Rust

Neil Kingdom

December 6, 2023

Contents

1 Preface

Rust is, (in my humble opinion), not a beginner's language. Despite the official documentation's best efforts at welcoming newcomers, the simple truth is that Rust expands upon languages of the past in numerous ways. It is syntactically rich and brings forth new concepts which are foreign to the average modern- day programmer. It is my recommendation that you learn at least three other languages prior to learning Rust. My recommendations are C, due to its low level concepts which lend well with regards to understanding how Rust works under the hood, C++, as it is a very syntax-heavy language, and a high level language such as Python, TypeScript, or Kotlin, as these will introduce you to modern programming idioms, most of which Rust adopts. This document will not waste time refreshing you on basics such as pointers, multithreading, enums, functions, classes, inheritance, or anything of the sort.

2 Introduction

I'm sure Rust requires little introduction, but I'll recap in case you're a bit behind. Rust is a procedural (and arguably OOP) language which offers incredible run-time speeds and memory safety. It is primarily targeted towards the systems programming crowd, but it has also been adopted widely in the web ecosystem thanks to its incredible compatibility with WASM. In order to implement memory-safe code and worry-free concurrency, Rust uses a system known as the borrow checker. We will discuss the borrow checker ad nauseam, as it is one of Rust's core features. Assuming that you've heeded my warnings in the preface, and have some fundamental knowledge of programming, Rust's learning curve will seem steep at first, but you will quickly begin to understand the reasoning behind the decisions made, and hopefully you will reap the benefits of the initial time sink it requires learning the language by spending less time debugging difficult memory problems.

3 Types

Rust is a type-safe language. One of my favorite aspects of Rust is that it forces the user to be precise about the size of types. Integer types are implicitly signed by default, but have unsigned variants which are prefixed by the letter 'u'. The standard integer type is i32, but we can also declare a variable to be of type i8, i16, i64, i128, u8, u16 , u32, u64, or u128. If you want a platform-dependent integer type, you may use isize or usize, which will occupy the length of a word. Integer literals can be represented in hexadecimal, octal, or binary using the standard prefixes (0x, 0o, 0b). Rust can also represent characters as bytes using the b prefix:

```
let a: u8 = b'A';
```

This will convert it to a u8. Decimals may include underscores to denote commas for readability e.g. 98_222. In order to use floats and doubles, you may use f32 and f64, respectively. Rust actually does not support the 'f' postfix for floats. Floating point literals require that a digit precede the decimal point, but they do not require that anything come after the decimal point. Characters in Rust (declared with the char keyword) are 4 bytes due to the fact that Rust ensures all strings are UTF-8 compliant. I will have a section dedicated to strings in Rust, as they are quite a complex and robust feature of the language.

4 Arrays and Vec<T>

Rust has built-in support for both statically sized arrays (allocated on the stack), as well as dynamically sized arrays via the Vec struct (allocated on the heap). A statically sized array is declared using square braces ([]).

These braces require two arguments delimited by a semicolon: the type and the length. Rust is very strict about knowing the length of stack-allocated variables at compile time. The only time that Rust will allow you to create dynamically sized objects is if they are heap-allocated. Therefore, to declare an array of bytes, we can do the following:

```
let bytes: [u8; 6] = [b'A', b'B', b'C', b'D', b'E', b'F'];
```

If the array in this example did not contain exactly 6 elements at compile time, the compiler would throw an error, even if the number of elements was less than the length specified. It is at this point that I should bring up the fact that Rust supports type inference. Rather than explicitly providing the array's type and length, we can instead create a dynamically sized array who's type is inferred like so:

```
let bytes = [b'A', b'B', b'C', b'D', b'E', b'F'];
```

In this example, Rust infers the type to be a [u8; 6] (and yes, the capacity is included as part of the type information). We can provide, as an rvalue, an array initializer to quickly initialize or re-initialize an array with a specific value. The syntax for this is once again, to use square braces with two parameters: the value that the buffer will be initialized with, and the length of the initializer list. For example, we could say

```
let a: [i32; 5] = [9; 5];
```

which would populate the variable a with five 9s. This is written more simplistically using type inferrence:

```
let vals = ["Hello"; 10];
```

Vectors, as mentioned, are heap-allocated, and thus Rust allows them to be dynamically sized. A variable can explicitly be marked as a vector using Vec<T> e.g.

```
let vector: Vec<bool>;
```

In order to initialize a Vec in Rust, we use the vec! macro. Macros in Rust function similar to how they work in other languages, but are in many ways, more extensible. We can initialize a Vec like so:

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

Rust will use type inference to infer the concrete type of Vec in all of the examples where I've omitted the explicit type. If you want to use type inference for a Vec, but you don't want to initialize it with any data, you can either assign it a value of vec![] or Vec::new().

5 Strings

Strings in Rust can be quite daunting for most programmers due to some of the rules by which Rust abides. Unlike C or C++, strings in Rust are not null-terminated. To expound upon this, there are two primary ways of determining where a string terminates. The traditional way is to use a null-terminator, which is more memory efficient, but less safe. The other method, which Rust uses, is to think of strings as a structure which first contains the string's length, followed by the actual data. This occupies more memory since now we must store the length within each string. In Rust, we additionally store the capacity of the string i.e. the maximum amount of bytes that the string can contain if it needs to grow. Strings in Rust are also guaranteed to be valid UTF-8. If you're not aware, UTF-8 is a text encoding standard which uses a variable-width encoding. Variable-width refers to the fact that a character may be represented as one or more bytes (in the case of UTF-8, up to 4). This allows it to be backwards compatible with ASCII, while also providing support for Unicode characters. Finally, Strings in Rust, alongside all datatypes in Rust, are immutable by default. As we'll come to see, mutability is something that Rust takes very seriously. Creating a string requires slightly more effort than one may expect. If we attempt to make one e.g.

```
let s: String = "hello world";
```

we get an error: "Expected String, found &str". In Rust, the str type a.k.a. the string slice, represents a view into a string. It represents the underlying string literal as a contiguous block of bytes. Unlike String, which is an owned type i.e. it owns the underlying data, str does not own the string literal that it points to, it simply has a read-only pointer to the data (thus it is a borrowed type). The str type also does not contain the capacity of the string, because unlike the String type, a str is not growable. Additionally, String always allocates its string literals on the heap, but str may reference string literals in heap memory, stack memory, or the .data section of the binary. The str type is usually seen in its borrowed form (&str) due to the fact that Rust requires that its size is known at compile time, meaning that we need to use either a reference or pointer (since references and pointers have known lengths). Although we'll talk about lifetimes in more detail later, it is important to note that string literals in Rust have an implicit static lifetime, meaning that they live for the duration of the program. We can type this explicitly like so:

```
let s: &'static str = "hello world!";
```

Back to creating a String though. The error arose due to the fact that we were trying to assign a borrowed type (&str) to an owned type (String). There are a few ways to solve this dilemma, but I'll cover just three. The first is to use String::from() which will allocate some heap memory and perform a copy of the string slice if necessary. The second is to_owned(), which will clone the string slice and return an owned String. Finally is into(), which is one way of performing type casting in Rust. We'll look into type casting in more detail later, but essentially, the str type in Rust implements a trait which tells the compiler how to convert between str and String. The following examples all have the same end result:

```
let s = String::from("hello world");
let s = "hello world".to_owned();
let s: String = "hello world".into();
```

6 Tuples

The final type we'll look at are tuples. In certain languages a tuple refers to a set of exactly two items. In Rust, a tuple is considered to be any set that contains two or more items. When we define a tuple, we're really just defining multiple variables at the same time, each of which can be its own type, unlike arrays or Vecs. These variables do not necessarily need to be coupled to each other. Here is an example of a tuple:

```
let tuple = (true, "word", 4.5);
```

The variable tuple contains three items. These three items are actually stored within the tuple struct as distinct member variables. The member variables are given a name according to their respective indices. For example, to access the first member variable of the tuple, we can do tuple.0; Note the use of the dot operator, indicating that this is not a list being indexed, but rather a struct with a distinct member variable, named 0, which is of type bool and stores the value true, as that is what we initialized it with. In the previous example, we used Rust's type inference to determine the types for each member variable. In order to do this explicitly, it would look something like:

```
let tuple: (bool, &str, f64) = (true, "word", 4.5);
```

Recall that I mentioned the items in a tuple did not necessarily need to be coupled. In the examples I've given thus far, each item in the tuple is bound to the tuple variable, meaning that they are dependent (or coupled) to the same variable. We can avoid this by using tuples to perform multiple assignment. For example, we could do:

```
let (x, y, z) = (10.5, 456.32, 358.3);
```

In this case, x is assigned 10.5, y is assigned 456.32, and z is assigned 358.3. The variables x, y, and z can be used completely independently, as if they were their own variables (because they are). What we've done here is use the tuple (10.5, 456.32, 358.3) to perform multiple assignment on distinct variables. We can cast a regular tuple into multiple variables as well:

```
let tuple = (true, "word", 4.5);
let (x, y, z) = tuple; // Assigns true to x, "word" to y, and 4.5 to z
```