Rust Safety

Lucas Hasting

University of North Alabama

Florence, Al, United States

INTRODUCTION

This paper will explore the idea around the safety concerns that exist when programming memory operations in Rust. Rust is a relatively new systems programming language focused on speed and safety [2]. First, the history of Rust is explored and an overview of Rust is given. Then, a review of the literature chosen for the paper is summarized and some security terms are defined for the methodology. The methodology of researching the purpose of the paper is shown. Finally, the paper discusses some questions that could be researched in the future.

BACKGROUND

Historical Background

Rust was created by Graydon Hoare, he started on the project in 2006 as a passion project and Mozilla started sponsoring the language in 2010. The first compiler for Rust was created in 2012. Rust is known as being a safe, yet efficient, systems programming language, ranked 18 on the TIOBE index in 2020 [2].

Rust Overview

Rust is a systems programming language, that supports many features which will be discussed throughout this section. To start, Rust uses blocks ("{}") to represent the scope and a semi-colon (";") to represent the end of an instruction [1], Figure 1 demonstrates this. Also, the keyword fn is used to define a function, and all programs need a main function which is called when the program is executed. Figures 1-17 show demo programs to demonstrate the syntax of the Rust language, these programs and all the tables can be found in the Programs and Tables section. The semantics of what these programs demonstrate will be discussed throughout this section [1].

Some Tables of interest in this section are Tables 1, 2, and 5. Table 1 contains the data types in Rust, Table 5 contains the two main types of strings, and Table 5 contains the operators in Rust. Both Tables contain a description of what they represent, and the operators and data types will be discussed throughout the rest of the section. One thing to keep in mind is that the println!() macro is used for output, macro creation is beyond the scope of this paper. To recognize the difference between a macro and a function, recognize that all macros have an "!" before "()" [1].

Another note about println! is that variable cannot be the first parameter, it has to be a string literal. Variables can be substituted in place of "{}" where every "{}" represents the next variable parameter, this shows up in many example programs, the first being Figure 3 [1].

Variables in Rust are bound at compile time using the keyword let, it is strongly typed and uses interpretation if a type is not specified. Variables are read-only by default, and adding the mut keyword allows the value to be changed. See Figure 2 for the syntax of declaring a variable in Rust. It shows two ways to bind a type without the use of type interpretation. Next, if, else if, and else statements can be used for control flow. if and else if check a condition (boolean statement) which can be either true or false, if true it will execute the block else if a different condition is true, it will execute the corresponding block, if all statements are false the else block will execute. Next, a match uses the syntax of "case => functionality" and will execute the functionality based on its case, "_" represents the default case, see Figure 2 for an example of match and conditionals. Also, Loops in Rust can be based on a condition, counter, or nothing at all [1].

Moreover, there are a few types of loops in Rust, these include a for each loop which loops through a structure iterating through each item in the structure, a for loop which is counter-controlled, a while loop which loops if a condition (boolean statement) is true, and a loop that loops forever. A for each and for loop can be found in Figure 6, and a while loop and loop can be found in Figure 3. The break and continue keywords help manage the flow of a loop, continue tells the loop to go to the next iteration, ignoring the rest of the loop, and break causes the loop to end [1].

Continuing, the as operator is used to cast a similar type as another, as long as it is allowed. An example where it would not be allowed is converting a negative signed integer to an unsigned integer, an example can be found in Figure 4 [1].

Rust has a few different ways to perform memory operations, it can involve a reference to a variable using a smart pointer, a reference using a raw pointer, or it can be allocated directly to the heap (using either pointer method). A raw pointer is not recommended and is unsafe, the only way to use a raw pointer is by enclosing it in an unsafe block as shown as an example in Figure 5 [1].

Arrays are a collection of items of a specific data type, which the number of items cannot change, while vectors are a collection of items of a specific data type where the number of items can change. They can be displayed using a for each loop, or a counter-controlled for loop, see Figure 6 for an example [1].

In Rust, a slice of a string is a portion of that string, it can include the whole string as that slice. The &str data type is a reference to a slice, while the String data type is a reference to a string. Enclosing a string using quotes creates a slice, it can be converted into a string using String::from(""). Slices are also a part of arrays and vectors and Figure 7 has a demonstration of this. The index of either structure can create a slice, for example, 1..3 will create a slice that includes indices 1 and 2 (stopping at 3), and 4.. will create a slice from index 4 to the end of the structure [1].

A tuple works the same way in mathematics, it works as a collection of items where a n-tuple has n items. Each index can be retrieved using the "." operator, an example is shown in Figure 8 [1].

In Rust generics are done similarly to other languages by using <VAR> where VAR represents a type placeholder, some features of a type may need to implement features from a trait, this is shown in Figure 9 where the function get_value takes in any type, displays it and returns it. The generic type needed to be able to implement the std::fmt::Display to use the println! macro [1], keep in mind the example only works if the type used has an implementation of std::fmt::Display.

In Rust, structures like arrays, vectors, and strings are moved to another variable and then dropped from the previous variable it is assigned. Figure 10 shows the problem. In the example, String is chosen and the clone() method is used to create an exact copy instead of assigning the variable [1].

A closure works similarly to a procedure and can be called like a function. variables from a higher level scope are borrowed by default and can be moved to that closure by using the keyword move [5], Figure 11 shows an example.

In Rust, structs are how objects are created, unlike languages like C and C++, a struct has permissions, they are private by default and the keyword pub makes them public. Traits are Rust's version of abstract classes/methods, an example is shown in Figure 13. All methods of a class are implemented in an impl block as shown in Figure 12 [1].

When references are used, the rustc compiler may believe the lifetime of the variables is ambiguous and a lifetime parameter is required, and it can happen with functions or classes. The lifetime parameter is done similarly to generics, an example is shown in Figures 12 and 13. The lifetime parameters in those figures are 'a and '_ [5], which were recommended by the compiler.

Table 3 includes utility traits that are a part of the Rust language.

Enums are state-like variables and contain constants that represent the state. A match statement that takes in an enum and performs operations based on the enum's state is referred to as a pattern, an example enum and pattern can be seen in Figure 14 [1].

Operators in Rust can be overloaded by implementing a method in the trait corresponding to the operator being overloaded, Table 6 shows each trait with the corresponding operator. An example of an operator being overloaded is shown in Figure 15 [1].

Rust uses readers and writers for input and output, this goes outside the scope of the paper, but getting input from STDIN can be useful for writing programs. The std::io library has a read_line method which takes in a reference to a string and will take input from the keyboard and store the string passed in. The trim method for strings is used to trim off any extra white space. Figure 16 shows an example of taking in input from the user and displaying that output [5].

Concurrency in Rust is done using threads, a new thread can be declared by using the thread::spawn method from the std::thread library. thread::spawn takes in a closure and will execute that closure separately from the rest of main, the thread::sleep method is used to make the current thread sleep. Although not needed, the Durration::from_millis method from the std::time::Duration library can make a machine-readable format for milliseconds which is used along with the concurrency example in Figure 17 [5].

To conclude this section, Rust has collections included in the standard library, which are implementations of specific data structures, a list of those can be found in Table 4 [1].

Literature Review

[1]: This is a textbook on the Rust language. The textbook is a comprehensive guide on the Rust language including the systems features of the language.

[2]: The paper provides an overview of the Rust language including some historical facts and specific features of the Rust language.

[3]: The paper provides an overview of the security of Rust, it discusses several software vulnerabilities and describes how Rust prevents those vulnerabilities from making a system vulnerable, however, a bad programmer can still make a system vulnerable as shown in the methodology section.

[4]: The paper describes Rust programs that make use of the unsafe feature, and reasons why a program may use unsafe by looking at projects that make use of the unsafe feature.

[5]: This is the official Rust documentation, the documentation includes resources on learning the language such as small projects, a few textbooks, and other materials. It also provides a guide to contributing to the Rust compiler called rustc.

[6]: The paper discusses what the Rust language is, the bugs found in projects using Rust, bugs that can occur when using thread, and how to build a Rust bug detector to avoid writing unsecure code.

METHODOLOGY

To understand the safety concerns in Rust, software vulnerabilities will be tested in Rust. Figures 18-27 demonstrate common software vulnerabilities attempted in the Rust language safely and unsafely, a description of these vulnerabilities is provided in the following subsection.

Security Definitions

Below are some software security vulnerabilities and their definitions, these terms are used in the methodology section to answer the research question.

Buffer Overflow: This vulnerability occurs whenever a memory slot is accessed outside of the allocated boundary, and that memory slot is being written to [3].

Buffer Over-read: This vulnerability occurs whenever a memory slot is accessed outside of the allocated boundary, and that memory slot is being read [3].

Use-after-free/Dangling pointer: This vulnerability occurs whenever a pointer variable that has been deallocated is accessed, the pointer is called a dangling pointer [3].

Double Free: This vulnerability occurs whenever a pointer variable is deallocated twice [3].

Uninitialized Memory Access: This vulnerability occurs whenever memory that has not been initialized is accessed [3].

Race Condition: This vulnerability occurs whenever a value is being written and read in memory at the same time, causing the behavior to be undesired [3].

Unsafe Rust

Unsafe Rust was mentioned briefly in the background section, involving Figure 5. Unsafe Rust can only be used if what the programmer is doing is considered unsafe by the compiler [1]. Unsafe Rust is used by many people to use libraries written for other programs, this can be done using foreign functions which act like interfaces to other languages [4] [1]. It is also used by programmers who are not comfortable with using some of the features in Rust like smart pointers, even though it is safer than raw pointers which requires Rust to be unsafe [4]. Also, if the Rust code is unsafe in a function and the function is used, the Rust program is referred to as interior unsafe [6].

The Research

First, A buffer overflow and over-read is shown in Figures 18 and 19. In Figure 18, an array with a size of 6 is used as a buffer. When trying to access an item out of bounds, a bound check is done on the array and Rust returns a compile time error when the program is compiled. In Figure 19, Rust tries to access/write an item from an array based on where it is located in memory using the std::ptr library. More specifically, the wrapping_add method offsets the memory address by the number passed in as an argument and the write method will write a value to a memory address. The over-read and overflow are both successful with undesired behavior occurring [3].

Then, Use-after-free/Dangling Pointer is attempted In Figures 20 and 21. In Figure 20, the value of the variable val is dropped when the function ends, so when the reference to the variable is returned, the value at the reference no longer exists and Rust provides a compile-time error. In Figure 21 we can use a raw pointer instead of a function using unsafe, using the drop_in_place function, the value in the referenced variable is dropped, and accessing the dereferenced value provides undesired behavior [3].

Next, Double Free is attempted in Figures 22 and 23. Rust does not allow a reference to be dropped twice, causing Figure 22 to provide a compile-time error. While in the unsafe block, Rust does not have a check for dropping a referenced value [3], executing the code in Figure 23 causes Rust to crash once the value is dropped again, and finished is never displayed on the screen.

Also, Rust does not allow uninitialized memory access, causing both Uninitialized Memory Access attempts to provide a compile-time error [3]. This is shown in Figures 24 and 25 to provide a compile-time error.

Finally, a race condition is attempted in Figures 26 and 27. The process this follows goes beyond the scope of the paper and the example (figures 26 and 27) was taken from the official Rust documentation. So, at a high level, Figure 26 attempts to retrieve a value and display that value to the screen in two separate threads. Rust does not allow 2 mutable references to the same memory address at once and uses a borrowed system to allow the value stored to be accessed once a previous operation is completed [1] [5]. In Figure 27, the same operation is done, but the access of the value being written is accessed by skipping the check that determines if the value is currently being used by another thread and therefore by definition (in the Security Definitions section), causes a race condition to occur [5].

RESULTS

Overall, Rust implements strict safety checks before allowing a program to be executed in order to maintain safe programs, this prevents any unsafe code utilizing memory operations in Rust (outside of the unsafe block) [4]. However, this safety check can be turned off by using the unsafe keyword and will make the program as safe as the programmer allows, and could cause the program to be exposed to one of the many vulnerabilities discussed and should be seldom used.

Therefore, the results of this study show that it is safe to perform memory operations in Rust by default, and it protects against many memory-related vulnerabilities. The memory operations in Rust can become unsafe if the programmer forces it to by using the unsafe keyword. Therefore, the concerns involved with performing memory operations are determined by the programmer and the choices made. If the programmer chooses to use an unsafe block, the programmer should be concerned with being exposed to one of the vulnerabilities mentioned in the methodology, otherwise, Rust will not allow unsafe code.

FUTURE WORK

Rust has many more topics that could be researched. These include the speed of processing Rust code compared to other languages like C, the suitability of Rust to software projects, and the compatibility between Rust and C++.

REFERENCES

[1] J. Blandy, J. Orendorff, L. Tindall, *Programming Rust Fast, Safe Systems Development*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[2] W. Bugden, A. Alahmar, "Rust: The Programming Language for Safety and Performance," 2022, doi: 10.48550/arXiv.2206.05503

[3] H. Xu, Z. Chen, M. Sun, Y. Zhou, M. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," 2021, doi: 10.1145/3466642

[4] V. Astrauskas, C. Matheja, F. Poli, P. Müller, A. Summers, "How do programmers use unsafe Rust?," 2020, doi: 10.1145/3428204

[5] Rust project. *Rust Documentation.* Accessed: April, 17, 2024. Available: https://doc.Rust-lang.org/beta/

[6] B. Qin, Y. Chen, Z. Yu, L. song, Y. Zhang, "Understanding memory and thread safety practices and issues in real-world Rust programs," 2020, doi: 10.1145/3385412.3386036

PROGRAMS AND TABLES

Figure 1: Hello Word

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

Figure 2: Variables/Conditionals/Match

```
fn main() {
    //initialize with a type
    let val1 : i8 = 6;
    let val3 = 7_i8;

    //type inference
    let val2 = 6;

    //conditional
    if (val1 == val3){
        println!("Block 1");
    } else if (val1 == val2){
        println!("Block 2");
    } else {
        println!("Block 3");
    }

    switch(val1);
    switch(val2);
    switch(val3);
}

fn switch(val: i8){
    //match statement
    match val {
        6 => println!("Case 1"),
        7 => println!("Case 2"),
        _ => println!("Default")
    }

    return;
}
```

Figure 3: While Loops and Loops

```
fn main() {

    //both loops display 1, 3, 5

    let mut count = 0_i8;

    while(count != 6){
        count += 1;
        if(count % 2 == 0){
            continue;
        }

        println!("{}", count);
    }

    count = 0_i8;

    loop{
        if(count == 6){
            break;
        }

        count += 1;

        if(count % 2 == 0){
            continue;
        }

        println!("{}", count);
    }

}
```

Figure 4: as operator

```
fn main(){
    let val1 = 5_u8;
    let val2 = val1 as u16;

    println!("Value: {}", val2);
}
```

Figure 5: Memory related operations

```
fn main(){
    //declare mutable variable
    let mut val = 5_i8;

    //pass in reference to the function
    read(&val);

    //pass in mutable reference to the function
    change(&mut val);
    read(&val);

    //allocate value in heap as a smart pointer
    let _v = Box::new(val);

    //raw pointer, unsafe
    unsafe {
        let _bad_val: *mut i8 = &mut val as *mut i8;
        println!("{}", *_bad_val);
    }
}

//mutable reference to a variabel
fn change(val: &mut i8){
    *val = *val + 1;
}

//Read-only reference to a variable
fn read(val: &i8){
    println!("val is {}", *val);
}
```

Figure 6: Arrays and Vectors

```
fn main(){
    //create and edit an array
    let mut arr1: [u16; 6] = [3, 2, 7, 4, 11, 13];
    arr1[0] = 4;

    //set all elements a specific value, t has 10000 0s
    let _arr2 = [0; 10000];

    //initialize the vector with values
    let vec1 = vec![3, 2, 7, 4, 11, 13];

    //create empty vector and push values onto it
    let mut vec2 = Vec::new();
    vec2.push(3);
    vec2.push(4);
    vec2.push(5);

    println!("ARRAY1 1:");
    //print array
    for i in arr1 {
        println!("{}", i);
    }

    println!("Vector 1:");
    //print vector
    for i in vec1 {
        println!("{}", i);
    }

    println!("Vector 2:");
    //another way to print
    for i in 0..vec2.len() {
       println!("{}", vec2[i]);
    }
}
```

Figure 7: Slicing

```
fn main(){
    //initilize the vector with values
    let vec1 = vec![3, 2, 7, 4, 11, 13];

    //create slices, works with arrays and vector
    let slice1: &[u16] = &vec1[1..3];
    let slice2: &[u16] = &vec1[4..];

    //print the slices
    println!("Slice 1:");

    for i in slice1 {
        println!("{}", i);
    }

    println!("Slice 2:");

    for i in slice2 {
        println!("{}", i);
    }
}
```

Figure 8: Tuples

```
fn main() {
    let tup = (4, 5, 6);
    println!("{}",tup.0);
    println!("{}",tup.1);
    println!("{}",tup.2);
}
```

Figure 9: Generics

```
fn main(){
    let mut val1 = 5_i8;
    let mut val2: String = "Hello".to_string();

    //call generic functions
    val1 = get_value::<i8>(val1);
    val2 = get_value::<String>(val2);

    println!("{} {}",val1, val2)
}

//define generic function
fn get_value<T: std::fmt::Display>(value: T) -> T{
    println!("{}", value);
    return value;
}
```

Figure 10: Move Process

```
fn main(){
    //printing s causes an error, the value has been
     moved to k
    let mut s = "John Doe".to_string();
    let k = s;
    println!("k: {}, s: {}", k, s)

    //problem fixed
    let s = "John Doe".to_string();
    let k = s.clone();
    println!("k: {}, s: {}", k, s)
}
```

Figure 11: Closures [5]

```
use  std::mem;

fn main() {
    let mut count = 0;

    //closure borrows count
    let mut add_one = || {
        count += 1;
        println!("`count`: {}", count);
    };

    add_one();

    //closure moves count
    let mut count2 = Box::new(0);

    let mut add_one_move = move || {
        *count2 += 1;
        println!("`count2`: {}", *count2);
        mem::drop(count2);
    };

    add_one_move();
}
```

Figure 12: Structs

```
fn main(){
    //struct object definition
    pub struct Animal<'a> {
        pub name: &'a str,
        action: &'a str
    }

    //struct object implementation
    impl Animal<'_> {
        pub fn talk(&mut self) {
            println!("{}", self.action);
        }
    }

    //create the struct object
    let mut dog = Animal { name: "Joe", action: "Bark"};
    println!("{}",dog.name);
    dog.talk();
}
```

Figure 13: Traits

```
fn main(){
    //trait definition
    trait AnimalMethods {
        fn talk(&mut self);
    }

    //struct object definition
    pub struct Animal<'a> {
        pub name: &'a str,
        action: &'a str
    }

    //implement the trait for the Animal class
    impl AnimalMethods for Animal<'_> {
        fn talk(&mut self) {
            println!("{}", self.action);
        }
    }

    //create the struct object
    let mut dog = Animal { name: "Joe", action: "Bark"};
    println!("{}",dog.name);

    //call the method implemented by the trait
    dog.talk();
}
```

Figure 14: Enums/Patterns

```
fn main() {
    //create enum
    let mut light : Light = Light::RED;

    //use the pattern
    println!("{}",pattern(light));

    //change the state
    light = Light::YELLOW;

    println!("{}",pattern(light));
}

//define enum
enum Light {
    RED,
    YELLOW,
    GREEN
}

fn pattern(light: Light) -> String{
    //define pattern
    match light {
        Light::RED => String::from("RED"),
        Light::YELLOW => String::from("YELLOW"),
        Light::GREEN => String::from("GREEN")
    }
}
```

Figure 15: Operator Overloading

```
//add trait
use std::ops::Add;

fn main() {
    //test Fraction class with the overloaded + operator
    let fraction1 = Fraction { numerator : 5 ,
     denomonator : 8};
    let fraction2 = Fraction { numerator : 5 ,
     denomonator : 8};
    let fraction3 = fraction1 + fraction2;
    println!("{}/{}", fraction3.numerator, fraction3.
     denomonator)
}

//create basic Fraction class
struct Fraction {
    pub numerator : i16,
    pub denomonator: i16
}

//implement the add trait for the Fraction class
impl Add<Fraction> for Fraction{
    type Output = Fraction;

    fn add(mut self, rhs: Fraction) -> Fraction {
        self.numerator = self.numerator * rhs.denomonator
     ;
        self.numerator = self.numerator + (rhs.numerator
     * self.denomonator);
        self.denomonator = self.denomonator * rhs.
     denomonator;

        return self;
    }
}
```

Figure 16: Input/Output from command line [5]

```
use std::io;

fn main() {
    //set empty string
    let mut input = String::new();

    //read input
    io::stdin().read_line(&mut input);

    //trim input
    println!("You typed: {}", input.trim());
}
```

Figure 17: Concurrency [5]

```
use std::thread;
use std::time::Duration;

fn main() {
    //create a new thread
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned
     thread!", i);

            //sleep
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i)
     ;
        thread::sleep(Duration::from_millis(1));
    }
}
```

Figure 18: Safe Buffer Overflow/Over-read

```
fn main(){
    let mut buff: [u16; 6] = [3, 2, 7, 4, 11, 13];

    //buffer over-read
    println!("{}",buff[6]);

    //buffer over-flow
    buff[6] = 5;
}
```

Figure 19: Unsafe Buffer Overflow/Over-read

```
use std::ptr;

fn main(){
    let mut buff = [3, 2, 7, 4, 11, 13];

    unsafe {
        let ptr: *mut i32 = buff.as_mut_ptr();

        //buffer over-read
        println!("{}",*(ptr.wrapping_add(6)));

        //buffer overflow
        ptr::write(ptr.wrapping_add(6), 5);
    }
}
```

Figure 20: Safe Use-after-free/Dangling Pointer

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

fn drop() -> &'static String{
    let val = String::from("Rust");
    return &val;
}
```

Figure 21: Unsafe Use-after-free/Dangling Pointer

```
use std::ptr::drop_in_place;

fn main(){
    let val: *mut String = Box::into_raw(Box::new(String
     ::from("Rust")));

    unsafe {
        println!("{}", *val);
        drop_in_place(val);
        println!("{}", *val);
    }
}
```

Figure 22: Safe Double Free

```
fn main(){
    let val: String = String::from("Rust");
    std::mem::drop(val);
    std::mem::drop(val);
}
```

Figure 23: Unsafe Double Free

```
use std::ptr::drop_in_place;

fn main(){
    unsafe {
        let val: *mut String = Box::into_raw(Box::new(
     String::from("Rust")));
        println!("{}", *val);
        drop_in_place(val);
        drop_in_place(val);
        println!("finished");
    }
}
```

Figure 24: Safe Uninitialized Memory Access

```
fn main(){
    let val: i8;
    println!("{}", val);
}
```

Figure 25: Unsafe Uninitialized Memory Access

```
fn main(){
    unsafe {
        let val: *mut i32;
        println!("{}", *val);
    }
}
```

Figure 26: Safe Race Condition [5]

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

thread::spawn(move || {
    other_idx.fetch_add(10, Ordering::SeqCst);
});

println!("{}", data[idx.load(Ordering::SeqCst)]);
```

Figure 27: Unsafe Race Condition [5]

```
use std::thread;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;

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

let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

thread::spawn(move || {
    other_idx.fetch_add(10, Ordering::SeqCst);
});

if idx.load(Ordering::SeqCst) < data.len() {
    unsafe {
        println!("{}", data.get_unchecked(idx.load(
     Ordering::SeqCst)));
    }
}
```

Table 1: Data Types in Rust [1]

Type	Description
i8	8 bit signed integer
i16	16 bit signed integer
i32	32 bit signed integer
i64	64 bit signed integer
i128	128 bit signed integer
u8	8 bit unsigned integer
u16	16 bit unsigned integer
u32	32 bit unsigned integer
u64	64 bit unsigned integer
u128	128 bit unsigned integer
isize	signed integer (32 or 64 bit depending on the system)
usize	unsigned integer (32 or 64 bit depending on the system)
f32	32 bit floating point number
f64	64 bit floating point number
bool	1 bit, true or false
char	32-bit unicode character
String	dynamically sized string of characters
&str	reference to a non owning string pointer

Table 2: String Types in Rust [1]

Type	Description	Example
Byte String	string of bytes, no Unicode characters	b"get"
String Literal	text wrapped in quotes	"string"

Table 3: Utility Traits in Rust [1]

Type	Description
Drop	Destructors
Sized	Trait for types with a fixed size
Clone	Trait for types that support cloning their value
Copy	Clone trait with a byte-by-byte copy
Deref and DerefMut	Trait for smart pointers
Default	Trait for values with a sensible default value
AsRef and AsMut	Conversion Traits (reference to mutable)
Borrow and BorrowMut	Consistent version of AsRef/AsMut
From and Into	Type conversion trait
TryFrom and TryInto	Type conversion trait that could fail
ToOwned	Conversion Traits (reference to owned)

Table 4: Collections in Rust [1]

Vec<T>	Vector
VecDeque<>	Deque
LinkedList<T>	Linked List
BinaryHeap<T>	Binary Heap
HashMap<T>	Hash Map
BTreeMap<T>	Binary Tree Map
HashSet<T>	Hash Set
BTreeSet<T>	Binary Tree Set

Table 5: Operators in Rust [1]

Type	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus division
=	assignment
«	bit shift left
»	bit shift right
&	bitwise and
|	bitwise or
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal
&&	and
||	or

Table 6: Operator Types in Rust [1]

Type	Operator
std::ops::Neg	-
std::ops::Not	!
std::ops::Add	+
std::ops::Sub	-
std::ops::Mul	*
std::ops::Div	/
std::ops::Rem	%
std::ops::BitAnd	&
std::ops::BitOr	|
std::ops::BitXor	ˆ
std::ops::Shl	«
std::ops::Shr	»
std::ops::AddAssign	+=
std::ops::SubAssign	-=
std::ops::MulAssign	*=
std::ops::DivAssign	/=
std::ops::RemAssign	%=
std::ops::BitAndAssign	&=
std::ops::BitOrAssign	|=
std::ops::BitXorAssign	ˆ=
std::ops::ShlAssign	«=
std::ops::ShrAssign	»=
std::cmp::PartialEq	==, !=
std::cmp::PartialOrd	<, <=, >, >=
std::ops::Index	x[y], &x[y]
std::ops::IndexMut	x[y] = z, &mut x[y]