Collections and String in Rust

Hitesh Kumar Sharma
Instructor, Pluralsight

Agenda

- About Collections
- Types of Collections
- Collection Examples
- About String
- String Literal
- String Object

Collection

A systems programming language called Rust is renowned for its speed, dependability, and robust memory safety guarantees. Rust's collection types, which offer effective and secure ways to handle and manipulate data, are one of its fundamental characteristics.

Rust’s collections can be grouped into following four categories:

Sequence	Map	Sets	Misc
Vec	HashMap	HashSet	BinaryHeap
VecDeque	BTreeMap	BTreeSet	
LinkedList			

Vec

A dynamic array or vector that allows you to store a variable number of elements of the same type. Vectors are analogous to arrays in other programming languages but with more flexibility.

Acceptance Use-Case:

Vec is a growable array type, which enables runtime size changes. When you require a dynamic array with quick random access and when the sequence of the elements matters, you should use a Vec.

Avoidance Use-Case :

If you require a fixed-size array or a collection with constant-time insertions or deletions at any location, we should avoid using Vec.

Vec (Example)

```
fn main() {
    // Create an empty vector to store integers
    let mut numbers: Vec<i32> = Vec::new();
    // Add elements to the vector
    numbers.push(10);
    numbers.push(20);
    numbers.push(30);
    // Access elements in the vector
    println!("Vector: {:?}", numbers);
    // Access individual elements by index
    let first_number = numbers[0];
    let second_number = numbers[1];
    println!("First number: {}", first_number);
    println!("Second number: {}", second_number);
```

In this example, we create a vector called numbers to store i32 (32-bit integer) elements. We add three elements (10, 20, and 30) to the vector using the push method. Then, we access and print the vector, individual elements

LinkedSet

LinkedList is a doubly-linked list that allows you to efficiently insert and remove elements at the beginning or end of the list.

Acceptance Use-Case:

A doubly-linked list, LinkedList offers constant-time insertions and removals at the list's beginning and end. When you require a collection with quick insertions and deletions on both ends, or when you have numerous elements and need to reduce reallocation costs, use LinkedList.

Avoidance Use-Case :

Avoid using LinkedList if you need a collection with quick random access because traversing the list is required to access elements by index.

LinkedSet (Example)

```
use std::collections::LinkedList;

let mut PL_LL = LinkedList::new();
PL_LL.push_back(“John");
PL_ LL.push_back(“Tom");
PL_ LL.push_front(“Jerry");
let first = PL_ LL.pop_front().unwrap();
println!("First removed: {}", first);
for PL_LL in &PL_LL {
    println!("PL_ LL : {}", PL_ LL);
}
```

In this example, we create a LinkedList called PL_LSet, add elements to it, remove elements, and iterate through the elements. This demonstrates how LinkedList can efficiently manage insertions and deletions at both ends of the list.

HashMap

A hash map is an unordered collection of key-value pairs, where each key must be unique. It provides fast lookup, insertion, and deletion of elements based on the keys.

Acceptance Use-Case :

A collection called a hash map stores key-value pairs in an unordered fashion. When you need to quickly access, insert, or remove elements using a certain key, use hash maps.

Avoidance Use-Case :

If you need a sorted collection or to keep the order of the items, stay away from hash maps.

HashMap (Example)

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

   let mut Cricket = HashMap::new();
   scores.insert(String::from(“Bat"), 2000);
   scores.insert(String::from(“Ball"), 200);
   for (key, value) in & Cricket {
       println!("{key}: {value}");
   }
```

This code will print each pair in an arbitrary order:

Bat: 2000
Blue: 200

BTreeMap

A balanced binary tree-based map that maintains the keys in sorted order. It's useful when you need the keys to be sorted.

```
use std::collections::BTreeMap;

let mut ps_btms = BTreeMap::new();
ps_btms.insert(“John", 10);
ps_btms.insert(“Tom", 20);
ps_btms.insert(“Jerry", 30);
if let Some(ps_btm) = ps_btms.get(“John") {
    println!(“John Record is: {}", ps_btm );
}
for (name, ps_btm) in & ps_btms {
    println!("{}: {}", name, ps_btm);
}
```

In the above Example, We construct a BTreeMap called ps_btms, add key-value pairs, access a value by its key, and iterate through the key-value pairs in sorted order in this example. This illustrates how range-based queries can be run on BTreeMap to manage sorted collections.

HashSet

HashSet is an unordered collection of unique values. It is useful when you want to store a unique set of items and perform set operations.

Acceptance Use-Case :

An unordered collection called a hash set is used to store distinct elements. When you need to quickly verify an element's existence, establish uniqueness, or carry out set operations like union and intersection, use hash sets.

Avoidance Use-Case :

If you need a collection with key-value pairs or the order of the elements to be maintained, stay away from hash sets.

HashSet (Example)

```
use std::collections::HashSet;

let mut shapes = HashSet::new();
shapes.insert(“square");
shapes.insert(“circle");
shapes.insert(“rectangle");
let contains_red = shapes.contains("square");
println!("Contains square? {}", contains_ square);
for shape in &shapes {
    println!(“Shape: {}", shape);
}
```

In Above example, we construct a HashSet called shapes, add items to it, check to see if an element is there, and then loop through the elements. This exemplifies how efficiently HashSet can manage unique elements and carry out set operations.

BTreeSet

A balanced binary tree-based Set that maintains the values in sorted order. It's useful when you need the value to be sorted.

```
use std::collections::BTreeSet;
let mut books = BTreeSet::new();
books.insert("A Dance With Dragons");
books.insert("To Kill a Mockingbird");
books.insert("The Odyssey");
books.insert("The Great Gatsby");
for book in &books {
    println!("{book}");
}
```

In Abovexample, we construct a BTreeSet called books, insert items to it, check to see if an element is there, and then loop through the elements. This exemplifies how efficiently BTreesSet can manage unique elements and carry out set operations.

String

The **`String`** type is used to represent a mutable, growable, and UTF-8 encoded text. It is one of the most commonly used data types for handling textual data and is part of the Rust standard library (std::string::String). Strings in Rust are different from string literals, which are static and fixed-length.

To use the String type, you need to bring it into scope with a use statement or use its fully qualified path.

String is mutable and requires memory allocation. When you are dealing with string manipulation, keep in mind the ownership, borrowing, and lifetimes to avoid unnecessary memory copies and ensure safe and efficient code.

String (Example)

```
fn main() {
    let mut my_string = String::new();
    my_string.push('H');
    my_string.push_str("ello");
    my_string += ", world!";
    println!("{}", my_string);
    let length = my_string.len();
    println!("String length: {}", length);
    let is_empty = my_string.is_empty();
    println!("Is the string empty? {}", is_empty);
}
```

In this example, we create a String called my_string, add characters and strings to it using push and push_str, respectively. We also use the += operator for appending more data to the string. The len method is used to get the length of the string, and is_empty is used to check if the string is empty.

String Literal

A set of characters that is hardcoded into a variable is referred to as a string literal. Let company=“Plural Sight" as an example. Module std::str contains literal strings. String slices are another name for string literals. When the value of a string is known at compile time, String literals (&str) are utilized.

String literals are by default static. Thus, the validity of string literals is ensured throughout the entire program.

String Literal (Example)

```
fn main() {
   let fname:&str=“John";
   let lname:&str = “Dalton";
   println!(“First Name: {} Last Name :{}",fname,lname);
}
```

Above example declares two string literals – *fname* and *lname*.

```
fn main() {
   let fname:&str=“John";
   let lname:&'static str = “Dalton";
   println!(“First Name: {} Last Name :{}",fname,lname);
}
```

We can also explicitly specify the variable as static as shown above.

String Object

The string object type is not a part of the core language, in contrast to the string literal. It is designated as a public structure in pub struct String in the standard library. String is a collection that can grow. It is changeable and of type UTF-8. String values that are supplied at runtime can be represented using the String object type. In the heap, a string object is allocated. The String object type is provided in Standard Library.

Syntax

Syntax to create an empty string:
`String::new()`

Syntax to create string with some default value passed as parameter to the **from()** method:
`String::from()`

String Literal (Example)

```
fn main(){
   let ps_empty_string  = String::new();
   println!("length is {}",ps_empty_string .len());
   let ps_content_string = String::from(“PluralSight");
   println!("length is {}",ps_content_string.len());
}
```

The above example creates two strings, an empty string object using the *new* method and a string object from string literal using the *from* method.

End of Module