Rust

Jeff Shen

Last revised May 5, 2020

Contents

Basic Concepts

Variable Bindings

Use `let` to introduce a **variable binding**. These are immutable by default. Use `mut` to make them mutable:

```
let mut x = 0;
```

Rust is **statically typed**: specify your types up front.

```
let x: i32 = 5;
```

Bindings cannot be accessed outside of the scope they are defined in.

Bindings can be **shadowed** (overwritten):

```
let mut y: i32 = 1;
y = 2; // mutate y
let y = y; // y now immutable and bound to 2
let y = "text"; // rebind y to different type
```

Functions

Define a function with `fn`:

```
fn foo() {
    // do stuff here
}
```

Every program has a `main` function.

Functions can take arguments. The type of the argument must be declared.

Functions can return arguments. Use `->` to indicate the return, and declare the type after the arrow. The last line in the function is what is returned. Do not insert a semicolon at the end of that line.

```
fn add(x: i32, y: i32) -> i32 {
    foo();
    x + y
}
```

Expressions and Statements

Expressions return a value, and **statements**, indicated by a semicolon, do not. Semicolons are used to turn expressions into statements (ie. suppress output).

Assignments to already-bound variables are expressions, but the value returned is `()` rather than the “expected” value. This is because the assigned value can only have one owner:

```
let mut y = 5;
let x = (y = 6); // x has value '()' rather than 6
```

Variable bindings can point to functions:

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
let f: fn(i32) -> i32 = add; // or, let f = add;
let six = f(1, 5);
```

Primitive Types

Boolean

`bool`: `true` or `false`

char

A single Unicode value. Created with `''`.

```
let x = 'x';
let x = '1';
```

Numerics

- **Signed vs unsigned**: Signed integers support both positive and negative values, whereare unsigned integers can only store positive values. For a fixed size, an unsigned integer can store larger positive values. Signed integers are denoted by `i` (eg. `i8` for a signed eight-bit number), and unsigned by `u` (eg. `u16`).
- **Fixed vs variable size**: Fixed size types have a specific number of bits they can store. Sizes can be `8`, `16`, `32` or `64` (eg. `i32`, `u16`). Variable size types are denoted by `isize` and `usize`.
- **Floating-point**: Denoted by `f32` (single precision) and `f64` (double precision).

Arrays

An array is a fixed-size list of elements of the same type. They are immutable by default.

```
let a = [1, 2, 3];
let b = [0; 20]; // 20 elements, each with a value of 0
let a_length = a.len();
let a_first = a[0]
```

Tuples

Tuples are ordered lists of fixed sizes. They can contain multiple types. Fields of tuples can be **destructured** using `let`:

```
let x: (i32, &str) = (1, "hello");
let (a, b) = x; // a gets 1, b gets "hello"
let (c, d) = ("test", 5);
```

Elements of a tuple can be accessed using dot notation:

```
let tup = (1, 2, 3, 4);
let x = tup.0;
let y = tup.3;
```

if

Use an `if` expression (not statement!) to conditionally run code:

```
let x = 5;

if x == 5 {
    println!("x is five")
} else if x == 6 {
    println!("x is six")
} else {
    println!("asdf")
}
```

Since `if` is an expression, it can return a value:

```
let x = 5;
let y = if 5 { 10 } else { 15 };  // y is 10
```

If there is no `else`, then the return value is `()`.

Loops

Use for loops to loop over an iterable:

```
for i in 0..10 {
    println!("{}", x);
}
```

where `0..10` gives an iterable range.

Use `.enumerate()` to keep track of how many times you have looped:

```
for (i, j) in (2..5).enumerate() {
    println!("{} {}", i, j)
}
```

```rust
// Output:
// 0 2
// 1 3
// 2 4
```

Use `while` for while loops. Keep looping while some condition holds.

```rust
let mut x = 5;
let mut done = false;

while !done {
    x += 1;
    if x % 10 == 0 {
        done = true;
    }
}
```

Use `loop` for infinite loops (instead of writing `while true`)

```rust
loop {
    println!("loop forever")
}
```

Use `break` to break out of the loop (can combine with `loop` instead of explicitly defining a `done` condition).

Use `continue` to skip to the next iteration.

Ownership

Rust follows three ownership rules:

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

Stack vs heap

The stack stores values in a stack-like structure: last in, first out. Adding data to the stack is called pushing to the stack, and removing data is called popping off the stack. Data stored on the stack must have a known, fixed size.

When storing data on the heap, a certain amount of memory is requested. The heap finds a place large enough, marks it as being used, and then returns a **pointer**, which gives the address of that place. This is called **allocation**. To get the data, you follow the pointer to get to the address.

Pushing to the stack is faster than allocating on the heap because there is no need to search for free space: the location is always the top of the stack. Similarly, accessing data is also faster, because you don't need to follow a pointer.

Function parameters and variables inside functions are pushed to the stack, and then popped off the stack once the function has completed.

Variable scope

The **scope** is the range in which an item is valid. A scope can be created with `{}`.

```
{ // create a new scope
    let s = "hello"; // s is valid here.
    // do stuff with s.
} // scope is over. s no longer valid.
```

A variable is valid when it comes into scope, and remains valid until it goes out of scope.

`String` type

The `String` type is stored on the heap (and thus is able to store an arbitrary amount of text). They are also mutable, whereas string literals are not. Strings are created from string literals as follows:

```
let mut s = String::from("hello");
s.push_str(", world");
// s has "hello, world"
```

Memory management

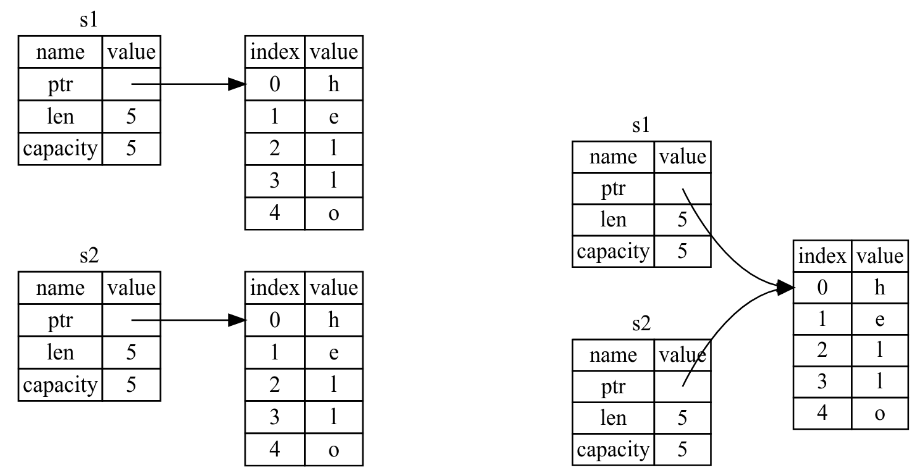
The reason why `String` types are mutable and literals are not has to do with memory. `String` types request memory from the OS during runtime (done with `String::from`), and return the memory when the `String` is finished being used.

Memory return is usually done with a **garbage collector (GC)**, which keeps track of memory that is no longer being used, and cleans it up automatically, or by allocating and freeing memory manually.

Rust takes a different approach and automatically (and deterministically) frees up memory once the variable goes out of scope by calling a special `drop` function (eg. at `}`).

Move, copy, and clone

There are two ways to bind a variable to another.



(a) Deep copy. The actual data is copied.

(b) Shallow copy. The metadata and pointer are copied, but the actual data itself is not.

For data types with a trait called `Copy`, which are usually known-size data that lives only on the stack (eg. ints, bool, floats, char, tuples containing only the previous types), such a binding copies the actual data into the second variable.

```
let s1 = "hello"; // s1 gets "hello"
let s2 = s1; // s2 gets "hello",and s1 remains unchanged.
```

This is fine because this data lives entirely on the stack, so copies of the actual values are quick to make. Here, shallow copy and deep copy are the same thing.

Data types without a known size at compile time live on the heap. For this data, deep copying may not be a great idea. The first variable can point to a large amount of data, and copying everything may be very expensive. Instead, we can do a shallow copy. The problem with this is that when `s1` and `s2` both go out of scope, they will both try to free the same memory. This is called a **double free error** and is not safe. To fix this, Rust **transfers ownership** of the data to `s2`, and invalidates `s1` immediately. Then, when `s2` goes out of scope, it and it alone will free the memory.

```
let s1 = String::from("hello");
let s2 = s1; // s2 gets `String` type "hello", and s1 is invalidated.
```

If we really want to do a deep copy of the heap data, then we can invoke the `clone` method:

```
let s1 = String::from("hello");
let s2 = s1.clone(); // s1 remains valid.
```

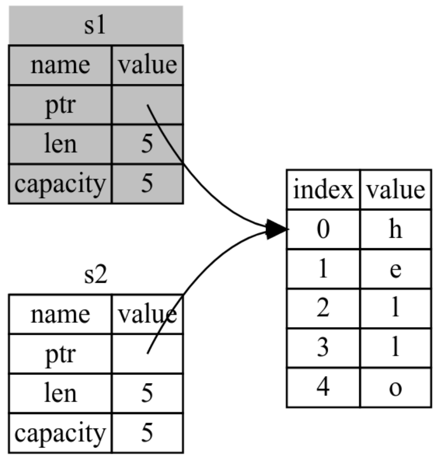


Figure 2: Representation in memory after ‘s1‘ is invalidated.

Functions

Passing a value into a function also transfers ownership of the data to the function as if it were a binding. The variable (unless it is `Copy` is invalid outside of that function. When that function completes, the variable goes out of scope. Function returns also transfer ownership in the same way.

```
fn main() {
    let s = String::from("hello"); // s comes into scope.
    f_take(s); // s moves into the scope of f_take
               // and is no longer valid in main.

    let x = 5;
    f_copy(x); // x moves into the scope of f_copy
               // but x is i32, which is Copy.
               // so x remains valid here.
    // do stuff with x.

    let s2 = String::from("hello"); // s2 comes into scope.
    let s3 = f_take_give_back(s2) // s2 moves into the scope of f_take_give_back
                                  // s2 is no longer valid in main.
                                  // f_take_give_back returns its value into s3.
} // s3 goes out of scope here. s2 is already invalid...
  // ...x goes out of scope. s is already invalid.

fn f_take(some_str: String) { // some_str comes into scope.
    // do stuff with some_str.
} //some_str goes out of scope.
```

```
fn f_copy(some_int: i32) { // some_int comes into scope.
    // do stuff with some_int
} // some_int goes out of scope...
  // ...nothing special happens because it is a copied valued.

fn f_take_give_back(some_str: String) -> String { // some_str comes into scope
    some_str // some_str is returned and ownership is transferred out of the function
}
```

References and Borrowing

In order to get the value of a variable without taking ownership, use `&`. This passes a **reference** of the object instead of the object itself. This is called **borrowing**.

```
fn main() {
    let s1 = String::from("hello");
    let len1 = calculate_len1(&s1); // pass a reference of s1.
    // s1 is still valid here.
    let len2 = calculate_len2(s1); // pass s1 itself.
    // ownership of s1 has been transferred to calculate_len2
    // s1 is no longer valid here.
}

fn calculate_len1(s: &String) -> usize {
    s.len()
} // function does not have ownership of s.
  // s goes out of scope but nothing special happens.

fn calculate_len2(s: String) -> usize { // s comes into scope.
    s.len()
} // function has ownership of s, and it goes out of scope here.
  // drop gets called, and the memory of s is cleared.
```

Mutable references

References are by default immutable. This is to prevent **data races**, which happen when: - two or more pointers access the same data at the same time, - at least one of the pointers is being used to write to the data, and - there is no synchronized access to the data.

Mutable references are allowed under some restrictions using `&mut`: - There can only be one mutable reference to a particular piece of data within a particular scope. - It is possible to create a new scope with `{}` and use multiple mutable references in different scopes. - It is not possible to combine mutable and immutable references in the same scope.

The scope of a reference starts from where it is introduced, and ends after the last time it is used. The following codde is permitted, because the scopes of the immutable reference `s1` ends before the mutable reference `s2` is introduced:

```
let mut s = String::from("hello");

let s1 = &s;
println!("{}", s1);
// s1 no longer being used.

let s2 = &mut s; // this is fine.
```

Dangling references

A **dangling pointer** is a pointer that references a location in memory that might have been given to someone else (ie. memory was freed while the pointer was preserved). Rust will automatically prevent this from compiling.

```
fn main() {
    let dead_reference = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello"); // s comes into scope.

    &s // return reference to s
} // s goes out of scope. but the reference (to this invalid String) has been stored.
```

Additional data types

Slice

Slicing allows a "view" into a collection of elements without ownership. Use `&` to indicate that slices are like references. We can also make references to a portion of the collection.

```
let a = [0, 1, 2, 3, 4];
let complete = &a[..] // slice with all elements
let middle = &a[1..4] // slice with 1, 2, 3

let s = String::from("hello");
let first_two = &a[..2] // slice with "he"
let last_two = &a[3..] // slice with "lo"
```

A string slice is denoted `&str`. String literals are actually string slices. This is why they are immutable: they are immutable references.

When writing a function to take in a string, it is better to use `&str` as the parameter instead of `&String`. Using `&str` means that if we have a `String`, we can pass a slice of the entire string, but if we only have a slice, then we can just pass the slice. It allows for more general use without any loss of functionality.

```
fn some_fn(s: &str) -> &str { // this is better.
...
fn some_fn(s: &String) -> &str { // dont do this.
```

Structs

Classic C structs

Structs are labelled and grouped collections of data (called **fields**). After defining a struct, we create an instance of it and specify concrete values for each of the fields. We can use dot notation to get the value of a particular field, or to change it. In order to change a field, the entire struct instance must be marked with `mut`.

```
struct User {
    username: String,
    n_logins: u32,
    active: bool,
};

let mut user1 = User {
    username: String::from("test"),
    n_logins: 16,
    active: true,
};

user1.active = false;
```

To create a new instance of a struct quickly using most of another instance's values, you can use the **struct update syntax**:

```
let user2 = User {
    username: String::from("other"),
    n_logins: 16, // unchanged
    active: false, // unchanged
}
... // equivalent to the following:
let user2 = User {
    username: String::from("other"),
    ..user1 // struct update
}
```

Tuple structs

Tuple structs are like named tuples, or C structs without field labels:

```
struct Point(i32, i32); // define struct.
let origin = Point(0, 0); // create instance.
let (x, y) = origin; // destructure.
```

Methods

To give a struct a method that it can call, use `impl`. The first parameter of a method is always `self`, which is the instance of the struct that the method is being called on. Multiple methods can be defined in an `impl` block.

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32) {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width >= other.width && self.height >= other.height
    }
}
```

Associated Functions

Associated functions are functions (not methods) defined within `impl` which do not take `self` as a parameter. `String::from` is an example of an associated function. These are often used for returning a new instance of the struct.

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle {
            width: size,
            height: size,
        }
    }
}

let sq5 = Rectangle::square(5);
```

Enums

Enums are used to define different possible variants of some type of data. A instance can only be one variant. Functions that are set up to take in an `enum` can take any variant. Each variant can have some data associated with it, and the types can differ. `impl` can also be used with `enum`.

```
enum Message {
    Quit,
    Move {x: i32, y: i32},
    Write(String),
    ChangeColor(i32, i32, i32),
}

impl Message {
    fn call(&self) {
        // method here.
    }
}

let m = Message::ChangeColor(5, 23, 52);
m.call();
```

Option

`Option` is a special `enum` that encodes the concept of a value being present or absent (like a null value, which Rust doesn't have (for safety purposes)). The `<T>` is a generic type which indicates that it can take any type.

```
enum Option<T> {
    Some(T),
    None,
}
```

Note that a variable of type `Option<T>` and one of type `T` are not the same. They cannot interact like two `T` variables can.

DO GENERICS FIRST. THEN REWRITE THIS.

match

`match` is used to compare a value against a series of patterns and conditionally execute code based on the match. Unlike `if`, the expression doesn't need to return a boolean. Each condition in the `match` is called an **arm**, which is comprised of a pattern and some code, separated by `=>`. It is possible to get the value inside the variant, and then perform some action on that value. Matches are exhaustive: all cases must be explicitly covered. In many cases, the equivalent of an `else` statement is the pattern `_`, which matches any value.

```
enum issue_year {
    2000,
    2001,
    2002,
    ...
}
enum coin {
    penny,
    nickel,
    dime(issue_year),
    quarter,
    loonie,
    toonie,
}

fn get_small_vals(c: coin) -> u8 {
    match c {
        coin::penny => 1,
        coin::nickel => 5,
        coin::dime(issue_year) => {
            println!("This dime was issued in {}", issue_year);
            10
        },
        coin::quarter => 25,
        _ => {
            println!("value too large");
            0
        },
    }
}
```

Generics

Generics can be used to write code that applies to many different types without knowing beforehand what the type will be. Generics are usually denoted by `<T>`. There is no performance cost to using generics because Rust applies **monomorphization** and turns the generic code into a concrete type during compilation.

Functions

```
fn largest<T>(list: &[T]) -> {
```

means that the function `largest` is generic over some type `T`. It has one parameter named `list`, which is a slice of values of type `T`. It returns a value of the same type `T`. Because this function is defined generically, it could be applied to slice of `ints` or a slice of `chars` in the same way.

Structs

```
struct Point<T> {
    x: T,
    y: T,
}

let integer = Point{x: 5, y: 10};
let float = Point{x: 1.2, y: 5.0};
```

Note that although generics can work with different types, for a given instantiation, the `T` is fixed. That is, we cannot define `Point` with `x` and `y` as different types, unless we define it as follows:

```
struct Point<T, U> {
    x: T,
    y: U,
}

let int_and_float = Point{x: 5, y: 10.5};
let both_floats = Point{x: 1.2, y: 5.0};
```

Methods

In order to declare that a method takes a generic, we use `impl<T>`:

```
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```