Rust - String

The String data type in Rust can be classified into the following −

- String Literal**(&str)**
- String Object**(String)**

String Literal

String literals (&str) are used when the value of a string is known at compile time. String literals are a set of characters, which are hardcoded into a variable. For example, *let company="Tutorials Point"*. String literals are found in module std::str. String literals are also known as string slices.

The following example declares two string literals − *company* and *location*.

```
fn main() {
   let company:&str="TutorialsPoint";
   let location:&str = "Hyderabad";
   println!("company is : {} location :{}",company,location);
}
```

String literals are static by default. This means that string literals are guaranteed to be valid for the duration of the entire program. We can also explicitly specify the variable as static as shown below −

```
fn main() {
   let company:&'static str = "TutorialsPoint";
   let location:&'static str = "Hyderabad";
   println!("company is : {} location :{}",company,location);
}
```

The above program will generate the following output −

```
company is : TutorialsPoint location :Hyderabad
```

String Object

The String object type is provided in Standard Library. Unlike string literal, the string object type is not a part of the core language. It is defined as public structure in standard library *pub struct String*. String is a growable collection. It is mutable and UTF-8 encoded type. The **String** object type can be used to represent string values that are provided at runtime. String object is allocated in the heap.

Syntax

To create a String object, we can use any of the following syntax −

```
String::new()
```

The above syntax creates an empty string

```
String::from()
```

This creates a string with some default value passed as parameter to the **from()** method.

The following example illustrates the use of a String object.

```
fn main(){
   let empty_string = String::new();
   println!("length is {}",empty_string.len());

   let content_string = String::from("TutorialsPoint");
   println!("length is {}",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.

The output is as shown below −

```
length is 0
length is 14
```

Common Methods - String Object

Sr.No.	Method	Signature	Description
1	new()	pub const fn new() → String	Creates a new empty String.
2	to_string()	fn to_string(&self) → String	Converts the given value to a String.
3	replace()	pub fn replace<'a, P>(&'a self, from: P, to: &str) → String	Replaces all matches of a pattern with another string.
4	as_str()	pub fn as_str(&self) → &str	Extracts a string slice containing the entire string.
5	push()	pub fn push(&mut self, ch: char)	Appends the given char to the end of this String.
6	push_str()	pub fn push_str(&mut self, string: &str)	Appends a given string slice onto the end of this String.
7	len()	pub fn len(&self) → usize	Returns the length of this String, in bytes.
8	trim()	pub fn trim(&self) → &str	Returns a string slice with leading and trailing whitespace removed.
9	split_whitespace()	pub fn split_whitespace(&self) → SplitWhitespace	Splits a string slice by whitespace and returns an iterator.
10	split()	pub fn split<'a, P>(&'a self, pat: P) → Split<'a, P> , where P is pattern can be &str, char, or a closure that determines the split.	Returns an iterator over substrings of this string slice, separated by characters matched by a pattern.
11	chars()	pub fn chars(&self) → Chars	Returns an iterator over the chars of a string slice.

Illustration: new()

An empty string object is created using the ***new()*** method and its value is set to *hello*.

```
fn main(){
   let mut z = String::new();
   z.push_str("hello");
   println!("{}",z);
}
```

Output

The above program generates the following output −

```
hello
```

Illustration: to_string()

To access all methods of String object, convert a string literal to object type using the ***to_string()*** function.

```
fn main(){
   let name1 = "Hello TutorialsPoint ,
   Hello!".to_string();
   println!("{}",name1);
}
```

Output

The above program generates the following output −

```
Hello TutorialsPoint , Hello!
```

Illustration: replace()

The ***replace()*** function takes two parameters − the first parameter is a string pattern to search for and the second parameter is the new value to be replaced. In the above example, *Hello* appears two times in the *name1* string.

The replace function replaces all occurrences of the string ***Hello*** with ***Howdy***.

```
fn main(){
   let name1 = "Hello TutorialsPoint ,
   Hello!".to_string();         //String object
   let name2 = name1.replace("Hello","Howdy");    //find and replace
   println!("{}",name2);
}
```

Output

The above program generates the following output −

```
Howdy TutorialsPoint , Howdy!
```

Illustration: as_str()

The **as_str()** function extracts a string slice containing the entire string.

```
fn main() {
   let example_string = String::from("example_string");
   print_literal(example_string.as_str());
}
fn print_literal(data:&str ){
   println!("displaying string literal {}",data);
}
```

Output

The above program generates the following output −

```
displaying string literal example_string
```

Illustration: push()

The **push()** function appends the given char to the end of this String.

```
fn main(){
   let mut company = "Tutorial".to_string();
   company.push('s');
   println!("{}",company);
}
```

Output

The above program generates the following output −

```
Tutorials
```

Illustration: push_str()

The **push_str()** function appends a given string slice onto the end of a String.

```
fn main(){
   let mut company = "Tutorials".to_string();
   company.push_str(" Point");
   println!("{}",company);
}
```

Output

The above program generates the following output −

```
Tutorials Point
```

Illustration: len()

The **len()** function returns the total number of characters in a string (including spaces).

```
fn main() {
   let fullname = " Tutorials Point";
   println!("length is {}",fullname.len());
}
```

Output

The above program generates the following output −

```
length is 20
```

Illustration: trim()

The trim() function removes leading and trailing spaces in a string. NOTE that this function will not remove the inline spaces.

```
fn main() {
   let fullname = " Tutorials Point \r\n";
   println!("Before trim ");
   println!("length is {}",fullname.len());
   println!();
   println!("After trim ");
   println!("length is {}",fullname.trim().len());
}
```

Output

The above program generates the following output −

```
Before trim
length is 24

After trim
length is 15
```

Illustration:split_whitespace()

The ***split_whitespace()*** splits the input string into different strings. It returns an iterator so we are iterating through the tokens as shown below −

```
fn main(){
   let msg = "Tutorials Point has good t
   utorials".to_string();
   let mut i = 1;

   for token in msg.split_whitespace(){
      println!("token {} {}",i,token);
      i+=1;
   }
}
```

Output

```
token 1 Tutorials
token 2 Point
token 3 has
token 4 good
token 5 tutorials
```

Illustration: split() string

The **split() string** method returns an iterator over substrings of a string slice, separated by characters matched by a pattern. The limitation of the split() method is that the result cannot be stored for later use. The ***collect*** method can be used to store the result returned by split() as a vector.

```
fn main() {
   let fullname = "Kannan,Sudhakaran,Tutorialspoint";

   for token in fullname.split(","){
      println!("token is {}",token);
   }

   //store in a Vector
   println!("\n");
   let tokens:Vec<&str>= fullname.split(",").collect();
   println!("firstName is {}",tokens[0]);
   println!("lastname is {}",tokens[1]);
```

```
    println!("company is {}",tokens[2]);
}
```

The above example splits the string ***fullname***, whenever it encounters a comma **(,)**.

Output

```
token is Kannan
token is Sudhakaran
token is Tutorialspoint

firstName is Kannan
lastname is Sudhakaran
company is Tutorialspoint
```

Illustration: chars()

Individual characters in a string can be accessed using the chars method. Let us consider an example to understand this.

```
fn main(){
   let n1 = "Tutorials".to_string();

   for n in n1.chars(){
      println!("{}",n);
   }
}
```

Output

```
T
u
t
o
r
i
a
l
s
```

Concatenation of Strings with + operator

A string value can be appended to another string. This is called concatenation or interpolation. The result of string concatenation is a new string object. The + operator internally uses an *add* method. The

syntax of the add function takes two parameters. The first parameter is *self* – the string object itself and the second parameter is a reference of the second string object. This is shown below −

```
//add function
add(self,&str)->String {
   // returns a String object
}
```

Illustration: String Concatenation

```
fn main(){
   let n1 = "Tutorials".to_string();
   let n2 = "Point".to_string();

   let n3 = n1 + &n2; // n2 reference is passed
   println!("{}",n3);
}
```

The Output will be as given below

```
TutorialsPoint
```

Illustration: Type Casting

The following example illustrates converting a number to a string object −

```
fn main(){
   let number = 2020;
   let number_as_string = number.to_string();

   // convert number to string
   println!("{}",number_as_string);
   println!("{}",number_as_string=="2020");
}
```

The Output will be as given below

```
2020
true
```

Illustration: Format! Macro

Another way to add to String objects together is using a macro function called format. The use of Format! is as shown below.

```
fn main(){
   let n1 = "Tutorials".to_string();
   let n2 = "Point".to_string();
   let n3 = format!("{} {}",n1,n2);
   println!("{}",n3);
}
```

The Output will be as given below

```
Tutorials Point
```