RUSTACEANSKENYA

BORROW CHECKER RULES & RUST TYPES

List of Types

Primitive types:
- Boolean
- Numeric
- Textual
- Never

Sequence types:
- Tuple
- Array
- Slice
- Vec & VecDeque

User-defined types:
- Struct
- Enum
- Union

Function types:
- Functions
- Closures

Pointer types:
- References
- Raw Pointers
- Function Pointers

Trait types:
- Trait Objects
- impl Trait

Rust's type system ensures safety and performance with primitives, compounds, enums, pointers, traits, and generics for flexible, reusable, and error-free code.

Source: https://doc.rust-lang.org/reference/types.html

Boolean Type

The **`bool`** type, short for boolean, is a *primitive* data type that can hold one of two values: **true** or **false**.

A boolean value occupies **1 byte** in size.

Like all primitives, the boolean type implements the traits **Default**, **Clone**, **Copy**, **Sized**, **Send**, and **Sync** and a many other traits.

Defining a default boolean type:

```
let foo = true;
let bar = false;

// The default is always false
let foo = bool::default();
```

Explicit type annotation:

```
let foo: bool = true;
let bar: bool = false;
```

Operations on Boolean Types

Logical not (!)

!true = false

!false = true

Logical or (|)

true | true = true
true | false = true
false | true = true
false | false = false

Logical and (&)

true & true = true
true & false = false
false & true = false
false & false = false

Logical xor (^)

true ^ true = false
true ^ false = true
false ^ true = true
false ^ false = false

https://doc.rust-lang.org/std/primitive.bool.html

Comparison on Boolean Types

Equality (==)

(true == true) = true
(true == false) = false
(false == true) = false
(false == false) = true

Greater (>)

(true > true) = false
(true > false) = true
(false > true) = false
(false > false) = false

Not Eq (!=)

a != b is the same as !(a == b)

Less (<)

a < b is the same as !(a >= b)

Greater Eq (>=)

a >= b is the same as a == b | a > b

Less Eq (<=)

a <= b is the same as a == b | a < b

Unsigned Integers

Can only represent non-negative whole numbers (i.e., zero and positive numbers)

u8

Min: 0
Max: $2^{8}-1$
Length: 8-bit

Definition:

```
let foo: u8 = 0;
let bar = 1_u8;
let baz = 1u8;
```

Parsing:

```
"0".parse::<u8>()?
```

u16

Min: 0
Max: $2^{16}-1$
Length: 16-bit

Definition:

```
let foo: u16 = 0;
let bar = 1_u16;
let baz = 1u16;
```

Parsing:

```
"0".parse::<u16>()?
```

u32

Min: 0
Max: $2^{32}-1$
Length: 32-bit

Definition:

```
let foo: u32 = 0;
let bar = 1_u32;
let baz = 1u32
```

Parsing:

```
"0".parse::<u32>()?
```

u64

Min: 0
Max: $2^{64}-1$
Length: 64-bit

Definition:

```
let foo: u64 = 0;
let bar = 1_u64;
let baz = 1u64;
```

Parsing:

```
"0".parse::<u64>()?
```

u128

Min: 0
Max: $2^{128}-1$
Length: 128-bit

Definition:

```
let foo: u128 = 0;
let bar = 1_u128;
let baz = 1u128;
```

Parsing:

```
"0".parse::<u128>()?
```

usize

Min: 0
Max: $2^{N}-1$
Length: arch

Definition:

```
let foo: usize = 0;
let bar = 1_usize;
let baz = 1usize;
```

Parsing:

```
"0".parse::<usize>()?
```

Definitions

arch, short for architecture, means it's maximum is limited by the memory address space and CPU register width (i.e., 32-bit and 64-bit CPUs).

.parse::<usize>() method is used to convert the textual representation of a number to it's Rust type.

Signed Integers

Can represent negative and positive whole numbers (i.e., negative, zero and positive numbers)

i8

Min: -2^{7}
Max: $2^{7}-1$
Length: 8-bit

Definition:
```
let foo: i8 = 0;
let bar = 1_i8;
let bar = 1i8;
```

Parsing:
```
"0".parse::<i8>()?
```

i16

Min: -2^{15}
Max: $2^{15}-1$
Length: 16-bit

Definition:
```
let foo: i16 = 0;
let bar = 1_i16;
let bar = 1i16;
```

Parsing:
```
"0".parse::<i16>()?
```

i32

Min: -2^{31}
Max: $2^{31}-1$
Length: 32-bit

Definition:
```
let foo: i32 = 0;
let bar = 1_i32;
let bar = 1i32;
```

Parsing:
```
"0".parse::<i32>()?
```

i64

Min: -2^{63}
Max: $2^{63}-1$
Length: 64-bit

Definition:
```
let foo: i64 = 0;
let bar = 1_i64;
let bar = 1i64;
```

Parsing:
```
"0".parse::<i64>()?
```

i128

Min: -2^{127}
Max: $2^{127}-1$
Length: 128-bit

Definition:
```
let foo: i128 = 0;
let bar = 1_i128;
let bar = 1i128;
```

Parsing:
```
"0".parse::<i128>()?
```

isize

Min: -2^{N-1}
Max: $-2^{N-1}-1$
Length: arch

Definition:
```
let foo: isize = 0;
let bar = 1_isize;
let bar = 1isize;
```

Parsing:
```
"0".parse::<isize>()?
```

Definitions

arch, short for architecture, means it's maximum is limited by the memory address space and CPU register width (i.e., 32-bit and 64-bit CPUs).

.parse::<isize>() method is used to convert the textual representation of a number to it's Rust type.

Integer Best Practices

Digit Seperators

Use an underscore `_` as a digit seperator when dealing with large numbers.

One million = 1_000_000
One Thousand = 1_000
999_999_999 instead of 999999999

Rust Reference Excerpt:

usize: Unsigned integer with the same size as the platform's pointer type. Can represent every memory address in the process.

isize Signed integer with the same size as the platform's pointer type. Used for pointer arithmetic and object/array addressing.

Both are at least **16 bits wide**, but most Rust code assumes 32-bit or 64-bit platforms. 16-bit support is limited and may require special handling.

Default Numeric Types

Rust defaults to using `i32` for numeric types and `isize` for architecture bound numeric types.

Integer Literals

Decimal: 999_999
Hex: 0xff
Octal: 0o77
Binary: 0b1111_0000
Bytes(u8 only): b'Hello'

Default Method

Using **`::default()`** method on an integer results in a **`0`** or **`0.0`** respectively.

```
let foo = u8::default();
let bar = i64::default();
```

Two's Complement

Signed numbers are stored using two's complement representation.

All integer types are stored on the stack.

Floating-point Integers

Rust has two primitive types for floating-point numbers, **f32** and **f64**.

These two types handle both signed and unsigned floating-point numbers.

IEEE 754-2008

Rust IEEE 754-2008 “binary32” and “binary64” floating-point types are f32 and f64, respectively.

f32

Definition:

```
let foo: f32 = 0.1;
let bar = 0.1_f32;
let bar = 0.1f32;
```

Parsing:

```
"0.1".parse::<f32>()?
```

f64

Definition:

```
let foo: f64 = 0.1;
let bar = 0.1_f64;
let bar = 0.1f64;
```

Parsing:

```
"0.1".parse::<f64>()?
```

Defaults

The default type is **f64** because on modern CPUs, it’s roughly the same speed as **f32** but is capable of more precision. Note that these two types are work on both 32-bit and 64-bit platforms.

Default Method

Using `**::default()**` method on an float results in a or `**0.0**` respectively.

```
let foo = f32::default();
let bar = f64::default();
```

Financial applications

Avoid using floating-point in financial applications due to loss of precision. Use integers to represent the integral part and fractional part of the monetary value seperately. You can use a struct to hold the 2 parts in one data structure.

All floating-point types are stored on the stack.

Textual Types: char and str

str type

Also called a `string slice` usually seen in it's borrowed for **&str**. It is also the type used to represent a string literal (constant string) **`&'static str`**.

Usage

```
let hello_world = "Hello, World!";
let hello_world: &'static str = "Hello, world!";
```

Rust libraries may assume that string slices are always valid **UTF-8,** non-UTF-8 string slice can lead to undefined behavior. Since str is a dynamically sized type, it can only be instantiated through a pointer type, such as **&str**.

https://doc.rust-lang.org/std/primitive.str.html

char type

The char type represents a single character, `char` is a 'Unicode scalar value'.

Usage

```
let c: char = 'a';
```

Rust libraries may assume that string slices are always valid **UTF-8,** non-UTF-8 string slice can lead to undefined behavior.

https://doc.rust-lang.org/std/primitive.char.html

https://doc.rust-lang.org/std/char/index.html

Textual Types: String

String is UTF-8-encoded, growable and stored on a heap-allocated buffer. **String**s are always valid UTF-8. In Rust ownership rules, a String owns all of it's contents.

std excerpt on components of a String:

A String is made up of three components: a pointer to some bytes, a length, and a capacity. The pointer points to the internal buffer which String uses to store its data. The length is the number of bytes currently stored in the buffer, and the capacity is the size of the buffer in bytes. As such, the length will always be less than or equal to the capacity.

This buffer is always stored on the heap.

You can look at these with the **as_ptr**, **len**, and **capacity** methods.

Usage

```
let hello_world = String::from("Hello, World!");
let into_a_string: String  = "Hello, world!".into();
let empty_string = String::new();
let default_empty_string = String::default();
let borrowed_str_to_a_string = "Hello, world!".to_string();


hello_world.len(); // Check the length of a String
hello_world.capacity(); // Check the capacity of a String
hello_world.ptr(); // Get the pointer to the heap buffer stroing the String
```

https://doc.rust-lang.org/std/string/struct.String.html

Textual Types: OsString & OsStr

To create non-UTF-8 Strings use a **OsString**. This is an owned, mutable platform-native strings. It can be cheaply converted to Rust strings. It is found in `std::ffi` module which are utilities related to Foreign Function Interface bindings. An **&OsStr** is a borrowed **OsString**.

std excerpt on components of a String:

The need for this type arises from the fact that:

- On Unix systems, strings are often arbitrary sequences of non-zero bytes, in many cases interpreted as UTF-8.
- On Windows, strings are often arbitrary sequences of non-zero 16-bit values, interpreted as UTF-16 when it is valid to do so.
- In Rust, strings are always valid UTF-8, which may contain zeros.

Usage

```
let os_string = OsString::new();
let from_a_string: OsString = String::from(“hello”).into();
```

https://doc.rust-lang.org/std/ffi/struct.OsString.html

Never Type

The never type ! is a type with no values, representing the result of computations that never complete.

They can be coerced into other types.

This type is common in embedded and bare metal code.

Presently it can only appear in function return types indicating that it is a diverging function that never returns.

Usage

```
fn foo() -> ! {
    panic!("This call never returns.");
}
```

Usage in unsafe contexts like FFIs

```
unsafe extern "C" {
    pub safe fn no_return_extern_func() -> !;
}
```

https://doc.rust-lang.org/reference/types/never.html

Tuple Type

A finite heterogeneous sequence, (T, U, ..). Finite means tuples have a length. The length of a tuples is also known as **`*arity*`**.

Tuples with the same data types are known as **homogeneous**.

https://doc.rust-lang.org/std/primitive.tuple.html

heterogeneous

Each element of the tuple can be of a different type

```
("hello world”, 'o', 'p', String::new(), 4u8);
```

tuple indexing

Since tuples are a sequence, they can be accessed by position.

```
let tuple = ("hello world”, String::new(), 4u8);
assert_eq!("hello world”, tuple.0)
```

common usage

Tuples are often used as return types to return more than one value or data type:

```
fn foo() → (i32,i32){
     (1,1)
}

fn bar() → (String, u8) {
     (String::from(“hello”), 9)
}
```

arrays to Homegeneous tuples

Tuples with the same data type can be created from an array of the same length as the tuple.

```
let array: [u32; 3] = [1, 2, 3];
let tuple: (u32, u32, u32) = array.into();
```

Array Type

An array is a fixed-size sequence of **N** elements of type **T**.

An Array = [T; N]

where

T is the data type

N is a usize describing the length of the array

An array is ***stack allocated*** and Constant size. An array can be allocated on the heap if it is wrapped in a **Box** type.

Copying Arrays into Arrays

An array of the same length can be copied to another one replacing all elements.

```
let mut array1: [u32; 3] = [1, 2, 3];
let array2: [u32; 3] = [4, 5, 6];
array1.copy_from_slice(&array2);
```

Creating an array

```
// A stack-allocated array
let array: [i32; 3] = [1, 2, 3];

// A heap-allocated array, coerced to a slice
let boxed_array: Box<[i32]> = Box::new([1, 2, 3]);
```

Array indexing

Arrays are sequence therefore can be indexed by position

```
let array = [1u8, 2, 3];

array[0]; // Panics on out-of-bounds
array.get(0); //  Never panics, `Option` type
```

Iter trait

Arrays can be looped through.

```
let array: [u32; 3] = [1, 2, 3];
array.iter().for_each();
array.iter().map()...;
for item in array{}
```

https://doc.rust-lang.org/std/primitive.array.html

Vec<T> Type

A ***Vec<T>*** is a contiguous growable array type containing items of type ***T***.
Vec is short for **vector**.

A ***Vec<T>*** is stored on the heap as a contiguous buffer whose length is increased whenever the length exceeds the vector's capacity. The capacity is the amount allocated for any future elements that will be added onto the vector.

Pushing items in a vector inserts them at the end.

Creating an Vec

```
let vector: Vec<i32> = vec![1,2,3];


// Creating an empty Vec
let empty_vec: Vec<u8> = vec![];
let empty_vec_with_default: Vec<u8> = Vec::default();
let empty_vec_with_default = Vec::<u8>::default();
let empty_vec_with_new: Vec<u8> = Vec::new();
let empty_vec_with_new = Vec::<u8>::new();
```

Vec indexing

Vec<T> is sequence therefore can be indexed by position

```
let vector = vec![1u8, 2, 3];

vector[0]; // Panics on out-of-bounds
vector.get(0); //  Never panics, returns `Option` type
```

Looping through

```
let vector = vec![1u8, 2, 3];

vector.iter().for_each();
vector.iter().map()...;
for item in vector{}
```

Mutable Operations

```
let mut vector = vec![1u8, 2, 3];
vector.push(4); // Inserting into a vec
vector.pop(); // Remove last element
vector.remove(0); //panics on out-of-bounds, O(n)
vector.swap_remove(0); //out-of-bounds panics, O(1)


// Remove element at index without panicking
if vector.get(4).is_some() {
     vec.remove(4);
}
```

https://doc.rust-lang.org/std/vec/struct.Vec.html

VecDeque<T> Type

A ***VecDeque<T>*** is a double-ended queue implemented with a growable ring buffer.

A ring buffer (also known as a circular buffer) is a fixed-size data structure that uses a single, contiguous block of memory and treats it as if it were connected end-to-end in a circular fashion.

This type is commonly used where one wants to push in front or at the back of the queue, unlike a Vec<T> which supports pushing only at the back.

Creating an VecDeque

```
use std::collections::VecDeque;

// Creating an empty VecDeque
let empty_with_default: VecDeque<u8> = VecDeque::default();
let empty_vec_with_default = VecDeque::<u8>::default();
let empty_with_new: VecDeque<u8> = VecDeque::new();
let empty_with_new = VecDeque::<u8>::new();
```

Mutable Operations

```
let mut queue: VecDeque<u8> = VecDeque::default();

queue.push_front(8); // Prepends an element to the deque.
queue.push_back(5); // Appends to the back of the deque.
queue.pop_back(5); // Removes last element from the deque.
queue.pop_front(5); // Removes first element from the deque.
```

Iterator trait

Just like a Vector, a VecDeque implements the iterator trait and can looped through using for loop and iterators.

https://doc.rust-lang.org/std/collections/struct.VecDeque.html

Slice Type

Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection. A slice is a kind of reference, so it does not have ownership.

A dynamically-sized view into a contiguous sequence, ***[T]*** where ***T*** is a type.

Contiguous here means that elements are laid out so that every element is the same distance from its neighbors.

Slices are a view into a block of memory represented as a pointer and a length.

Example Coercing A Vec into a Slice

```
// slicing a Vec
let vec = vec![1, 2, 3];
let int_slice = &vec[..];
// coercing an array to a slice
let str_slice: &[&str] = &["one", "two", "three"];
```

Mutable Slices

```
let mut x = [1, 2, 3];
let x = &mut x[..]; // Take a full slice of `x`.
x[1] = 7;
```

Range support

```
let x = [1, 2, 3];
let x = &x[0..=1];
```

Iterator trait

A Slice implements the iterator trait and can looped through using for loop and iterators.

https://doc.rust-lang.org/std/primitive.slice.html

HashMap Type

A hash table-based collection that stores key-value pairs.

Keys are hashed, and values are stored based on their hash. The order of keys is not guaranteed because it depends on the hash function.

Defining a hashmap

```
use std::collections::HashMap;

let map: HashMap<u8, String> = HashMap::new();
let map: HashMap::<u8, String> = HashMap::new();
```

Inserting key value pairs

```
let mut map: HashMap<u8, String> = HashMap::new();
map.insert(4, "Greeting".to_string());
```

Fetching a value based on a key

```
map.get(4);
```

Checking if a key exists

```
map.contains_key(&4);
```

Iterator trait

A HashMap implements the iterator trait and can looped through using for loop and iterators.

https://doc.rust-lang.org/std/collections/struct.HashMap.html

BTreeMap Type

A BtreeMap is a balanced binary tree-based collection that stores key-value pairs.

Keys are sorted in ascending order according to their natural ordering (or a custom comparator). Keys are always sorted.

Defining a BTreeMap

```
use std::collections::BTreeMap;

let map: BTreeMap<u8, String> = BTreeMap::new();
let map: BTreeMap::<u8, String> = BTreeMap::new();
```

Inserting key value pairs

```
let mut map: BTreeMap<u8, String> = BTreeMap::new();
map.insert(4, "Greeting".to_string());
```

Fetching a value based on a key

```
map.get(4);
```

Checking if a key exists

```
map.contains_key(&4);
```

Iterator trait

A HashMap implements the iterator trait and can looped through using for loop and iterators.

https://doc.rust-lang.org/std/collections/struct.HashMap.html

New Type

The Newtype Pattern in Rust is a design pattern used to create a new, distinct type by wrapping an existing type.

The new type is distinct from the original type, even though it contains the same data, allowing you to enforce type safety, add custom behavior, or provide semantic meaning to the wrapped type.

This pattern is called "newtype" because it creates a new type that is distinct from the original type, while still reusing the underlying data representation.

Syntax

```
type TypeName = DataType;
```

Example

```
type Kilometers = f64;
type Litres = u64;
type CustomFoo = Foo; // Same as `Foo` but as a new type

struct Foo {
    bar: u8,
    baz: u16,
}
```

Creating a New-Type

```
    let foo: Kilometers = 4.0;
    let bar: Litres = 1000;
    let bar: CustomFoo = Foo { bar: 0, baz: 1 };
```

Struct Type

A struct is like a tuple in that it is a type that can contain heterogenous or homogenous types but unlike a tuple, all it's fields must be named.

Structs group related fields and data together.

Creating a Struct

```
struct Foo{
     bar: u8,
     baz: String,
}
```

Implementing Methods for a Struct

```
impl Foo {
     // Self  keyword is a type alias for `Foo`
     pub fn new() → Self {
          Self {bar: 4, baz: String::from("Hello") }
     }
}
```

Creating and Accessing a Struct

```
let foo = Foo {bar: 0, baz: String::new() };

// Creating a struct via its method.
let foo = Foo::new();

// accessing a field of a struct;
let counter = foo.bar;
let greeting - foo.baz;
```

Enum Type

Enums are a way of saying a value is one of a possible set of values.

Some built-in enum types are:
Result<T, E> - describe where a value is a success `**Ok(T)**` *or failure* `**Err(T)**`

Option<T> - describe whether a value exists `***Some(T)***` or it dosen't `***None***`.

Creating an Enum

```
enum Foo{
     Bar,
     Baz,
}
```

Implementing Methods for a Struct

```
impl Foo {
     // Self  keyword is a type alias for `Foo`
     pub fn new() → Self {
          Self::Bar
     }
}
```

Creating and Accessing a Struct

```
let foo = Foo::Baz;

// Creating a struct via its method.
let foo = Foo::new();

// matching on the fields of an enum;
match foo{
     Foo::Bar => { // do something },
     Foo::Baz => { // do something },
}
```

Function Type

Describes the signature of a function, including its input parameters (arguments) and return type.

It defines what a function accepts as input and what it produces as output, without specifying the actual implementation.

Syntax

```
fn(param1_type, param2_type, ...) -> return_type
```

Examples

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}


// Example with `impl Trait`
fn add(a: cfg core::fmt::Display) → String {
    a.to_string()
}


// Example with conditional arguments based on various
// features, architecture, etc
pub fn my_function(
    input1: usize,
    #[cfg(debug_assertions)] input2: String,
    #[cfg(feature = "my-feature")] input3: Vec<u32>,
) {
}
```

Closure Type

Closures are anonymous functions that can capture variables from their surrounding environment.

They are similar to regular functions but are more flexible because they can:

- Be defined inline without a name.
- Capture and use variables from the scope in which they are defined.
- Be stored in variables, passed as arguments to other functions, or returned from functions.

Traits

Fn: closure borrows values immutably.
FnMut: closure borrows values mutably.
FnOnce: The closure takes ownership of captured values (can only be called once).

Syntax

```
|parameters| -> return_type { body };
```

Examples

```
// Types are inferred based on usage
let multiply = |a, b| a * b;
println!("Result: {}", multiply(3, 4));


// Capturing Variables by Reference (&T)
let x = 5;
let closure = || println!("x: {}", x); // Borrows `x` immutably
Closure();


// Capturing Variables by Mutable Reference (&mut T)
let mut x = 5;
let mut closure = || {
    x += 1; // Mutates `x`
    println!("x: {}", x);
};
closure(); // Output: x: 6

// Capturing Variables by Value (T)
let x = 5;
let closure = move || println!("x: {}", x); // Takes ownership
of `x`
closure();
```

Pointer Types

References (& and &mut)

Shared references (&) point to memory which is owned by some other value. It prevents direct mutation of the value unless via interior mutability. Copying a reference is a “shallow” operation.

Mutable references point to memory which is owned by some other value. A mutable reference type is written &mut type or &'a mut type. A mutable reference (that hasn’t been borrowed) is the only way to access the value it points to, so is not Copy.

Raw pointers (*const and *mut)

Raw pointers in Rust (*const T and *mut T) are low-level pointers that lack safety or liveness guarantees. They allow direct memory access but bypass Rust's ownership rules, making dereferencing them an unsafe operation. Raw pointers can be created using &raw const or &raw mut, and they can be converted into references via reborrowing (&* or &mut *). While copying or dropping a raw pointer has no effect on other values, they are primarily used for interoperability with foreign code or performance-critical scenarios.

When comparing raw pointers, they are evaluated by their memory address rather than the data they point to, with additional metadata compared for dynamically sized types. Although raw pointers provide fine-grained control over memory, they are discouraged in idiomatic Rust due to their potential for unsafety and are typically reserved for low-level programming or interfacing with external libraries.

https://doc.rust-lang.org/reference/types/pointer.html#raw-pointers-const-and-mut

Function Pointers

Function pointer types, written using the fn keyword, refer to a function whose identity is not necessarily known at compile-time.

Syntax)

```
fn(param1_type, param2_type, ...) -> return_type
```

Examples

```
type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);

fn add(x: fn(i32) -> i32, y: i32) -> i32 {
    x(4) + y
}
```

https://doc.rust-lang.org/reference/types/function-pointer.html

Ownership Rules: Intro

Rust ownership rules allow Rust to make memory safety guarantees without needing a garbage collector. Memory can be freed whenever data goes out of scope.

Ownership Rules

Each value in Rust has an owner.
There can only be one owner at a time.
When the owner goes out of scope, the value will be dropped.

Stack and the Heap

The stack and heap are memory regions used at runtime: the stack stores fixed-size data in a fast, last-in-first-out manner, while the heap stores dynamically-sized or unknown-size data, requiring allocation and pointer access.

Stack operations are faster due to their simplicity, whereas heap access is slower because it involves following pointers. Ownership in Rust manages how data is stored on the heap, ensuring efficient memory use and cleanup.

Copy and Clone

The Copy and Clone traits define how types are duplicated. Types implementing Copy are stored on the stack and use a cheap bitwise copy, as they don't involve heap memory. Types requiring Clone perform an explicit deep copy of heap-allocated data. A bitwise copy of a type needing Clone would only duplicate the pointer, risking double-free errors when ownership rules are violated.

Ownership Rules: Copy & Clone Examples

Copy

```
// Integers are stack allocated and implement copy. Their
size is also known at compile time.
let foo = 1u8;
let bar = foo;
assert_eq!(foo, bar);

// Constants and static strings are also stack allocated
let foo = "Hello World";
let bar = foo;
assert_eq!(foo, bar);

const baz: &str = "const Hello World";
let bar = baz;
assert_eq!(bar, baz);
```

Clone

```
// String and Vec are heap allocated and do not implement
// Copy, instead they implement clone
let foo = String::from("Hello World");
let bar = foo.clone();
assert_eq!(foo, bar);

let foo = HashMap::<u8, String>::new();
let bar = foo.clone();
assert_eq!(foo, bar);
```

Ownership Rules: References and Borrowing

References (&)

A reference is an address pointing to data owned by another variable, guaranteed to be valid and of a specific type for its entire lifetime, unlike a pointer. To use a reference for a type, simply prepent **(&)** symbol.

```
let foo = String::new();
let bar = &foo;

// A reference as function argument
fn foo(bar: &String) {}

// Attempting to modify a borrowed value results in an error:
// cannot borrow `*some_string` as mutable,
// as it is behind a `&` reference

let foo = String::from("Hello World");
let bar = &foo;
bar.push_str("!");
```

Mutable references (&mut)

Mutable references allow a value to be borrowed and modified at the same time. To make a value mutable just prepend ***&mut*** to the value. Only one mutable borrow can happen in the same scope.

```
// This results in an error:
// cannot borrow `s` as mutable more than once at a time
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

// We can fix this by moving `r1` to another scope
{
    let r1 = &mut s;
} // The mutable borrow is dropped here
let r2 = &mut s;
```

Ownership Rules: Dangling References

🦀 Lifetimes (') & Dangling References

A dangling reference in Rust is a reference that points to a memory location that has already been deallocated or gone out of scope, making it invalid and unsafe to access.

Rust uses lifetimes to prevent dangling references. Lifetimes in Rust are a mechanism to ensure that references remain valid as long as they are used, by specifying how long a reference is guaranteed to exist relative to other references or data.

```
fn dangling_reference() -> &i32 {
    let x = 5; // `x` is created on the stack
    &x         // Attempt to return a reference to `x`
} // `x` goes out of scope here and is dropped


// An example that fixes this
let foo = 4i32;
dangling_reference(&foo);

// Here, `x` lifetime is constrained to the lifetime
// of the main function therefore lives beyound
// the scope of this function
fn dangling_reference<'a>(x: &'a i32) -> &'a i32 {
    x
} // `x` lifetime is still valid outside this scope
```

Smart Pointers

Smart pointers, on the other hand, are data structures that act like a pointer but also have additional metadata and capabilities. Smart pointers have no synchronization mechanism to guarantee thread-safety.

https://doc.rust-lang.org/beta/book/ch15-00-smart-pointers.html

Refcell<T>

Interior mutability is a Rust design pattern that allows data to be mutated even when immutable references exist, bypassing normal borrowing rules by using unsafe code within a data structure. This unsafe code manually enforces borrowing rules at runtime, and the pattern is wrapped in a safe API, ensuring the outer type remains immutable while enabling controlled mutation.

Example:

```
use std::cell::RefCell;

let array = RefCell::new(1u8);
*array.borrow_mut() = 5; //Mutate the value
```

Box<T>

Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data. Box<T> is used for types with unknown sizes at compile time, transferring ownership of large data without copying, or owning a value that implements a specific trait.

```
let store_array_on_heap = Box::new([1u8, 2, 3]);
```

Rc<T> (Reference Counted)

A smart pointer that enables multiple ownership by tracking the number of references to a value on the heap, ensuring the value is only cleaned up when no references remain. It's useful in scenarios like graph data structures where a node may have multiple owners, but it's limited to single-threaded contexts.

Example:

```
let array = Rc::new([1u8, 2, 3]);
let cloned_array = Rc::clone(&array);
let cloned_array = array.clone();
```

Cell<T>

std::cell::Cell<T> is a type in Rust that provides interior mutability for values of type T, allowing mutation of data even when the Cell itself is immutable enabling you to mutate the value inside the Cell without requiring a mutable reference.

```
 let c = std::cell::Cell::new(5);
 c.set(10); // Mutate the value inside the Cell
```

Concurrency Types: Arc, Mutex & RwLock

All concurrency primitives are part of the **std::sync** module providing a way to synchronize data across threads while guaranteeing thread safety.

Mutex<T>

Ensures **mutual exclusion** when accessing shared data, allowing interior mutability. Panicking inside a mutex makes it poisoned.

```
use std::sync::Mutex;

let res_mutex = Mutex::new(0_u32);
*res_mutex.lock().unwrap() += 1;
```

Arc<T>

Arc is a thread-safe, reference-counted pointer for shared ownership of heap-allocated data, ensuring immutability unless combined with types like Mutex, RwLock, or Atomic for interior mutability.

```
use std::sync::Arc;

let foo = Arc::new(vec![1.0, 2.0, 3.0]);
// The two syntaxes below are equivalent.
let a = foo.clone();
let b = Arc::clone(&foo);
// a, b, and foo point to the same memory location
```

RwLock<T>

A reader-writer lock allows multiple readers or one writer at a time, with read-only access for readers (shared) and exclusive access for writers; it becomes poisoned if a panic occurs during a write lock.

```
use std::sync::RwLock;

let lock = RwLock::new(5);
{
    let r1 = lock.read().unwrap();
    let r2 = lock.read().unwrap();
} // many readers, read locks are dropped at this point

{
    let mut w = lock.write().unwrap();
    *w += 1;

} // one writer, write lock is dropped here
```

Condvar Type

A condition variable allows a thread to block efficiently until an event occurs, typically associated with a predicate and a mutex; misuse with multiple mutexes may cause a runtime panic.

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = Arc::clone(&pair);

// Inside of our lock, spawn a new thread, and then wait
for it to start.
thread::spawn(move || {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    // We notify the condvar that the value has changed.
    cvar.notify_one();
});

// Wait for the thread to start up.
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
    started = cvar.wait(started).unwrap();
}
```

Barrier Type

A barrier enables multiple threads to synchronize the beginning of some computation.

```
use std::sync::{Arc, Barrier};
use std::thread;

let n = 10;
let mut handles = Vec::with_capacity(n);
let barrier = Arc::new(Barrier::new(n));

for _ in 0..n {
    let c = Arc::clone(&barrier);
    // The same messages will be printed together.
    // You will NOT see any interleaving.
    handles.push(thread::spawn(move || {
        println!("before wait");
        c.wait();
        println!("after wait");
    }));
}

// Wait for other threads to finish.
for handle in handles {
    handle.join().unwrap();
}
```

mpsc & mpmc types

These types are channels used to pass messages. They contain a **Sender** and a **Receiver** part.

mpsc

multi-producer, single-consumer allows multiple threads to send data to one receiver.

```
use std::thread;
use std::sync::mpsc::channel;

// Create a simple streaming channel
let (tx, rx) = channel();
thread::spawn(move || {
    tx.send(10).unwrap();
});

while let Ok(sent_value) = rx.recv() {
     // Do something
}
```

mpsc (Nightly Rust as of writing)

multi-producer, multi-consumer enables multiple threads to both send and receive data concurrently.

```
#![feature(mpmc_channel)]

use std::thread;
use std::sync::mpmc::channel;

// Create a simple streaming channel
let (tx, rx) = channel();
thread::spawn(move || {
    tx.send(10).unwrap();
});

while let Ok(sent_value) = rx.recv() {
     // Do something
}
```

Once, OnceLock & LazyLock types

These are thread-safe one-time initializion types.

Once

A low-level synchronization primitive for one-time global execution.

```
use std::sync::Once;

static INIT: Once = Once::new();
static mut VAL: usize = 0;

let foo = unsafe {
     INIT.call_once(|| { VAL = 400; });
     VAL
};
```

OnceLock

A synchronization primitive which can nominally be written to only once.

```
use std::sync::OnceLock;

static CELL: OnceLock<usize> = OnceLock::new();
let value = CELL.get_or_init(|| 12345);
CELL.get(); // Get the value stored, returns an Option<T>
```

LazyLock

A value which is initialized on the first access.

```
use std::sync::LazyLock;

static DEEP_THOUGHT: LazyLock<String> = LazyLock::new(|| {
     // Initialize something here
});

let _ = &*DEEP_THOUGHT; // Access the value stored
```

Inferred Types

The inferred type asks the compiler to infer the type if possible based on the surrounding information available.

🦀 `_` underscore when used in place of type definitions tells the compiler to infer the data type.

🦀 Example: Infer type T of Vec<>

```
let x: Vec<_> = (0..10).collect();
```

🦀 Example: Infer a lifetime

```
struct Foo<'a>{
     bar:&'a str
}

impl<'a> for Foo<’_> {

}
```

Resources

Rust Book: Common Programming Concepts
https://doc.rust-lang.org/beta/book/ch03-00-common-programming-concepts.html

Rust Reference: Types
https://doc.rust-lang.org/reference/types.html

Rust By Example: Primitives
https://doc.rust-lang.org/rust-by-example/primitives.html

RustaceansKenya Events Code & Presentation Archive
https://github.com/RustaceansKenya/Events