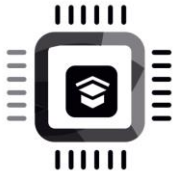
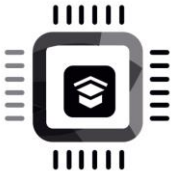


Master The Rust Programming Language : Beginner To Advanced



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-II

This presentation contains slides for the below sections

13) Vectors

14) HashMap

15) Error Handling

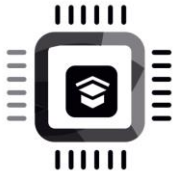
16) Generics

17) Lifetimes

18) Const. and Static

19) Traits

20) Closures



Vectors

Vectors

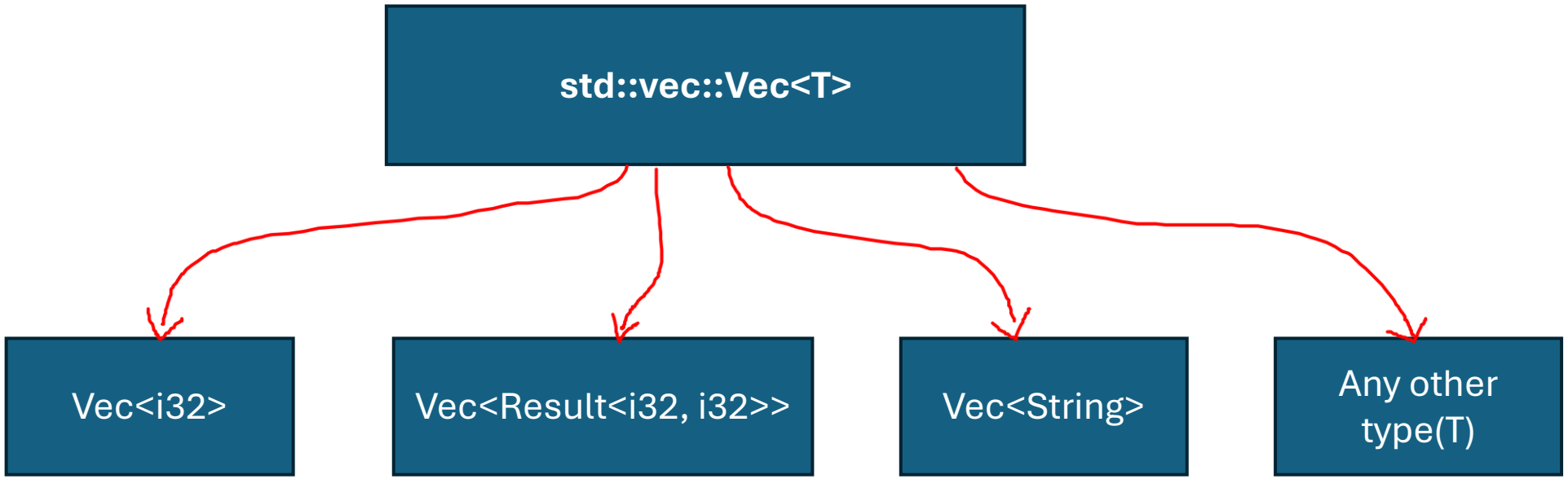
- A growable list type that stores elements contiguously in memory.
- Vectors are growable and shrinkable at run time, which means you can modify their size at runtime by adding or removing elements. (Remember arrays in Rust ([T; N]) have a fixed size that's determined at compile time?)
- Supports random access, push, pop, and other list-like operations

Vector in standard library

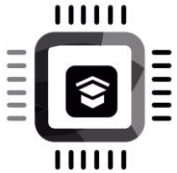
Vectors are defined in the **std::vec** module. The primary struct for vectors is **std::vec::Vec<T>,** where T is the type of elements the vector contains.

Vec<**i32**>
Vec<**f32**>
Vec<**String**>
Vec<**Option<i32>>**

Vector is implemented as a Generic Structure in ‘std’ crate



You can use vectors to hold elements of any type, as long as all elements in a vector are of the same type



Important methods to be used with Vec

Create and initialize a vector

```rust
fn main() {
    // Create an empty vector of i32 type
    let mut v = Vec::new();

    // Push elements into the vector
    v.push(1);
    v.push(2);
    v.push(3);
    v.push(4);
    v.push(5);

    // Iterate over the vector and print each element
    for i in v {
        println!("{}", i);
    }
}
```

The **vec!** syntax in Rust is a macro that allows you to easily create and initialize a vector

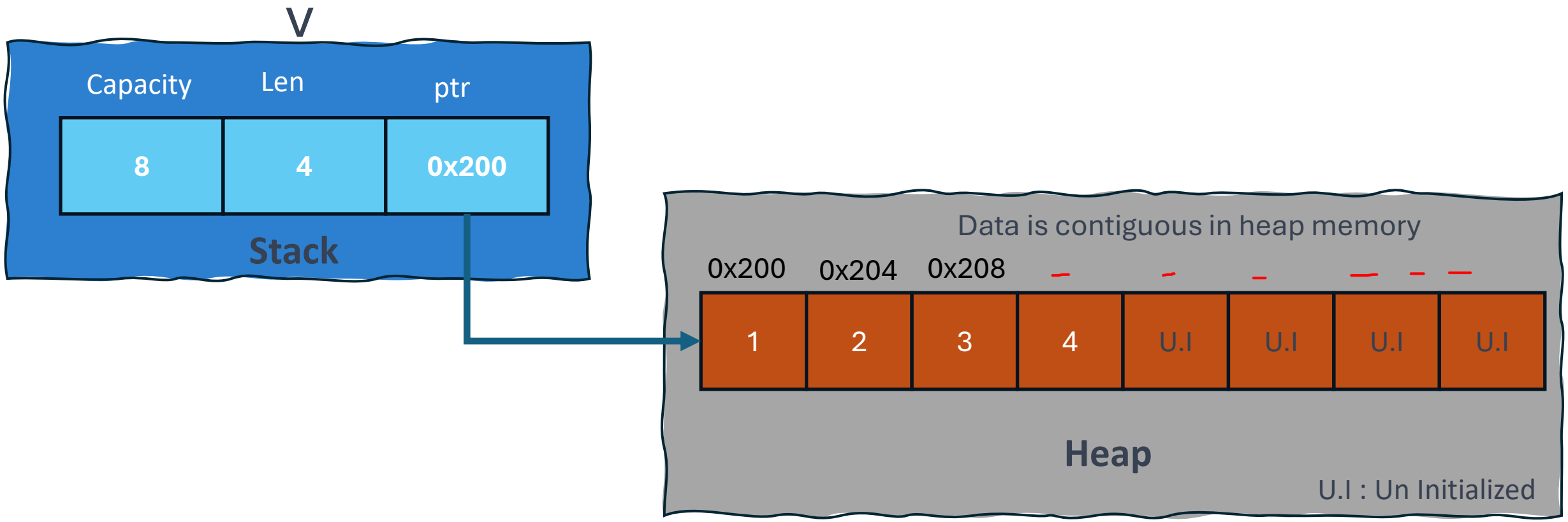
```rust
let v = vec![1, 2, 3, 4, 5];
```

Array to Vec

```rust
fn main() {
    // Define an array
    // type of arr : [i32; 4]
    let arr = [1, 2, 3, 4];

    // Convert the array to a vector
    // type of vec1 : Vec<i32>
    let vec1 = arr.to_vec();
    let _vec2 = Vec::from(arr);
    let _vec3 = Vec::from([1, 2, 3, 4]);

    // Print the vector
    println!("{:?}", vec1);
}
```

Vector under the hood

```
let v = vec![1, 2, 3, 4];
```



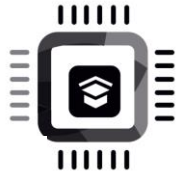
Method : Vec::with_capacity();

When to use?:
When you have an estimation or knowledge about the number of elements the vector will hold, this can be especially beneficial in performance-critical sections of code where minimizing memory reallocations can make a noticeable difference.

```
let mut v = Vec::with_capacity(10);
```

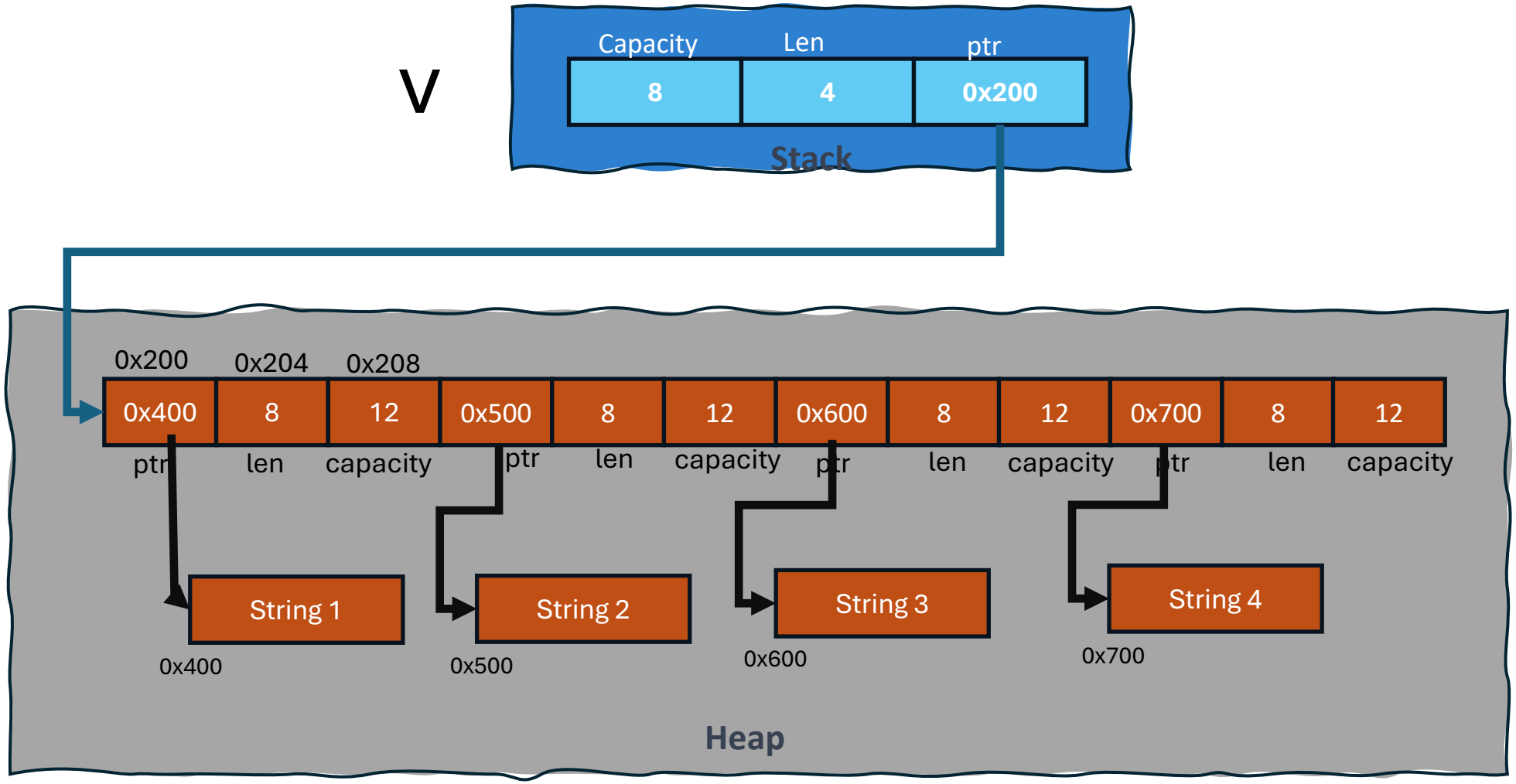
```rust
fn main() {
    let mut v = vec![1, 2, 3, 4];
    let initial_ptr = v.as_ptr();
    let initial_capacity = v.capacity();

    println!("Initial heap address: {:p}", initial_ptr);
    println!("Initial capacity: {}", initial_capacity);

    // Pushing another element to force a potential reallocation
    v.push(5);
    let after_push_ptr = v.as_ptr();
    let after_push_capacity = v.capacity();

    println!("Heap address after push: {:p}", after_push_ptr);
    println!("Capacity after push: {}", after_push_capacity);

    if initial_ptr == after_push_ptr {
        println!("Memory address did not change.");
    } else {
        println!("Memory address changed, indicating reallocation.");
    }
}
```

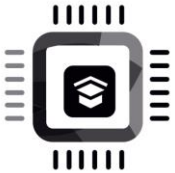
```
Initial heap address: 0x556b5aebb9d0
Initial capacity: 4
Heap address after push: 0x556b5aebb9d0
Capacity after push: 8
Memory address did not change.
```



```rust
fn main() {
    let mut v = Vec::with_capacity(4);  // Start with a capacity of 4

    for i in 1..=4 {
        v.push(format!("String {}", i));
    }

    // At this point, the vector's length equals its capacity
    let vector_base_addr_in_heap_1 = v.as_ptr();
    println!("Heap address (capacity {}): {:p}", v.capacity(), vector_base_addr_in_heap_1);

    // Now, we push another string, which should trigger reallocation
    v.push("Another String".to_string());
    let vector_base_addr_in_heap_2 = v.as_ptr();
    println!("Heap address (capacity {}): {:p}", v.capacity(), vector_base_addr_in_heap_2);

    if vector_base_addr_in_heap_1 == vector_base_addr_in_heap_2 {
        println!("No reallocation occurred.");
    } else {
        println!("Reallocation occurred.");
    }
}
```

Memory layout of Vec of Strings



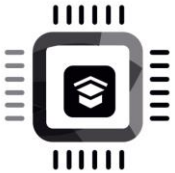


Is vector copy or move?

Vectors (Vec<T>) in Rust are move-by-default and do not implement the Copy trait. This is because vectors own heap-allocated memory and copying them requires a deep copy of the underlying data, which is done by the Clone trait.

Vector indexing

```
fn main() {
    let v = vec![1, 2, 3];

    println!("{}",v[0]);
    // This will panic at runtime with an 'index out of bounds' error
    println!("{}",v[5]);

}
```



Safe to vector indexing

The safest way to access elements in a Vec is using the **get** and **get_mut** methods. They return an Option<&T> or Option<&mut T>, respectively, instead of panicking when faced with out-of-bounds indices.

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

    // Using get
    // val_ref is of type Option<&i32>
    let val_ref = vec.get(1);
    if let Some(val) = val_ref {
        // Dereferencing the reference to get the actual value
        println!("Value: {}", *val);
    }

    // Using get_mut
    // val_mut_ref is of type Option<&mut i32>
    let val_mut_ref = vec.get_mut(2);
    if let Some(val) = val_mut_ref {
        // Dereferencing and modifying the value in place
        *val *= 10;
    }

    println!("{:?}", vec);

}
```

Both **get()** and **get_mut()** return references, but they differ in their mutability:

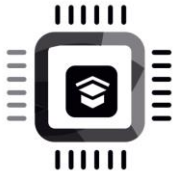
1. **get()**: Returns an immutable reference (Option<&T>).

2.**get_mut()**: Returns a mutable reference (Option<&mut T>)

Neither of these methods gives direct ownership of the value. Instead, they give you a reference to the value, allowing you to either read it (with get) or modify it (with get_mut).

Slice of a vector

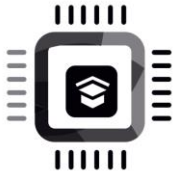
```
let vec = vec![1, 2, 3, 4, 5];


let slice = &vec[1..4]; // A slice containing [2, 3, 4]
```



```rust
let vec = vec![0, 1, 2, 3, 4];

// a slice containing all the elements: [0, 1, 2, 3, 4]
let a = &vec[..];

// a slice starting from the second element: [1, 2, 3, 4]
let b = &vec[1..];

// a slice of the first three elements: [0, 1, 2]
let c = &vec[..3];

// a slice starting from the second and up to the third element: [1, 2]
let d = &vec[1..3];
```

Iterating

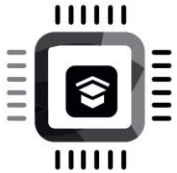
- by value
- by reference



Iterating by value

When iterating through a vector using .into_iter() or consuming it in a for loop, the elements are moved out of the vector.

This means that for non-Copy trait types, such as String, the original vector cannot be used once the values have been moved out.



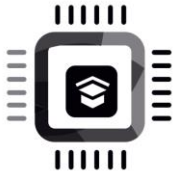
```rust
fn main() {
    let vec_of_strings = vec![
        String::from("first"),
        String::from("second"),
        String::from("third"),
    ];

    // Iterating over the vector by reference, borrowing each String
    for s in &vec_of_strings {
        println!("{}", s);
    }

    // Now this line will work, because vec_of_strings was not moved.
    println!("{:?}", vec_of_strings);
}
```

Using &vec_of_strings in the for loop allows you to iterate over references to the String values inside the vector, rather than the values themselves.

push()

The push method is used to append an element to the end of a Vec. It will increase the length of the Vec by one

```
fn main() {
    let mut numbers = Vec::new();

    numbers.push(1);
    numbers.push(2);
    numbers.push(3);

    println!("{:?}", numbers);  // Outputs: [1, 2, 3]
}
```

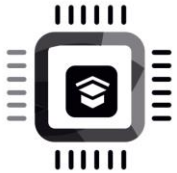


pop()

When you call pop() on a Vec, it returns an **Option<T>** where **T** is the type of elements stored in the vector.

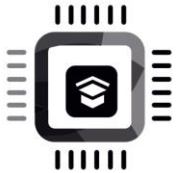
If the vector is not empty, pop() removes the last element, returns **Some(value),** and decreases the vector's length by one. If the vector is empty, pop() returns **None**.

Does the Vec shrink with every pop()?
In terms of its length (len()), but not in terms of its capacity (capacity())



shrink_to_fit()

If you want to reduce the capacity of a Vec to match its length, you can use the shrink_to_fit() method



drain()

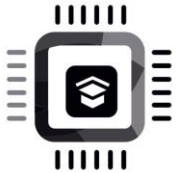
The drain method on a Vec takes a single parameter which is a range. This range specifies the beginning and end of the section of the vector to be drained.

```
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    let _  = v.drain(1..3); // This will drain values 2 and 3
    println!("{:?}", v);
}
```

drain()

Use case scenarios?

1. Remove a range of elements from a vector without deallocating the entire vector
2. Move a subset of elements from one vector to another
3. Remove elements one at a time from a vector
4. Removing specific elements, such as all elements greater than 5

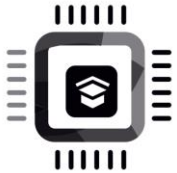


drain() and collect()

```rust
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];
    // This will drain values 2 and 3
    let v_drain: Vec<_>  = v.drain(1..3).collect();
    println!("{:?}", v_drain);
}
```

drain() and extend()

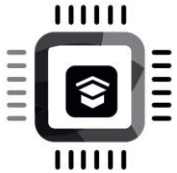
```
fn main() {
    let mut v1 = vec![1, 2, 3, 4, 5];
    let mut v2: Vec<i32> = Vec::new();
    v2.extend(v1.drain(2..));
    println!("v1: {:?} and v2: {:?}", v1, v2);
}
```



Drain specific elements (extract_if)

experimental API you have to use #![feature(extract_if)]

retain() and retain_mut()



It allows you to keep only the elements in the vector that satisfy a given predicate while removing all others. In essence, it's an in-place filter for a Vec.

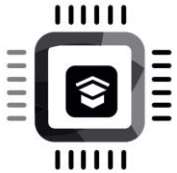
```
fn main() {
    let mut numbers = vec![-3, -2, -1, 0, 1, 2, 3];

    // This will retain only positive numbers in the Vec
    numbers.retain(|x| *x > 0);

    println!("{:?}", numbers); // Prints: [1, 2, 3]
}
```

Retains only positive elements in a vector.

Methods related to split

Splitting vectors in various ways

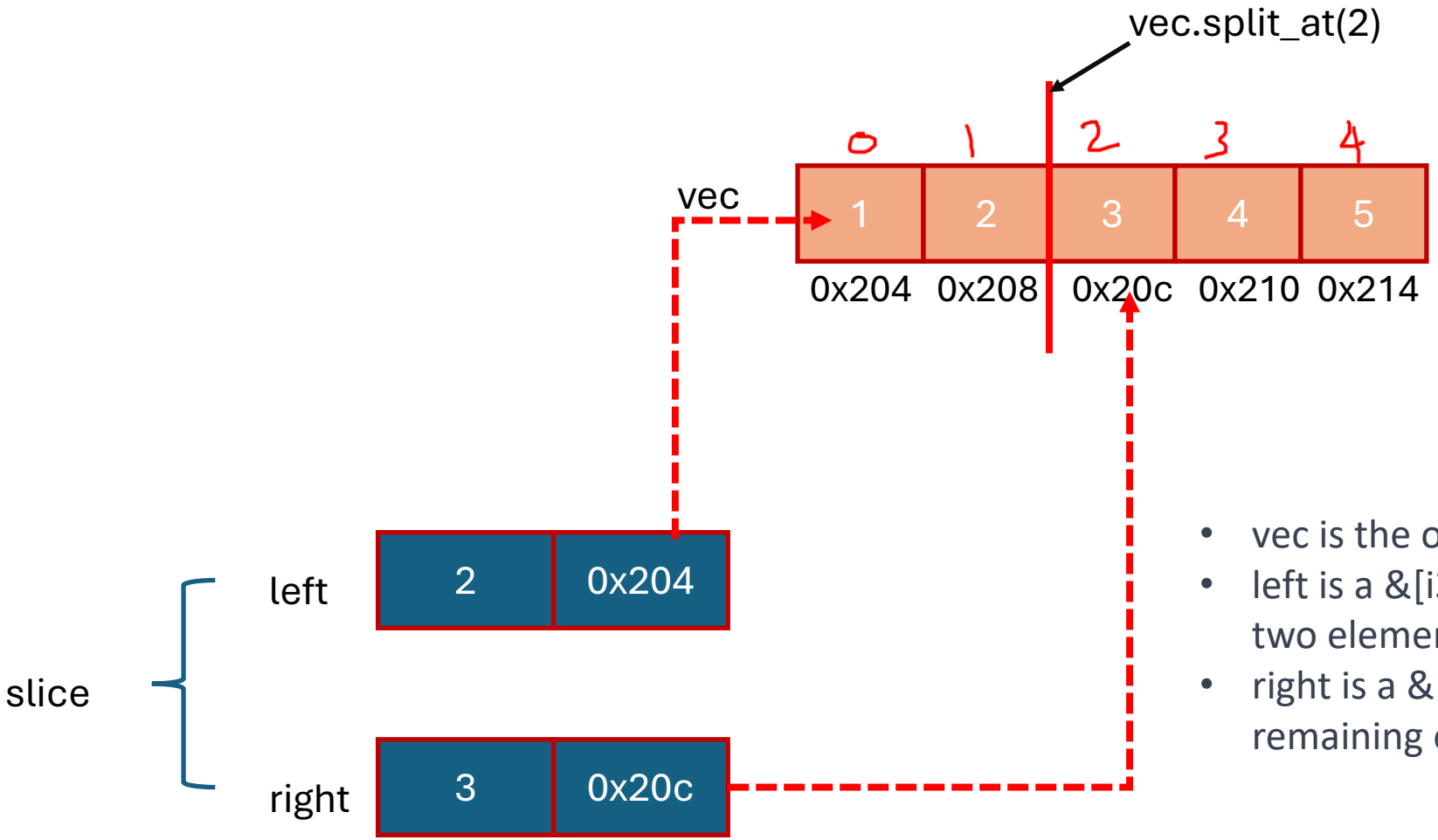
- split_at()
- split_at_mut()
- split()
- rsplit()
- splitn()
- rsplitn()



split_at()

Divides one slice into two at an index.

```
fn main() {
    let vec = vec![1, 2, 3, 4, 5];
    let (left, right) = vec.split_at(2);
    println!("Left: {:?}", left);   // Left: [1, 2]
    println!("Right: {:?}", right); // Right: [3, 4, 5]
}
```

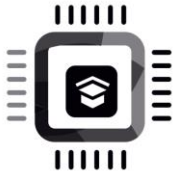


- vec is the owned Vec<i32>.
- left is a &[i32] slice referencing the first two elements of vec.
- right is a &[i32] slice referencing the remaining elements of vec

split()

The split method creates an iterator over **subslices** separated by elements matching a predicate. It's a way to tokenise or divide a slice based on a given condition.

- Note: The matched element is not contained in the subslices



```
let vec = vec![1, 2, 3, 7, 11, 4, 33, 67, 8, 10]
```

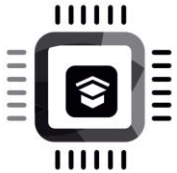
Condition: Split when an element is even.

[1], 2, [3, 7, 11], 4, [33, 67], 8, 10

```rust
fn main() {
    let vec = vec![1, 2, 3, 7, 11, 4, 33, 67, 8, 10];
    let subslices = vec.split(|e| e % 2 == 0);
    for slice in subslices {
        println!("{:?}", slice);
    }
}
```

Whenever the closure returns true for an element, that element is treated as a split point, and the original slice is divided at that point (excluding the element itself).

splitn()



rsplit()

- It's quite similar to ***split***, but with a difference: while split iterates from the beginning to the end of the slice, ***split*** iterates from the end to the beginning.
- This can be useful when you want to split a sequence based on a predicate but you're interested in working with the elements in reverse order.

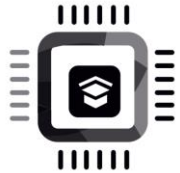
split_off()

If you want to divide a ***Vec*** into two parts at a certain index and intend to use both parts independently afterwards.

```rust
fn main() {
    // Initialize a vector with numbers from 1 to 5.
    let mut vec = vec![1, 2, 3, 4, 5];

    // Split off the vector from index 3 (0-based), so elements [4, 5] will be transferred
    let split_vec = vec.split_off(3);

    // Print both vectors after the split.
    println!("Original vector: {:?}", vec);          // Original vector: [1, 2, 3]
    println!("Split-off vector: {:?}", split_vec);   // Split-off vector: [4, 5]
}
```

splice()

The ***splice*** method is used to replace a specified range of elements in a vector with elements from an iterator.



```rust
fn main() {
    // Primary temperature readings (in Celsius) over the last 10 hours
    let mut primary_readings = vec![22.5, 23.0, 22.8, 0.0, 0.0, 0.0, 23.2, 22.9, 22.4, 22.0];

    // Backup readings for hours 4 to 6
    let backup_readings = vec![22.7, 22.6, 23.0];

    // This will not compile because `splice` expects an iterator
    let faulty_readings: Vec<_> = primary_readings.splice(3..6, backup_readings).collect();

    println!("Corrected primary readings: {:?}", primary_readings);
    println!("Faulty readings: {:?}", faulty_readings);
}
```

append() and extend()

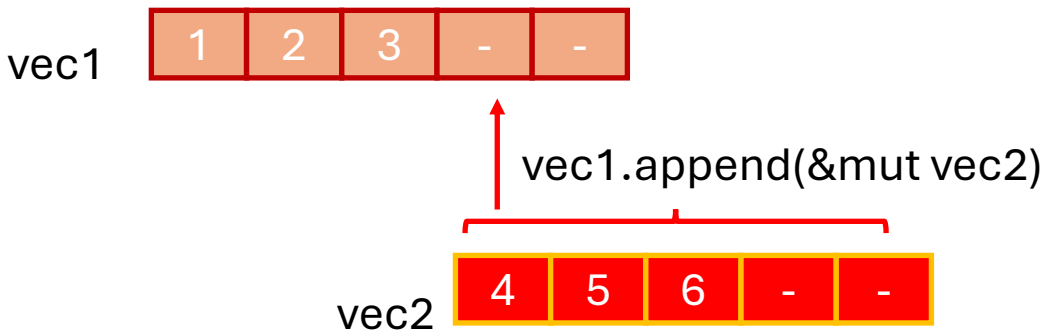
append():

Takes all elements from one vector and appends them to another. The source vector is left empty after this operation.

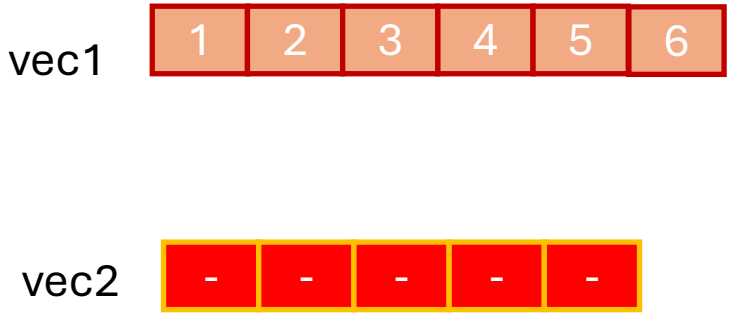
Note :

When you use the append method, the elements from vec2 are moved to vec1, but the capacity of vec2 doesn't automatically shrink. Therefore, vec2 will have a length of 0, but its capacity remains unchanged. This means that the underlying memory allocation of vec2 remains, even though it's now empty.

append()



Before



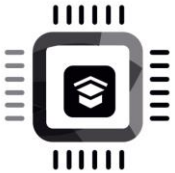
After

extend()

Takes an iterator and extends the vector with elements from that iterator

Note:

- extend() does not empty the source (if you pass a vector or any other collection). It either consumes items (for consuming iterators) or clones/copies items (for cloning iterators).

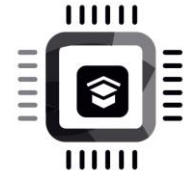


```rust
fn main() {
    let mut vec = vec![1, 2, 3];
    let vec1 = vec![-23, 45];
    let arr = [11, 22, 33];
    let vec2 = vec![44, 55, 66];

    vec.extend(&[4, 5]);            // extend from a slice
    vec.extend(6..9);               // extend from a range
    vec.extend(vec![9, 10, 11]);    // extend from another vector
    vec.extend(vec1);                // extend from another vector
    vec.extend(&arr);                // extend from an array

    // Using a consuming iterator from vec2 to extend vec.
    vec.extend(vec2.into_iter());

    println!("{:?}", vec);          // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

    // The next line would be an error because vec2 has been consumed.
     println!("{:?}", vec2);
}
```

Is comparing vectors possible?

- Vec<T> implements the PartialEq trait as long as the type T stored in the vector also implements PartialEq

- If the elements inside the vectors also implement the PartialOrd trait (like i32, f32, String, etc.), you can also compare vectors lexicographically (similar to how words are sorted in a dictionary)



```rust
fn main() {
    let vec1 = vec![1, 2];
    let vec2 = vec![1, 2, 3];
    if vec1 < vec2 {
        println!("vec1 is less than vec2");
    } else {
        println!("vec2 is less than vec1")
    }

}
```

Prepend

- A Vec<T> in Rust grows unidirectionally, meaning it expands or contracts only at its end. The starting address of the vector remains fixed, ensuring that the address of the first element doesn’t change, even as the vector’s capacity increases.

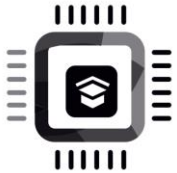
- Although direct prepend operation isn't supported, you can insert elements at the beginning of a Vec<T> using the insert method, effectively achieving a prepend operation.

insert()

Use the insert method on a Vec<T> to insert an element at a specified index, effectively pushing all the subsequent elements towards the tail. If you use insert with an index of 0, it will prepend the element to the vector:

```
fn main() {
    let mut vec = vec![1, 2, 3];
    vec.insert(0, 25);
    println!("{:?}", vec); // Outputs: [25, 1, 2, 3]
}
```

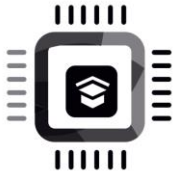
The time complexity is O(n), where **n** refers to the number of elements to the right of the specified index where the new element is being inserted



Prepend using splice

- Splice() can be used to prepend one vector with another vector's elements.

```
fn main() {
    let mut main_vec = vec![4, 5, 6];
    let vec_to_prepend = vec![1, 2, 3];

    main_vec.splice(0..0, vec_to_prepend);

    println!("{:?}", main_vec);  // [1, 2, 3, 4, 5, 6]
}
```

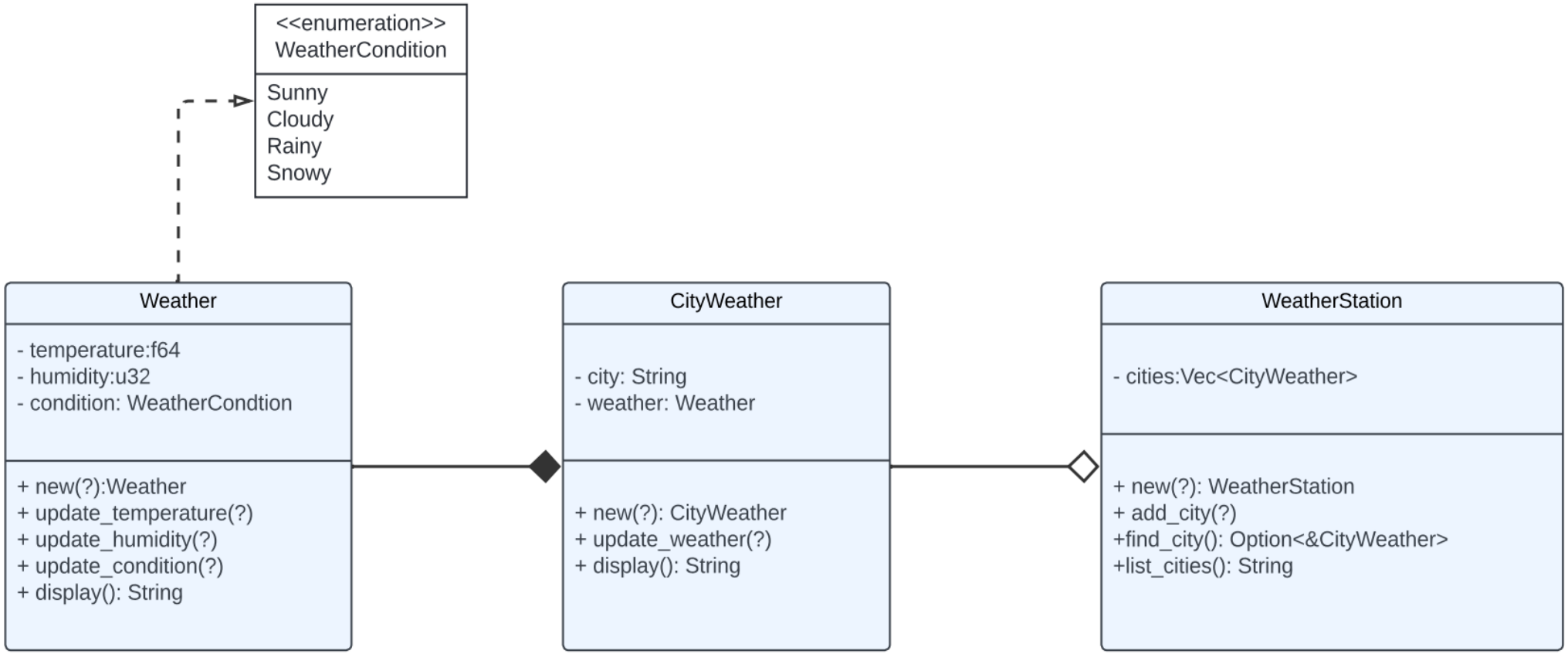


Exercise: Build a simple weather station application

Write a program to manage weather data for different cities.

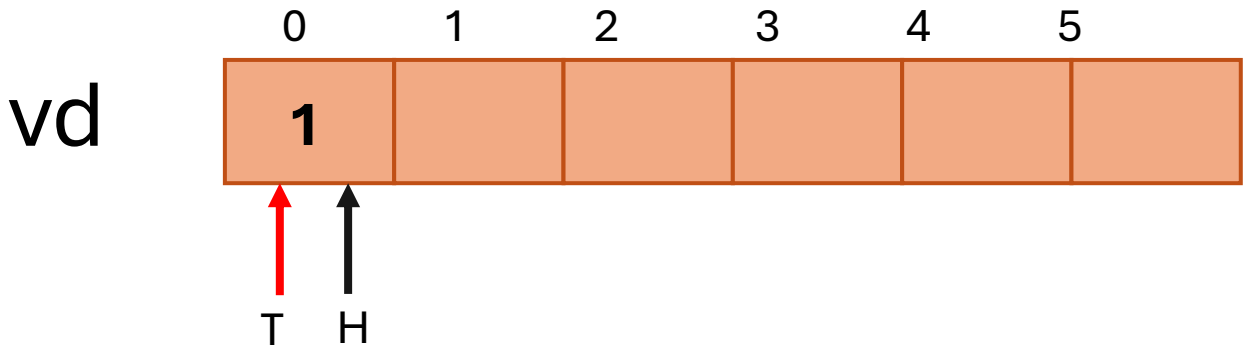
Hints: Check the demo video , UML diagrams and partial code shared with this lecture

UML(Unified Modeling Language] Class diagrams

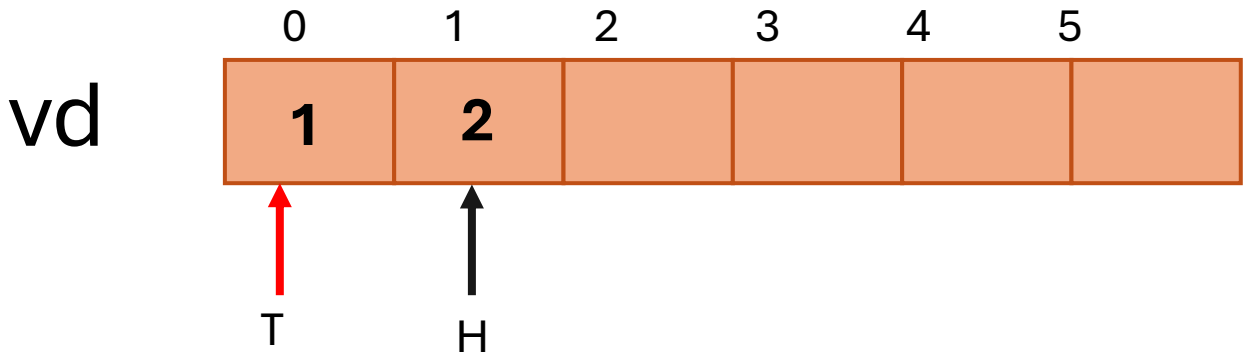


VecDeque<T>

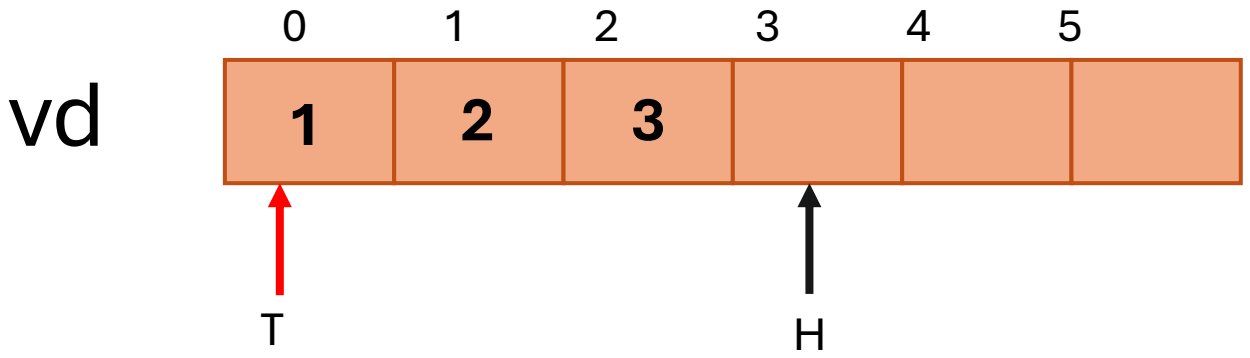
- VecDeque stands for “Vector Deque" where “Deque" stands for “Double-ended queue.“
- A double-ended queue is implemented with a growable ring buffer.
- Particularly suitable for use cases that require a first-in-first-out (FIFO) queue because it allows for efficient pushing and popping of elements at both ends.



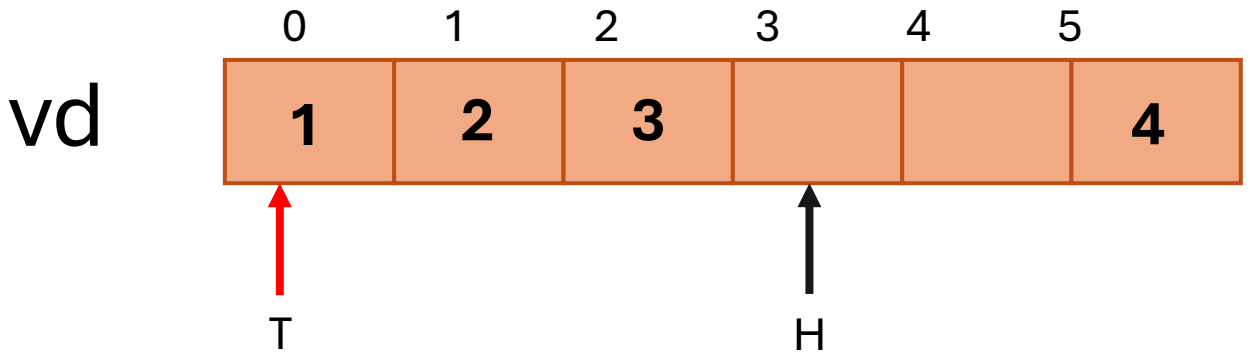
vd.push_back(1)



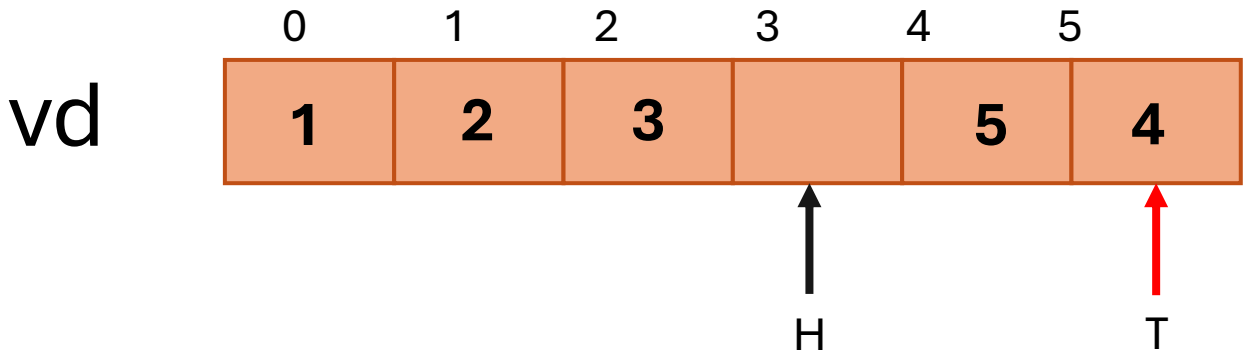

```
vd.push_back(1)
vd.push_back(2)
```




```
vd.push_back(1)
vd.push_back(2)
vd.push_back(3)
```




```
vd.push_back(1)
vd.push_back(2)
vd.push_back(3)
vd.push_front(4)
```



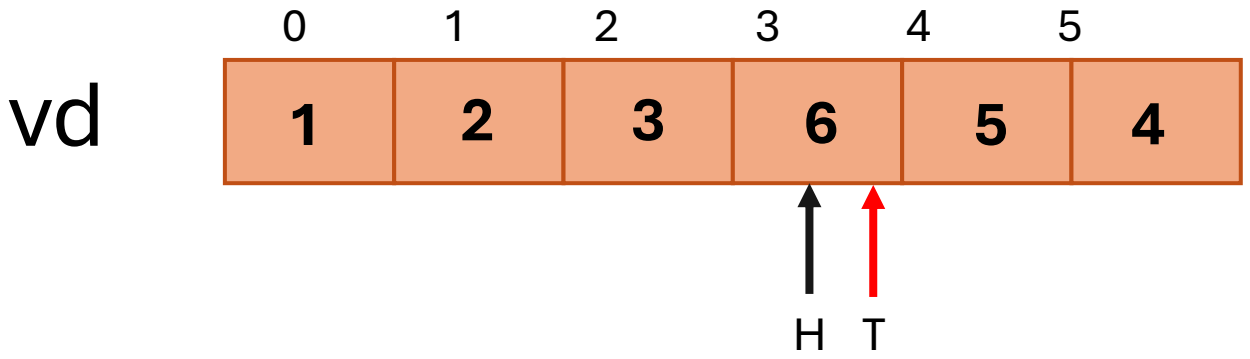
vd.push_back(1)

vd.push_back(2)

vd.push_back(3)

vd.push_front(4)

vd.push_front(5)



vd.push_back(1)

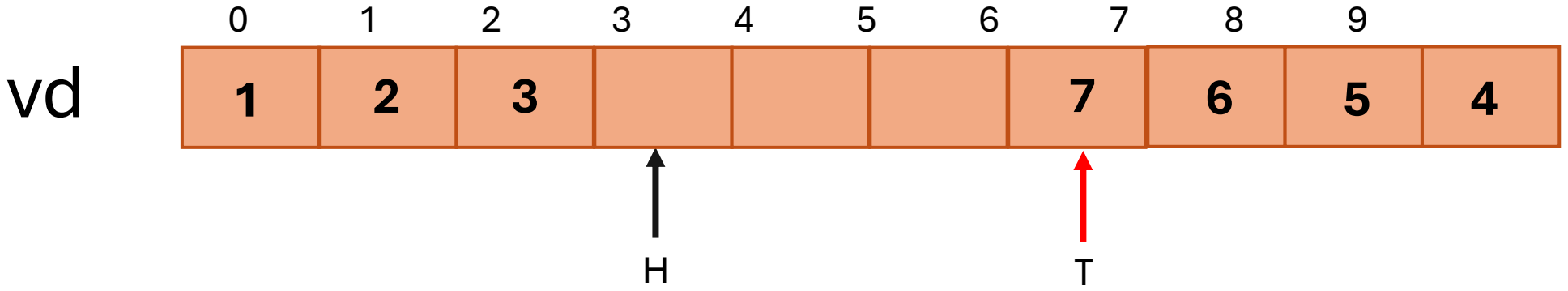
vd.push_back(2)

vd.push_back(3)

vd.push_front(4)

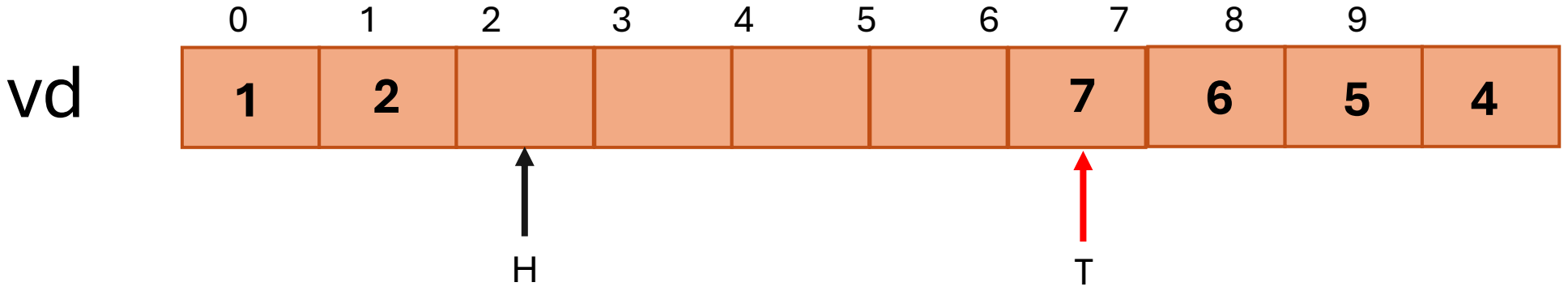
vd.push_front(5)

vd.push_front(6)

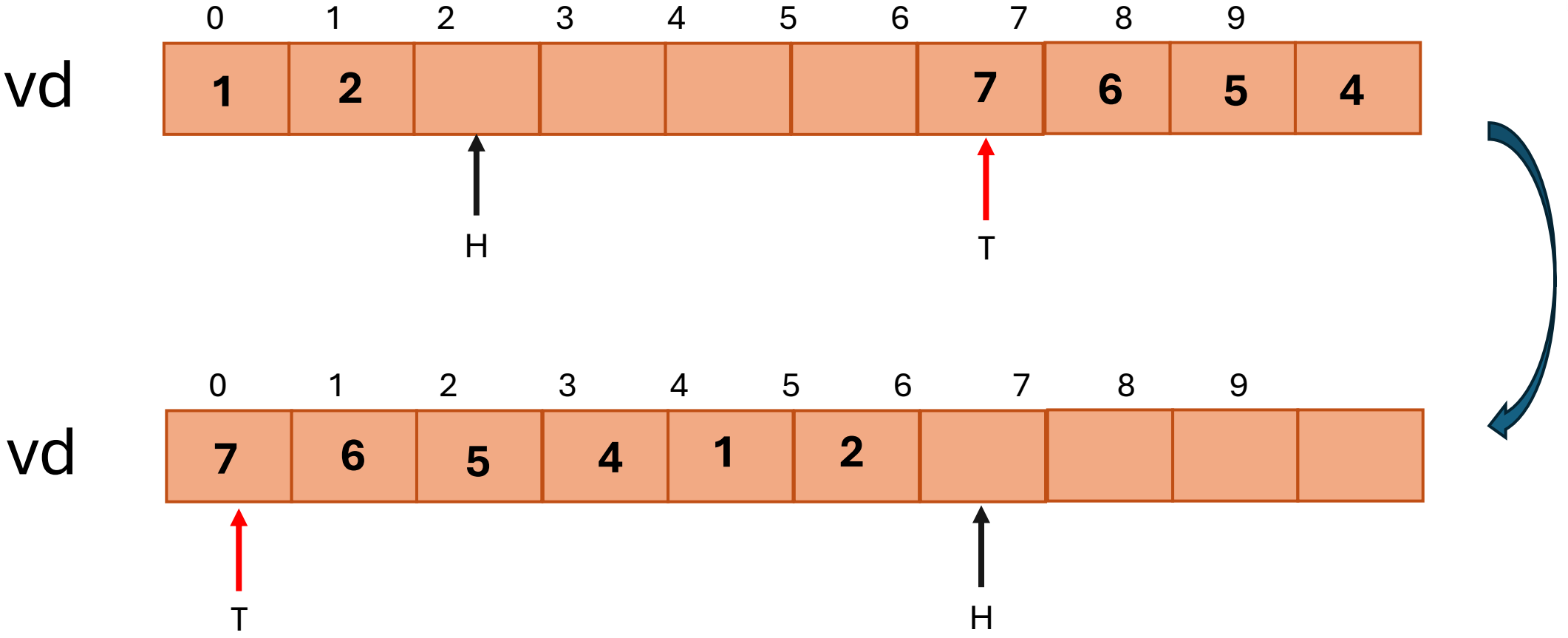


vd.push_back(1)
vd.push_back(2)
vd.push_back(3)
vd.push_front(4)

vd.push_front(5)
vd.push_front(6)
vd.push_front(7)

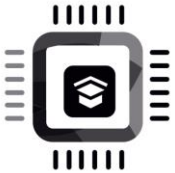


vd.push_back(1)
vd.push_back(2)
vd.push_back(3)
vd.push_front(4)
vd.push_front(5)
vd.push_front(6)
vd.push_front(7)
vd.pop_back(7)



VecDeque<T> vs Vec<T>

1. **Design:** VecDeque is designed to be a double-ended queue.
2. **Insertion Efficiency:** VecDeque supports efficient insertion at both the front and back, while Vec supports efficient insertion at the back.
3. **Removal Efficiency:** VecDeque supports efficient removal from both the front and back, while Vec supports efficient removal from the back.
4. **Access Efficiency:** Both VecDeque and Vec offer O(1) complexity for direct access/indexing.
5. **Insertion/Removal in the Middle:** Both have O(n) complexity for inserting or removing elements in the middle. However, VecDeque can sometimes be more efficient if the operation is closer to the ends because it can choose the end that requires fewer elements to be moved.



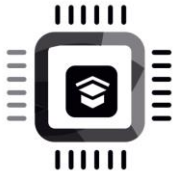
HashMap

HashMap

- A **HashMap** stores a collection of **key-value** pairs, and you can access the **value** associated with a given **key** in constant time.

Working with HashMap

1. Creating a new HashMap
2. Inserting key-value pair
3. Accessing value by key
4. Iterating over HashMap



```rust
use std::collections::HashMap;

fn main() {
    // Create a new HashMap
    let mut fruit_prices = HashMap::new();

    // Insert key-value pairs (fruit name as key and price as value)
    fruit_prices.insert("apple", 1.2);
    fruit_prices.insert("banana", 0.8);
    fruit_prices.insert("cherry", 2.5);

    // Access the price of a specific fruit
    let apple_price = fruit_prices.get("apple").unwrap_or(&0.0);

    // Print the price
    println!("The price of an apple is ${}", apple_price);

    // Iterate over all entries and print them
    for (fruit, price) in &fruit_prices {
        println!("{}: ${}", fruit, price);
    }
}
```

Indexing vs get

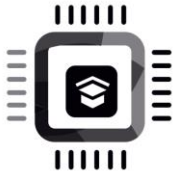
```rust
use std::collections::HashMap;

fn main() {
    // Create a HashMap to store fruit prices.
    let mut fruit_prices = HashMap::new();

    // Insert some fruits with their prices.
    fruit_prices.insert("apple", 1.20);
    fruit_prices.insert("banana", 0.50);
    fruit_prices.insert("cherry", 2.50);

    // Accessing values using the indexing syntax.
    // This will panic if the key is not present.
    println!("Price of an apple: ${}", fruit_prices["apple"]);
    println!("Price of a banana: ${}", fruit_prices["banana"]);

    // Safe way to access values: using the get method.
    match fruit_prices.get("orange") {
        Some(price) => println!("Price of an orange: ${}", price),
        None => println!("Orange price doesnt exist"),
    }

}
```

Iterating over HashMap

```
for (fruit, price) in &fruit_prices {
    println!("{}: ${}", fruit, price);
}
```

Iterating over *references* to the key-value pairs in the HashMap. This avoids transferring ownership

During each iteration, the for loop retrieves the next (key, value) tuple from the HashMap and destructures it by matching the tuple against the (fruit, price) pattern.



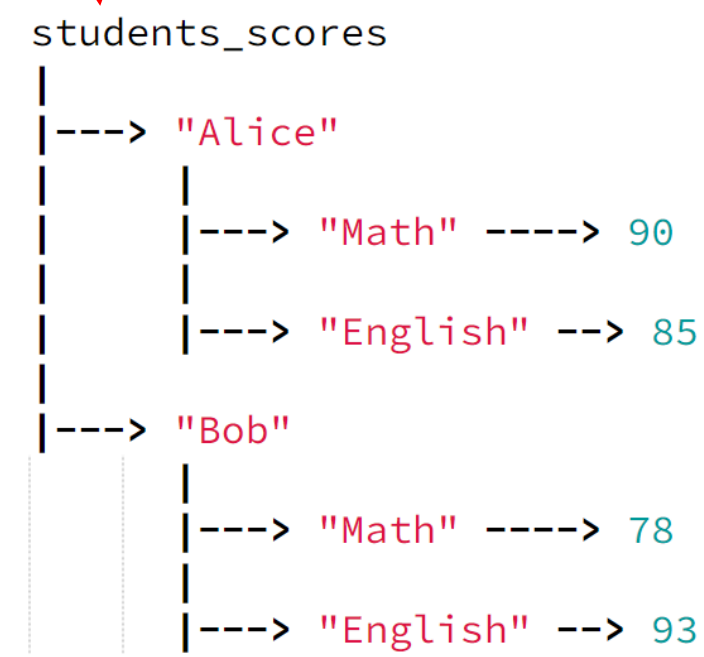
Iterating over keys and values

The keys() and values() methods on a HashMap return iterators over the map's keys and values, respectively.

Nested HashMaps

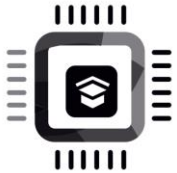
‘value’ of a HashMap could be another HashMap

```
let mut students_scores: HashMap<String, HashMap<String, i32>> = HashMap::new();
```



Student	Subject	Score
Alice	Math	90
Alice	English	85
Bob	Math	78
Bob	English	93

Implement a nested HashMap in Rust to store and display student scores for various subjects in a table-like format.



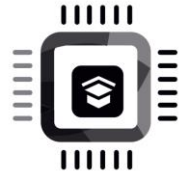
Methods of HashMap

- **new()**: Creates a new, empty HashMap.
- **insert()**: Inserts a key-value pair Into the map
- **get()**: Returns a reference to the value corresponding to the key
- **contains_key()**: Checks if a key is present in the map
- **remove()**: Removes a key from the map
- **len()**: Returns the number of elements in the map.
- **is_empty()**: Returns true if the map contains no elements.
- **clear()**: Clears the map, removing all key-value pairs.
- **entry()**: Gets the given key’s corresponding entry in the map for in-place manipulation
- **iter()**: Returns an iterator over the map's key-value pairs.
- **keys()** and **values()**: Return iterators over the map's keys or values.

entry()

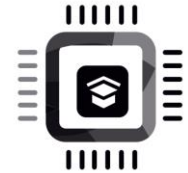
- It's useful for scenarios where you want to modify a value for an existing key or insert a new **key-value** pair if the key doesn't exist. (in-place manipulation)

If "apple" is not in the HashMap, add the key "apple" with a value of 1
If "apple" is already in the HashMap, increment the value by 1



```
*fruits.entry("apple").or_insert(0) += 1;
```

```rust
use std::collections::HashMap;

fn main() {
    let mut fruits = HashMap::new();

    if fruits.contains_key("apple") {
        // If "apple" exists in the map, get its value and increment it by 1
        let count = fruits.get_mut("apple").unwrap();
        *count += 1;
    } else {
        // If "apple" doesn't exist in the map, insert it with a value of 1
        fruits.insert("apple", 1);
    }

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



```rust
use std::collections::HashMap;

fn main() {
    let mut counts = HashMap::new();
    *counts.entry("apple").or_insert(0) += 1;
    println!("{:?}", counts)
}
```

‘Entry’ enum

Methods:

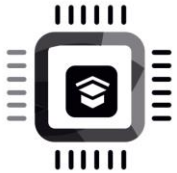
or_insert()

or_insert_with()

and_modify()

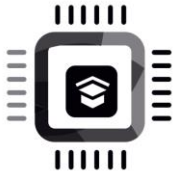
or_insert()

- If the entry is ***Vacant***, this method inserts the given value for the key and returns a mutable reference to the value. If the entry is ***Occupied***, it returns a mutable reference to the existing value.



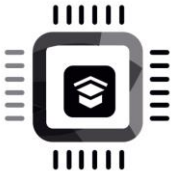
or_insert_with()

Similar to *or_insert()*, but *or_insert_with()* method expects a closure that returns the value to be inserted if the key doesn't exist in the HashMap.



Note:

- When you use ***or_insert(expensive_computation()),*** the *expensive_computation()* function is executed *before* the ***or_insert()*** is called, regardless of whether the ***key*** exists in the map or not. So, every time this line runs, *expensive_computation()* is called.
- On the other hand, with ***or_insert_with(expensive_computation),*** the *expensive_computation()* function is executed only when the ***key*** does not already exist in the map. This is because ***or_insert_with()*** accepts a closure and only invokes it when needed.

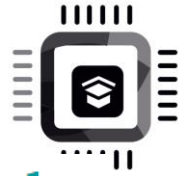


```rust
use std::collections::HashMap;

fn expensive_computation() -> i32 {
    println!("Expensive computation called!");
    //simulate some complex calculations

    42
}


fn main() {
    let mut data = HashMap::new();
    data.insert("key", 10);

    //or_insert_with()
    data.entry("key").or_insert_with(|| expensive_computation());
    println!("HashMap contents: {:?}", data);
}
```

and_modify()

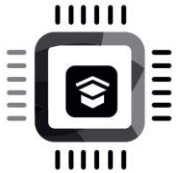
- The ***and_modify()*** method on the **Entry** enum is useful for modifying the ***value*** associated with a ***key*** in a HashMap when that ***key*** exists. It allows for in-place manipulation of the value without needing to remove the key-value pair and then reinsert it

- Use ***and_modify()*** in conjunction with methods like ***or_insert()*** to handle both cases:
  - When a key is present (modify its value) and when it's not (insert a default value).

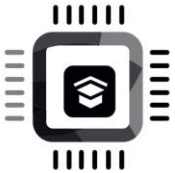


```
*counts.entry("apple").or_insert(0) += 1;
```

```rust
use std::collections::HashMap;

fn main() {
    let mut fruits = HashMap::new();

    if fruits.contains_key("apple") {
        // If "apple" exists in the map, get its value and increment it by 1
        let count = fruits.get_mut("apple").unwrap();
        *count += 1;
    } else {
        // If "apple" doesn't exist in the map, insert it with a value of 1
        fruits.insert("apple", 1);
    }

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



```rust
use std::collections::HashMap;

fn count_words(text: &str) -> HashMap<&str, u32> {
    let mut word_counts = HashMap::new();

    for word in text.split_whitespace() {
        word_counts.entry(word).and_modify(|count| *count += 1).or_insert(1);
    }

    word_counts
}

fn main() {
    let text = "this is a sample text this text is just a sample";
    let counts = count_words(text);

    for (word, count) in &counts {
        println!("{}: {}", word, count);
    }
}
```



Error Handling

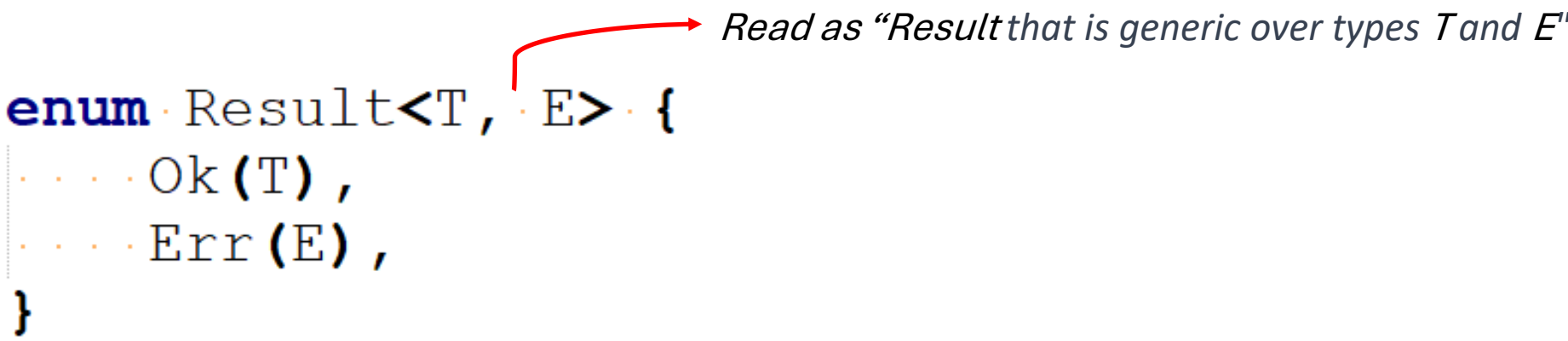
Error handling in Rust

Rust offers several mechanisms for programmers to handle errors in a safe and efficient manner.

1. Result Enum
2. Option Enum
3. Panic macro
4. unwrap() and expect() methods
5. ? Operator

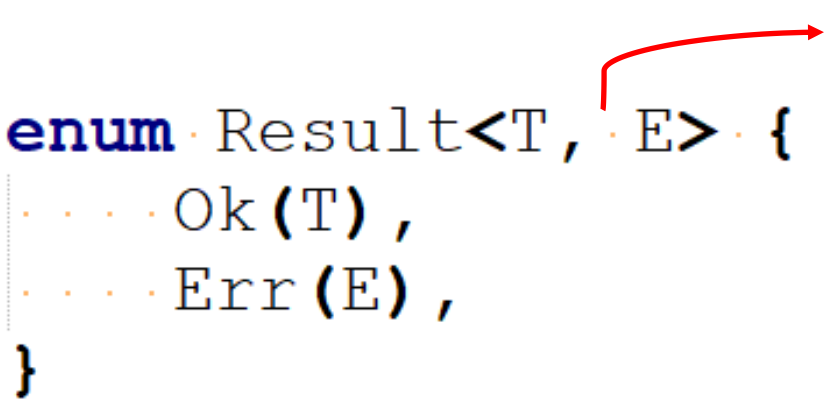
The ‘Result’ enum

The Result enum is a built-in type in Rust that represents the outcome of a computation that can succeed or fail.



- The **Result** type has two possible variants: **Ok** and **Err**.
- The **Ok** variant represents a successful result and contains a value of type T, which is the type of the value that the operation returns if it succeeds.
- The **Err** variant represents a failed result and contains a value of type E, which is the type of the error that the operation returns if it fails.

The ‘Result’ enum



- The **Result** type has two possible variants: **Ok** and **Err**.
- The **Ok** variant represents a successful result and contains a value of type T, which is the type of the value that the operation returns if it succeeds.
- The **Err** variant represents a failed result and contains a value of type E, which is the type of the error that the operation returns if it fails.

Error handling using ‘Result’ type

- The Result type is primarily used for handling recoverable errors, where it's possible for a function to return either a successful value or an error value, and the calling code can handle the error in a meaningful way

- For example, if a program needs to read data from a file, it might return a ***Result*** type that indicates whether the operation was successful or not. The calling code can then check the result and handle the error appropriately.

Used for handling recoverable errors

Panic

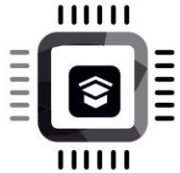
- **panic!** is a macro that can be used to indicate that an unrecoverable error has occurred, and the program should stop executing

- It's important to note that panicking should be used sparingly and only for unrecoverable errors, such as when a critical file or resource is missing, or when a precondition for a function has been violated. In most cases, using **Result** or **Option** is a better choice because it allows for more graceful error handling.

When a function uses panic!, it's essentially making a decision to stop the program execution due to an unrecoverable error

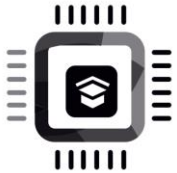
There are some scenarios where panicking may be more appropriate

1. **Out of memory errors:** If the program cannot allocate enough memory to continue running, there is no way to recover from this error.
2. **Invalid inputs or state:** If the program is given invalid inputs or enters an invalid state, it may be appropriate to panic.
3. **Unexpected errors:** If the program encounters an unexpected error that it does not know how to handle, it may be appropriate to panic.
4. **Debugging and testing:** During development, it can be useful to use panics to identify errors and bugs in the code. For example, if a test case fails, we can use a panic to signal that the test has failed. The **assert!** macro is implemented using panics. The **assert!** macro is used to check whether a condition is true, and if it is not true, the program will panic. This can be useful during testing and debugging to ensure that our assumptions about the code are correct.



```rust
fn add_strings(s1: &str, s2: &str) -> Result<String, String> {
    if s1.is_empty() || s2.is_empty() {
        return Err(String::from("Empty string detected!"));
    }

    let result = format!("{} {}", s1, s2);
    Ok(result)
}

fn main() {
    let s1 = "";
    let s2 = "";

    match add_strings(s1, s2) {
        Ok(concatenated) => println!("{}", concatenated),
        Err(error) => eprintln!("{}", error),
    }
}
```

This example shows a function ***add_strings*** that takes two string references as arguments and returns a **Result** that either contains the concatenated string or an error message if one of the strings is empty.



Exercise

Refactor the ***add_strings*** function to return ***Result<String, AddStringError>***

Implement ***description()*** method on the **AddStringError** enum to return description about the error.

```
enum AddStringError {
    EmptyString,
    LengthMismatch,
}
```

The '?' Operator (Error propagation)

- The ? operator is used for propagating errors or early returning from a function when an error occurs. It works by unwrapping the ***Result*** or ***Option*** value and returning early with an error if it contains ***Err*** or ***None***, respectively. If the value is ***Ok*** or ***Some***, the inner value is returned, and execution continues

- This also avoids the verbosity of ***match*** statements when handling errors in Rust

- The ? operator in Rust can also be used for error type conversion

```rust
fn add_strings(s1: &str, s2: &str) -> Result<String, String> {
    if s1.is_empty() || s2.is_empty() {
        return Err(String::from("Both strings must be non-empty."));
    }
    Ok(format!("{} {}", s1, s2))
}

fn main() -> Result<(), String> {
    let s1 = String::new();
    let s2 = String::from("world!");

    match add_strings(&s1, &s2) {
        Ok(final_str) => {
            println!("{}", final_str);
            Ok(())
        }
        Err(e) => Err(e),
    }
}
```

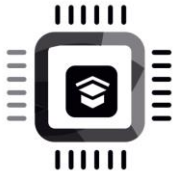
```rust
let final_str = add_strings(&s1, &s2)?;
println!("{}",final_str);
Ok(())
```

Converting *Option* type to *Return type*

You can use the ***ok_or*** method to convert the **Option** to a **Result** with an error value that is compatible with the expected **Result** type.

```
fn add_strings(s1: &str, s2: &str) -> Option<String> {
    if s1.is_empty() || s2.is_empty() {
        None
    } else {
        Some(format!("{} {}", s1, s2))
    }

}

fn main() -> Result<(), String> {
    let s1 = String::new();
    let s2 = String::from("world!");

    let final_str = add_strings(&s1, &s2).ok_or("Both strings must be non-empty")?;
    println!("{}",final_str);
    Ok(())
}
```

This argument only used when add_strings() returns the ***None*** variant



ok_or()

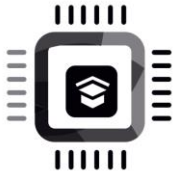
```
[–] pub fn ok_or<E>(self, err: E) -> Result<T, E>                    const: unstable · source
```

Transforms the `Option<T>` into a `Result<T, E>`, mapping `Some(v)` to `Ok(v)` and `None` to `Err(err)`.

Arguments passed to `ok_or` are eagerly evaluated; if you are passing the result of a function call, it is recommended to use `ok_or_else`, which is lazily evaluated.

Examples

```
let x = Some("foo");
assert_eq!(x.ok_or(0), Ok("foo"));

let x: Option<&str> = None;
assert_eq!(x.ok_or(0), Err(0));
```



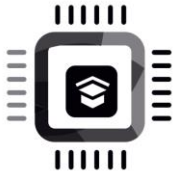
Unwrap() and expect()

- Both ***expect*** and ***unwrap*** are methods provided by the ***Option*** and ***Result*** enums for extracting their values.
- **unwrap()** simply returns the contained value of the ***Some*** or ***Ok***, and if the value is ***None*** or an ***Err***, it will panic and terminate the program.
- **expect()** works in the same way as ***unwrap*** but allows the programmer to provide a custom error message as a parameter to the method. This error message is printed to the console when a panic occurs, making it easier to identify the cause of the problem.
- It's worth noting that, while ***unwrap*** and ***expect*** can be useful when you're confident that your code will always produce a **Some** or an **Ok**, it's generally better to handle potential errors explicitly by using pattern matching or the ? operator. This helps to make your code more robust and easier to maintain and can prevent unexpected panics from occurring.

unwrap_err()

unwrap_err() will panic if the Result is not an ***Err***. It is the counterpart to the ***unwrap()*** method, which panics if the ***Result*** is not an ***Ok***.

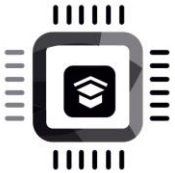
```
fn add_strings(s1: &str, s2: &str) -> Result<String, String> {
    if s1.is_empty() || s2.is_empty() {
        return Err(String::from("Both strings must be non-empty."));
    }
    Ok(format!("{} {}", s1, s2))
}

fn main() {
    let s1 = String::new();
    let s2 = String::from("world!");

    let result = add_strings(&s1,&s2);
    if result.is_ok() {
        println!("Concatenated string : {}",result.unwrap());
    } else {
        println!("{}",result.unwrap_err());
    }
}
```



unwrap_or(fallback)

unwrap_or() is a method defined for ***Option*** and ***Result*** enums that returns the contained value of the ***Some*** or ***Ok*** variant or a default value provided as an argument if ***None*** or ***Err***.

as_ref()



This code does not compile, find out why!

```
fn add_strings(s1: &str, s2: &str) -> Result<String, String> {
    if s1.is_empty() || s2.is_empty() {
        return Err(String::from("Both strings must be non-empty."));
    }
    Ok(format!("{}{}", s1, s2))
}

fn main()  {
    let s1 = String::new();
    let s2 = String::from("world!");

    let result = add_strings(&s1,&s2);
    match result {
        Ok(s) => println!("Concatenated string : {}",s),
        Err(e) => println!("Error: {}",e),
    }
    println!("{:?}",result);
}
```

Standard library error types

There are several types of errors that commonly occur while coding, regardless of the programming language being used.

I/O Errors

- File not found errors
- Permission denied errors
- Connection reset by peer errors (in the case of network I/O)
- Timeout errors
- Protocol errors
- DNS lookup fails

Numeric and Conversion Errors

- Parsing user input
- Reading data from files or network streams
- Converting between number types

Formatting and data validation Errors

- Converting a String from a UTF-8 byte vector
- Formatting a message into a stream

Network Errors

- Parsing an IP address or a socket address

```rust
use std::fs::File;
use std::io::Error;
use std::io::Read;
//std::io::{Error, Read}

fn read_file_contents(filename: &str) -> Result<String, Error> {

    let file_status = File::open(filename);

    let mut file = match file_status {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    // Read the file contents into a string
    let mut contents = String::new();
    match file.read_to_string(&mut contents) {
        Ok(_) => Ok(contents),
        Err(e) => Err(e),
    }
}

fn main() {
    let contents_result = read_file_contents("example.txt");

    match contents_result {
        Ok(contents) => println!("File contents: {}", contents),
        Err(error) => eprintln!("Failed to read file: {}", error),
    }
}
```

std::fs::File is a struct that represents a file. It provides methods for reading and writing to files.

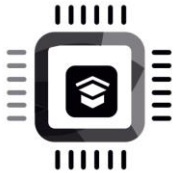
std::io is the standard library module for performing I/O operations in Rust. It defines a variety of types and traits for working with input and output streams.

std::io::Read is actually a trait that defines a method read that allows for reading data from a source into a buffer. It doesn't actually have any implementations of its own, but rather is implemented by various types that represent input sources, such as files, network sockets, and in-memory buffers.

In the code you provided, std::io::Read is imported to enable the use of the read_to_string method provided by the File type.
The File type in Rust provides methods for reading and writing to files. One of these methods is read_to_string, which reads the entire contents of a file into a String.
The read_to_string method is defined for the Read trait, which means that it can be called on any type that implements Read. Since File implements Read, we can call read_to_string on a File object to read the contents of the file into a String.
By importing std::io::Read, we make the read_to_string method available in our code so that we can use it to read the contents of files into strings.

Handling I/O Errors

In Rust std::io::Error error type is used for I/O-related errors, such as failed file operations or network errors



Exercise

Write a function to rename a given file .

If renaming successful return Ok(())

If the given file doesn’t exist return Err(std::io::Error)

std::io::Result

In std::io module you will find the type definition :

```
pub type Result<T> = Result<T, Error>;
```

- The ***std::io::Result*** is a type alias for the standard ***Result*** type in Rust, with the error type fixed to ***std::io::Error***. The purpose of this type alias is to make it easier to work with ***Result*** values that may return I/O errors.

- If you're returning an error of type ***std::io::Error***, you can use either the standard ***Result<T, E>*** type or the ***std::io::Result<T>*** type. Both types are essentially the same, with the only difference being the error type: the standard ***Result<T, E>*** type can hold any type **E** as the error, whereas the ***std::io::Result<T>*** type is a type alias for ***Result<T, std::io::Error>,*** which means it can only hold ***std::io::Error*** objects as the error.

- So, if your function returns a ***Result<T, E>*** where **E** is an ***std::io::Error***, you can use the ***std::io::Result<T>*** type alias instead, which can make your code more concise and easier to read.

Summary

- **std::io::Error** is an error type provided by the Rust standard library's io module. The std::io::Error type is used to represent errors that occur during I/O operations, such as reading or writing to a file or network socket

- You can use the std::io::Error type directly in your own **Result** type, or you can use the **std::io::Result** type alias provided by the io module, which is a shorthand for **Result<T, std::io::Error>.**

- **rename** is a function provided by the **fs** module of the standard library, and in order to use it, you need to import the **fs** module using the **use** statement.

- By including **use std::fs;** at the beginning of your code, you're bringing all the items defined in the **fs** module into the current scope.

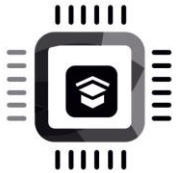
std::io::ErrorKind

- Different kinds of I/O errors tat can occur when performing various I/O operations.
- **std::io::ErrorKind** is an enum type that is defined inside the std::io module. It represents the different kinds of I/O errors that can occur when performing various I/O operations.
- You can use **ErroKind** by bringing the **std::io** module into scope, like this

```
use std::io;
use std::io::ErrorKind;
```

```
use std::io::{self, ErrorKind}
```

Instead of writing these 2 statements separately you can write in one line like this



Numeric Errors : std::num::ParseIntError

This is an error which can be returned when parsing an integer.

Exercise

Implement a function called **parse_integer_from_string** that takes a string(&str) as input and attempts to parse it as a **32-bit signed integer (i32).** The function should return a **Result** type, where a successful parsing will be represented by **Ok(num)** with the parsed integer value **num**, and an unsuccessful parsing will be represented by **Err(e)** where **e** is corresponding error message(**String**)

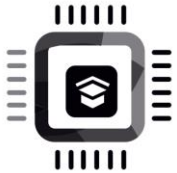
Inside the **parse_integer_from_string** function, the provided string is parsed using the **i32::from_str** method. If the parsing succeeds, the function should return the parsed integer value. However, if an error occurs during parsing**, from_str()** method returns the error value of type **std::num::parseIntError**. To handle different error cases, use the **kind()** method of the **std::num::parseIntError**

```rust
use std::str::FromStr;

fn parse_integer_from_string(input: &str) -> Result<i32, String> {
    //TODO
}

fn main() {
    let result = parse_integer_from_string("9");
    match result {
        Ok(num) => println!("Parsed number: {}", num),
        Err(e) => println!("Error parsing number: {}", e),
    }
}
```

#[non_exhaustive]

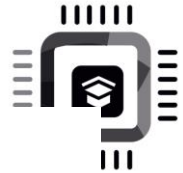
- When an enum is marked as #[non_exhaustive], it means that the enum is intentionally designed to be non-exhaustive, allowing additional variants to be added in the future without causing a compilation error or breaking existing code that handles the enum.

- This feature comes in handy when the enum is included in a publicly available API and the creators wish to maintain the option of adding new options in future updates without compromising the reliability promised to API users.



Exercise

Write a function that accepts a number from the user as a string, parses the string to convert it into a number, and returns 'Ok(number)'. If parsing fails, return 'Err(std::io::Error)' with 'ErrorKind' set to 'InvalidData'.



```rust
match i32::from_str(&user_input.trim()) {
    Ok(num) => Ok(num),
    Err(e) => Err(io::Error::new(io::ErrorKind::InvalidData, e)),
}
```

```rust
i32::from_str(user_input.trim())
                .or(Err(io::Error::new(io::ErrorKind::InvalidData, "Invalid input")))

user_input.trim().parse().map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))
```

- The **or/map_err** method concisely converts a Result<T, E> to a Result<T, F> where F is a different error type you want to use.

- In a situation where you want to convert one type of error to another, using **or/map_err** can be a more concise and readable approach than using a match expression with multiple arms.

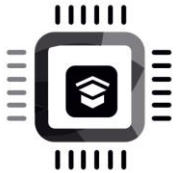
Closure

A closure is an anonymous function that can capture variables from its surrounding environment. It allows you to create a function on the fly without explicitly defining a separate named function

Auto conversion of errors

```rust
use std::ffi::{CString, NulError};
use std::io;
/*
 * Defining a function that returns a Result type with a unit type ()
 * indicating success or a NulError indicating failure
 * Return type : Result<(), NulError>
 */
fn some_function() -> Result<(), NulError> {
    /*
     * Creating a C-style string using CString::new()
     * and a string literal with a null terminator "\0"
     */
    let c_string = CString::new("foo\0bar")?;
    Ok(())
}

/*
 * Return type : Result<(), std::io::Error>
 */
fn main() -> io::Result<()> {
    some_function()?;
    Ok(())
}
```

In the main() function, the ***From*** trait implementation for ***std::io::Error*** would be used to convert the ***NulError*** to an ***std::io::Error*** type, as specified in the return type of main()



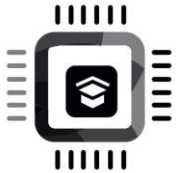
So, when is auto-conversion of error types possible?

The automatic conversion of error types is possible when there is an implementation of the **From** trait available for converting from the original error type to the desired error type.

```rust
use std::ffi::{CString, NulError};
use std::io;

fn some_function() -> io::Result<()> {
    let c_string = CString::new("foo\0bar")?;
    Ok(())
}

fn main() -> Result<(), NulError> {
    some_function()?;
    Ok(())
}
```

Since there is no explicit ***From*** trait implementation to convert ***std::io::Error*** to ***NulError*** in ***NulError*** type , you will encounter a compilation error, as Rust will not be able to perform the conversion automatically

```
   Compiling playground v0.0.1 (/playground)
error[E0277]: `?` couldn't convert the error to `NulError`
  --> src/main.rs:11:20
   |
10 | fn main() -> Result<(), NulError> {
   |              -------------------- expected `NulError` because of this
11 |     some_function()?;
   |                    ^ the trait `From<std::io::Error>` is not implemented for `NulError`
   |
```



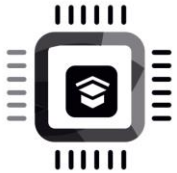
Generics: A Key to Flexible and Reusable Code

Generics

- Using generics, you can write functions or types that can work with any type that meets certain requirements rather than being tied to a specific type.
- This makes your code more reusable and flexible since you can write code that can be used with various data types.

Many constructs in the language can take advantage of Generics.

1. Functions
2. Structs
3. Enums
4. Traits
5. Functions and closures as arguments
6. Methods

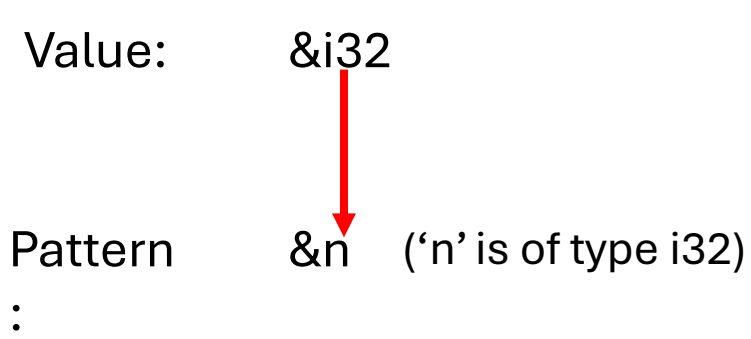


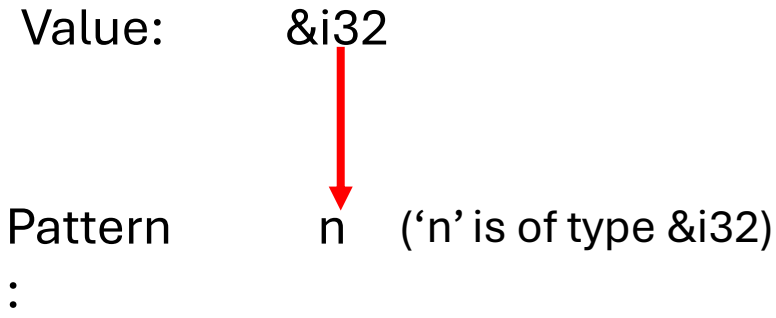
Functions and Generics

Writing a function that works with a wide range of input types

```rust
fn find_max_int(v: &[i32]) -> Option<i32> {
    if v.is_empty() {
        return None;
    }
    let mut max = v[0];
    for &n in v {
        if n > max {
            max = n;
        }
    }
    Some(max)
}
```

Pattern matching with references





The & operator has two different meanings in Rust, depending on whether it's used in an expression or in a pattern.

- In an expression, the & operator creates a reference to a value. This is called the "borrowing" operator, and it's used to give temporary access to a value without transferring ownership.

- On the other hand, in a pattern, the & operator is used to match a reference to a value, which means destructuring a reference into its underlying value.

This specifies a generic type parameter named T that can be used in the function's signature.

This specifies a parameter named 'v' of type &[T], which is a slice of the generic type T.

```
fn find_max<T>(v: &[T]) -> Option<T>
    where T: std::cmp::PartialOrd + Copy
{
    if v.is_empty() {
        return None;
    }
    let mut max = v[0];
    for &n in v {
        if n > max {
            max = n;
        }
    }
    Some(max)
}
```

Option value of the generic type T

Trait bounds to restrict the types that can be used with a generic function

Generic type parameters and Generic lifetime parameters are specified using angle brackets < >

Monomorphization

- Using generics in Rust does not cause any significant performance overhead. Rust's generics are implemented using monomorphization, meaning the compiler generates separate code for each concrete type used with a generic function or type.
- This may increase the binary size of the compiled code, but it also results in faster execution because the code is specialized for the specific types being used.

Code to be optimized by the compiler as if it were written specifically for that type, without any runtime overhead

Summary : function_name<T>

- The <T> after the function name is ‘generic type parameter syntax’
- <T> is required to specify that the function is generic and can accept any type T. Without it, the compiler would interpret T at the parameter side as a specific type instead of a generic type parameter.
- When the function is called with **find_max(&[1, 2, 3])**, T will be inferred as &i32. Similarly, when the function is called with **find_max(&["apple", "banana", "cherry"]),** T will be inferred as &str.
- You can use any valid identifier as a type parameter in place of T. However, it is common practice to use single uppercase letters such as T, U, V, etc. to denote type parameters in Rust

Generic function with two type parameters of same type T

```rust
fn combine<T>(a: T, b: T) -> Vec<T> {
    let mut v = Vec::new();
    v.push(a);
    v.push(b);
    v
}

fn main() {
    let v1 = combine(3, 4);
    println!("{:?}",v1);
    let v2 = combine("hello".to_string(), "world".to_string());
    println!("{:?}",v2);
}
```

Output

```
[3, 4]
["hello", "world"]
```

Generic function with two different type parameters T and U:

```
fn combine<T, U>(a: T, b: U) -> (T, U) {
    (a, b)
}


fn main() {
    let t1 = combine(3, "three");
    let t2 = combine("two",2.0);
    let t3 = combine("one","two".to_string());
    println!("{:?}\n{:?}\n{:?}",t1,t2,t3);
}
```

Output

```
(3, "three")
("two", 2.0)
("one", "two")
```

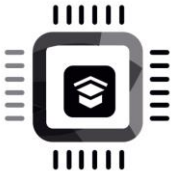
<T> Vs <T, U>

- <T> could be used to define a function that takes n arguments of the same type, whereas <T, U> could be used to define a function that takes two arguments of potentially different types.
- If a function specifies the same generic type parameter twice, e.g. **combine<T>(a: T, b: T),** it means that both arguments passed to the function must have the same type. If you pass two different types, the Rust compiler will give you a type mismatch error.
- If a function accepts <T, U> as generic type parameters, and the arguments passed are the same type, the compiler will not give an error.

Turbofish syntax

The turbofish syntax ***::<Type>*** can be used to explicitly specify the type parameter when calling a generic function.

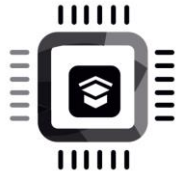
Generics with struct



```rust
#[derive(Debug)]
struct Pair<T> {
    first: T,
    second: T,
}

fn main() {
    let pair_of_ints = Pair { first: 1, second: 2 };
    let pair_of_strings = Pair { first: "hello", second: "world" };

    println!("{:?}", pair_of_ints);
    println!("{:?}", pair_of_strings);
}
```

This allows the struct to use the type parameter T in its fields, methods, and implementation.

struct with 2 generic type parameters

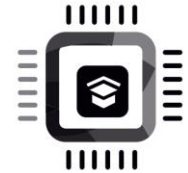


```rust
#[derive(Debug)]
struct Pair<T, U> {
    first: T,
    second: U,
}

fn main() {
    let pair_of_ints = Pair { first: 1, second: "hello" };
    let pair_of_strings = Pair { first: "hello", second: 4.5 };

    println!("{:?}", pair_of_ints);
    println!("{:?}", pair_of_strings);
}
```

Methods of generic type struct

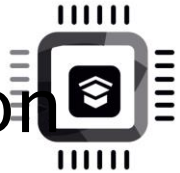
```
struct Pair<T> {
    first: T,
    second: T,
}

impl<T> Pair<T> {

}
```

The type parameter again after ***impl*** to indicate that you're implementing methods that work for any instantiation of Pair<T>, where T can be any type



```rust
impl<T> Pair<T> {
    fn new(first: T, second: T) -> Pair<T> {
        Pair { first, second }
    }

    fn get_first(&self) -> &T {
        &self.first
    }

    fn get_second(&self) -> &T {
        &self.second
    }
}
```

Concrete method implementation



```rust
struct Pair<T> {
    x: T,
    y: T,
    label: String,
}

impl Pair<i32> {
    fn new(x: i32, y: i32, label: &str) -> Pair<i32> {
        Pair {
            x,
            y,
            label: String::from(label),
        }
    }

    fn sum(&self) -> i32 {
        self.x + self.y
    }
}

fn main() {
    let pair_i32 = Pair::new(3, 5, "Integer Pair");
    println!("{} + {} = {}", pair_i32.x, pair_i32.y, pair_i32.sum());

    let pair_str = Pair::new("Hello", "Rust", "String Pair");
}
```

```rust
impl<T> Pair<T> {
    fn new(x: T, y: T, label: &str) -> Pair<T> {
        Pair {
            x,
            y,
            label: String::from(label),
        }
    }

    fn get_label(&self) -> &str {
        &self.label
    }

    fn set_label(&mut self, new_label: &str) {
        self.label = String::from(new_label);
    }
}

impl Pair<i32> {
    fn sum(&self) -> i32 {
        self.x + self.y
    }
}
```

Mix of generic and concrete methods

```rust
fn main() {
    let mut pair_i32 = Pair::new(3, 5, "Integer Pair");
    println!("{} + {} = {}", pair_i32.x, pair_i32.y, pair_i32.sum());

    let pair_str = Pair::new("Hello", "Rust", "String Pair");
    println!("Label: {}", pair_str.get_label());

    pair_i32.set_label("New Label");
    println!("Label: {}", pair_i32.get_label());
}
```

The 'Self' keyword

```rust
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T>{

    fn new(x: T, y: T) -> Point<T> {
        Point { x, y }
    }

    fn distance(&self, other: &Point<T>) -> f64
        where
            T: std::ops::Sub<Output = T>
            + std::ops::Mul<Output = T>
            + std::ops::Add<Output = T>
            + std::convert::Into<f64> + Copy
    {
        let dx = self.x - other.x;
        let dy = self.y - other.y;
        let sum = (dx * dx) + (dy * dy);
        (sum.into() as f64).sqrt()
    }

}

fn main() {
    let p1 = Point::new(3, 4);
    let p2 = Point::new(6, 8);
    let distance = p1.distance(&p2);
    println!("Distance between {:?} and {:?} is {}", p1, p2, distance);
}
```

Sel
f

&Self

Here, Self refers to the type Point<T>, which is the type for which the methods are being implemented.

Self with a capital "S" refers to the type that the implementation is for, while ***self*** with a lowercase "s" refers to an instance of that type.

Generics with enum

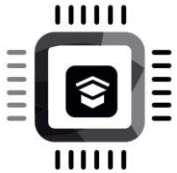
```rust
enum MyEnum<T> {
    A(T),
    B(T),
}

impl<T> MyEnum<T> {
    fn get(&self) -> &T {
        match self {
            MyEnum::A(t) => t,
            MyEnum::B(t) => t,
        }
    }

    fn set(&mut self, t: T) {
        match self {
            MyEnum::A(ref mut x) => *x = t,
            MyEnum::B(ref mut x) => *x = t,
        }
    }
}

fn main() {
    let mut e1 = MyEnum::A(5);
    let e2 = MyEnum::B("hello");
    println!("e1.get(): {}", e1.get());
    println!("e2.get(): {}", e2.get());
    e1.set(10);
    println!("e1.get() after set(): {}", e1.get());
}
```

MyEnum is a generic enum, parameterized over the type T

ref mut x, we can match the value by reference, and then modify it in a mutable way.

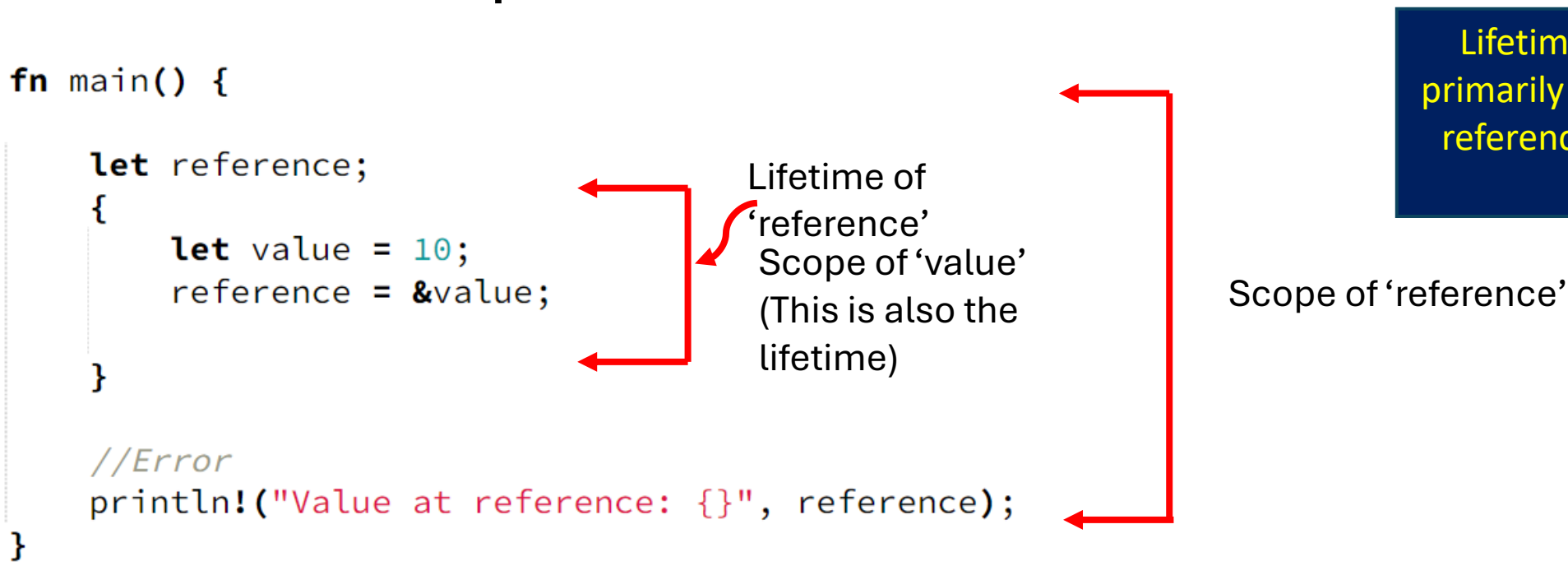
MyEnum::A(5) creates a concrete instance of the **MyEnum<T>** enum. The type of e1 will be inferred by the compiler as MyEnum<i32>.

```
e1.get(): 5
e2.get(): hello
e1.get() after set(): 10
```



Lifetimes

Lifetime Vs Scope



Lifetimes in Rust are primarily associated with references rather than values

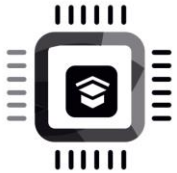
- Scopes refer to a portion of the code where a variable is valid and accessible
- Lifetimes, on the other hand, refer to the duration for which a reference is valid. They ensure that references do not outlive the data they point to, preventing dangling references.

Lifetime

- Lifetime refers to the duration for which a reference is valid. They ensure that references do not outlive the data they point to, preventing dangling references.
- Lifetimes in Rust are denoted using apostrophes (') followed by a lowercase character, typically 'a, 'b, 'c, and so on (common convention)

Why rust uses lifetimes?

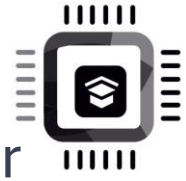
By using lifetimes, Rust guarantees memory safety without the need for garbage collection. It enables the compiler to perform compile-time checks and ensure that references are always valid within their designated scopes. This prevents common issues such as accessing freed memory or dangling references that can lead to undefined behavior and bugs



```rust
fn main() {

    let reference;
    {
        let value = 10;
        reference = &value;

    }

    //Error
    println!("Value at reference: {}", reference);
}
```

Rust's borrow checker analyzes the code during compilation to ensure that references are always valid and don't point to deallocated memory

The scope of the reference reference is the entire main() function. However, the lifetime of the reference is determined by the lifetime of the value it borrows, which is value in this case.
Therefore, the issue is that the lifetime of the reference is shorter than the scope of the reference itself



Let's consider a scenario where Rust fails to analyze lifetimes and requires your assistance

```
fn extend_string(original: &mut String, data: &str) -> &str {
    original.push_str(data);
    original
}


fn main() {

    let mut original = String::from("hello");
    let data = String::from("world");
    let result = extend_string(&mut original, &data);

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

Why error?

- Remember every reference has a lifetime associated with it
- The program fails because rust fails to infer the lifetime of the reference which the function is returning
- When a function returns a reference, it must have a known and valid lifetime associated with it
- To fix the compilation error, you need to specify the correct lifetime for the return value, and you can do that using explicit lifetime annotations in the function signature
- Lifetime annotations indicate the relationships between the input references and the return value

Explicit Lifetime annotation

```
fn extend_string<'a>(original: &'a mut String, data: &'a str) -> &'a str {
    ...
}
```

Note:
When a function in Rust returns a reference, the lifetime parameter for the return type must be identical to the lifetime parameter of one of the function's parameters. This ensures that the reference being returned remains valid and does not extend beyond the lifespan of the data it refers to.

Lifetime annotation syntax

Lifetimes are denoted using apostrophes followed by a lowercase name or in most cases just a character

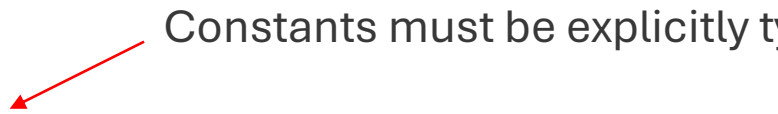
Here are a few examples of lifetime annotations

- ✓ **'a**: Represents a lifetime named a.
- ✓ ‘xyz: Represents a lifetime named xyz.
- ✓ 'b: Represents a lifetime named b.
- ✓ '_: Represents an anonymous lifetime.

These lifetime annotations are used in function signatures, struct definitions, and other places where lifetimes need to be specified to ensure proper borrowing and ownership rules in Rust.

Constant in Rust

Constants in Rust are values that are fixed and immutable throughout the entire execution of a program.




```rust
const MAX_VALUE: u32 = 100;
const MESSAGE: &str = "Hello, world!";

fn main() {
    let ref_to_const = &MAX_VALUE;
    println!("The maximum value is: {}", MAX_VALUE);
    println!("Message: {}", MESSAGE);
    println!("value: {}", ref_to_const);
}
```

Here are a few characteristics of constants in Rust:

1. **Immutable**: Constants are immutable, meaning their values cannot be modified once assigned. This ensures that the value remains constant throughout the program's execution.

2. **Known at compile time**: The value of a constant must be computable at compile time. This allows the Rust compiler to optimize the program and directly substitute the constant's value wherever it is used.

3. **Scoped globally**: Constants are usually defined at the global scope and are accessible from any part of the program..

4. **Type annotations**: Constants require explicit type annotations to indicate the type of the value they hold

5. **Inlined**: Constants in Rust do not have fixed memory locations. Instead, they are usually "inlined" by the compiler. Inlining means that the compiler replaces the usage of the constant with its actual value wherever it is referenced in the code

Const lifetime

The lifetime of a constant in Rust is **'static**, also known as the static lifetime. **'static** is a special lifetime that represents the entire duration of the program. It means that the constant is valid for the entire lifetime of the program execution

```
const MAX_VALUE: u32 = 100;
const MESSAGE: &str = "Hello, world!";

fn main() {
    let ref_to_const = &MAX_VALUE;
    println!("The maximum value is: {}", MAX_VALUE);
    println!("Message: {}", MESSAGE);
    println!("value: {}", ref_to_const);
}
```

Naming convention

- it is a convention to use all capital letters for the names of constants. This convention helps to distinguish constants from other variables and makes them easily recognizable in the code.

Static items in Rust

- A static item is a value that has a **'static** lifetime, which means it is valid for the entire duration of the program's execution. Static items have a fixed memory address, and their values are accessible throughout the program

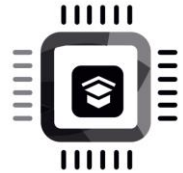
```
static GLOBAL_VALUE: i32 = 42;

fn main() {
    println!("Global value: {}", GLOBAL_VALUE);
}
```

Non mutable static variable

'static' variables represent a location in memory, and they can be referenced and potentially modified, making them similar to global variables

Mutable static value

- If a ***static*** item is declared with the ***mut*** keyword, it indicates that the static item is mutable and can be modified by the program. However, accessing and modifying mutable statics can introduce potential issues and undefined behavior, especially data races in a multithreaded context.
- To access and modify mutable statics, the code must be wrapped within an unsafe block. The unsafe block indicates that the code contains potentially unsafe operations and bypasses Rust's safety guarantees. It allows the programmer to take responsibility for ensuring the correctness and safety of the code.



```rust
static mut MUTABLE_STATIC: i32 = 42;

fn main() {
    unsafe {
        // Modifying mutable static requires an unsafe block
        MUTABLE_STATIC = 10;
        println!("Updated mutable static: {}", MUTABLE_STATIC);
    }
}
```

Lifetime elision

lifetime elision is a set of rules implemented by the Rust compiler that allows you to omit explicit lifetime annotations in certain cases. These rules make the code more concise and readable by automatically inferring the correct lifetimes based on the structure and usage of the code.

Lifetime elision rules

lifetime elision rules are not intended as rules for programmers to follow. The first rule of lifetime elision specifically applies to input lifetimes, while the other two rules apply to output lifetimes.

1. An individual lifetime parameter is assigned to each reference parameter in order to establish proper lifetimes for function parameters.

2. If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters.

3. If there are multiple input lifetime parameters, but one of them is **&self** or **&mut self** as part of a method, the lifetime of ***self*** is assigned to all output lifetime parameters.

Input lifetime: An input lifetime represents the lifetime of a reference that is passed as an argument to a function

Output lifetime: An output lifetime represents the lifetime of a reference returned by a function

Lifetime elision rule #1

An individual lifetime parameter is assigned to each reference parameter in order to establish proper lifetimes for function parameters.

```
fn extend_string(original: &mut String, data: &str) -> &str {
    original.push_str(data);
    original
}
```

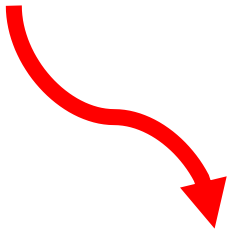
```
fn extend_string<'a, 'b>(original: &'a mut String, data: &'b str) -> &str {
    original.push_str(data);
    original
}
```

Explicit lifetime annotation required here

Lifetime elision rule #2

If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters.

```
fn process_input(input: &str) -> (&str, &str) {
    (input, input)
}
```



```
fn process_input<'a>(input: &'a str) -> (&'a str, &'a str) {
    (input, input)
}
```

No explicit lifetime annotations required

Lifetime annotation with Structs

For struct , explicit lifetime annotations are necessary when a struct contains references, and the compiler needs explicit information to determine the lifetimes of those references.

```rust
struct MyStruct<'a, 'b> {
    data1: &'a str,
    data2: &'b str,
}

fn main() {
    let data1 = "Hello";
    let data2 = "World";

    let my_struct = MyStruct {
        data1: &data1,
        data2: &data2,
    };

    println!("Data 1: {}", my_struct.data1);
    println!("Data 2: {}", my_struct.data2);
}
```

The field data1 is of type **&'a str**, indicating a reference to a string slice with a lifetime 'a. Similarly, the field data2 is of type **&'b str**, indicating a reference to a string slice with a lifetime 'b.

By using generic lifetime parameters, the struct MyStruct can accept references with different lifetimes for its fields. This allows you to create instances of the struct where the lifetimes of data1 and data2 can be distinct or independent from each other

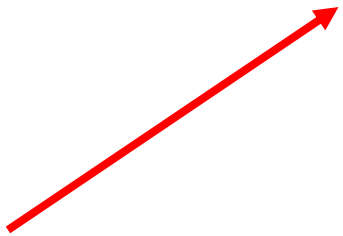
The instance of MyStruct cannot outlive the lifetimes of the references it contains.

```rust
struct MyStruct<'a, 'b> {
    data1: &'a str,
    data2: &'b str,
}

fn main() {
    let data1 = String::from("Hello");
    let data2 = String::from("World");

    let my_struct = MyStruct {
        data1: &data1,
        data2: &data2,
    };

    let ret = fun(&my_struct);

    println!("ret = {}",ret);

}


fn fun(data: &MyStruct) -> &str {
    data.data1
}
```

Will this code compile?

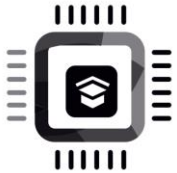
```rust
struct MyStruct<'a, 'b> {
    data1: &'a str,
    data2: &'b str,
}

fn main() {
    let data1 = String::from("Hello");
    let data2 = String::from("World");
    let ret;

    {
        let my_struct;

        my_struct = MyStruct {
            data1: &data1,
            data2: &data2,
        };

        ret = fun(&my_struct);
    }

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

```rust
fn fun<'a, 'b, 'c>(data: &'a MyStruct<'b, 'c>) -> &'a str {
    data.data1
}
```



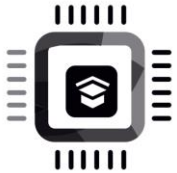
The lifetime constraint in the provided code is that the reference returned by the fun function must have a lifetime that is at least as long as the ‘a lifetime parameter of the MyStruct reference passed as an argument to fun.

Lifetime annotation for struct methods



```rust
#[derive(Debug)]
struct MyStruct<'a, 'b> {
    data1: &'a str,
    data2: &'b str,
}


impl MyStruct{
    fn set_data1(&mut self, input: &str) {
        self.data1 = input;
    }
}

fn main() {

    let mut struct_ins = MyStruct {
        data1 : "Hi",
        data2 : "World",
    };

    struct_ins.set_data1("bye");
    println!("{:?}",struct_ins);
}
```

Lifetime annotation for struct methods

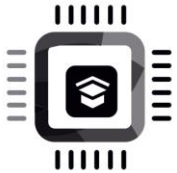


```
#[derive(Debug)]
struct MyStruct<'a, 'b> {
    data1: &'a str,
    data2: &'b str,
}


impl<'a> MyStruct<'a, 'a>{
    fn set_data1(&mut self, input: &'a str) {
        self.data1 = input;
    }
}

fn main() {

    let mut struct_ins = MyStruct {
        data1 : "Hi",
        data2 : "World",
    };

    struct_ins.set_data1("bye");
    println!("{:?}",struct_ins);
}
```

The impl<'a> MyStruct<'a, 'a> block specifies that the implementation applies to instances of MyStruct where both the data1 and data2 fields have the same lifetime 'a.

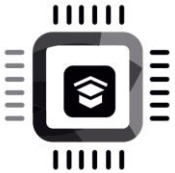
```rust
#[derive(Debug)]
struct MyStruct<'a, 'b> {
    data1: &'a str,
    data2: &'b str,
}


impl<'a, 'b> MyStruct<'a, 'b>{
    fn set_data1(&mut self, input: &str) {
        self.data1 = input;
    }
}

fn main() {

    let mut struct_ins = MyStruct {
        data1 : "Hi",
        data2 : "World",
    };

    struct_ins.set_data1("bye");
    println!("{:?}",struct_ins);
}
```

```rust
#[derive(Debug)]
struct MyStruct<'a, 'b> {
    data1: &'a str,
    data2: &'b str,
}


impl<'a, 'b> MyStruct<'a, 'a>{
    fn set_data1(&mut self, input: &'b str) {
        self.data1 = input;
    }
}

fn main() {

    let mut struct_ins = MyStruct {
        data1 : "Hi",
        data2 : "World",
    };

    struct_ins.set_data1("bye");
    println!("{:?}",struct_ins);
}
```



Lifetime elision rule #3

If there are multiple input lifetime parameters, but one of them is **&self** or **&mut self** as part of a method, the lifetime of ***self*** is assigned to all output lifetime parameters.



Traits

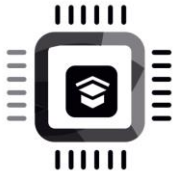
What is a trait?

Trait bounds

Trait objects

Generic traits

Examples



What is a trait?

- A ‘Trait’ in Rust is a collection of method signatures that can be implemented by any type, similar to an interface in other languages.

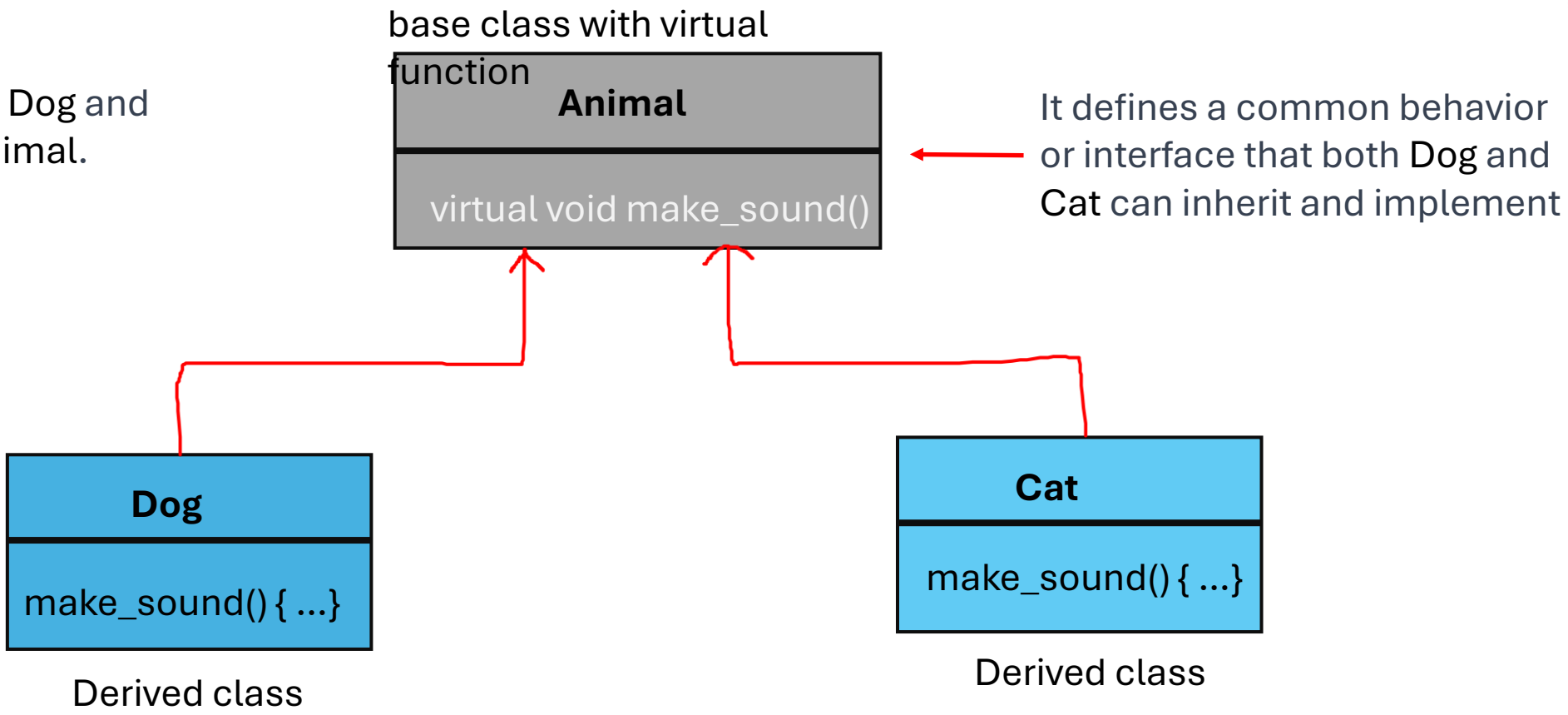
- By implementing a ‘Trait’ for a type, that type gains the ability to utilize the methods defined in the trait. This enables polymorphism, where different types can be treated interchangeably as long as they implement the same trait.

You can think of *traits* in Rust as the equivalent of interfaces in C++

Interface

An interface in programming is like a blueprint that outlines what a class or structure should have, such as methods, properties, and behaviors. It sets the rules that other classes must follow, but it doesn't tell them exactly how to do it. It's like a contract that says, "You need to have these things, but I won't tell you how to make them."

inheritance, where Dog and Cat inherit from Animal.



make_sound() method is defined in the *Animal* class as a virtual method, which means it can be overridden by the child classes

Polymorphism

- Polymorphism means that objects of different classes can be treated as if they belong to a common type, even though they may have different specific behaviors. This allows you to write code that can work with different objects interchangeably, based on their shared interface

- An interface helps achieve polymorphism by providing a common set of methods that classes can implement. This allows different objects to be used interchangeably based on the shared interface, making the code more flexible and reusable.

C++ implementation

```cpp
#include <iostream>

class Animal {
public:
    virtual void make_sound() {
        std::cout << "Generic animal sound\n";
    }
};

class Dog : public Animal {
public:
    void make_sound() {
        std::cout << "Bark!\n";
    }
};

class Cat : public Animal {
public:
    void make_sound() {
        std::cout << "Meow!\n";
    }
};

int main() {
    Animal* a1 = new Dog();
    Animal* a2 = new Cat();

    a1->make_sound(); // Outputs "Bark!"
    a2->make_sound(); // Outputs "Meow!"

    delete a1;
    delete a2;

    return 0;
}
```

Interface

Implementing interface defined methods

Polymorphism

```
int main() {
    Animal* a1 = new Dog();
    Animal* a2 = new Cat();

    a1->make_sound(); // Outputs "Bark!"
    a2->make_sound(); // Outputs "Meow!"

    delete a1;
    delete a2;

    return 0;
}
```

Because **Dog** and **Cat** classes both have the **make_sound()** method that is declared as **virtual** in the **Animal** class, they can be treated as **Animal** objects.

In this code, polymorphism allows the **Animal** pointer to point to either a **Dog** object or a **Cat** object, and the correct implementation of the **make_sound()** method is called at runtime depending on which object the pointer is pointing to.

Benefits of using interfaces in programming

- **Loose coupling:**
  - The code exhibits loose coupling, which is the practice of designing code that minimizes dependencies between components.
  - e.g. The code achieves loose coupling by using base class pointers (*Animal**) to refer to derived class objects (*Dog and Cat*).
  - The *main()* function only interacts with the base class interface (*Animal*), without needing to know the specific derived classes or their implementations

- **Flexibility and extensibility:**
  - The code becomes more flexible and extensible by programming to the interface (*Animal*).
  - New 'Animal' types can be added by creating derived classes that implement the *make_sound()* function without modifying existing code.
  - This allows for the easy addition of new behaviour and accommodating future changes.

Equivalent code in Rust

In Rust, Dog and Cat implement the Animal trait. 'Traits' serve the same purpose as interfaces in C++. They define a set of methods a type must implement, allowing for polymorphism and abstraction.

trait keyword is used to define an interface

```
trait Animal {
    fn make_sound(&self);
}
```

Whichever types implement this trait must give definitions for this method or set of methods.

```rust
trait Animal {
    fn make_sound(&self);
}

struct Dog;

impl Animal for Dog {
    fn make_sound(&self) {
        println!("woof-woof!");
    }
}

struct Cat;

impl Animal for Cat {
    fn make_sound(&self) {
        println!("Meow!");
    }
}

fn produce_sound(animal: &dyn Animal) {
    animal.make_sound();
}

fn main() {
    produce_sound(&Dog);
    produce_sound(&Cat);
}
```

self in a trait method signature is mandatory. It is used to represent the instance of the type implementing the trait, and allows you to access and modify the object's properties or call other methods on it

Dog struct gives its implementation of **Animal** trait.

Cat struct gives its implementation of **Animal** trait.

produce_sound() function takes a reference to a trait object of type **dyn Animal**, which can be a reference to any object that implements the Animal trait.

Trait objects

Trait objects enable writing polymorphic code by providing a common interface through traits.

When a type that implements a trait provides an implementation for a method that is also defined in the trait, the implementation provided by the type takes precedence over the default implementation defined in the trait.

Associated type of a trait

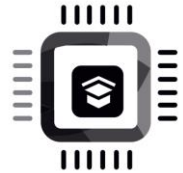
```
trait Animal {
    fn make_sound(&self);
    type Weight;
    fn set_weight(&mut self, weight: Self::Weight);
    fn get_weight(&self) -> Self::Weight;
    fn set_age(&mut self, age: u8);
    fn get_age(&self) -> u8;
}
```

- ✓ The associated type **Weight** serves as a placeholder for a specific concrete type that implementing structs or classes will define. It is a way to make the trait more generic and flexible, allowing each implementing type to specify its own concrete type for Weight.
- ✓ The specific type that Weight represents will be determined by the implementing structs or classes when they provide their own implementations for the associated type
- ✓ Using PascalCase (also known as UpperCamelCase) for associated type names is a common convention in Rust

Associated type of a trait

```rust
trait Animal {
    fn make_sound(&self);
    type Weight;
    fn set_weight(&mut self, weight: Self::Weight);
    fn get_weight(&self) -> Self::Weight;
    fn set_age(&mut self, age: u8);
    fn get_age(&self) -> u8;
}
```

Self is a Rust keyword that refers to the type of the struct or the enum that implements the trait. For example, if we have a ***Dog*** struct that implements the Animal trait, then ***Self*** in the trait methods will refer to the type ***Dog***



```
struct Dog {
    weight: u8,
    age: u8,
}
```

```
struct Cat {
    weight: f32,
    age: u8,
}
```

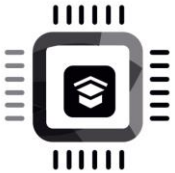
- In the *Animal* trait definition, the ***get_age()*** and *set_age()* methods have a concrete return type of ***u8***, which means that any type that implements the ***Animal*** trait must have an age that is represented as a ***u8***.

- The ***Weight*** associated type in the ***Animal*** trait is generic, meaning it can be any type. The implementor of the trait can choose the concrete type that will be used for ***Weight***. In the examples we have seen so far, we have used ***u8*** for ***Dog*** and ***f32*** for ***Cat***,

```rust
impl Animal for Dog {
    fn make_sound(&self) {
        println!("woof-woof!");
    }
    type Weight = u8;
    fn set_weight(&mut self, weight: u8) {
        self.weight = weight;
    }
    fn get_weight(&self) -> u8 {
        self.weight
    }
    fn set_age(&mut self, age: u8) {
        self.age = age;
    }
    fn get_age(&self) -> u8 {
        self.age
    }
}
```

The implementation of the trait provides the concrete type for the associated type

Type aliasing



In rust, you can use the '**type**' keyword to create type aliases. This allows you to give a new name to an existing type, which can improve code readability.

```rust
// Creating a type alias for u8
type Byte = u8;

fn main() {
    let a: Byte = 255;
    println!(
    "The value of a is: {}",
    a
    );
}
```

```rust
fn main() {
    let origin: (i32, i32) = (0, 0);
    println!(
    "The origin is at: ({}, {})",
    origin.0,
    origin.1
    );
}

// Creating a type alias for a tuple
type Point = (i32, i32);

fn main() {
    let origin: Point = (0, 0);
    println!(
    "The origin is at: ({}, {})",
    origin.0,
    origin.1
    S);
}
```

```rust
fn produce_sound(animal: &dyn Animal) {
    animal.make_sound();
}
```

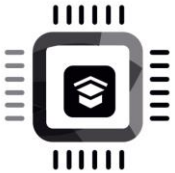
```rust
fn produce_sound<W>(animal: &dyn Animal<Weight = W>) {
    animal.make_sound();
}
```

The generic type W in the produce_sound function is resolved when the function is called. When calling the produce_sound function, the W type is explicitly specified by the caller. For example, in the main function of the code we have been discussing, the produce_sound function is called with &my_dog, and W is resolved to be u8 because Dog implements the Animal trait with type Weight = u8.

Exercise

Implement a **Display** trait to print a structure using the format specifier {}

```
struct Dog {
    weight: u8,
    age: u8,
    name: String,
}


fn main() {
    let my_dog = Dog {
        weight: 2,
        age: 2,
        name: "Bow Wow".to_string(),
    };

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



Implementing the **Display** trait

- The ***Display*** trait cannot be derived automatically. It needs to be implemented manually for each struct or enum that you want to display in a formatted manner.

- The ***std::fmt::Display*** trait specifically defines the behavior for formatting an object for display as a string. It is commonly used when you want to customize how an object is printed or converted to a string.

- By implementing the ***std::fmt::Display*** trait for a type, you can define how that type should be formatted when it is displayed using formatting macros like ***println!, format!(),write!()*** or when explicitly converting it to a string

Module std::fmt

*The **std::fmt** module defines several traits that you can implement to customize the formatting behavior of your types. The most commonly used trait is **fmt::Display**, which enables you to define how your type is formatted when it is displayed as a string.*

Trait std::fmt::Display

1.0.0 · source · [-]

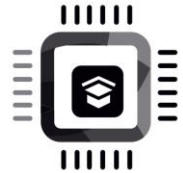
```
pub trait Display {
    // Required method
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

The **fmt** method is a required method of the **fmt::Display** trait. Inside the **fmt** method, you would write the code to format the self object according to your desired format.

The **fmt::Formatter** parameter provides methods for applying formatting options and writing the formatted output.

Ok(()): Represents a successful formatting operation

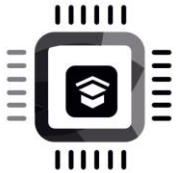
Err(std::fmt::Error): Represents an error that occurred during the formatting process



‘fmt’ module from the Rust standard library

```
use std::fmt;


impl fmt::Display for Dog {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {

    }
}
```

Implementing the **fmt::Display** trait for a Dog type



Struct std::fmt::Formatter

- The ***Formatter*** struct provides methods and using which you can specify the formatting options, set the width, precision, alignment, and perform other formatting operations

- It serves as a destination for writing formatted data :
  - The **Formatter** object is typically provided by the formatting macros (e.g., println!, write!, format!) when they are called, and you don't create it explicitly yourself. It encapsulates the output stream(String, File, Standard output,etc) and provides methods for writing formatted data.

write!()

```
let mut buffer = String::new();
write!(buffer, "Your message {} {}", arg1, arg2, ...)
```

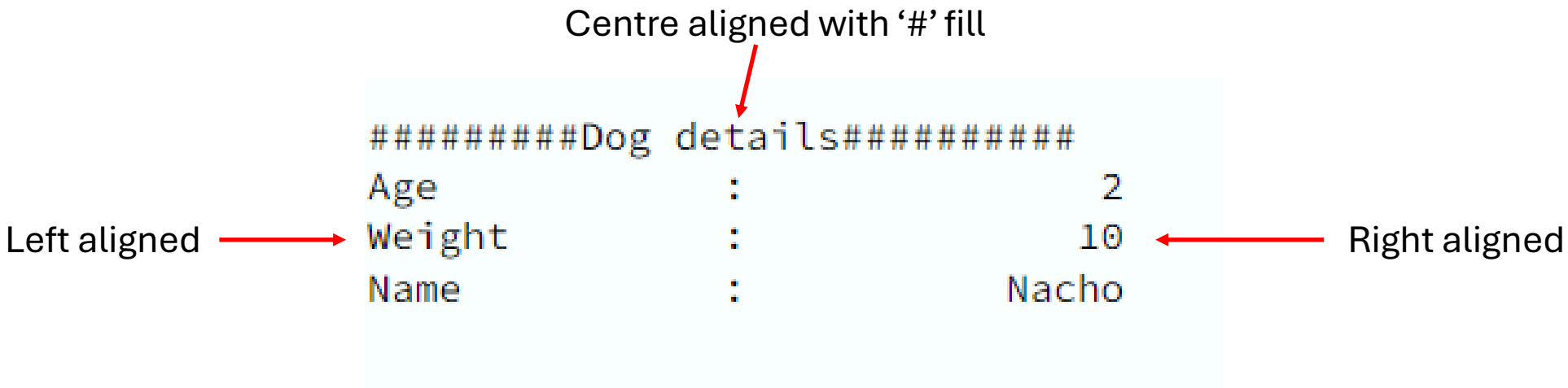
- The **write!** macro is used to write formatted data to a specified output stream.
- **buffer** is the destination(output stream) where the formatted output will be written. It can be any type that implements the **std::fmt::Write** trait, such as String, Vec<u8>, or a file stream

```rust
fn main() {

    let my_dog = Dog {
    weight: 10,
    age: 2,
    name: String::from("Nacho"),
    };

    println!("{}",my_dog);
    println!("{:15}",my_dog);
}
```

```
  Dog details
Age      :          2
Weight   :         10
Name     :      Nacho
```

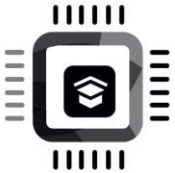
```
        Dog details
Age             :               2
Weight          :              10
Name            :           Nacho
```

Customize how your type is printed or displayed when using formatting macros like **println!** or **format!** by implementing the **fmt::Display** trait, you define how your type is formatted as a string.



- **{}**: Default alignment. Left-aligns the argument within the specified width.
- **{:>width}**: Right-aligns the argument within the specified width.
- **{:width}**: Left-aligns the argument within the specified width.
- **{:^width}**: Center-aligns the argument within the specified width.
- **{:=^width}**: Center-aligns the argument within the specified width with = as the fill character

Trait bounds



```
impl<T> Rectangle<T> {
    fn area(&self) -> T {
        self.width * self.height
    }
}

struct Rectangle<T> {
    width: T,
    height: T,
}

fn main() {
    let rect1 = Rectangle { width: 5.0, height: 10.0 };
    let rect2 = Rectangle { width: 7.5, height: 3.2 };

    println!("Area of rect1: {}", rect1.area());
    println!("Area of rect2: {}", rect2.area());
}
```

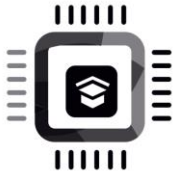
Multiplication operation (*) used in the ***area*** method cannot be applied to generic type **T** directly. The multiplication operation requires the **T** type to implement the ***std::ops::Mul*** trait, which is not guaranteed for all possible types **T**.

T is a generic type that must satisfy two constraints: It must implement the **std::ops::Mul** trait with **Output = T**, and it must implement the **Copy** trait.

Trait bound

```
impl<T: std::ops::Mul<Output = T> + Copy> Rectangle<T> {
    fn area(&self) -> T {
        self.width * self.height
    }
}
```

The **Output = T** constraint is required in the **std::ops::Mul** trait bound (T: std::ops::Mul<Output = T>) to ensure that the multiplication operation returns the same type as the operands.



Trait std::ops::Mul

1.0.0 · source · [-]

```
pub trait Mul<Rhs = Self> {
    type Output;

    // Required method
    fn mul(self, rhs: Rhs) -> Self::Output;
}
```

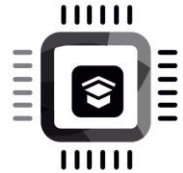
The resulting type after applying the * operator

Trait bounds

- Trait bounds in Rust specify the set of traits that a generic type parameter must implement. They define the behavior and capabilities required for the generic type to be used in a certain context.
- you can think of trait bounds as a way to filter or constrain the types that can be used with a generic function or struct. By specifying trait bounds, you are narrowing down the set of possible types that can be used for a generic parameter to only those types that satisfy the specified traits.

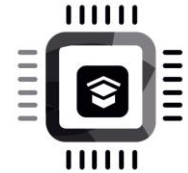
‘where’ clause

You can use the ***where*** clause to improve clarity and readability when specifying multiple trait bounds for a generic type parameter. The ***where*** clause allows you to separate the trait bounds from the rest of the code and provide a clearer definition of the requirements.



```rust
impl<T> Rectangle<T>
    where T: std::ops::Mul<Output = T> + Copy,
{
    fn area(&self) -> T {
        self.width * self.height
    }
}
```

```rust
fn compare<T>(item1: T, item2: T) -> bool
    where T: PartialOrd,
{
    item1 > item2
}
```



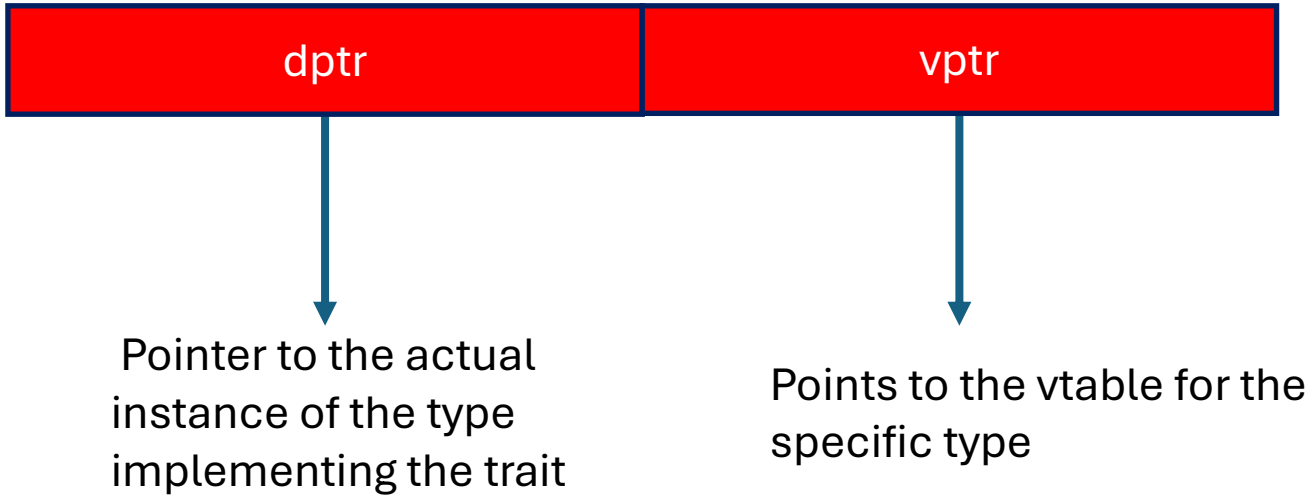
```rust
fn combine<T, U>(item1: T, item2: U) -> (T, U)
    where T: std::fmt::Debug,
          U: std::fmt::Display,
{
    (item1, item2)
}
```

Disambiguating between traits

- When you want to use a trait that is defined in a different module, you typically need to use the full path to specify the trait. This is to disambiguate between multiple traits with the same name that may exist in different modules or crates.
- e.g., The **Display** trait is defined in the **std::fmt** module, so to use it in your code, you need to specify the full path **std::fmt::Display**. This helps the Rust compiler understand which **Display** trait you are referring to, especially when there might be multiple **Display** traits defined in different modules.
- By using **fmt::Display** instead of just **Display**, you are being explicit about which **Display** trait you are implementing or using, which helps prevent confusion and ensures the correct trait is used.
- If a trait is in the Rust prelude, there is no need to use the full path when implementing it. e.g. The **PartialOrd** trait is part of the Rust prelude, and therefore, you don't need to use the full path to implement it.

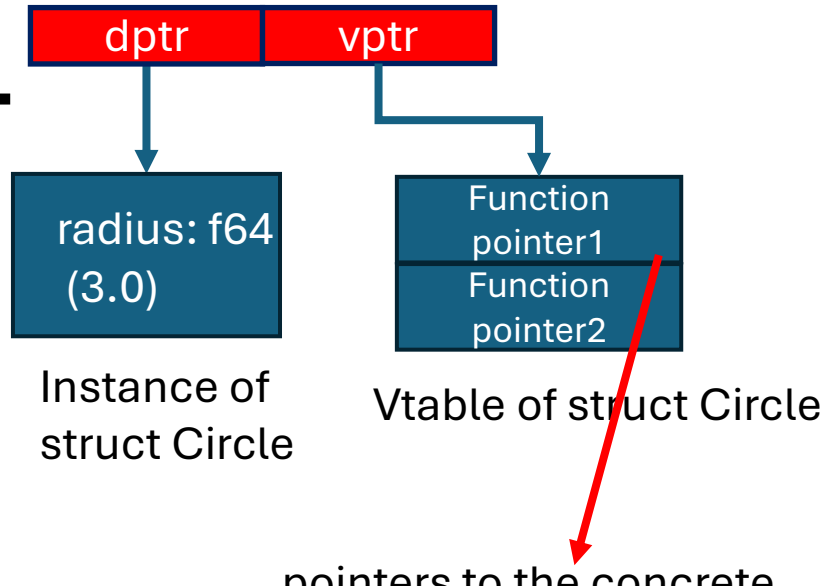
Memory layout of Reference to a Trait Object (&dyn Trait):

- **&dyn Trait** is a reference to a trait object.
- It is a fat pointer that includes a **data pointer** (pointing to the actual data) and a **vtable pointer** (Points to the vtable for the concrete type's implementation of the trait).



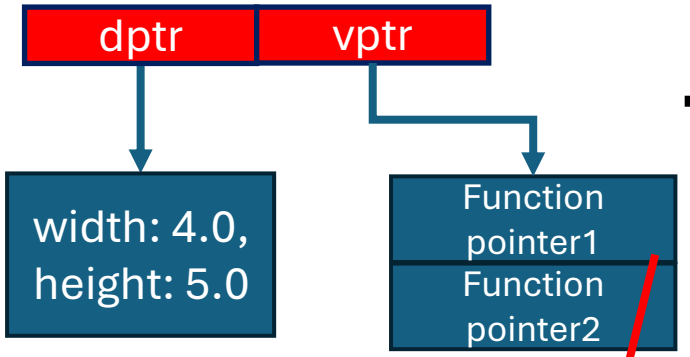
let vec_shapes =

vec![



Instance of
struct Circle

Vtable of struct Circle



Instance of
struct Rectangle

Vtable of struct Rectangle

]

pointers to the concrete implementations of the trait's methods for a this type (struct Circle).

pointers to the concrete implementations of the trait's methods for a this type (struct Rectangle).

```rust
//vec of Shapes
// type of &circle is &Circle but
//it can be coerced to &dyn Shape automatically by the compiler
let vec_shapes: Vec<&dyn Shape> = vec![&circle, &rectangle];
```

Here the vector borrows the trait objects

```rust
let shapes: Vec<Box<dyn Shape>> = vec![ Box::new(circle), Box::new(rectangle)];
```

Here vector own the trait object

Box is a heap-allocated smart pointer type that can be used to store dynamically sized types (DSTs) like trait objects (dyn Trait) and slices ([T]).

Box<**dyn** Shape>

This is called boxing the DST

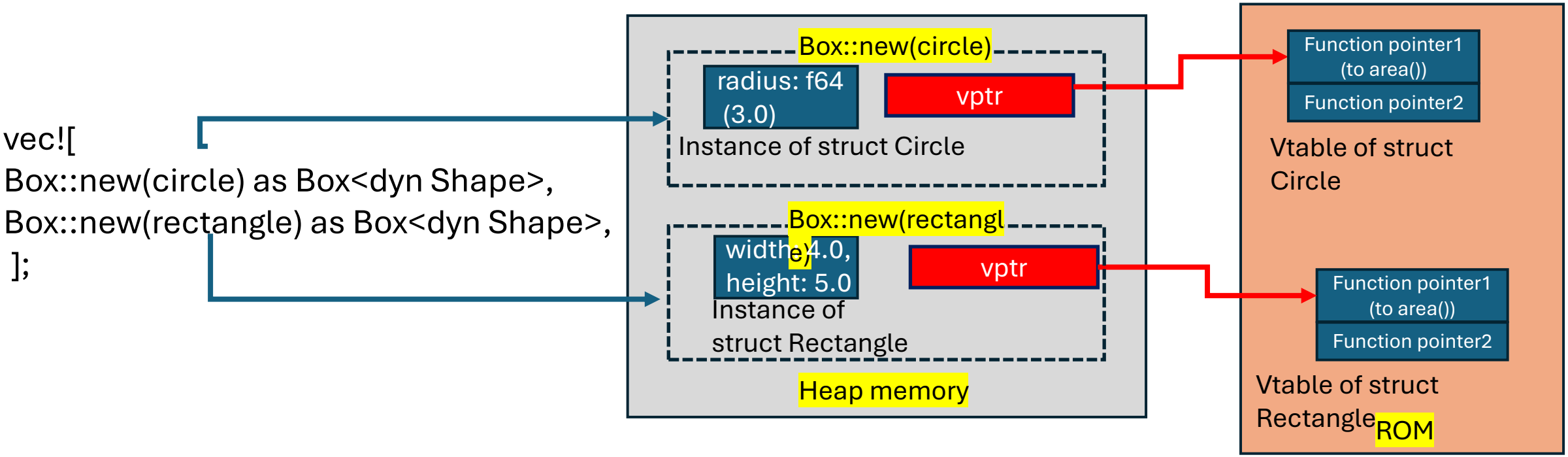
1) Box<T> is a smart pointer in rist that allows you to allocate data on the heap rather than the stack

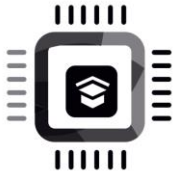
2) Why it is called smart pointer because, Box<T> can automatically handles the allocation and deallocation of heap memory

3) Box adheres to rust's ownership principles, ensuring that there is a single owner for the heap-allocated data at any given time.

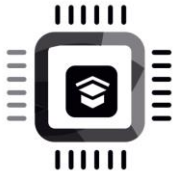
4) Basically , Box<T> encapsulates a raw pointer and makes it "smart" by adding several features and guarantees through its interface

Memory representation of Box<dyn Trait>





Closures



What is a closure?

- A closure is an anonymous function that can capture values from its environment.
- Closures are a powerful tool for writing concise and reusable code in Rust. They can be used to:
  - Pass functions as arguments to other functions
  - Return functions from functions
  - Iterate over collections
  - Write event handlers
  - Create custom iterators
  - store a function in a variable

Closure syntax

The parameters can be any number of variables separated by a comma, and they can be of any type. The closure's body can be any code you want to execute.

Closures in Rust are defined using the pipe |...| syntax to specify the parameters

```
|parameters| -> return-type {
  // body of closure
}
```

```rust
fn main() {

    // Define a closure that takes an 'i32' parameter 'x' and returns 'x + 1'.
    let closure = |x: i32| {
        return x + 1;
    };

    // Call the closure with the argument '10'.
    let result = closure(10);

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

```rust
let closure = |x: i32| x + 1;
```

You often don't need to mention the parameter and return types in the closure definition because the compiler can infer them most of the time

```rust
fn main() {
    let get_z = |x: i32| {
        let y = x + 1;
        let z = y * 2;
        z
    };

    println!("{}", get_z(10));
}
```

A closure with multi-line body

Closure with multiple parameters

```rust
fn main() {
    // Closure with two parameters: x and y
    let add = |x: i32, y: i32| x + y;

    let result = add(5, 10);
    println!("Result: {}", result);
}
```

Conciseness

Closures can be used to write concise code because they can be defined inline. This means that you can define a closure within the body of another function, without having to define a separate function

equivalent named function
for the given closure

```rust
fn main() {
    // Closure with two parameters: x and y
    let add = |x: i32, y: i32| x + y;

    let result = add(5, 10);
    println!("Result: {}", result);
}
```

```rust
fn add(x: i32, y: i32) -> i32 {
    x + y
}

fn main() {
    let result = add(5, 10);
    println!("Result: {}", result);
}
```

Type inferencing of parameters of closures

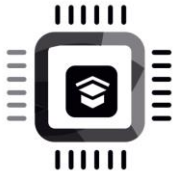
Rust closures do not always require type annotations. The Rust compiler's type inference system can often deduce the types of closure parameters and return values automatically based on how the closure is used.

```rust
fn main() {
    //Explicit type annotation for parameters and return type
    let add = |x: i32, y: i32| -> i32 {x + y};
    let result = add(5, 10);
    println!("Result: {}", result);
}
```

```rust
fn main() {
    let add = |x, y| x + y;
    // Compiler can infer the types of 'x' and 'y' as i32.
    let result = add(5, 10);
    println!("Result: {}", result);
}
```

Attempting to use the closure with different types in subsequent calls

When you define a closure without providing explicit type annotations for its parameters, Rust will try to infer the types from the first usage of the closure. Once the type inference is done, the compiler expects all subsequent uses of that closure to have the same types for its parameters.



```rust
fn main() {
    let add = |a, b| a + b;

    let result1 = add(2_u8, 3_u8);
    println!("Result 1: {}", result1);

    // Error: literal out of range for `u8`
    // Rust expects the closure to take two arguments of the same type (inferred from the first usage).
     let result2 = add(5, 256);
     println!("Result 1: {}", result2);
}
```

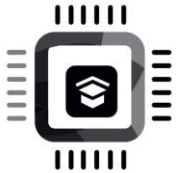
```rust
fn main() {
    let add = |a, b| a + b;

    let result1 = add(2, 3);
    println!("{}", result1);

    //OK
    let result2 = add(5, 7u32);
    println!("{}", result2);
}
```

In this case, the 7u32 value can be implicitly converted to an i32 to match the inferred type of the closure

```rust
fn main() {
    let add = |a, b| a + b;

    let result1 = add(2, 3);
    println!("{}", result1);

    //Error : Mismatched types
    let result2 = add(5, 7.0);
    println!("{}", result2);
}
```

Capturing environment

Closures have the unique ability to capture and use variables from their surrounding environment, which allows them to reference and use variables that are defined outside of their own scope. while regular functions are limited to accessing only the variables passed as arguments or defined within their own scope



Closures without capturing an environment

```
fn main() {
    let add = |a, b| a + b;

    let result1 = add(2, 3);
    println!("{}", result1);

     //OK
    let result2 = add(5, 7u32);
    println!("{}", result2);
}
```

The closure *Add* does not capture any variables from its surrounding environment main()

“Surrounding environment" refers to the scope of the main function

Closure Capturing an Environment

```rust
fn main() {
    let x = 10; // Variable in the outer scope

    // Closure capturing 'x' from the outer environment
    let add = |a| a + x;

    let result1 = add(2); // Closure uses 'x' from the outer scope: 2 + 10 = 12
    println!("Result 1: {}", result1);

    let result2 = add(5); // Closure uses 'x' from the outer scope: 5 + 10 = 15
    println!("Result 2: {}", result2);
}
```

Closure Capturing an Environment

1. Read-only (Immutable) Capture
2. Mutable Capture
3. Transfer of ownership to a closure

Read-only (Immutable) Capture

```
fn main() {
    let x = 10;
    // The closure borrows 'x' immutably.
    let print = || println!("Read x: {}", x);

    print();
    // The closure can be called multiple times as it only has read access to 'x'.
    print();
}
```

Ownership of the variable remains with the original scope. The closure can read the value of the captured variable but cannot modify it.

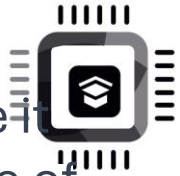
Mutable Capture

If the closure needs to modify the captured variable (mutable access), it borrows the variable mutably. In this case, the closure temporarily holds a mutable reference to the variable, but the ownership still remains with the original scope. The closure can modify the value of the captured variable.

```
fn main() {
    let mut x = 10;
    let mut print = || {
        // The closure borrows 'x' mutably and modifies it.
        x += 1;
        println!("Modified x: {}", x);
    };
    print();
    // The closure can still be called multiple times and modify 'x'.
    print();
}
```

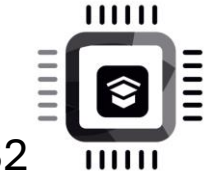
Transfer of ownership to a closure

- A closure takes ownership of a captured variable that owns a non-copy type when it moves the variable inside its body, resulting in a transfer of ownership from the original variable to the closure.
- **move** keyword is used to transfer ownership of variables to a closure when capturing them.



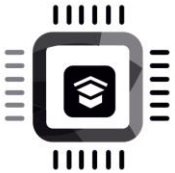
The closure ‘print’ has its own copy of x because it took ownership, not just a reference. The lifetime of this owned x is now tied to the closure, not to the scope in which x was originally defined.

```rust
fn main() {
    let mut print;

    {
        // Creating a new scope to limit the lifetime of 'x'.
        let mut x = 10;

        // Defining a closure 'print' and capturing 'x' with the 'move' keyword.
        print = move || {
            // The closure takes ownership of 'x' and mutates it.
            x += 1;
            println!("Modified x: {}", x);
        };
    } // The scope ends here, 'x' goes out of scope and is dropped.

    print();
    print();
}
```



when a variable implements the Copy trait, as i32 does, this ownership transfer effectively results in a copy of the value, not a move in the traditional sense.

```rust
fn main() {
    let mut x = 10;
    let mut print = move || {
        x += 1;
        println!("Modified x: {}", x);
    };
    print(); //Output : Modified x: 11
    println!("{}",x); //Output : 10
}
```



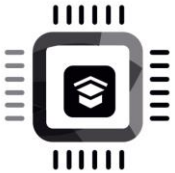
```rust
fn main() {
    let mut x = "10".to_string();
    let mut print = move || {
        x += "22";
        println!("Modified x: {}", x);
    };
    print();
    println!("{}",x);//Error
}
```

Unlike i32, String in Rust does not implement the Copy trait, so the move keyword will take ownership of x rather than just copying it.

Will this code compile?

```
fn main() {
    let mut x = "10".to_string();
    let print = || {
        x += "22";
        println!("Modified x: {}", x);
        x
    };

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

The closure modifies x and then returns it, consuming **x** in the process. This behaviour means the closure implements the ***FnOnce.***



Closures in Rust can implement one of three traits*:*

Fn, FnMut, *and* ***FnOnce.***

Understanding Rust closures and their traits.

1. There are three traits related to closures: **FnOnce, FnMut, and Fn**.
2. All closures implement the **FnOnce** trait, meaning they can all be called at least once.
3. The **FnOnce** trait allows a closure to consume the variables it captures from its environment. After being called once, a **FnOnce** closure cannot be called again as the environment has been consumed.
4. If a closure does not consume its environment, it can also implement the **FnMut** trait. This means the closure can be called multiple times and can mutate the environment.
5. If a closure does not mutate its environment, it can also implement the **Fn** trait. This means that the closure can be called multiple times without mutating the environment

Closure trait hierarchy

Rust determines the appropriate trait for a closure based on how it interacts with its surrounding values.

Fn -> FnMut -> FnOnce

1. If a closure implements **Fn,** it also implements **FnMut** and **FnOnce**.
2. If a closure implements **FnMut**, it also implements **FnOnce**.
3. A closure implementing **FnOnce** may not necessarily implement **FnMut** or **Fn**.
4. A closure implementing **FnMut** may not necessarily implement **Fn**.

Passing a closure as an argument to a function

1. Each closure in Rust has its own unique type that doesn't have a type name, so you can't directly write out the type like you can with i32 or String.

2. To be able to use closures as parameters in functions, we use what's called a "generic function". This is a function that can work with multiple types, not just one specific type.

3. Within these generic functions, we use "trait bounds" to specify what types of closures (or other types) the function can accept.

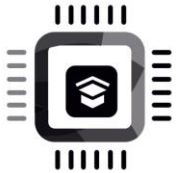
4. For example, we might say the function can take any type that can be called like a function with a certain type of argument and return value, which we write as Fn(i32) -> i32.

Passing a closure as an argument to a function

When passing closures to functions in Rust, you generally have two options:

1.Using function pointers: closures that do not capture any variables from their environment can be coerced to ***fn*** function pointer types. This means the closure does not hold any state and can be represented as a simple function pointer.

2.Using generic functions with trait bounds: This method is used when your closure does capture values from its scope. These closures implement one of the Fn, FnMut, or FnOnce traits, depending on how they use the values they capture.



Using fn pointers

```rust
// A function that takes a function pointer as an argument
fn apply(f: fn(i32) -> i32, x: i32) -> i32 {
    f(x) // Call the function
}

fn main() {
    let y = 2;
    // A closure that doesn't capture anything from its environment.
    let multiply = |x| x * x;
    // Pass the closure to 'apply'
    let result = apply(multiply, 5);//OK
    println!("Result: {}", result);
}
```

Closures that do not capture any variables from their environment can be coerced to ***fn*** function pointer types.

Passing a closure as an argument to a function

1. **For Functions Expecting FnOnce Closures**:
    1. You can pass any type of closure (Fn, FnMut, or FnOnce).
    2. The function might call the closure only once, and it might consume the captured variables.

2. **For Functions Expecting FnMut Closures**:
    1. You can pass a Fn or FnMut closure.
    2. The function can call the closure multiple times, and the closure might mutate the captured variables.
    3. A purely FnOnce closure (that consumes its environment in a way that makes it unsuitable for multiple calls) might not be suitable.

3. **For Functions Expecting Fn Closures**:
    1. You can only pass a Fn closure.
    2. The function can call the closure multiple times.
    3. The closure does not mutate or consume its captured variables.

Using generic functions with trait bounds

```rust
// This function accepts a closure that takes an i32 and returns an i32
apply<F>(f: F, x: i32) -> i32
where
    F: Fn(i32) -> i32,
{
    f(x) // Call the closure
}

fn main() {
    let y = 2;
    // This closure captures 'y'
    let multiply = |x| x * y;
    let result = apply(multiply, 5); // Now it can be passed to 'apply'
    println!("Result: {}", result); // Prints: "Result: 10"
}
```

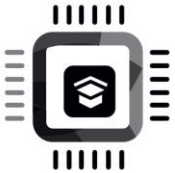
Closures as struct member fields

Storing a closure in a struct can be useful in several scenarios:

1. Event handling
2. Customizable behavior
3. Deferred execution
4. Iterator adapters

Methods to store a closure in a struct

1) As a trait object
2) As a generic struct



As a trait object

We can store closures in struct fields by using a trait object, like *Box<dyn Fn()>, Box<dyn FnMut()> or Box<dyn FnOnce()>*. This will allow you to store a closure of any type that implements the specific trait.

```
struct MyStruct {
    val: i32,
    action: dyn Fn(i32) -> i32
}
```

Trait object

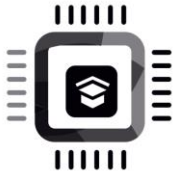
Using `dyn` before a trait name signals that you want to use it as a trait object, allowing dynamic dispatch for any type that implements the trait.

Its size isn't known at compile time because it can represent any type that implements the Write trait

dyn Trait, it represents a trait object. But to use a trait object, it must be behind some pointer-like type, such as:

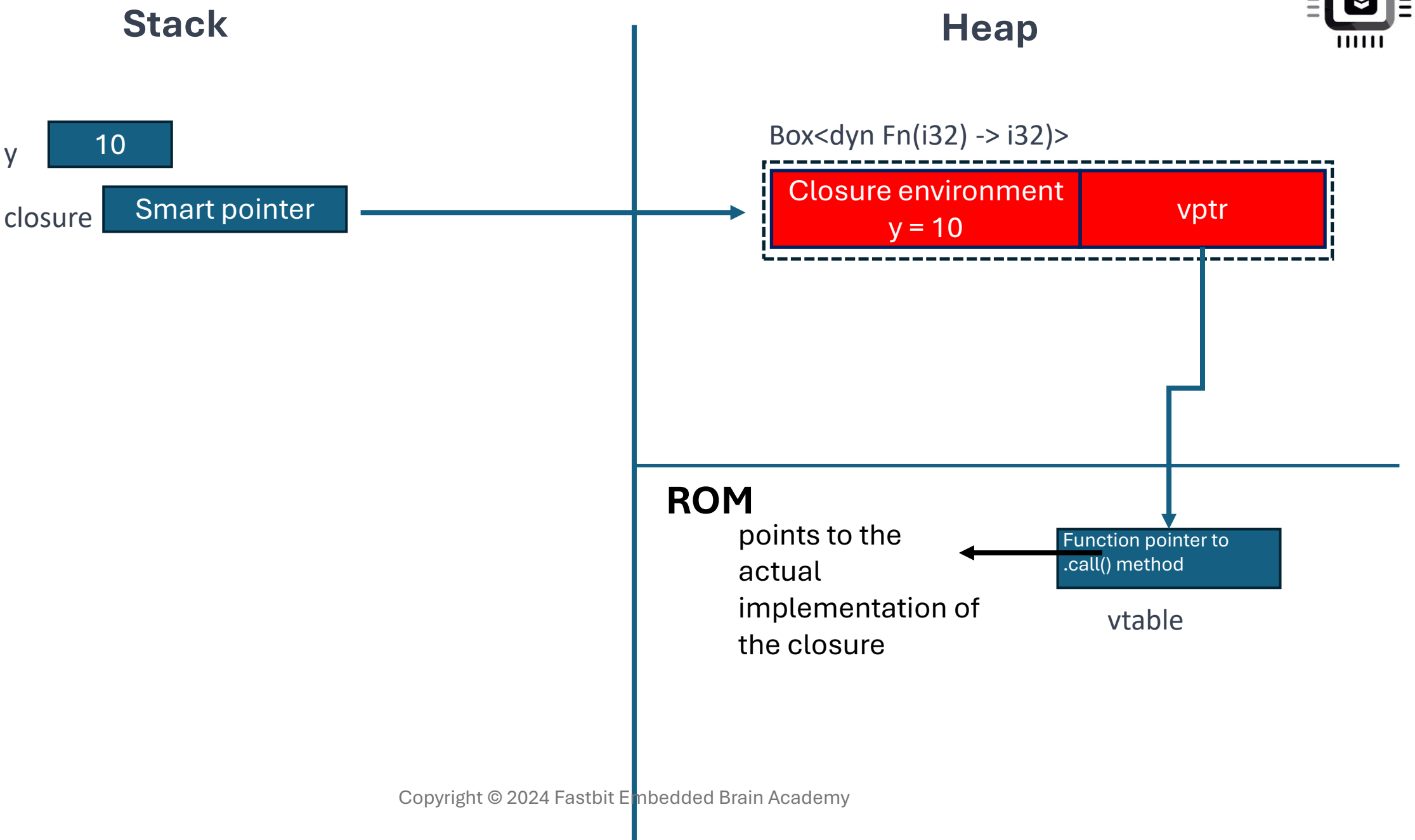
- A reference: &dyn Trait or &mut dyn Trait
- A box: Box<dyn Trait>
- Other smart pointers or custom types that implement the Deref trait, like Rc<dyn Trait>, Arc<dyn Trait>, etc.

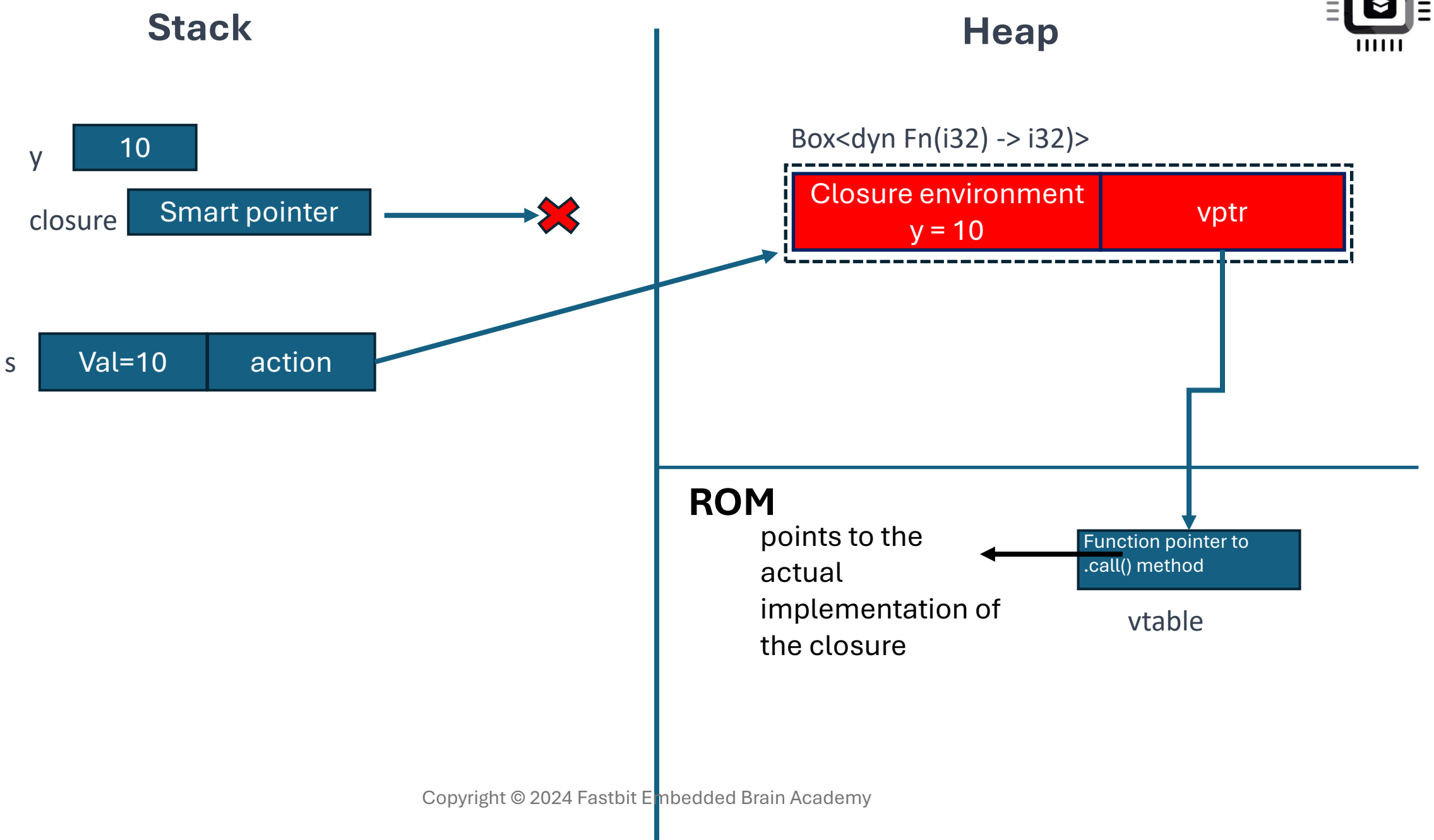
```
struct MyStruct {
    val: i32,
    action: Box<dyn Fn(i32) -> i32>,
}
```



```
struct MyStruct {
    val: i32,
    action: Box<dyn Fn(i32) -> i32>,
}
```

action is a Box that contains a trait object for the Fn(i32) -> i32 trait. This means that action can hold any type of closure (or other object) that implements the Fn(i32) -> i32 signature. The use of Box here is necessary because trait objects have a dynamic size, so they need to be boxed (or behind some other kind of pointer) to be stored.





Type of a closure

When you write a closure in Rust, you don't explicitly mention its type because closures have an anonymous, compiler-generated type. Each closure expression produces a unique type, even if the behaviour and captured environment of two closures are identical.

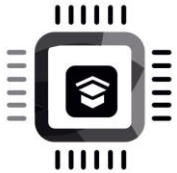
Exercise

```
struct TaxCalculator {
    calculation: Box<dyn Fn(f32) -> f32>,
}
```

VAT = amount * 0.2
IncomeTax = amount * 0.3

You can create multiple instances of TaxCalculator with different closure behaviors, allowing each instance to perform tax calculations based on different logic or rules.

This is an example of how closures can provide flexibility and allow for behaviour to be customized at runtime. It's also a good example of how boxed trait objects can be used to store closures with different behaviours in the same struct.



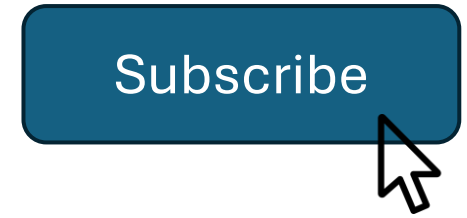
Closures as struct member fields

Storing a closure in a struct can be useful in a number of scenarios:

1. Event handling
2. Customizable behavior
3. Deferred execution
4. Iterator adapters

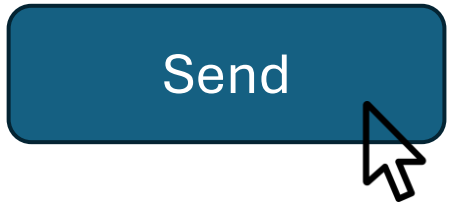
Event handling

Closures are commonly used in event handling, especially in GUI libraries and frameworks.



Subscribed! Total subscriptions: 1

Hi, Please call me



Your message sent: Hi, Please call me.