };

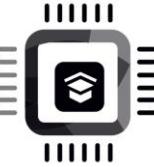
    println!("result = {}", result);
}
```



break with return value

```
fn find_index_of_first_even_number(numbers: &[i32]) -> Option<usize> {
    let mut index = 0;

    loop {
        if index >= numbers.len() {
            break None;
        }
        if numbers[index] % 2 == 0 {
            break Some(index);
        }
        index += 1;
    }
}
```



break with label and return value

```
fn main() {
    'outer: loop {
        println!("Outer loop");

        'inner: loop {
            println!("Inner loop-1");

            loop {
                println!("Inner loop-2");
                break 'outer;
            }
        }
    }

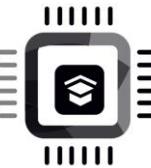
    println!("Exited outer loop");
}
```

```
fn main() {
    let result = 'outer: loop {
        println!("Outer loop");

        'inner: loop {
            println!("Inner loop-1");

            loop {
                println!("Inner loop-2");
                break 'outer;
            } //inner-2 loop ends
        } //inner-1 loop ends
    }; //outer loop ends

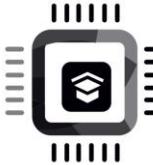
    println!("Exited outer loop with result = {}", result);
}
```



for in

```
for element in collection {  
    // code to execute for each element in the collection  
}
```

- ‘element’ is a variable that will be assigned each element of the collection on each iteration of the loop. The code inside the loop will be executed once for each element in the collection.
- The collection can be an array, a vector, a range, or any type that implements the Iterator trait.

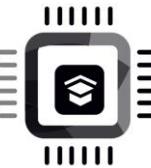


for loop with a range of numbers

```
fn main() {
    'outer: for i in 0... {
        println!("{}", i);
        if i == 10 {
            break 'outer;
        }
    }
}
```

```
fn main() {
    for i in 2..=5 {
        println!("{}", i);
    }
}
```

- ‘i’ is a loop variable. Its scope is limited to the loop block in which it is declared
- ‘i’ is of the same type as the values in the range
- **break** with a value cannot be used with for loop.



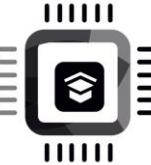
```
fn main() {
    let s = "Hello, World!";
    let target_char = 'o';
    let mut count = 0;

    for ch in s.chars() {
        if ch == target_char {
            count += 1;
        }
    }

    if count > 0 {
        println!("The character '{}' was found {} times in '{}', target_char, count, s);
    } else {
        println!("The character '{}' was not found in '{}'", target_char, s);
    }
}
```

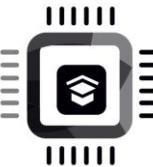
chars() returns an iterator and can be used with a for loop to iterate over its elements(chars of string slice)

The program counts the number of occurrences of a specific character in a given string



for in ... loop

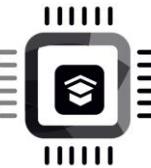
- **for** loop iterating over an iterator will stop when the iterator yields **None**. The **None** value signifies the end of the sequence in the context of an iterator.
- Rust's iterators are built on the Iterator trait, which defines a method **next()**. This method returns an **Option** type, which can be either **Some(value)** or **None**. When the iterator has more values to yield, **next()** returns **Some(value)**, and when the iterator has no more values to yield, it returns **None**



```
fn main() {  
    let words = ["hello", "world", "how", "are", "you"];  
  
    for s in &words {  
        println!("{}" , s.to_uppercase());  
    }  
}
```

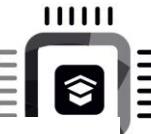
when you use `&words` in the for loop, it returns an iterator over references to each element in the array `words`.

```
fn main() {  
    let words = ["hello", "world", "how", "are", "you"];  
  
    // Create an iterator for the array using the iter() method.  
    let mut iterator = words.iter();  
  
    println!("{:?}" , iterator); // Output: Iter(["hello", "world", "how", "are", "you"])  
  
    // Get the next item in the iterator and assign it to a variable named "first".  
    let first = iterator.next();  
  
    println!("{:?}" , first); // Output: Some("hello")  
  
    println!("{:?}" , iterator); // Output: Iter("world", "how", "are", "you")  
}
```



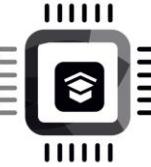
```
fn main() {
    let words = ["hello".to_string(), "world".to_string(), "how".to_string(), "are".to_string(), "you".to_string()];
    for s in words {
        println!("{}" , s.to_uppercase());
    }
    println!("{:?}", words); //Error
}
```

Here **for** loop consumes the array and moves each string element into the **s** variable. This means that after the loop is done, **words** is no longer accessible because it has been fully consumed.



```
fn main() {
    let words = ["hello".to_string(), "world".to_string(), "how".to_string(), "are".to_string(), "you".to_string()];
    let mut iterator = words.into_iter();
    let first = iterator.next();
    println!("{:?}", first);
    println!("{:?}", words);
}
```

- In the example , *words* is an array of String objects. By calling `into_iter()` on it, creates an iterator that will return owned String objects.
- Note that after calling `into_iter()`, the original array is no longer valid and cannot be used, because `into_iter()` takes ownership of the array and consumes it, just like for loop does.



while loop

The while loop in Rust allows for repeated execution of a block of code as long as a condition is true

```
while condition {
```

```
    ...
```

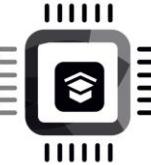
```
}
```

The while let loop in Rust allows for repeated execution of a block of code as long as a pattern matches a value.

```
while let pattern = value {
```

```
    ...
```

```
}
```

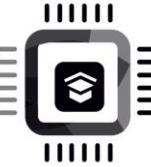


```
fn main() {
    let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut index = 0;

    while index < numbers.len() {
        let number = numbers[index];

        if number % 2 == 0 {
            println!("{} is even", number);
        } else {
            println!("{} is odd", number);
        }

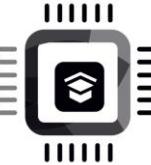
        index += 1;
    }
}
```



When to use *for*?

- **Iterating over collections:** When you have a collection (like an array, vector, or anything that implements the Iterator trait) and you want to perform an operation on each element. The for loop is the most idiomatic and straightforward way to iterate over collections

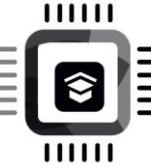
```
for element in collection {  
    // perform operations on element  
}
```



When to use *for*?

- **Range based iteration:** When you want to iterate a specific number of times, using a range. This is common for traditional indexed loops.

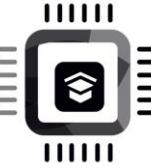
```
for i in 0..10 {  
    // repeated 10 times  
}
```



When to use *while* loop?

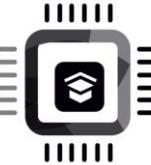
- **Condition based loops:** When you want to continue looping as long as a certain condition is true, and you cannot determine the number of iterations upfront.

```
while water_level_not_full == true {  
    // keep pumping  
}
```



When to use *while let* loop?

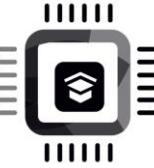
- Use this when you want to loop as long as a pattern continues to match



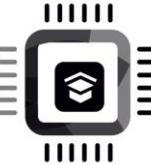
```
fn main() {
    let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut iterator = numbers.iter();

    while let Some(number) = iterator.next() {
        if number % 2 == 0 {
            println!("{} is even", number);
        } else {
            println!("{} is odd", number);
        }
    }
}
```

The **while let** loop is a more concise way of expressing a loop that uses pattern matching to de-structure an *Option* or *Result*. It is useful when the number of iterations is unknown and depends on the input.



Tuples



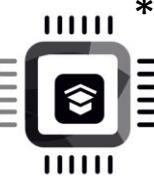
Tuples

- A tuple is a data structure that allows you to group together multiple values of different types into a single, ordered collection.

```
let my_tuple = (1, "hello", true);
```

```
let my_tuple: (i32, &str, bool) = (1, "hello", true);
```

- The type of the tuple 'my_tuple' is : (i32, &str, bool).
- Tuples in Rust do not have named parameters. Instead, the elements of a tuple are accessed by their position, using indexing starting at 0.



Accessing tuple elements

```
let my_tuple = (1, "hello", true);
```

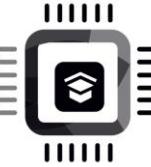
```
let my_int = my_tuple.0;  
let my_string = my_tuple.1;  
let my_bool = my_tuple.2;
```

or

```
//Tuple destructuring  
let (my_int, my_string, my_bool) = my_tuple;
```

Tuple index number cannot be variable. Tuple indices are fixed at compile-time, and you must use a constant integer literal to access an element in a tuple. It is not possible to use a variable or expression as the index value for a tuple.

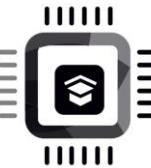
You can access the elements of a tuple by using dot notation and the index of the element you want to access



Return multiple values from a function

```
fn my_function() -> (i32, String, bool) {  
    let my_int = 42;  
    let my_string = "hello".to_string();  
    let my_bool = true;  
    (my_int, my_string, my_bool)  
}
```

```
fn main() {  
    let result = my_function();  
    println!("{}:{:?}", result);  
  
    // Destructuring the tuple  
    let (value, msg, is_ok) = result;  
    println!("{}:{}:{:?}", value, msg, is_ok);  
}
```

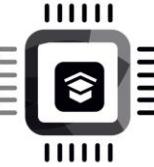


Passing tuple by reference

```
fn increment_element(tuple: &mut (i32, i32, i32)) {  
    tuple.0 += 1;  
    tuple.1 += 1;  
    tuple.2 += 1;  
}
```

```
fn main() {  
    let mut the_tuple = (1, 2, 3);  
    increment_element(&mut the_tuple);  
    println!("{:?}", the_tuple);  
}
```

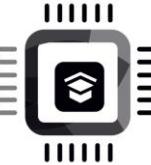
The compiler automatically dereferences the reference to the tuple or struct. we don't need to explicitly use the dereferencing operator (*) in this case.



Tuples can be nested

```
let grid = ((1, 2, 3), (4, 5, 6), (7, 8, 9));  
println!("Middle element: {}", grid.1.1);
```

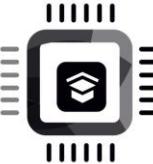
Tuples can be nested inside other tuples to create more complex data structures. For example, we could use a tuple of tuples to represent a 3x3 grid of numbers:



Tuples comparison

Tuples can also be compared lexicographically using operators like `==`, `!=`, `<`, `>`, `<=`, and `>=` as long as all the types in the tuple implement the ***PartialOrd*** and the ***PartialEq*** trait

The **PartialOrd** trait is used to define the ordering relationship between two values, while the **PartialEq** trait is used to define the equality relationship between two values.



```
fn main() {
    let tuple1 = (2, 3, 4);
    let tuple2 = (8, 3, 4);
    let tuple3 = (1, 2, 3);

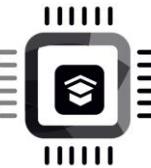
    // Possible case: all elements are comparable
    if tuple1 < tuple2 {
        println!("tuple1 is smaller");
    } else {
        println!("tuple2 is smaller")
    }

    if tuple1 == tuple3 {
        println!("Tuples equal")
    } else {
        println!("Tuples not equal")
    }

    let tuple4 = (1, "world.x");
    let tuple5 = (1, "world.z");
    let tuple6 = (1, "world.x");

    if tuple4 > tuple5 {
        println!("tuple4 is bigger")
    } else {
        println!("tuple5 is bigger")✓
    }

    if tuple4 == tuple6 {
        println!("Tuples equal")
    } else {
        println!("Tuples not equal")
    }
}
```

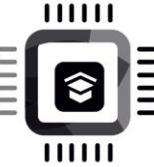


Pattern matching using tuple

Process the middle element only when the first element is greater than 0, the last element is less than 10

```
fn main() {
    let rcvd_data = (5, "hello", 8);

    match rcvd_data {
        (a, s, c) if a > 0 && c < 10 => {
            println!("Valid data: s = {}", s);
        }
        _ => println!("Invalid data"),
    }
}
```

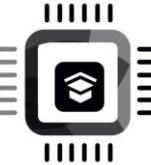


(**a**, **s**, **c**) **if** **a** > 0 **&&** **c** < 10
(5, "hello", 8)

Pattern matches and condition succeeds.
Successful match

(**a**, **s**, **c**) **if** **a** > 0 **&&** **c** < 15
(5, "hello", 15)

Pattern matches but condition fails.
Not a successful match



```
fn main() {
    let tup = (5, "SOP", 8);
    ...

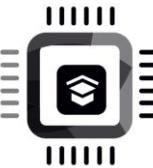
    match tup {
        (_, "SOP", _) => println!("Middle element is 'SOP'"),
        _ => println!("Middle element is not 'SOP'"),
    }
}
```

This will match any tuple with the middle element equal to "SOP", regardless of the values of the first and third elements

```
let tup = (1, "hello", true);

match tup {
    (1, s, true) => println!("The second element is {}", s),
    _ => (),
}
```

match against specific elements of specific value

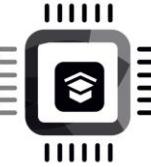


Tuple match with range expression

```
fn main() {
    ...
    let tup = (1, 2, 20);

    match tup {
        (_, _, c @ 10..=20) => {
            println!("c is between 10 and 20");
            println!("C is {}", c);
        }
        _ => (),
    }
}
```

The @ symbol followed by a range expression is used in pattern matching to create a binding to the value of the pattern being matched, while also checking that the matched value is within the specified range.

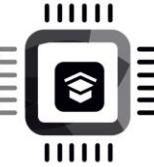


Pattern matching with tuple using the rest operator(..) to ignore some elements

```
fn main() {
    let tup = (1, "hello", 2.5, true, 'a');

    match tup {
        (_, _, c, ...) if c > 2.0 => println!("The third element is greater than 2.0"),
        _ => println!("The third element is less than or equal to 2.0"),
    }
}
```

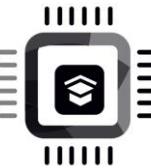
The .. syntax is called the "rest" operator in Rust. It can be used in destructuring patterns to match any remaining elements in a tuple, array, or struct



Pattern matching with tuple using variable binding and rest operator.

```
fn main() {
    let tup = (10, "hello", 2.5, true, 'a');

    match tup {
        (a @ 10, b @ "hello", ...) => println!("The first and second element: {} and {}.", a, b),
        _ => println!("The tuple does not match the pattern."),
    }
}
```

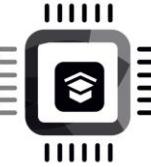


Move while matching

```
fn main() {
    let the_date = ("Monday".to_string(), "25".to_string(), "June".to_string(), "2023".to_string());

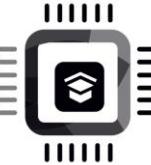
    match the_date {  
        (day, ...) if day == "Sunday" => {  
            println!("Its Sunday");  
        }  
        ...  
        _ => println!("Someother date"),  
    }  
  
    //Error : borrow of partially moved value: `the_date`  
    println!("{}:?", the_date);  
}
```

This gives error because the **the_date** tuple was partially moved within the match statement. The first element was moved to 'day' and consumed, making **the_date** inaccessible for further use outside of the match statement.



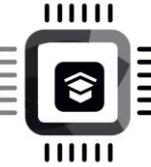
'ref' keyword

- When you use **ref** with a variable, it means that the variable is a reference to the value being matched, rather than taking ownership of the value. This allows you to work with a borrowed reference to the value and avoids possible move.



Summary : common use cases for tuples

1. Grouping related data
2. Returning multiple values from a function
3. Passing multiple arguments to a function
4. Enum variants : Some((index, item))
5. Less verbose alternative to a struct in some cases



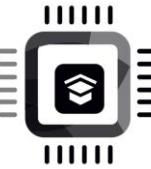
Tuple with defaults

Create a tuple with 5 elements, containing 2 i32 values and 3 bool values, all initialized to their default values.

```
let tuple = <(i32, i32, bool, bool)>::default();
```



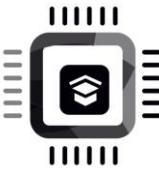
Here, <Type> explicitly tells the compiler that you are using the **default()** function from the **Default** trait for the type (i32, i32, bool, bool).



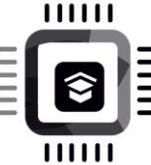
Tuples are not Iterable

A tuple can contain elements of different types. For example, (i32, f64, String) is a valid tuple.

This heterogeneity makes it impossible to create a single iterator type that can yield elements of a common type, unlike arrays or vectors that are homogeneous.

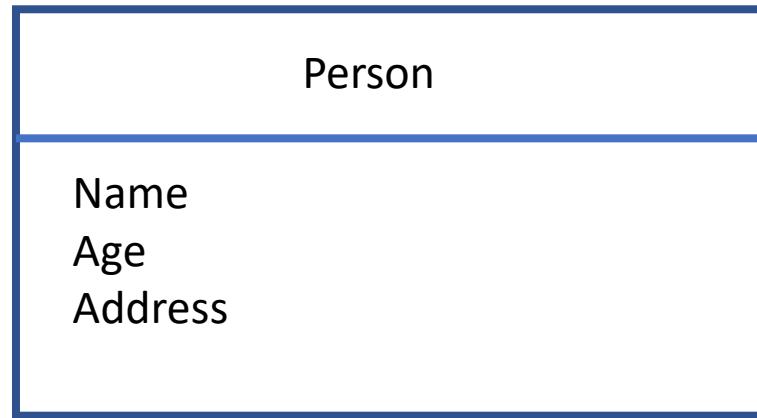
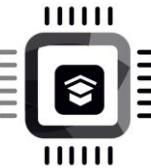


Structs



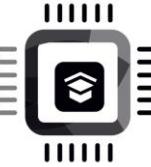
Struct

- Structs are user-defined data types that enable you to organize and encapsulate related pieces of data with different data types into a single unit, using named member fields for convenient access and manipulation
- By defining a struct, the programmer can create a custom data type that is tailored to the needs of their program



```
struct Person {  
    name: String,  
    age: u32,  
    address: String,  
}
```

Defining a struct in Rust



Different types of structs in Rust

There are three main types of structs in Rust:

- 1.Tuple struct:** Struct that has unnamed fields
- 2.Named struct:** Struct that has named fields
- 3.Unit Struct:** Struct that has no fields



Named Struct

```
//Defining a struct
struct Person {
    name: String,
    age: u32,
    address: String,
}

fn main() {
    //creating an instance of the 'struct Person'
    let person = Person {
        name: String::from("Alice"),
        age: 25,
        address: String::from("123 Main St"),
    };

    //use the dot(.) operator to print the values of each member element
    println!("Name: {}", person.name);
    println!("Age: {}", person.age);
    println!("Address: {}", person.address);
}
```

- According to Rust's naming conventions, the name of a struct should be in ***UpperCamelCase*** (PascalCase). ***UpperCamelCase*** means that the first letter of each word in the name is capitalized, and there are no underscores between words.
e.g. : SensorData, User, CustomerDB
- The names of member elements (fields) in a struct should be in ***snake_case***. ***snake_case*** means that the words in the name are separated by underscores, and all letters are lowercase.
e.g. : address_curr, address_perm



Create an instance of the struct using variables

```
let tmp_name = String::from("Alice");
let tmp_age = 25;
let tmp_addr = String::from("123 Main St");

//creating an instance of the struct Person
let person = Person {
    name: tmp_name,
    age: tmp_age,
    address: tmp_addr,
};
```

Field initialization shorthand syntax allows to create new instances of a struct using variables with the same name as the struct's fields

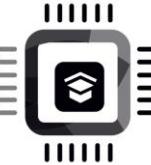
```
let name = String::from("Alice");
let age = 25;
let address = String::from("123 Main St");

//creating an instance of the struct Person
let person = Person {
    name,
    age,
    address,
};

struct Point {
    x: i32,
    y: i32,
}

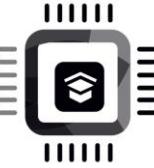
fn create_point(x1: i32, y1: i32) -> Point {
    Point { x: x1, y: y1 }
}

fn create_point(x: i32, y: i32) -> Point {
    Point { x, y }
}
```



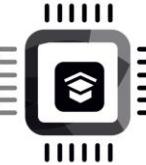
#[derive(Debug)]

- The **#[derive(Debug)]** annotation tells the Rust compiler to automatically generate the code to implement the **Debug** trait for the struct.
- This allows us to print out a struct for debugging purposes using `{:?}` format specifier
- The **derive** attribute in Rust allows developers to automatically generate implementations for various traits for a data structure. These traits can include **Debug**, **Clone**, **Copy**, **PartialEq**, and others. By adding the **derive** attribute with the name of the trait(s) to the data structure definition, Rust will automatically generate the implementation of that trait for the data structure.



Mutable struct

When you declare a struct variable with the ***mut*** keyword, you make the entire variable mutable, including all its fields.



```
#[derive(Debug)]
struct Person {
    name: String,
    age: u32,
    address: String,
}

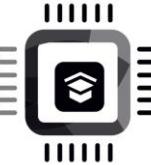
fn main() {
    // Create a new mutable Person struct
    let mut person = Person {
        name: String::from("Alice"),
        age: 25,
        address: String::from("123 Main St"),
    };

    // Update the name field
    person.name = String::from("Bob");

    // Print the updated struct
    println!("{:?}", person);

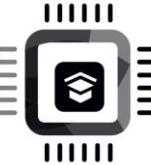
    // Move the 'name' field of struct is uninitialized
    let _name = person.name;

    // OK
    println!("{}" , person.age); //OK
    // Error
    println!("{:?}" , person);
}
```



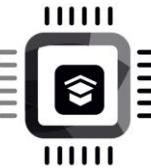
Is struct ‘copy’ or ‘move’ ?

- Whether a struct is **Copy** or **Move** depends on the types of its member fields
- When you assign a struct variable to another, it is a ‘move’ by default because structs *do not implement the Copy trait*.



By default, variable bindings have ‘move semantics’

```
· #[derive(Debug)]
· struct Point {
·     · · · x: i32,
·     · · · y: i32,
· }
·
· fn main() {
·     · · · let p1 = Point { x: 1, y: 2 };
·     · · · let p2 = p1; // p1 is moved to p2
·     · · · println!("p1: {:?}", p1, p2); // Error
· }
```



```
#[derive(Copy,Clone,Debug)]
struct Point {
    x: i32,
    y: i32,
}

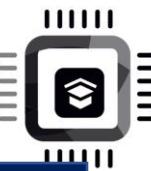
fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = p1; // p1 is copied to p2
    let p3 = p1.clone(); // p1 is cloned to p3
    println!("p1: {:?}, p2: {:?}, p3: {:?}", p1, p2, p3); //OK
}
```

'Clone' is a supertrait of 'Copy', so everything which is Copy must also implement Clone

In this particular case, both ***clone()*** and ***Copy*** are producing the same result, which is making a bitwise copy of the data in the struct. So, the values of p1 are being copied to p2, and p1 is being cloned to p3.

But note that this is not always the case, and in some situations, the behavior of Clone and Copy can be different.

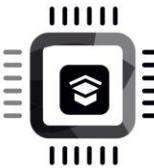
<https://doc.rust-lang.org/std/marker/trait.Copy.html>



```
//Error
#[derive(Debug, Copy, Clone)]
struct Person {
    name: String,
    age: u32,
}
```

```
fn main() {
    let p1 = Person {
        name: String::from("Alice"),
        age: 25,
    };
    let p2 = p1; // p1 is copied to p2
    let p3 = p1.clone(); // p1 is cloned to p3
    println!("p1: {:?}, p2: {:?}, p3: {:?}", p1, p2, p3);
}
```

The *Copy* trait cannot be implemented for types that contain a non-Copy field, like *String*



```
#[derive(Debug, Clone)]
struct Person {
    name: String,
    age: i32,
}

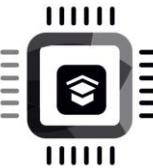
fn main() {
    let mut person = Person {
        name: String::from("John"),
        age: 30,
    };

    let user = person.clone();

    // 'person' is still valid
    let name = person.name.clone();
    person.age = 40;
    println!("Name: {}", name);

    // 'person' moved to 'admin'
    let admin = person;

    println!("User: {:?}", user);
    println!("Admin: {:?}", admin);
    println!("person: {:?}", person); // Error
}
```



##[derive(Default)]

```
#![allow(dead_code)]
#![allow(unused_variables)]

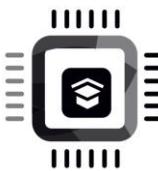
#[derive(Default, Debug)]
struct Person {
    name: String,
    age: u8,
    is_male: bool,
    height: f32,
    initial: char,
}

fn main() {
    let person = Person::default();

    println!("Person: {:?}", person);
}

Output:
Person: Person { name: "", age: 0, is_male: false, height: 0.0, initial: '\0' }
```

In Rust, struct types do not have a default implementation for the **Default** trait. If each field of a struct implements the **Default** trait, you can easily implement the Default trait for the struct automatically using the **##[derive(Default)]** attribute annotation



Updating a struct using another struct

```
#[derive(Debug)]
struct Process {
    name: String,
    pid: u32,
    group: String,
}

fn main() {
    let process1 = Process {
        name: String::from("Ping"),
        pid: 0x1234,
        group: String::from("Networking"),
    };
    println!("Process 1: {:?}\n", process1);

    // Create a new process by updating the name field
    let process2 = Process {
        name: String::from("Route"),
        ..process1
    };
    println!("process 2: {:?}\n", process2);

    // Create a new process by updating the pid and group fields
    let process3 = Process {
        pid: 0x3456,
        group: String::from("Security"),
        ..process2
    };
    println!("process 3: {:?}\n", process3);
}
```

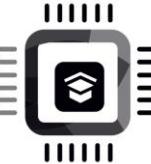
Struct literal update syntax

In struct update syntax, the source and destination structs must be of the same type

```
Process 1: Process {
    name: "Ping",
    pid: 0x1234,
    group: "Networking",
}

process 2: Process {
    name: "Route",
    pid: 0x1234,
    group: "Networking",
}

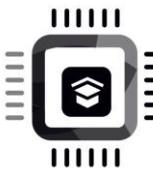
process 3: Process {
    name: "Route",
    pid: 0x3456,
    group: "Security",
}
```



Tuple struct

Tuple structs are similar to tuples in that they don't have named fields. Instead, the fields are accessed by their index within the tuple. However, unlike a plain tuple, tuple structs possess a name, which can be advantageous in providing additional context and clarity to the code

```
struct Point(i32, f64, u8);
fn main() {
    let point = Point(10, 3.5, 1);
    println!("x: {}, y: {}, z: {}", point.0, point.1, point.2);
}
```



```
struct Size(i32, i32, i32);
struct Point(i32, i32, i32);

fn refactor_point(point: &mut Point) {
    point.0 += 5;
    point.1 += 10;
}

fn print_point(point: Point) {
    println!("x: {}, y: {}, z: {}", point.0, point.1, point.2);
}

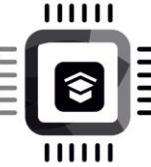
fn main() {
    let size = Size(0, 0, 0);
    let mut origin = Point(0, 0, 0);

    refactor_point(&mut origin);
    print_point(origin);

    print_point(size); //Error
}
```

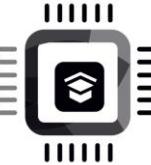
If two tuple structs have the same sequence of field types, they are considered as different types, even if the field types have the same name and value. Therefore, you cannot pass one tuple struct to another tuple struct, even if their field types are the same.

You can achieve type safety over collection of data using tuple structs



Summary

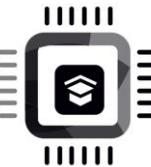
Tuple structs are used when you want to give a name to a whole tuple and make it a distinct type from other tuples, but naming each field as in a regular struct would be verbose or redundant. They can also be used when you need a lightweight struct-like data structure without the overhead of named fields.



A Unit type struct

A Unit type struct, also known as a zero-sized struct, is a struct in Rust that has no fields or size. It is declared as ***struct Name;*** without any fields. It is useful in situations where a type needs to be represented but no data needs to be stored, such as marker traits or indicating a state.

1. Can be used to implement marker traits, which convey information about the type without requiring any additional data
2. As Enum Variants: Can be used as variants within an enum to represent cases where no additional data is required
3. As placeholders in module-level constants

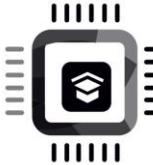


```
#[derive(Debug)]
struct NoData;

fn do_something() -> NoData {
    // do something that doesn't return any data
    NoData
}

fn main() {
    let result = do_something();
    println!("{:?}", result); // prints "NoData"
}
```

Using a struct as the return type instead of a unit type allows us to give the return value a name, which can make our code more self-documenting and easier to understand



Passing a struct by reference

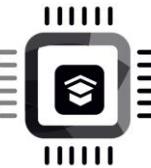
The dot operator implicitly dereferences the pointer and accesses the struct member. So, we don't need to explicitly dereference the pointer using the `*` operator before accessing the member.

```
struct Person {
    String name;
    u8 age;
}

fn update_person_age(person: &mut Person, new_age: u8) {
    person.age = new_age;
    // (*person).age = new_age; //OK
}

fn main() {
    let mut p = Person { name: String::from("Alice"), age: 25 };
    update_person_age(&mut p, 30);
    println!("Name: {}, Age: {}", p.name, p.age);
}
```

It is the explicit way of dereferencing a pointer to a struct to access its field.



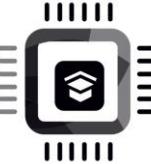
Methods and associated functions of a struct

Methods

- Methods are associated with a specific instance of the struct or enum
- Methods are defined within the `impl` block for the struct or enum, and their first parameter is always a reference to the instance of the struct or enum
- struct methods are accessed using an instance of the struct followed by the dot (.) operator. For example, if we have an instance of the **Person** struct named `person` with a method named `introduce`, we can access it as `person.introduce()`.

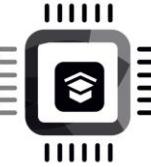
Associated functions

- Associated functions are not tied to any particular instance and are called on the type itself.
- Associated functions are defined within the `impl` block as well, but their first parameter is not a reference to the type.
- Associated functions are accessed using the struct's name followed by the double colon (::) operator
- For example, if we have a struct named **Person** with an associated function `new`, we can access it as `Person::new()`.



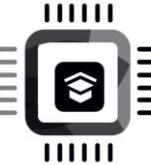
www.fastbit^~
//Struct definition

```
struct MyStruct {  
    ....  
}  
  
//f1() and f2() are methods/associated functions of 'MyStruct'  
//Put them inside impl <Struct-Name> block  
impl MyStruct {  
  
    fn f1(...) {  
        ....  
    }  
  
    fn f2(...) {  
        ....  
    }  
  
}  
  
fn main() {  
    //Create an instance of 'MyStruct'  
    let struct_inst = MyStruct {  
        ....  
    } ;  
  
    //Method call  
    struct_inst.f1();  
  
}
```



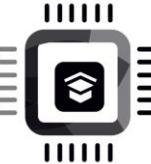
Writing Struct methods

- 1) Method that borrows 'self' immutably
- 2) Method that borrows 'self' mutably
- 3) Method that takes ownership of the 'self'

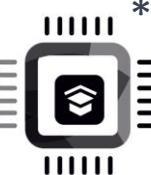


```
struct Point {  
    x: f32,  
    y: f32,  
}  
  
impl Point {  
    // Method that borrows self immutably  
    fn distance_from_origin(&self) -> f32 {  
        (self.x.powi(2) + self.y.powi(2)).sqrt()  
    }  
}  
  
fn main() {  
    let p = Point { x: 3.0, y: 4.0 };  
  
    let dist = p.distance_from_origin();  
    println!("Distance from origin: {}", dist);  
}
```

Short hand notation for
`self: &Point`



```
impl Point {  
  
    // Method that borrows self immutably  
    fn distance_from_origin(&self) -> f32 { ... }  
  
    // Method that borrows self mutably  
    fn translate(&mut self, dx: f32, dy: f32) {  
        self.x += dx;  
        self.y += dy;  
    }  
  
}  
  
fn main() {  
    let mut p = Point { x: 3.0, y: 4.0 };  
  
    p.translate(1.0, 1.0);  
    println!("Translated point: ({}, {})", p.x, p.y);  
}
```



```
#[derive(Debug)]
struct Point {
    x: f32,
    y: f32,
}

impl Point {

    // Method that borrows self immutably
    fn distance_from_origin(&self) -> f32 { ... }

    // Method that borrows self mutably
    fn translate(&mut self, dx: f32, dy: f32) { ... }

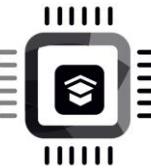
    // Method that takes ownership of self
    fn into_tuple(self) -> (f32, f32) {
        (self.x, self.y)
    }
}

fn main() {
    let p = Point { x: 3.0, y: 4.0 };

    let tuple = p.into_tuple();
    println!("Point as tuple: {:?}", tuple);

    println!("{:?}", p); //Error
}
```

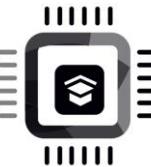
Using `self` as the first parameter of a method means that the method takes ownership of the instance and consumes it, transforming it into something else. This is useful when you want to prevent the caller from using the original instance after the transformation.



Remember calling **new** and **from** functions of the String?

They are associated function of the standard library ‘String’ type which is why you can call them directly on the type name without needing an instance of the String.

```
let empty_string: String = String::new();  
  
let message: String = String::from("hello");
```



```
impl Point {
    // Method that borrows self immutably
    fn distance_from_origin(&self) -> f32 { ... }

    // Method that borrows self mutably
    fn translate(&mut self, dx: f32, dy: f32) { ... }

    // Method that takes ownership of self
    fn into_tuple(self) -> (f32, f32) { ... }

    // Associated function
    fn from_tuple(coords: &(f32, f32)) -> Point {
        Point { x: coords.0, y: coords.1 }
    }
}

fn main() {
    let tuple = (10,20);
    let q = Point::from_tuple(&tuple);
    println!("Point from tuple: ({}, {})", q.x, q.y);
}
```

```
struct Point {
    x: f32,
    y: f32,
}
```

Constructor of a Struct

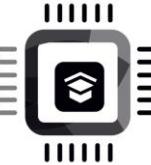


```
struct Rectangle {  
    width: u32,  
    height: u32,  
}
```

```
impl Rectangle {  
    fn new(width: u32, height: u32) -> Rectangle {  
        Rectangle { width, height }  
    }  
}
```

```
fn main() {  
    let rect = Rectangle::new(10, 20);  
}
```

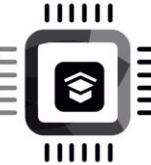
- In Rust, the convention is to name the constructor method using the word **new**, but this is not a requirement. You can choose any name you like for your constructor method as long as it makes sense for your struct.
- **new** is not a keyword in Rust. It is just a naming convention that has become common in Rust code.



Pattern matching using Struct

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let p = Point { x: 1, y: 2 };  
  
    match p {  
        // Note: the arguments inside { ... }  
        // must be field names of the struct  
        Point { x, ... } => println!("x is {}", x),  
    }  
}
```

Destructuring a struct

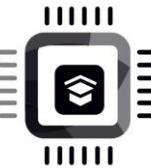


```
struct Rectangle {
    width: i32,
    height: i32,
}

fn main() {
    let rect = Rectangle { width: 10, height: 20 };

    match rect {
        Rectangle { width: w, height: h } if w == h => {
            println!("The rectangle is square!")
        },
        Rectangle { width: _, height: _ } => {
            println!("The rectangle is not square.")
        },
    }
}
```

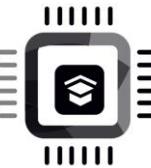
Matching multiple fields with guard



```
fn main() {
    let person = Person {
        name: "Ram".to_string(),
        age: 35,
    };

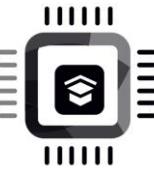
    match person {
        Person { name: _, age: 30 } => println!("A person with age 30 found."),
        Person { name, age: 35 } if name == "Ram" => {
            println!("Ram with age 35 found."),
        }
        _ => println!("Not sure who the person is."),
    }
}
```

Guard and variable binding

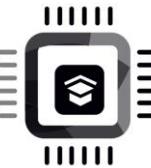


```
#![allow(dead_code)]  
  
struct Point {  
    x: i32,  
    y: i32,  
}  
  
struct Rectangle {  
    top_left: Point,  
    bottom_right: Point,  
}  
  
fn main() {  
    let rect = Rectangle {  
        top_left: Point { x: 0, y: 10 },  
        bottom_right: Point { x: 20, y: 0 },  
    };  
  
    match rect {  
        Rectangle { top_left: Point { x: 0, ... }, ... } => {  
            println!("The top-left corner of the rectangle is on the x-axis.")  
        },  
        Rectangle { ... } => println!("The rectangle is somewhere else."),  
    }  
}
```

Matching against nested structs



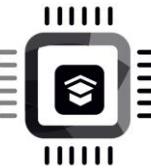
Enums



Enumerations

- An enum is typically used in any programming language to represent a value that can be one of several possible variants.
- When you define an enum in Rust, you are essentially creating a new type that can have one of several possible variants. Each variant can be associated with its own set of data, which allows you to represent complex data structures using a single enum.
- Enums can have methods associated with them, just like structs

Enums in Rust are powerful because
they can be used to define variants
with associated data

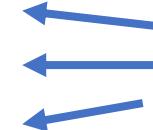


enum definition

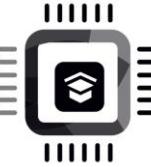
If you are working with a physical button or switch, you can use the `ButtonState` enum to represent the current state of the button

```
enum ButtonState {  
    Pressed,  
    NotPressed,  
    Bouncing(u32),  
}
```

Variants



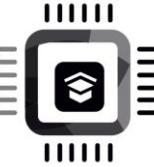
- *ButtonState* enum defines 3 possible variants for the state of the button
- The Bouncing variant includes a `u32` value that represents the number of milliseconds the button has been bouncing for



enum definition

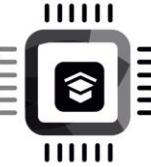
```
/* Defining an enum in Rust */
enum CarStatus {
    MovingUp(u32, u32, u32), //This variant has associated data
    MovingDown,
    NotMoving
}

enum GameState {
    Restart,
    End,
    Pause
}
```



Naming convention

- Refer : <https://doc.rust-lang.org/std/keyword.enum.html>
- Enum name : UpperCamelCase(PascalCase)
- Enum variants : UpperCamelCase (PascalCase)



Creating an instance of the enum

- To create an instance of the `ButtonState` enum, you can use one of its variants, either `Pressed`, `NotPressed`, or `Bouncing`. If you're creating an instance of the `Bouncing` variant, you'll need to provide a value of type `u32` that indicates how long the button has been bouncing for.

```
// Create an instance of the Pressed variant
```

```
let current_state = ButtonState::Pressed;
```

Or

```
// Create an instance of the NotPressed variant
```

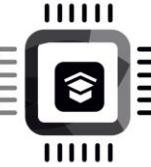
```
let current_state = ButtonState::NotPressed;
```

or

```
// Create an instance of the Bouncing variant with a value of 100 milliseconds
```

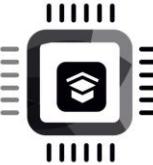
```
let current_state = ButtonState::Bouncing(100);
```

Variable *current_state* type:
'ButtonState'



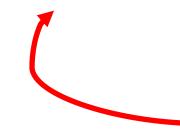
Creating an instance of Struct vs Enum

- When creating an instance of an enum , you need to specify which variant you want to use and provide any associated data that the variant require.
- On the other hand, when creating an instance of a struct, you need to initialize all of its member fields. You need to provide a value for each of them when creating an instance of the struct.



```
enum Shape {  
    Circle { x: f32, y: f32, radius: f32 },  
    Rectangle { x: f32, y: f32, width: f32, height: f32 },  
}
```

```
let new_shape = Shape::Circle { x: 0.0, y: 0.0, radius: 1.0 };
```



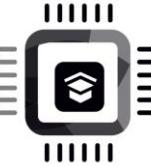
To represent a new shape in your program, you can choose from two variants, Circle and Rectangle, using the Shape enum.

```
struct Point {  
    x: f32,  
    y: f32,  
}
```

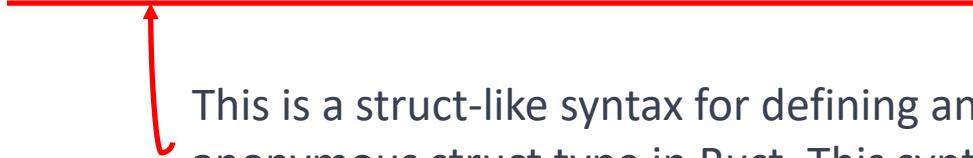
```
let new_point = Point { x: 1.0, y: 2.0 };
```

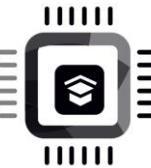
While creating a new point, you have to create new instance of the *Point* structure giving appropriate values for its member elements x and y





```
enum Shape {  
    Circle { x: f32, y: f32, radius: f32 },  
    Rectangle { x: f32, y: f32, width: f32, height: f32 },  
}
```

This is a struct-like syntax for defining an anonymous struct type in Rust. This syntax is often used when defining named associated values for enum variants



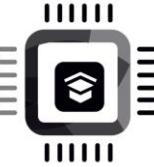
Enum and match

```
#[allow(dead_code)]
enum ButtonState {
    Pressed,
    NotPressed,
    Bouncing(u32),
}

fn main() {
    let current_state = ButtonState::Bouncing(100);

    match current_state {
        ButtonState::Pressed => println!("Button is pressed"),
        ButtonState::NotPressed => println!("Button not pressed"),
        ButtonState::Bouncing(ms) => println!("Button is bouncing for {} ms ",ms),
    }
}
```

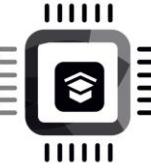
This match is exhaustive and covers all possible variants of the ButtonState enum. Therefore, there is no need to include a catch-all arm.



Methods and associate functions of an Enum

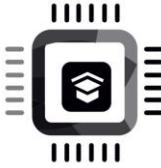
Write a method to calculate area of the Shape

```
enum Shape {  
    Circle { x: f32, y: f32, radius: f32 },  
    Rectangle(Rectangle),  
    Square(f32,f32,f32),  
}
```



Pattern matching with Enums

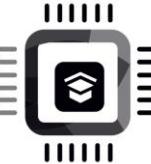
pattern matching to handle different variants of the enum



```
enum LightState {
    On { brightness: u8 },
    Off,
}

fn main() {
    let bulb = LightState::Off;

    match bulb {
        LightState::On { brightness } => println!("The light is on at brightness {}", brightness),
        LightState::Off => println!("The light is off"),
    }
}
```



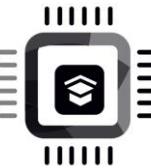
```
enum LightState {
    On { brightness: u8 },
    Off,
}

fn main() {
    let bulb = LightState::On { brightness: 180 };

    match bulb {
        LightState::On { brightness: 180 } => println!("The light is on at brightness 180"),
        //LightState::On { brightness: level @ 180 } => println!("The light is on at brightness {}", level),
        LightState::Off => println!("The light is off"),
    }
}
```

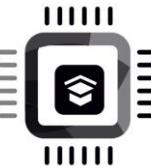
Non-exhaustive match. Fix it !

Pattern to match a specific value



matching with wildcard patterns ('_'):

```
· enum · MyEnum · {  
·     · VariantA(i32, i32),  
·     · VariantB(String),  
·     · VariantC(bool, f64),  
· }  
  
· fn · main() · {  
·     · let · my_enum · = · MyEnum::VariantA(10,20);  
  
·     · match · my_enum · {  
·         ·     · MyEnum::VariantA(_, _) · => · println!("VariantA"),  
·         ·     · MyEnum::VariantB(_) · => · println!("VariantB"),  
·         ·     · _ · => · println!("Other variant"),  
·     · }  
· }
```

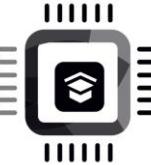


```
enum MyEnum {
    VariantA(i32, i32),
    VariantB(String),
    VariantC(bool, f64),
}

fn main() {
    let my_enum = MyEnum::VariantB("ggg".to_string());

    match my_enum {
        VariantA(_, b) => println!("VariantA with second field {}", b),
        VariantB(s) => if s == "hello" {
            println!("VariantB with value {}", s);
        }
        VariantC(true, _) => println!("VariantC with first field true"),
        VariantC(false, f) => println!("VariantC with second field {}", f),
    }
}
```

Ignoring fields with underscore patterns

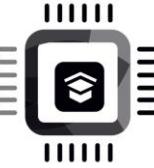


```
enum MyEnum {
    VariantA(i32, i32, u8),
    VariantB(String),
    VariantC{is_ok: bool, x: f32, y: f64},
}

fn main() {
    let my_enum = MyEnum::VariantC{is_ok: false, x: 6.0, y: 1.0};

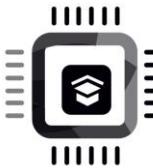
    match my_enum {
        VariantA(a, ..) => println!("VariantA with first field {}", a),
        VariantB(s) => println!("VariantB with value {}", s),
        VariantC{x, ..} => println!("VariantC with x field {}", x),
    }
}
```

Using the .. operator to match any remaining fields



The ‘Option’ enum type

- The ***Option*** enum encapsulates the possibility of either having a value or not having a value. It represents the concept of an optional value in Rust, which can be either ***Some(value)*** or ***None***.
- The ***Option*** type is defined in the Rust standard library **std::option::Option**



Enum std::option::Option

1.0.0 · [source](#) · [-]

```
pub enum Option<T> {
    None,
    Some(T),
}
```

[-] The **Option** type. See [the module level documentation](#) for more.

Variants

§None

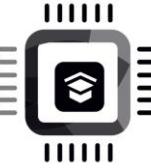
No value.

Some(T)

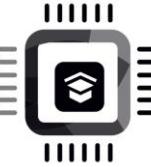
Some value of type T.

- The **Option** type is a generic enum that can be used with any type T. The type parameter T specifies the type of the value that may or may not be present.
- This allows you to use **Option** with any type, including primitive types like integers and floats, as well as more complex types like structs and enums.

<https://doc.rust-lang.org/std/option/enum.Option.html>



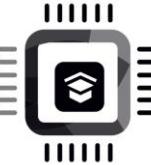
By using Option, Rust encourages developers to handle nullability explicitly and forces them to account for the possibility of absence of value in their code. This approach helps prevent null reference errors and encourages safer and more robust software



The Rust Prelude

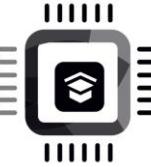
*In Rust, the Option type is included in the prelude, which means you don't need to explicitly import it using the **use** keyword. The prelude is a set of items that are automatically imported into every Rust module.*

<https://doc.rust-lang.org/std/prelude/index.html>



Option<T> helps avoiding null access errors

- In Rust, trying to access a value that doesn't exist or is set to ***None*** using the Option<T> type results in a compile-time error, not a runtime error like in other languages that have ***null***.
- The Rust type system mandates that you must check that an ***Option*** is ***Some*** before you can use its contents.

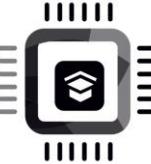


Exercise

Write a function that returns the longest string item from the array of strings.

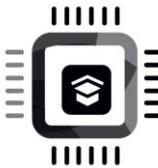
The return value must be of type **Option**.

None if the array is empty, and ***Some(longest_string)*** otherwise.



```
www.fastb fn fing_biggest_item(strings: &[&str]) -> Option<&str> {  
    let mut longest: Option<&str> = None;  
    for item in strings {  
        if longest.is_none() || (item.len() > longest.unwrap().len()) {  
            longest = Some(item);  
        }  
    }  
  
    longest  
}
```

```
fn main() {  
    let strings = ["apple", "banana", "mango"]; // [&str; 3]  
    let result = fing_biggest_item(&strings); // & [&str; 3] ==> & [&str]  
    if let Some(value) = result {  
        println!("Biggest item is : {}", value)  
    } else {  
        println!("Array was empty");  
    }  
}
```



The ‘Result’ enum type

Module std::result

1.0.0 · source · [-]

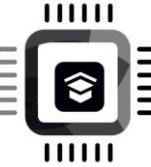
[–] Error handling with the `Result` type.

`Result<T, E>` is the type used for returning and propagating errors. It is an enum with the variants, `Ok(T)`, representing success and containing a value, and `Err(E)`, representing error and containing an error value.

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

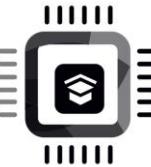
Functions return `Result` whenever errors are expected and recoverable. In the `std` crate, `Result` is most prominently used for I/O.

<https://doc.rust-lang.org/std/result/>



Rust enum variant naming

- In Rust, enum variants are not standalone types
- Generally, you need to specify the enum name followed by the variant name
(*MyEnum::VariantName*)
- This is because enum variants only have meaning in the context of their containing enum.



'Option' and 'Result' Shorthand

- Rust provides a built-in shorthand for the ***Option*** and ***Result*** types.
- You can use ***Some(value)*** instead of ***Option::Some(value)*** and ***Ok(value)*** instead of ***Result::Ok(value)***.
- This shorthand makes code using these types much cleaner and easier to read.
- **Option<T>** and **Result<T, E>** is included in the Rust prelude, which means it's automatically available in all Rust programs without needing to bring it into scope explicitly, making it easier for developers to use this useful enum.

The '**prelude**' is the list of things
that Rust automatically imports
into every Rust program