A Cursory Introduction To Ownership in Rust

David Johnston

Contents

The Rust Programming Language

Rust is a new systems programming language. It is comparable to C++ in that it provides the programmer with a high degree of control over application performance. However, unlike C++, Rust provides this control while also ensuring some very strong safety guarantees.

Included in these safety guarantees is *memory safety*. The design of Rust enables the programmer to make efficient programs without needing to worry about unforeseen space leaks, buffer overruns, and dangling pointer dereference bugs. Unlike many other memory safe languages (e.g. Java), Rust does not require a runtime garbage collector.

The idea of *ownership* is one of the core ideas in the programming model, and it is foundational in enabling the Rust type system to provide a number of its safety guarantees. This document is meant to be only a cursory introduction to *ownership*, not a complete introduction to the whole language.

- For a much more complete introductory description of the language, see The Rust Programming Language.
- For a much more complete set of instructive of Rust examples, see Rust By Example.
- For an effective introductory presentation on some core ideas of Rust, see Guaranteeing Memory Safety in Rust (Video)

Some Background on Ownership In C++

Stack Allocation Discipline in C++

As will be discussed later in this document, object ownership is a concept which is an essential and formal part of Rust's type system. The ideas behind object ownership in Rust grew out of common design patterns which have been widely used in other languages.

In order to introduce ownership as it is used in Rust, we will first consider some C++ examples. We have chosen C++ both because of its familiarity to many programmers and because many of Rust's design decisions have come from C++.

C++ programmers need to always be conscious of the possibility of memory bugs. Much modern C++ development follows certain code conventions or patterns to mitigate the risk of these bugs. One such convention is stack allocation discipline, where most objects are declared on the program execution stack.

According to this convention, client code should not call `new` or `malloc()` explicitly, even if a dynamic amount of memory is needed for that object.

Instead, an fixed-size object should be declared on the stack as a handle to heap-allocated memory. The implementation of this handle object should initialize fields of the object and allocate resources (e.g. memory) in its constructor and also release these resources in its destructor. This convention is called *Resource Acquisition Is Initialization*, or RAII. (Stroustrup, 2013, p. 64)

Below is an example which demonstrates one way to instantiate a `vector` object. This class and this example follow RAII discipline.

```
#include <vector>

int main() {
    // `vec` is a stack-allocated data structure which points to
    // three heap-allocated `int` values, each initialized to `0`.
    std::vector<int> vec (3, 0);
}
```

The `vec` variable's lifetime spans from its declaration to wherever it goes out of scope. The `vec` handle object is responsible for

- Acquiring the dynamically allocated backing store,
- Managing the reallocation of the backing store as the vector needs to grow and shrink, and
- Freeing the backing store when it is no longer being used.

One informal way to characterize the relationship between the `vec` handle and its backing store is to say that `vec` *owns* or *uniquely owns* its backing store.

One can make efficient and safe C++ code by placing the responsibilities of acquiring and freeing resources into the implementation of such handle objects thus encapsulating the memory management logic. The main reason that this is considered safer than "naked" `new` and explicit `malloc()` is that it hides the details of acquiring and freeing resources from the code which is *using* the data structure.

C++ Is Not A Memory Safe Language

Consider the following code example, and suppose `foo()` is a function from the `foobar` library.

```
#include <vector>
#include <foobar>

int main() {
```

```
5       std::vector<int> vec (3, 0);
6       foo(vec);  // What can `foo()` do with the objects which `vec` owns?
7   }
```

When looking at this code, one might reasonably ask, "What is `foo` *supposed* to be doing with `vec` and the memory which `vec` owns?"

Is `foo()` supposed to be temporarily borrowing access to to `vec`? Should it be taking ownership of `vec`? Is `foo` taking ownership of some part of the data structure or taking ownership of its elements? The programmer may well have precise answers to these questions, either from code documentation or code conventions. However, the only way to know whether the implementation actually adheres to this *expected* ownership behavior is to look closely at the full implementation of `foo()`.

As we have said, this very useful pattern of ownership is only a convention in C++. It is not *enforced* by the language itself. In general, there is no simple to use analysis to check whether the program's expected ownership behavior is the *actual* ownership behavior.

The essential reason that C++ is unsafe is that even when trying to follow a safe code patterns, it is still possible to introduce memory bugs which go undetected.

Ownership Basics In Rust

Some of the most intriguing contributions of the Rust programming language are how a number of safe ownership patterns (including RAII-like safe resource management) are checked by the compiler. The Rust compiler uses code analyses to check that every Rust program follows a standard set of safe code conventions. The language has been designed with set analyses such that, unlike most other memory-safe languages, a Rust program does not require a runtime garbage collector or any sort of language runtime; this is in spite of the fact that Rust offers the programmer pointers and explicit memory allocation.

Many C/C++ programmers may be surprised at the idea that a language can have explicit memory management and memory safety without some sort of runtime overhead. However, Rust shows that this is possible. [1]

In the following sections, we demonstrate some of the building blocks of object ownership in Rust.

What Is Dropped At The End Of A Lifetime

The following Rust program creates a vector like in the C++ examples above. Like in C++, `vec` is a fixed-sized, stack-allocated data structure which includes a pointer to heap-allocated memory.

```
fn main() {
    // `vec` is a stack-allocated data structure which points to
    // three heap-allocated integer values, each initialized to `0`.
    let vec = vec![0, 0, 0];
}
```

To use Rust terminology, we say that a variable's lifetime spans from the point at which it is declared to the point at which it is dropped (i.e. freed). In this case, `vec` is simply dropped when it goes out of scope at the end of the `main()` function. By identifying when each variable is dropped in Rust, the compiler can then infer where to add code to free an object.

A Rust variable is dropped at the end of its lifetime, but more importantly, every object which that variable owned is dropped at that point as well. In this way, if the data structure is recursively defined, its children will be freed as well.

To make this idea more explicit, suppose our `vec` object has three fields: an integer `len`, an integer `cap`, and a pointer `ptr`, which points into heap memory. In Rust, the `vec` object would *own* each of these fields.

A vector's `ptr` could reasonably be implemented as a `Box` pointer, a *unique* pointer to a heap-allocated object [2]. A `Box` pointer is unique in the sense that no other pointer can point to the boxed object. The object cannot be aliased; the type system will guarantee it [3]. The `Box` uniquely *owns* the `T`s to which it points and the memory which is used to hold those `T`s.

Similarly, we consider `vec` to *own* each of its fields. So, when a `vec` is dropped, each of its fields will be dropped as well. When the vector's `Box` pointer is dropped, the heap-allocated object which it owns will be dropped as well. The Rust compiler infers and generates code to do the appropriate garbage collection. (A consequence of this is that the programmer does not need to manually implementing a destructor for `vec`.)

In C++, the idea that the vector owned its heap-allocated backing store was just an informal idea in our program design. We have now seen that object ownership has been made an explicit part of the Rust programming model and that it is used in order to generate code to free heap-allocated objects.

Borrowing Ownership Is Explicit

We now introduce the idea of borrowing an object using borrow references. Consider the following example, and notice the use of the `&` operator.

```
fn foo_borrow(vec: &Vec<u32>) {
    // ...
}

```

```
fn main() {
    let vec = vec![0, 0, 0];
    foo_borrow(&vec);
}
```

In C/C++, the `&` represents the address-of operator; in C++, it also represents the reference operator and is used to denote reference types. However, in Rust, it does not mean any of these things. Instead, the `&` represents the declaration of a *borrow reference*.

According to the type signature of `foo_borrow()` the function only *borrows* ownership of its `vec` argument. This is a guarantee that the function body will not stash a reference to any memory owned by `vec`. It is in the rules of the type system that ownership cannot be taken in a context where ownership is being borrowed. The consequence is that the callee (that is, `foo_borrow()`) is guaranteed to give back all ownership of the object to the caller (in this case, the `main()` function).

Moving Ownership Is Explicit

An alternative to borrowing ownership is moving ownership. Consider the following example. Note that it is essentially the same as the last example, except that the `&` operators have been removed.

```
fn foo_move(vec: Vec<u32>) {
    // ...
}

fn main() {
    let vec = vec![0, 0, 0];
    foo_move(vec);
    // Cannot use `vec` after moving it into `foo_move`.
}
```

Because of the its type signature, `foo_move()` takes ownership of its argument, `vec`. After `foo_move(vec)` has been called, `vec` can no longer be used in the body of `main()`.

When an object's ownership is moved up into to a function call (e.g. `foo_move()`, that object will be freed at the end of that function call. So in the example above, the lifetime of the `vec` declared in `main()` actually spans until the end of the body of `foo_move()`. Therefore, `vec` will be dropped at the end of `foo_move()`.

Some Basic Move And Borrow Examples

The best way to get an intuition for the type-checking rules in Rust is by experimenting with the Rust compiler. An easy way to start doing this is with the Rust Playpen. Here is a series of simple examples which may help you get started with how that borrows and moves work in Rust. [4]

(Note that if a definition of either `foo_move()` or `foo_borrow()` is omitted, then it is defined the same as in the examples above.)

Borrow Then Move

```
fn main() {
    let vec = vec![0, 0, 0];
    foo_borrow(&vec);
    foo_move(vec);
}
```

This example *does* type check, because ownership of `vec` has been returned to the `main()` function by the time that `foo_move()` is called. [5]

Move Then Move Again

```
fn main() {
    let vec = vec![0, 0, 0];
    foo_move(vec);
    foo_move(vec);  // Type Error!
}
```

This example *does not* type check, because ownership of `vec` was moved to the first call to `foo_move()`. In fact, `vec` was dropped at the end of that function call. An object cannot be moved again in this way.

Move Then Borrow

```
fn main() {
    let vec = vec![0, 0, 0];
    foo_move(vec);
    foo_borrow(&vec);  // Type Error!
}
```

This example *does not* type check, because ownership of `vec` was moved to `foo_move()` and the `vec` object was dropped at the end of `foo_move()`. A borrow reference cannot be created for a object which has already been dropped.

Borrow Then Borrow Again

```
fn main() {
    let vec = vec![0, 0, 0];
    foo_borrow(&vec);
    foo_borrow(&vec);
}
```

This example *does* type check, because ownership of `vec` was returned to `main()` before attempting to perform another borrow.

Moving When A Borrow Reference Exists

```
fn main() {
    let vec = vec![0, 0, 0];
    let vec_ref = &vec;
    foo_move(vec);  // Type Error!
}
```

This example *does not* type check. Notice that the lifetime of `vec_ref` lasts until the end of `main()`. If we were to move `vec` to `foo_move()`, then `vec` will have been dropped by the end of `foo_move()`. The type system cannot allow this because then a borrow reference would *outlive* the object to which it points. That could be the source of use-after-free bugs.

Borrowing When A Borrow Reference Exists

```
fn main() {
    let vec = vec![0, 0, 0];
    let vec_ref = &vec;
    foo_borrow(vec_ref);
    foo_borrow(&vec);
}
```

This example *does* type check, because there is no problem simultaneously having two immutable borrow references to an object. [6]

Borrow From Move Context

```
fn foo_move(vec: Vec<u32>) {
    foo_borrow(&vec);
}

```

```rust
fn main() {
    let vec = vec![0, 0, 0];
    foo_move(vec);
}
```

This example *does* type check. However, what would happen if we first passed it to a borrowed context which then tried to move it to another context?

Move From Borrow Context

```rust
fn foo_borrow(vec: &Vec<u32>) {
    foo_move(*vec);  // Type Error!
}

fn main() {
    let vec = vec![0, 0, 0];
    foo_borrow(&vec);
}
```

This example *does not* type check, because you cannot give away ownership of some object that has only been borrowed.

An Example Of When To Borrow

Below is a less trivial Rust program. Notice that in each of the functions, the vector arguments are *moved*. Do these moves seem like good design decisions?

```rust
/// Returns a new vector containing the element-wise difference between
/// `left` and `right`. To be clear, this returns a vector `rv` such that
/// for every `i`, `left[i] - right[i] == rv[i]`. Note that the implementation
/// assumes that `left.len() == right.len()`.
fn vec_diff(left: Vec<i32>, right: Vec<i32>) -> Vec<i32> {
    let mut rv = Vec::new();
    for (l, r) in left.iter().zip(right.iter()) {
        rv.push(l - r);
    }
    return rv;
}

/// Returns true if and only if all elements in `vec` are equal to `value`.
fn all(vec: Vec<i32>, value: i32) -> bool {
    for elem in vec.iter() {
        if *elem != value {
```

```
            return false;
        }
    }
    return true;
}

fn main() {
    let v1 = vec![0, 1, 2];
    let v2 = vec![3, 4, 5];
    let diff = vec_diff(v2, v1);
    println!("{:?}", all(diff, 3));
}
```

Looking at the bodies of `vec_diff()` and `all()`, there does not seem to be any reason to have the arguments move. By moving an object to a function, that function is essentially *consuming* the object (i.e., the object cannot be used again). By using the above definitions for `vec_diff()` and `all()`, the following (very natural) modification of `main()` will not type check.

```
fn main() {
    let v1 = vec![0, 1, 2];
    let v2 = vec![3, 4, 5];
    let v3 = vec![6, 7, 8];
    let diff1 = vec_diff(v2, v1);
    let diff2 = vec_diff(v3, v2);  // Type Error!
    println!("{:?}", all(diff1, 3));
    println!("{:?}", all(diff2, 3));
}
```

This code will not type check. `v2` was consumed in the first call to `vec_diff()`, meaning that it cannot be passed to the second call to `vec_diff()`. This unnecessary restriction indicates that the definitions of `vec_diff()` and `all()` should be modified to use borrow references instead.

This demonstrates how it is usually a good idea to use a borrow reference when implementing functions unless there is a good reason to move the object. Moving an object consumes the object, so it should only be moved if necessary.

Endnotes

[1] Rust is by no means the first programming language to have these features. See the Cyclone Programming Language for one influential predecessor.

[2] The actual implementation of `Vec` does not actually use a box pointer for optimization reasons. Note also that the actual implementation of `Vec` uses

mutable backing storage, however, this document does not dive into the very important distinction between mutable and immutable types.

[3] The key to this guarantee is that a `Box` pointer cannot be copied, only moved.

[4] We recommend that you try for yourself those examples which do not type check in order to see the errors thrown by the compiler.

[5] Technically, what is happening here is slightly more subtle than this description suggests. First a borrow reference is created. Next this reference is moved to the `foo_borrow()` function. Then, this reference will be dropped at the end of its lifetime, that is, at the end of the scope of `foo_borrow()`. Finally, the `vec` can be moved into `foo_move()` because there do not exist any other claims on its ownership, i.e. live borrow references to `vec`.

[6] One can make an arbitrary number of borrow references so long they are all *immutable* borrow references. If a borrow reference is mutable, then it must be unique, that is, there can be no other borrow references, either mutable or immutable.