
RUST PROGRAMMING
Ownership, Borrowing and Reference
Kamal kumar mukiri, Apt Computing Labs

FEBRUARY 26, 2024

Press Release: Future Software Should Be Memory Safe

ONCD ▸ BRIEFING ROOM ▸ PRESS RELEASE

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

Read the full report here

WASHINGTON – Today, the White House Office of the National Cyber Director (ONCD) released a report calling on the technical community to proactively reduce the attack surface in cyberspace. ONCD makes the case that technology manufacturers can prevent entire classes of vulnerabilities from entering the digital ecosystem by adopting memory safe programming languages. ONCD is also encouraging the research community to address the problem of software measurability to enable the development of better diagnostics that measure cybersecurity quality.

https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

Contents

Data types in Rust

Primitive Data Types

Scalar Types

- **Integers**:
  - Signed: i8, i16, i32, i64, i128, isize
  - Unsigned: u8, u16, u32, u64, u128, usize
- **Floating-Point Numbers**:
  - f32, f64
- **Character**:
  - char (Unicode scalar value, 4 bytes)
- **Boolean**:
  - bool (true or false)

Compound Types

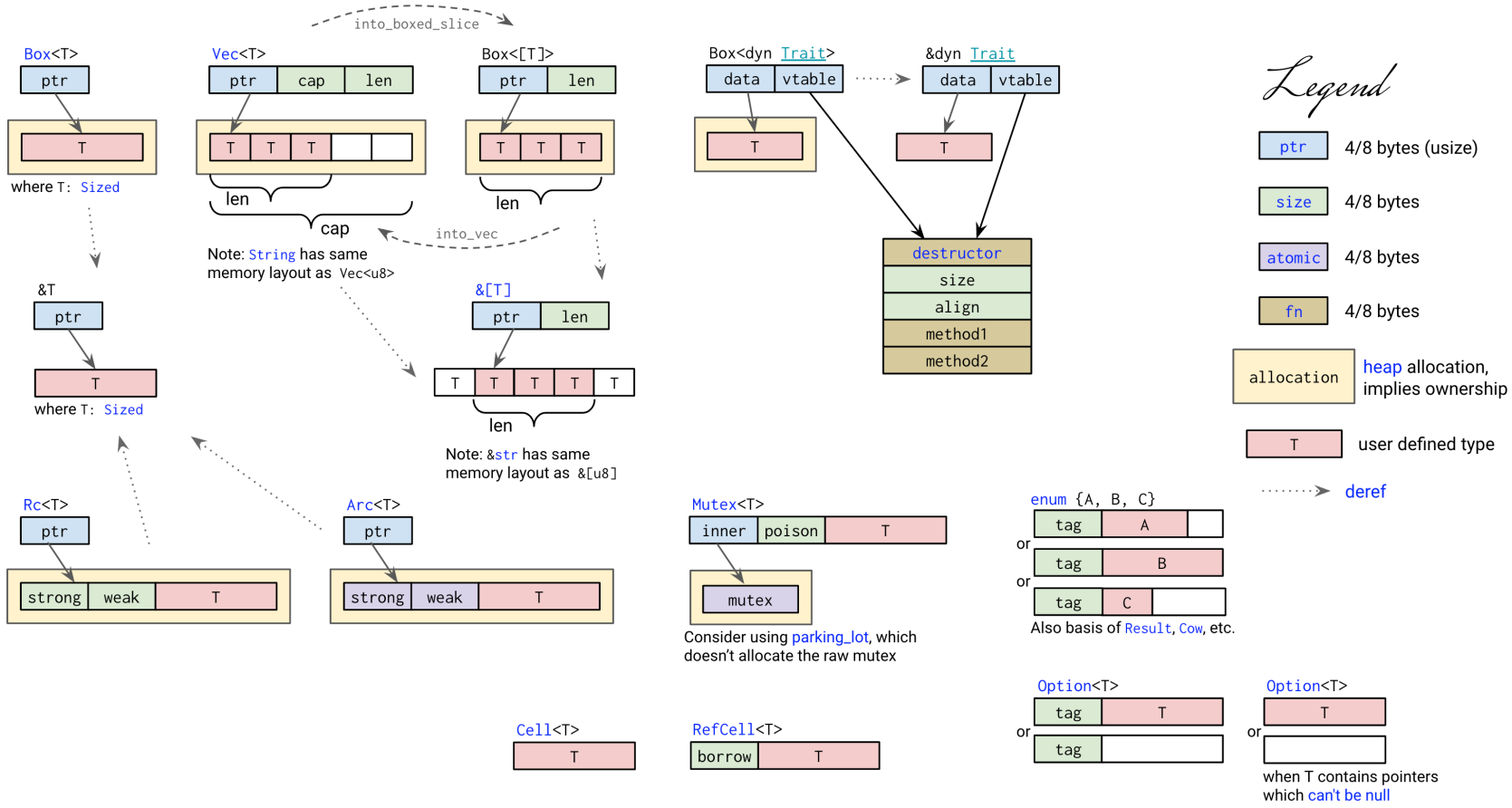
- **Tuples**:
  - A collection of values of different types. Example: (i32, f64, char)
- **Arrays**:
  - A collection of elements of the same type, with a fixed length. Example: [i32; 5] (an array of 5 integers)

Non-Primitive Data Types

Containers

- String
- Vector
- Enums
- Option and Result
- HashMap
- References
- Box, Rc, Arc
- Closures
- Traits
- Etc...

Data types in Rust: Cheat Sheet

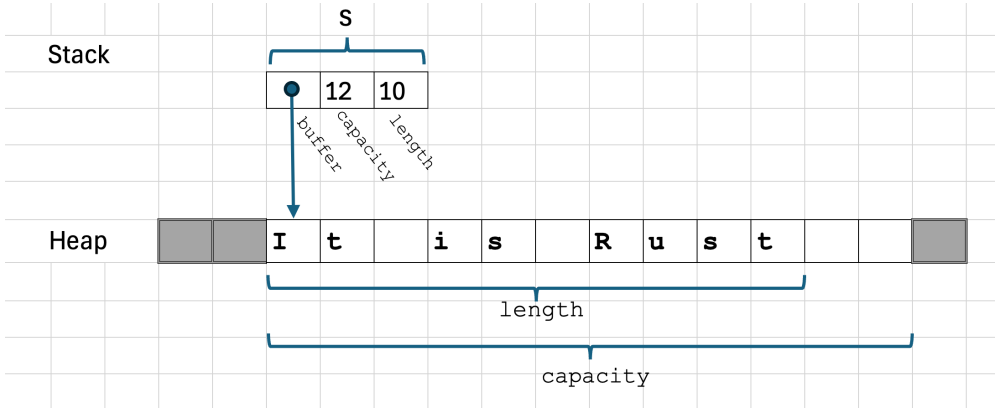


Memory Management in Python


```python
import sys
List1 = ["Linux", "Windows", "Mac"]
List2 = List1 # Increases Ref Count
List3 = List1 # Increases Ref Count
print(sys.getrefcount(List1)) # Prints 4
```

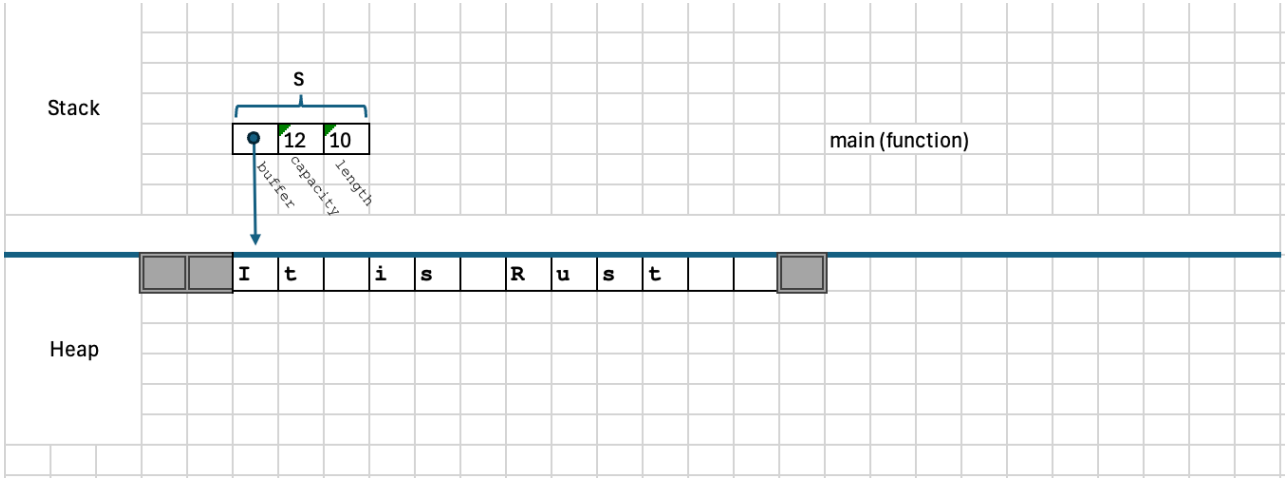
Simple String in C++


```cpp
#include <iostream>
using namespace std;
int main() {
    string s = "It is Rust";
    cout << s << endl;
}
```



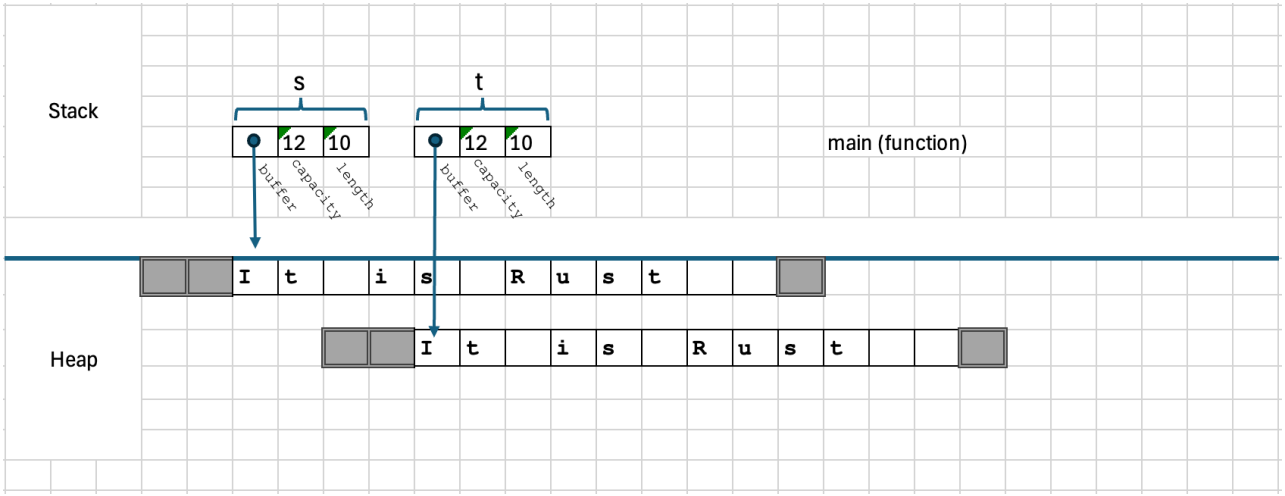
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



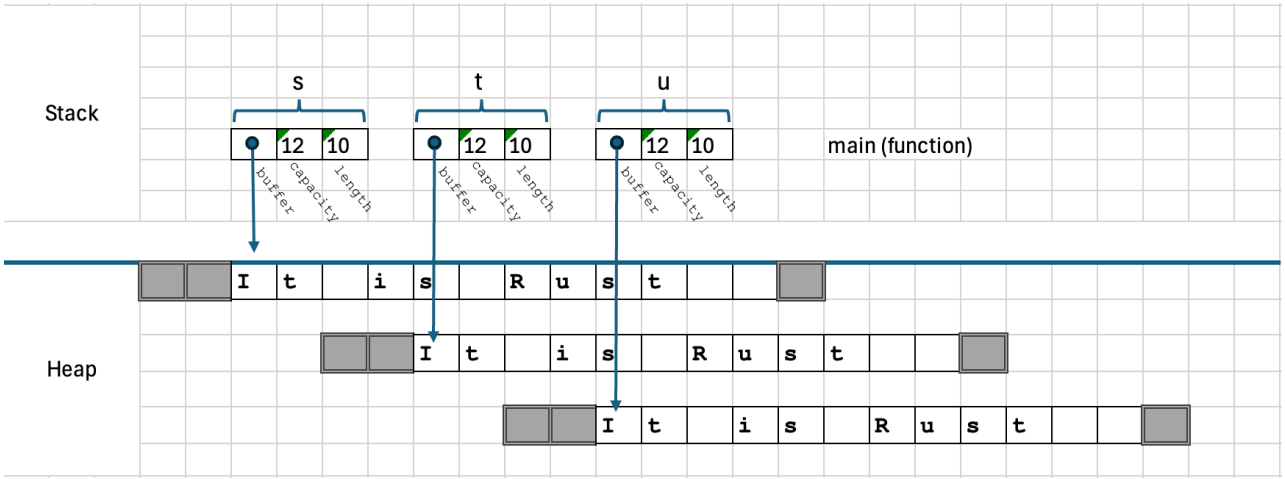
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



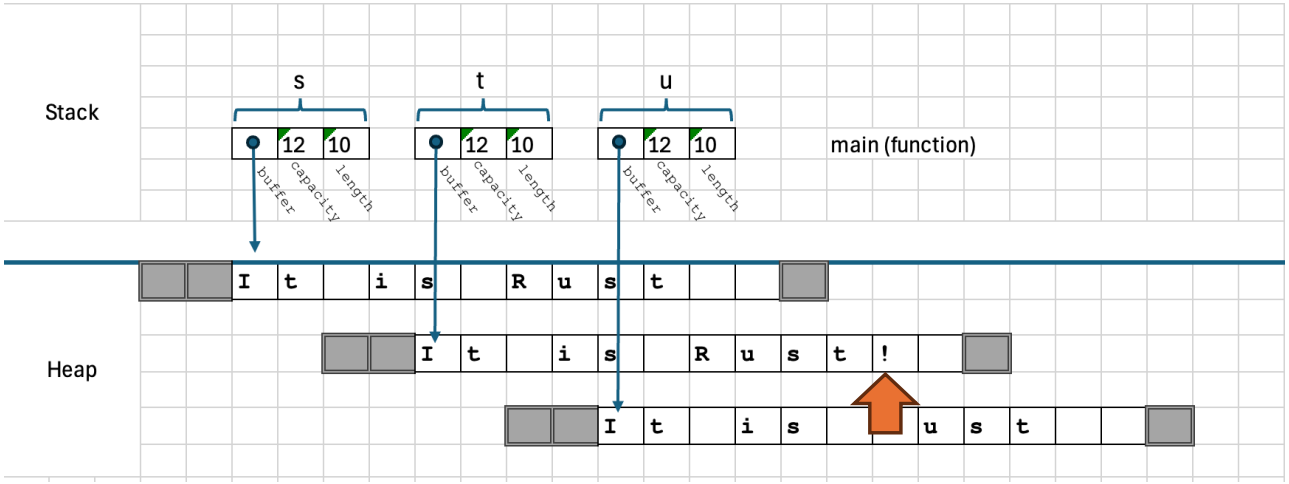
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



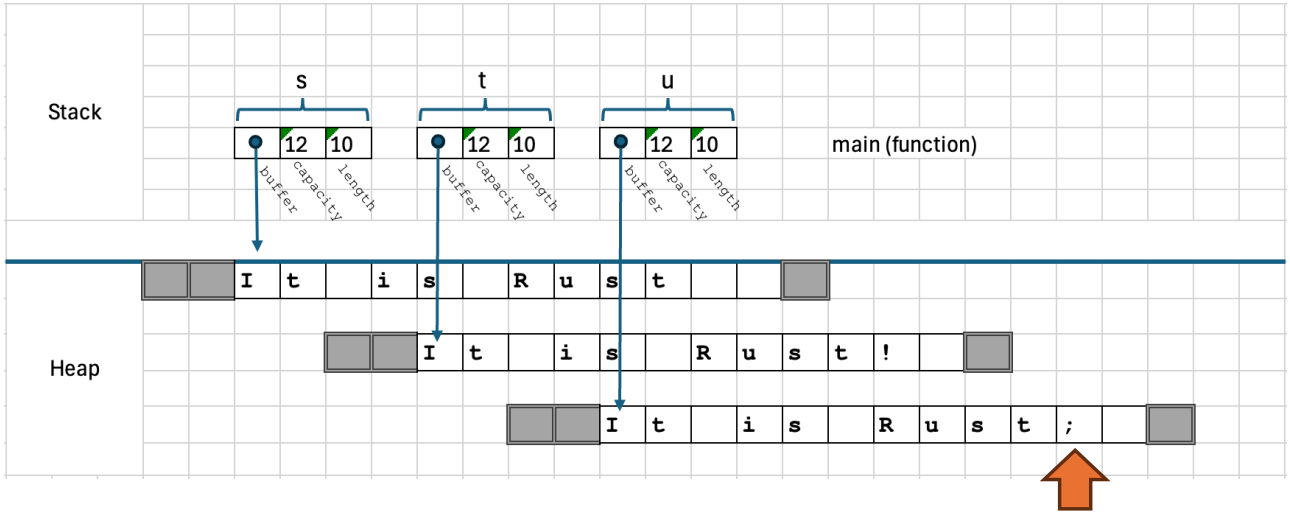
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



Output

```
s = It is Rust
t = It is Rust!
u = It is Rust;
```

Vector (similar to string) ops in C++


```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> s = {1,2,3};
    vector<int> t = s;
    vector<int> u = s;

    t.push_back(4);
    u.push_back(5);
    for (auto num:s)
        cout << "s = "  << num << endl;
    for (auto num:t)
        cout << "t = "  << num << endl;
    for (auto num:u)
    cout << "u = "  << num << endl;
}
```

How vectors are implemented in C++


```
1
2   class MyVector {
3   private:
4       int* buffer;        // Pointer to dynamically allocated array
5       std::size_t capacity; // Total allocated memory
6       std::size_t length;     // Number of elements in the vector
7
8   public:
9       // Constructor
10      MyVector() : data(nullptr),
11
12      // Destructor to free alloca
13      ~MyVector() {
14          delete[] data;
15      }
16
17      // Function to add elements
18      void push_back(int value);
19
20      // Access an element at a gi
21      int& operator[](std::size_t
22  };
```

```
    // Function to resize the internal array when necessary
    void resize() {
        capacity = (capacity == 0) ? 1 : capacity * 2;  // Start with capacity 1 or double it
        int* new_data = new int[capacity];              // Allocate new memory

        // Copy old data to the new array
        for (std::size_t i = 0; i < size; ++i) {
            new_data[i] = data[i];
        }

        // Delete the old memory and point data to the new array
        delete[] data;
        data = new_data;
    }
```

Recall “strdup”

Recall “strtok”

Who do we expect from Programming Language ??

1. Memory should be **freed properly**

2. We never want to use a pointer to an object after it's been freed (**Undefined behavior**).

1. We do not want to read/write **unauthorized memory**.

What RUST compiler offers ???

Safety + Control: Being strict at compile time achieves free of memory safety errors like;

- Dangling pointers
- Double free
- Memory leaks
- Uninitialized memory access
- Sharing data without lock
- etc...

RUST compiler offers...

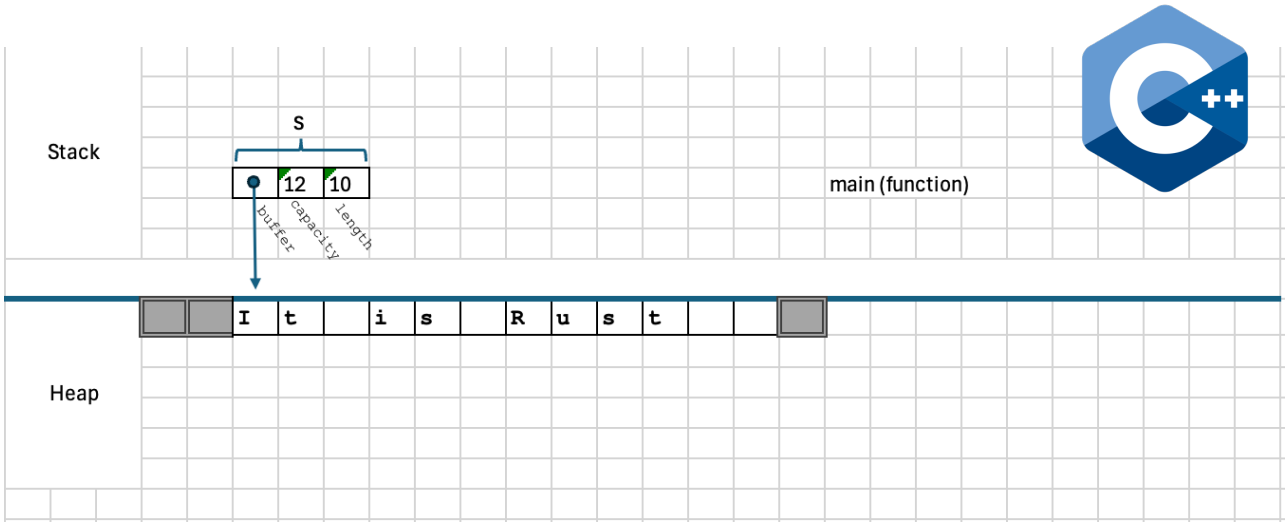
Rust's ownership, borrowing, and references work together to ensure memory safety without the need for garbage collection.

Introduction to Ownership

1. **Each value in Rust has a variable that is its owner.**
2. **There can only be one owner at a time.**
3. **When the owner goes out of scope, the value is dropped.**

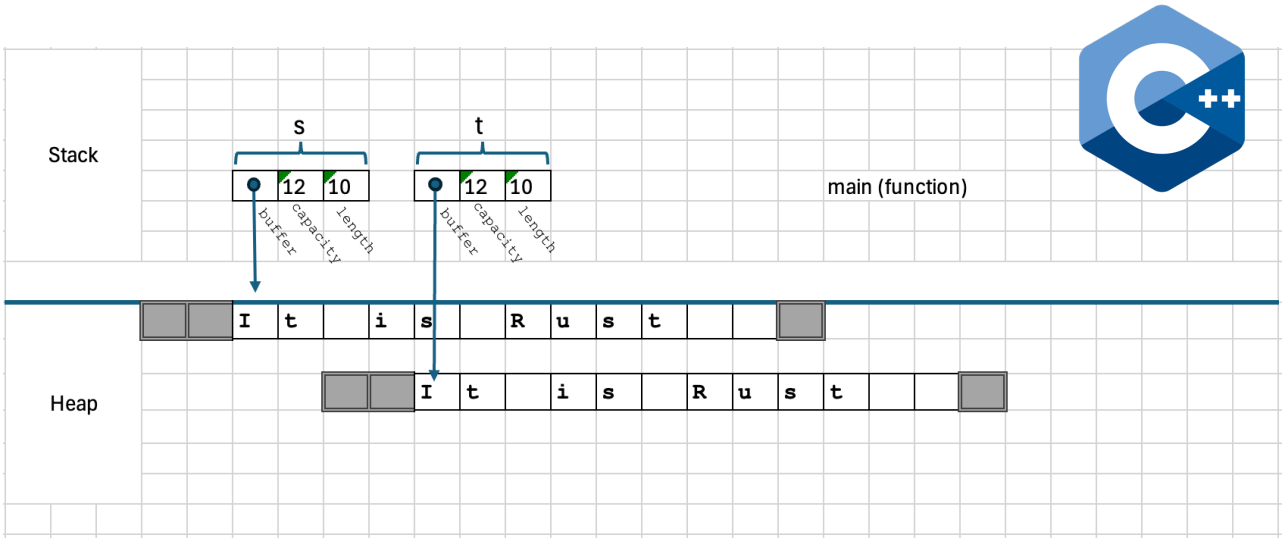
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



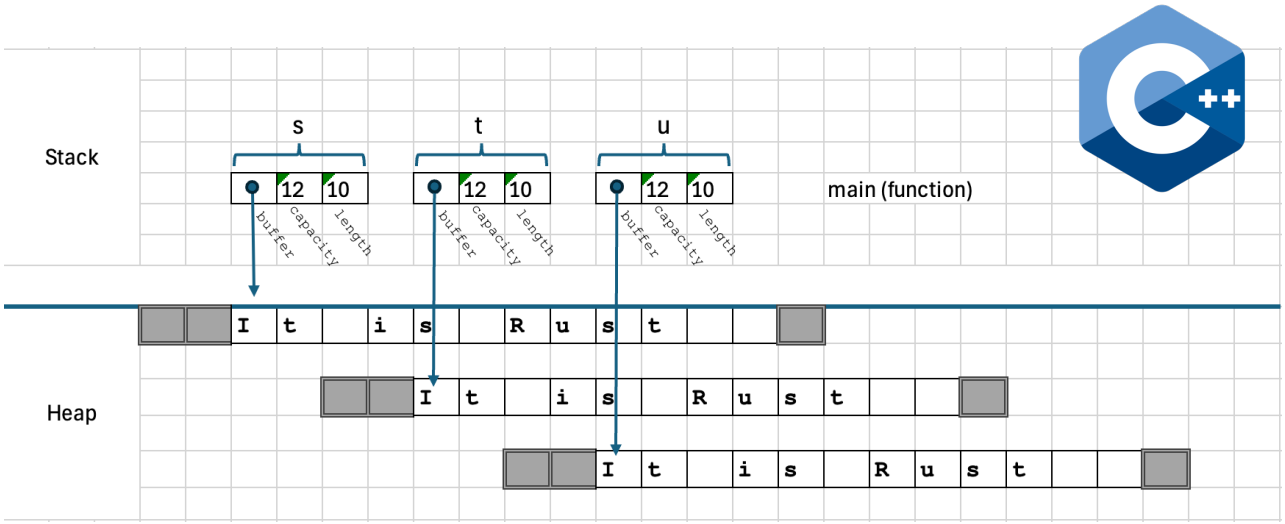
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



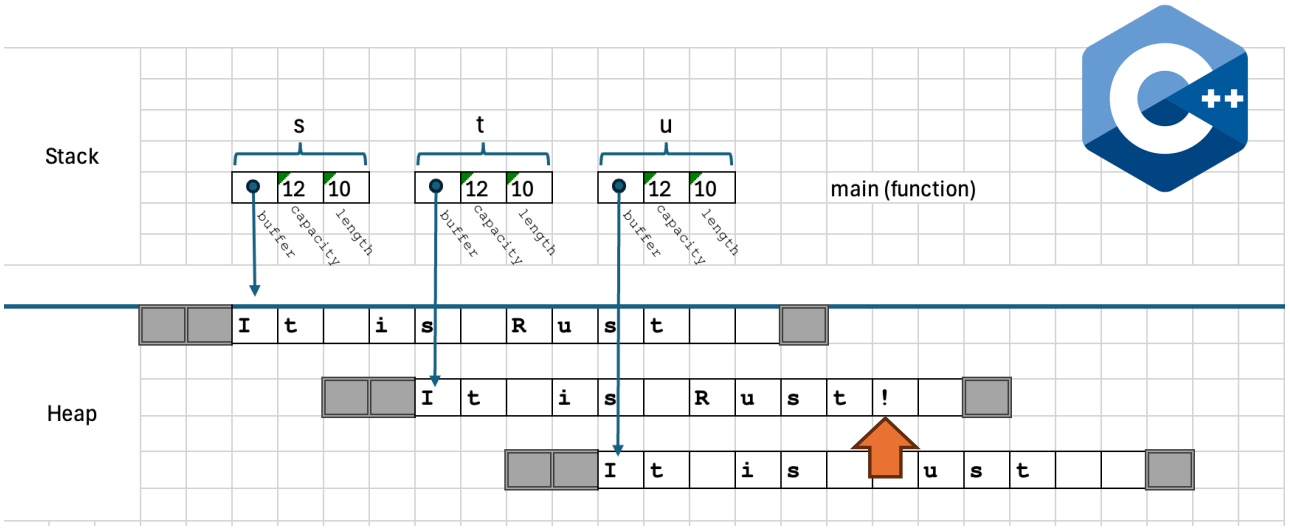
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



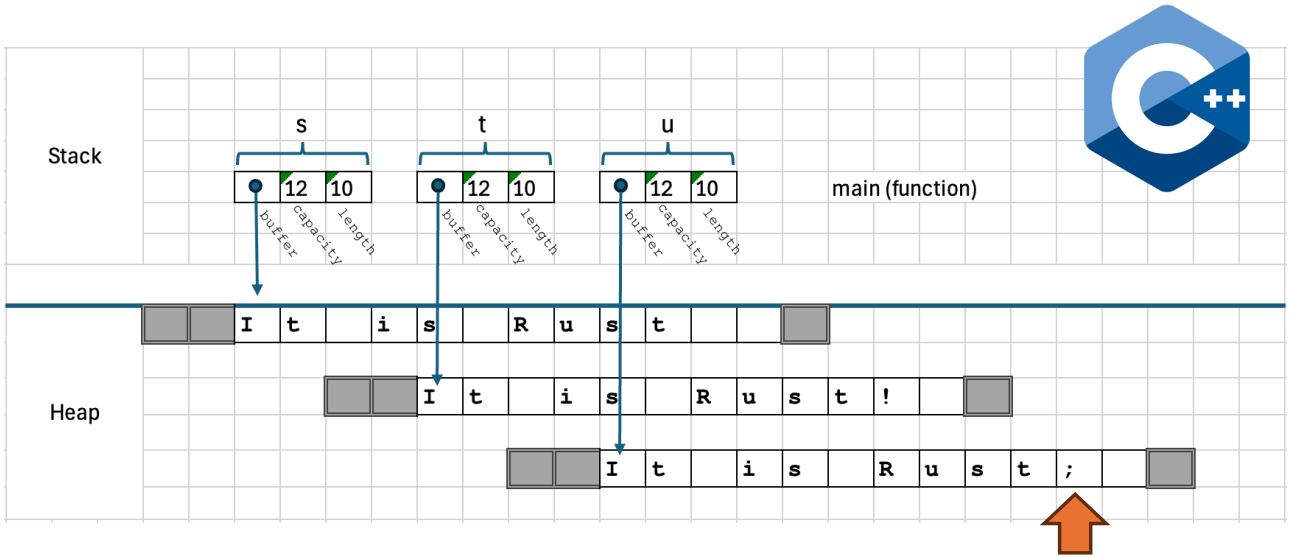
String ops in C++


```
1   #include <iostream>
2   using namespace std;
3   int main() {
4       string s = "It is Rust";
5       string t = s;
6       string u = s;
7
8       t.push_back('!');
9       u.push_back(';');
10      cout << "s = "  << s << endl;
11      cout << "t = "  << t << endl;
12      cout << "u = "  << u << endl;
13  }
```



String ops in C++


```cpp
#include <iostream>
using namespace std;
int main() {
    string s = "It is Rust";
    string t = s;
    string u = s;

    t.push_back('!');
    u.push_back(';');
    cout << "s = "  << s << endl;
    cout << "t = "  << t << endl;
    cout << "u = "  << u << endl;
}
```

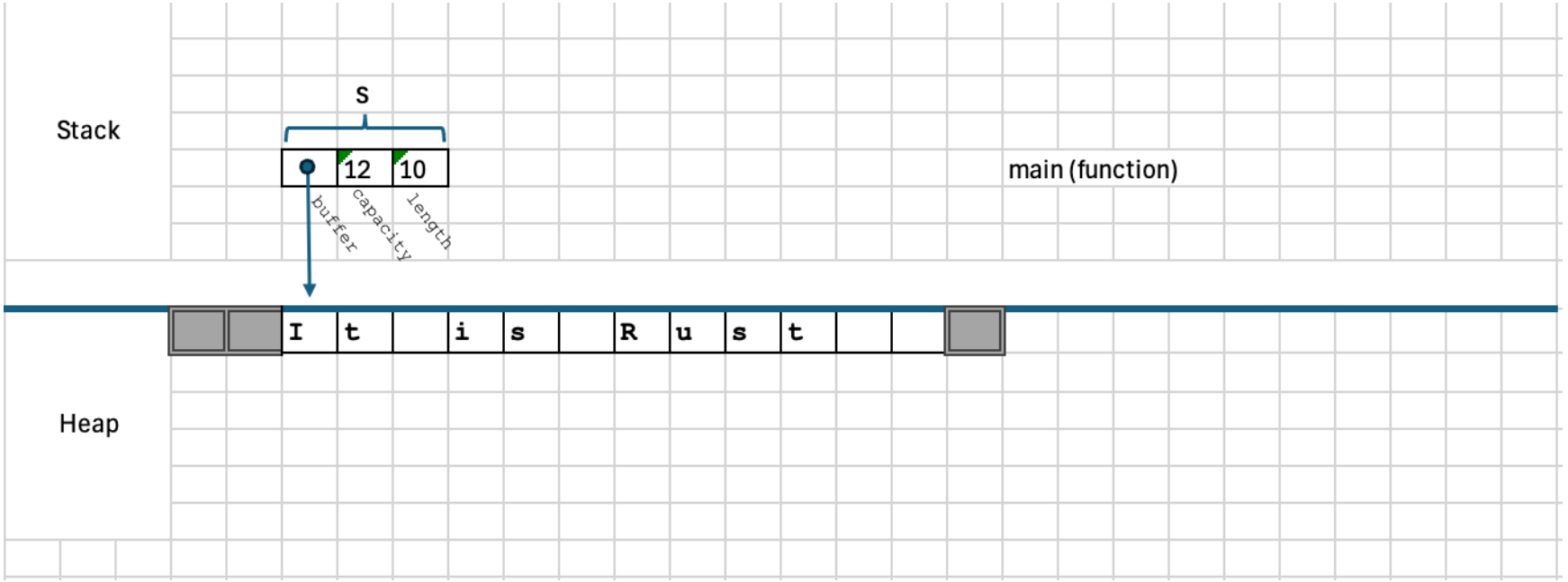


Output

```
s = It is Rust
t = It is Rust!
u = It is Rust;
```

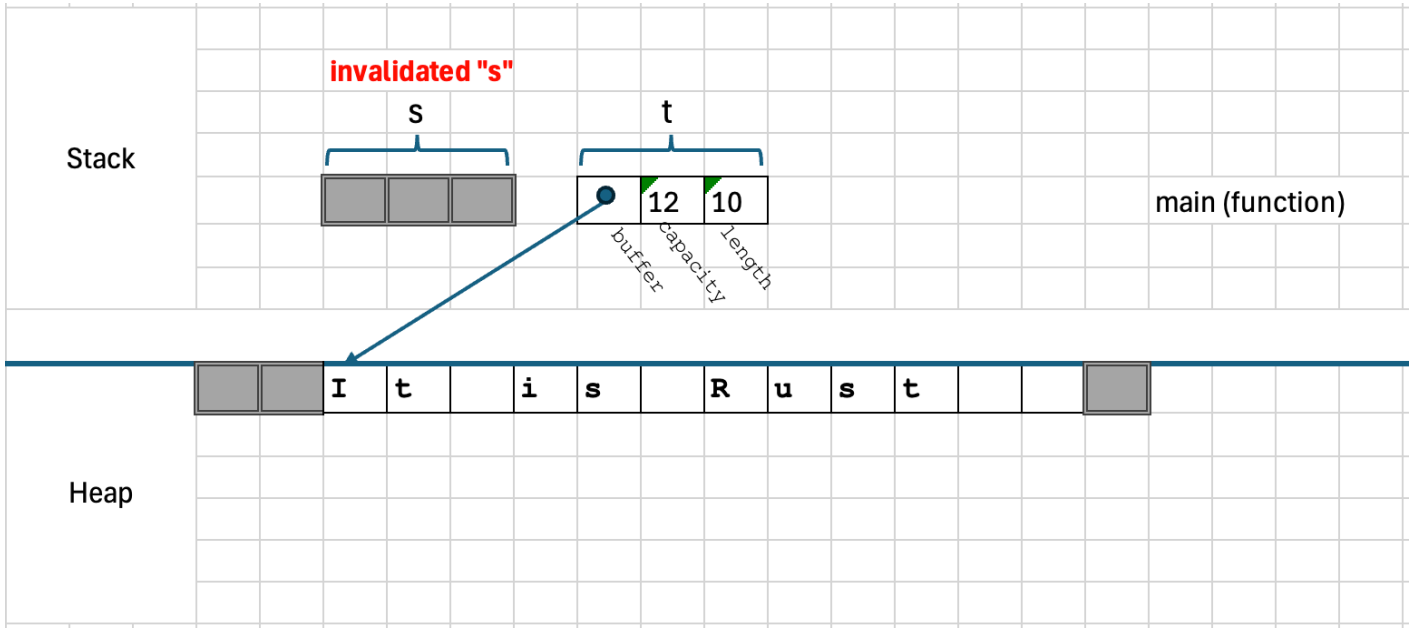
Creating String and ”s” is owner


```
1  fn main() {
2      let s = String::from("It is Rust");
3      let t = s; // moving to t
4      let u = t; // moving to u
5
6      println!("{:?}", u);
7  }
```



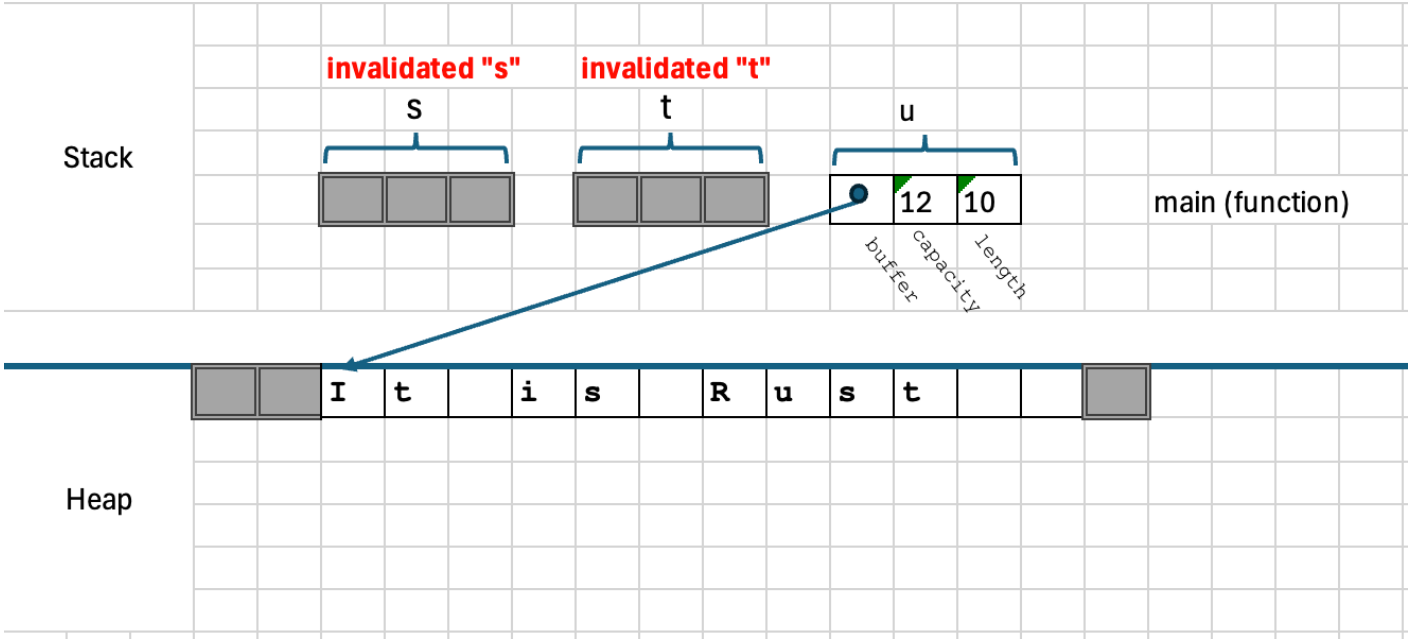
Moving ownership to ”t”, “s” is invalidated


```
1   fn main() {
2       let s = String::from("It is Rust");
3       let t = s; // moving to t
4       let u = t; // moving to u
5
6       println!("{:?}", u);
7   }
```



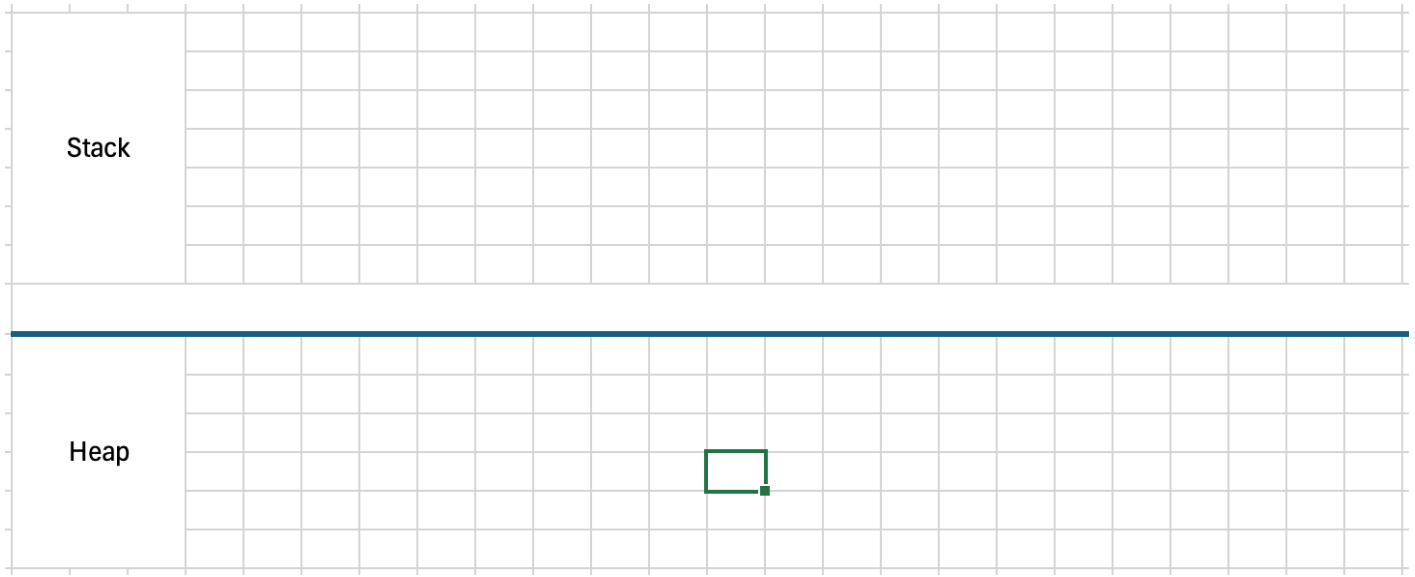
Moving ownership to ”u”, invalidate “t”


```
1   fn main() {
2       let s = String::from("It is Rust");
3       let t = s; // moving to t
4       let u = t; // moving to u
5
6       println!("{:?}", u);
7   }
```



Dropping the value


```
1   fn main() {
2       let s = String::from("It is Rust");
3       let t = s; // moving to t
4       let u = t; // moving to u
5
6       println!("{:?}", u);
7   }
```



Try with vectors

Drop, Move and Invalidate

`Drop` : This refers to **freeing or releasing** the value from memory when its owner goes out of scope. In Rust, this is handled automatically through the **Drop trait**, which can be customized.

`Move` : This is when ownership of a value is **transferred** from one variable to another. After a move, the original variable becomes **invalid**, and the new variable takes over the ownership.

`Invalidate` : This refers to the process where the original variable, after moving ownership to another, **loses the ability to access** the value. The original variable is no longer valid and cannot be used unless the ownership is restored or another operation is performed.

Checking the rules with own structure:


```rust
struct MyStruct {
    name: String,
}
impl MyStruct {
    // Constructor for the struct
    fn new(name: &str) -> MyStruct {
            MyStruct {
        name: name.to_string(),
        }
    }
}
// Implementing the Drop trait for MyStruct
impl Drop for MyStruct {
    fn drop(&mut self) {
            println!("Dropping MyStruct with name: {}", self.name);
    }
}

fn main() {
    // Rule1: Creating an instance of MyStruct
    let my_struct_1 = MyStruct::new("Struct 1");

    // Rule2: Moving ownership of my_struct_1 to my_struct_2
    let my_struct_2 = my_struct_1;

    // my_struct_1 is no longer valid here, but my_struct_2 owns the value.
    println!("Ownership moved to my_struct_2 with name: {}", my_struct_2.name);

    // Rule3: my_struct_2 will be dropped when it goes out of scope (end of main).
}
```

Reassigning a variable

String::from creates an instance of String

```
1  fn main() {
2      let mut a = String::from("Hello");
3      a = String::from("World"); // Above value is dropped.
4      println!("{a}");
5  }
```

Reassigning a variable


```rust
fn main() {
    let mut s = String::from("Hello");
    let t = s; // Moved to "t"
    s = String::from("World"); // 's' assigned with new value
    println!("{s}, {t}"); // Nothing is dropped.
}
```

Moving value to function


```
2   fn print(val: Vec<i32>) {
3       println!("{}", val[0]);
4   }
5   fn main() {
6       let s = vec![1,2,3];
7       let flag = false;
8
9       if flag == true {
10          print(s);
11      }
12      // println!("{:?}", s); // ERROR: value borrowed here after move
13  }
```

Moving to Vector as an element


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

fn main() {
    let mut composers: Vec<Person> = Vec::new(); // Allocating empty vector
    let p1 = Person{name:String::from("David"), birth:30};
    composers.push(p1);
    //println!("{:?}",p1); // ERROR: It is already moved.
    println!("{:?}", composers[0]);
}
```

As simple as ownership is given to function.

Scope matters: {}


```
2   fn print(val: Vec<i32>) {
3       println!("{}", val[0]);
4   }
5   fn main() {
6       let s = vec![1,2,3];
7       let flag = false;
8
9       if flag == true {
10          print(s);
11      }
12      // println!("{:?}", s); // ERROR: value borrowed here after move
13  }
```

Rust optimization is not done yet

Taking the value from vector


```rust
fn main() {
    let names = Vec::new();

    for i in 0..100 {
        names.push(i.to_string());
    }

    let first = names[0];
    //println!("{}", names[0]); //ERROR: move occurs because
    //value has type `String`, which does not implement the `Copy` trait
}

```

Home / Tech / Security

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

Written by **Catalin Cimpanu,** Contributor

Feb. 11, 2019 at 7:48 a.m. PT

Can ownership be given back???

```rust
fn main() {
    // Original ownership
    let original = String::from("Hello, Rust!");

    // Ownership is moved to new_owner
    let new_owner = original;

    // Ownership can be passed back manually
    let original = new_owner;

    println!("{}", original);  // Now original owns the value again
}
```

Answer: YES

Can we use Global variables in Rust

```
2    static mut counter:i32 = 10;
3
4    fn main() {
5        unsafe {
6            counter = counter+1;
7            println!("Counter = {}", counter);
8        }
9    }
```

Answer: YES, we shall discuss later about "unsafe" 

Can we edit the element in a vector ???

```
44  fn main() {
45      let mut s1: Student = Student::new(name: "Kamal", sid: 13);
46      let mut class = Vec::new();
47      s1.name.push_str(" kumar"); // Modified.
48      println!("{:?}", s1);
49      class.push(s1);
50
51      //Mutable Borrow
52      let mut element: &mut {unknown} = &mut class[0];
53      //Edit the content: No error
54      element.name.push_str(" mukiri");
55      println!("{:?}", element);
56      println!("{:?}", class[0]);
57  }
```

Answer: YES, it is part of ”mutable reference” (which is next topic)

Can we modify the data in Structure without “mut”?

```
#[derive(Debug)]
struct Student {
    name: String, // No mut is added
    id:i32, // No mut is added
}

impl Student {
    fn new(name :&str, sid: i32) -> Self {
        Student {
            name: name.to_string(),
            id: sid
        }
    }
}

fn main() {
    let mut s1 = Student::new("Kamal", 13);
    let mut class = Vec::new();
    s1.name.push_str(" kumar"); // Modified.
    println!("{:?}", s1);
    class.push(s1);
}
```

Answer: YES, no need to make member as mut.

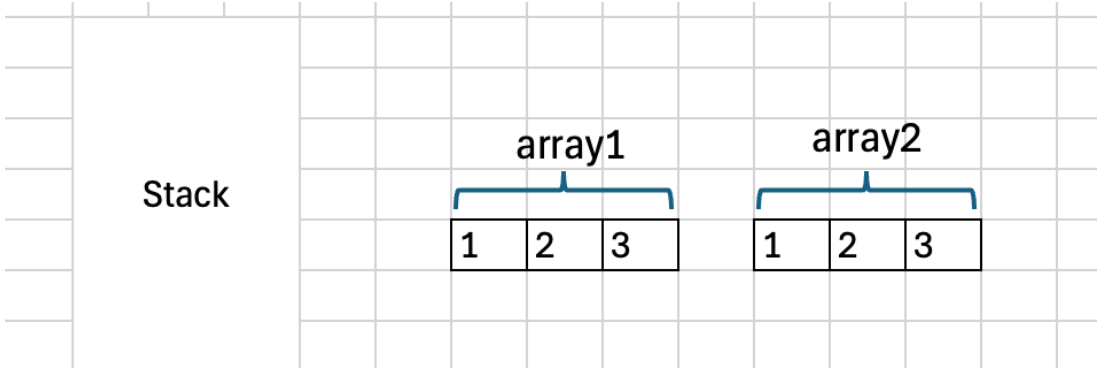
Check the below code and conclude....!


```rust
fn main() {
    let s = String::from("It is Rust");
    let t = s;
    //println!(" s = {:?}", s); //ERROR

    let array1: [i32; 4] = [1,2,3,4];
    let array2: [i32; 4] = array1; //No issue: No ownership is moved
    println!("array1 = {:?}", array1);
}
```

Conclusion:

Copy types (arrays)


```rust
fn main() {
    let array1: [i32; 4] = [1,2,3,4];
    let array2: [i32; 4] = array1; //  No ownership is moved

    println!("array1 = {:?}, array2 = {:?}", array1, array2);
}
```



What about Structure????

What about Structure????

Thumb rules of Copy Types

By default User defined types are non-Copy types.

If all the fields of structure are themselves Copy type, then the structure can be Copy as well by placing the attribute #[derive(Copy, Clone)]

Principles of Rust Programming

- Costs should be apparent to the Programmer
- Basic operations should remain simple.
- Potentially expensive operations should be explicit.

Simple Example of Reference in “C++”


```cpp
#include <iostream>

void feed_seed(int &num) {
    num +=10;
}

int main() {
    int a = 10;
1   int &b = a; // Caution: Purpose is not clear.
    b = 200; // Updating
    std::cout << a << "," << b << std::endl;

2   feed_seed(a); // Caution: No trust on third party lib.
    std::cout << a << "," << b << std::endl;
}
```

Issues:

1. While giving reference to a variable, there is no control on value.

2. Caller does not know, passing value or reference

Simple Example of Reference in “C++”


```cpp
int& fun() {
    int a = 10;
    return a;
}

int main() {
    int &ref = fun();
    // UNDEFINED
    std::cout << ref << std::endl;
}
```

Issues:

1. Collecting the reference of non-existing data
2. Dereferencing the same.

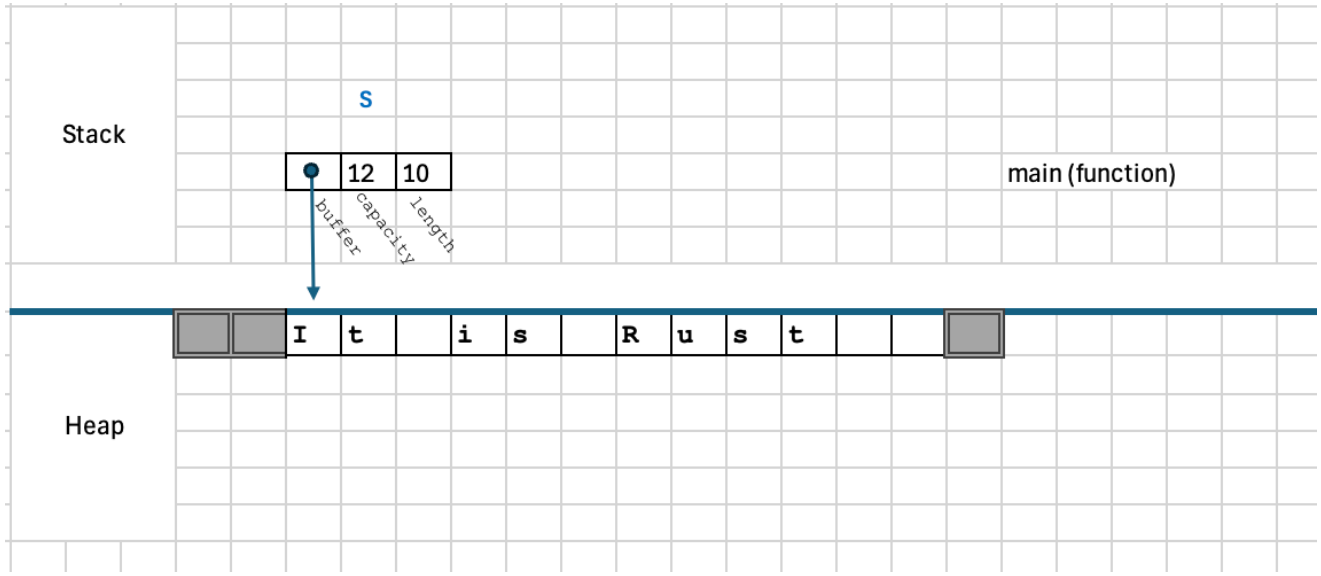
References in ”rust”

1. **References do not own the data they point to.**
   You can create a reference to a value, but the value's ownership remains with the original owner.
2. **Immutable references (&) allow read-only access.**
   You can create as many immutable references as you like, but they cannot be used to modify the data.
3. **Mutable references (&mut) allow modification of the data.**
   Only one mutable reference can exist at a time, preventing multiple pieces of code from modifying the data simultaneously.
4. **There can only be one mutable reference or multiple immutable references to a value at the same time.**
   This ensures there is no data race or concurrent modification while reading or writing.
5. **References must always be valid.**
   Rust ensures that references never point to invalid memory. This is enforced by Rust's **lifetime** system, which ensures that references do not outlive the data they point to.


When you have a security policy,
but no controls to enforce it.

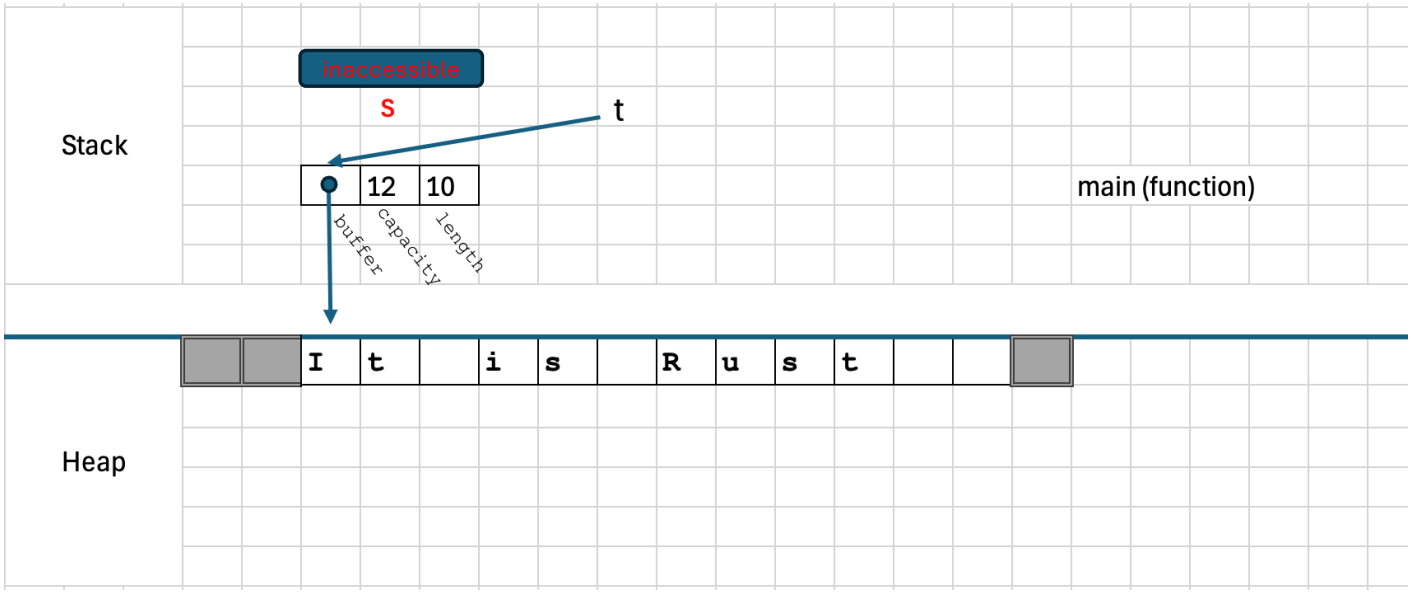
Memory layout


```
2   fn main() {
3       let mut s = String::from("It is Rust");
4       let mut t: &mut {unknown} = &mut s;
5       //println!("s = {}\n", s); // ERROR: s is inaccessible
6       println!("t = {}", t);
7   }
```



Memory layout


```
2   fn main() {
3       let mut s = String::from("It is Rust");
4       let mut t: &mut {unknown} = &mut s;
5       //println!("s = {}\n", s); // ERROR: s is inaccessible
6       println!("t = {}", t);
7   }
```

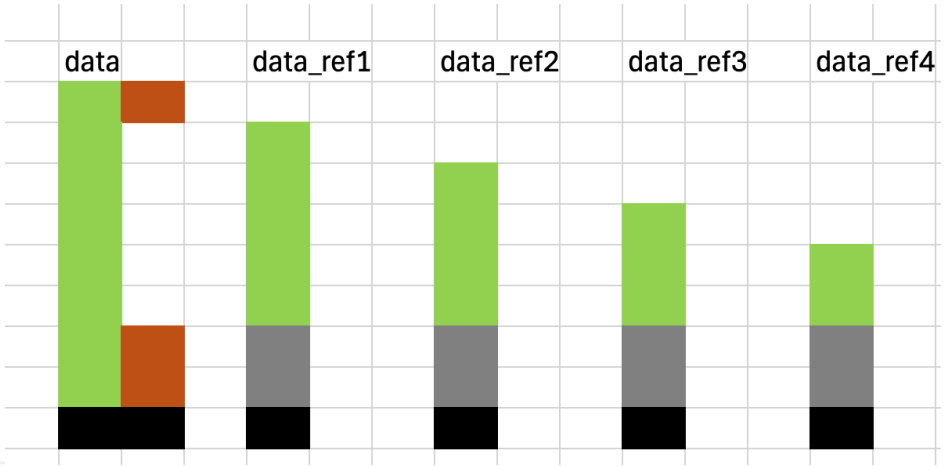


Check the rules

3. Mutable references (&mut) allow modification of the data.

4. There can only be one mutable reference or multiple immutable references to a value at the same time.

```
2   fn main() {
3       let mut data = String::from("hello");
4       let data_ref1: &{unknown} = &data;
5       let data_ref2: &{unknown} = &data;
6       let data_ref3: &{unknown} = &data;
7       let data_ref4: &{unknown} = &data;
8
9       data.push_str("End");
10      println!("{data}")
11  }
```



```
2  fn main() {
3      let mut data = String::from("hello");
4      let data_ref1: &{unknown} = &data;
5      let data_ref2: &{unknown} = &data;
6      data.push_str("... modified by owner");
7      let data_ref3: &{unknown} = &data;
8      let data_ref4: &{unknown} = &data;
9      println!("{data}, {data_ref4}");
10
11
12     let mut_ref: &mut {unknown} = &mut data;
13     mut_ref.push_str(".. added by mutable reference");
14     println!("{data}");
15 }
```

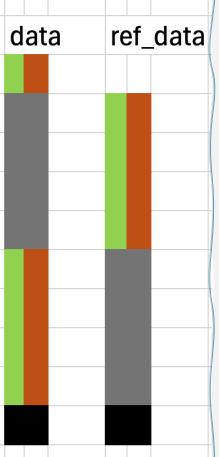
Will it compile ????

```rust
fn main() {
    let mut data = String::from("hello");
    let data_ref1: &{unknown} = &data;
    let data_ref2: &{unknown} = &data;
    data.push_str("... modified by owner");
    let data_ref3: &{unknown} = &data;
    let data_ref4: &{unknown} = &data;
    println!("{data_ref3}, {data_ref4}");

    let mut_ref: &mut {unknown} = &mut data;
    mut_ref.push_str(".. added by mutable reference");
    println!("{data}");
}
```

- Although created several **immutable references**, those references are **no longer used** after the point where you modify the original string.
- Rust's borrow checker notices this and makes the immutable references **out of scope** (i.e., it knows they are no longer needed), which allows you to borrow the data mutably after that.
- References to be dropped once they're no longer used, even if they are still technically in scope.

Simple Example of Mutable Reference in “Rust”


```
2   fn add_tail(data:&mut String) {
3   (3) data.push_str("...appended by function");
4   }
5
6   fn main() {
7       let mut data = String::from("hello");
8   (1) let ref_data:&mut String = &mut data;
9       ref_data.push_str("...appended by ref_data");
10      println!("ref_data = {}", ref_data);
11
12  (2) add_tail(&mut data); // data becomes active
13      data.push_str("world");
14      //ref_data.push_str("..."); // ERROR: ref_data is inaccessible.
15      println!("data = {}", data);
16  }
```



Output

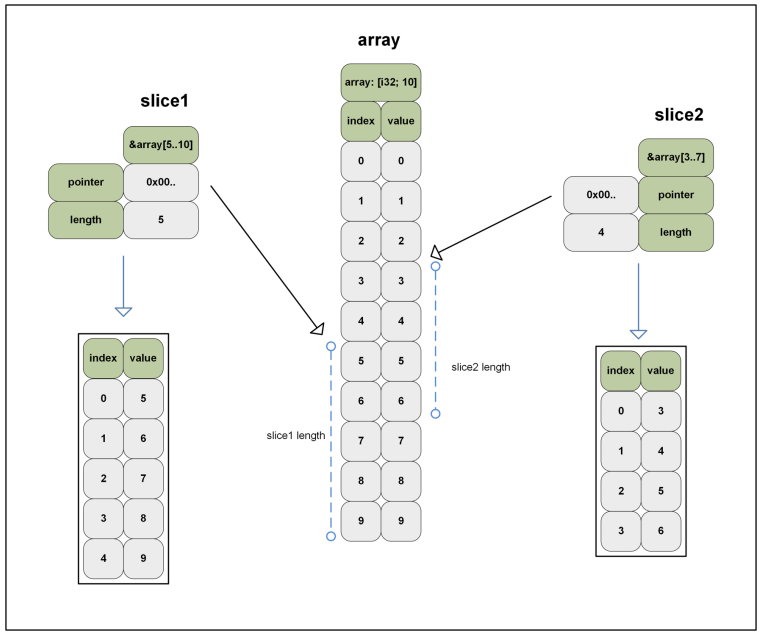
```
ref_data = hello...appended by ref_data
data = hello...appended by ref_data...appended by functionworld
```

1. Getting reference to a variable “a” along with permission to modify (Called Mutable reference).
2. Passing Mutable reference to a function. (but not ownership)
3. Modifying the value which should get reflected in caller.

Note: Try without “mut” keyword.

Reference to Slices:


```rust
fn main() {
    let array: [i32; 7] = [0, 1, 2, 3, 4, 5, 6];
    let slice: &{unknown} = &array[..]; // [ 0, 1, 2, 3, 4, 5, 6 ]
    let slice: &{unknown} = &array[0..3]; // [ 0, 1, 2 ]
    let slice: &{unknown} = &array[..3]; // [ 0, 1, 2 ]
    let slice: &{unknown} = &array[2..4]; // [ 2, 3 ]
    let slice: &{unknown} = &array[2..]; // [ 2, 3, 4, 5, 6 ]
}
```



Source: https://saidvandeklundert.net/learn/2021-08-14-rust-slice/

Reference to Slices:


```rust
fn main() {
    // Example with a string slice
    let my_string = String::from("Hello, Rust!");
    let slice: &str = &my_string[0..5]; // Create a sli
    print_slice(slice); // Pass the slice reference to t

    // Example with an array slice
    let my_array: [i32; 5] = [1, 2, 3, 4, 5];
    let array_slice: &[i32] = &my_array[1..4]; // Creat
    print_array_slice(array_slice); // Pass the array s
}

// Function that takes a string slice
fn print_slice(s: &str) {
    println!("String slice: {}", s);
}

// Function that takes an array slice
fn print_array_slice(s: &[i32]) {
    println!("Array slice: {:?}", s);
}
```

Output

```
String slice: Hello
Array slice: [2, 3, 4]
```

- Data Type of Slice is same as its source
- References to Slices are called as “fat pointers”

Misc


```rust
// #Multiple level immutable references
fn main() {
    let mut a: i32 = 10;
    let ra: &i32 = & a;
    let rra: &&i32 = &ra;
    println!("{},{},{}",a,*ra,**rra);
}
```

```rust
// # Multiple level mutable referece
fn main() {
    let mut a: i32 = 10;
    let mut ra: &mut i32 = &mut a;
    let rra: &mut &mut i32 = &mut ra;
    **rra = 200; // Modifyng double reference
    let result: i32 = **rra;
    println!("{rra}, {result}");
}
```

```rust
// # References and scope
fn main() {
    let mut a: i32 = 10;
    let fun: (a) = {
        let ra: &mut i32 = &mut a;
        *ra = 20;
        println!("{ra}");
    }
    println!("{a}");
}
```

```rust
// Reference to struct
// Nno need of *
0 implementations
struct data {
    id:i32,
    name :String
}
fn main() {
    let mut s1: data = data{name:String::from("Kamal"), id:10};
    let mut rs1: &mut data = &mut s1;
    println!("rst1.id = {}", rs1.id);
    let rrs1: &mut &mut data = &mut rs1;
    rrs1.id = 101;
    println!("rrs1.id = {}", rrs1.id);
}
```

Shared ownership with “Rc” and “RefCell”


```rust
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let data = Rc::new(RefCell::new(String::from("Hello")));

    let data1 = Rc::clone(&data);
    let data2 = Rc::clone(&data);

    data1.borrow_mut().push_str(", Rust!");

    println!("data1: {}", data1.borrow());
    println!("data2: {}", data2.borrow());
}
```

Output

```
data1: Hello, Rust!
data2: Hello, Rust!
```

Note: RefCell<T> in Rust is a type that provides interior mutability.

This means that it allows you to mutate data **even when the RefCell itself is immutable.**

One of the Principles of Rust Programming

- Costs should be apparent to the Programmer
- Basic operations should remain simple.
- Potentially expensive operations should be explicit.
  - Like calling “clone” to make deep copy.

Practice Time!!!

1. Write a function which returns histogram.