Rust Programming Language Tutorial (Basics)

Written by:
Apriorit Inc.

Author:
Alexey Lozovsky,
Software Designer in System Programming Team

https://www.apriorit.com

info@apriorit.com

Introduction

This Rust Programming Language Tutorial and feature overview is prepared by system programming professionals from the Apriorit team. The Tutorial goes in-depth about main features of the Rust programming language, provides examples of their implementation, and a brief comparative analysis with C++ language in terms of complexity and possibilities.

Rust is a relatively new systems programming language, but it has already gained a lot of loyal fans in the development community. Created as a low-level language, Rust has managed to achieve goals that are usually associated with high-level languages.

Main advantages of Rust are its increased concurrency, safety, and speed, that is achieved due to the absence of a garbage collector, eliminating data races, and zero-cost abstractions. Unlike other popular programming languages, Rust can ensure a minimal runtime and safety checks, while also offering a wide range of libraries and binding with other languages.

This tutorial is divided into sections, with each section covering one of the main features of the Rust language:

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

In addition, we have added a detailed chart comparing feature set of Rust to C++. As a leading language for low-level software development, C++ serves as a great reference point for illustrating advantages and disadvantages of Rust.

This tutorial will be useful for anyone who only starts their journey with Rust, as well as for those who want to gain a more in-depth perspective on Rust features.

Table of Contents

Introduction

Rust is focused on safety, speed, and concurrency. Its design allows you to develop software with great performance by controlling a low-level language using the powerful abstractions of a high-level language. This makes Rust both a safer alternative to languages like C and C++ and a faster alternative to languages like Python and Ruby.

The majority of safety checks and memory management decisions are performed by the Rust compiler so the program's runtime performance isn't slowed down by them. This makes Rust a great choice for use cases where more secure languages like Java aren't good:

- Programs with predictable resource requirements
- Embedded software
- Low-level code like device drivers

Rust can be used for web applications as well as for backend operations due to the many libraries that are available through the Cargo package registry.

Summary of Features

Before describing the features of Rust, we'd like to mention some issues that the language successfully manages.

Issue	Rust's Solution
Preferring code duplication to abstraction due to high cost of virtual method calls	Zero-cost abstraction mechanisms
Use-after-free, double-free bugs, dangling pointers	Smart pointers and references avoid these issues by design Compile-time restrictions on raw pointer usage
Null dereference errors	Optional types as a safe alternative to nullable pointers
Buffer overflow errors	Range checks performed at runtime Checks are avoided where the compiler can prove they're unnecessary
Data races	Built-in static analysis detects and prevents possible data races at compilation time
Uninitialized variables	Compiler requires all variables to be initialized before first use All types have defined default values
Legacy design of utility types heavily used by the standard library	Built-in, composable, structured types: tuples, structures, enumerations Pattern matching allows convenient use of structured types The standard library fully embraces available pattern matching to provide easy-to-use interfaces

Embedded and bare-metal programming place high restrictions on runtime environment	Minimal runtime size (which can be reduced even further) Absence of built-in garbage collector, thread scheduler, or virtual machine
Using existing libraries written in C and other languages	Only header declarations are needed to call C functions from Rust, or vice versa No overhead in calling C functions from Rust or calling Rust functions from C

Now let's look more closely at the features provided by the Rust programming language and see how they're useful for developing system software.

Rust Language Features

In the first part of this Rust language programming tutorial, we'll describe such two key features as zero-cost abstractions and move semantics.

Zero-Cost Abstractions

Zero-cost (or zero-overhead) abstractions are one of the most important features explored by C++. Bjarne Stroustrup, the creator of C++, describes them as follows:

> *"What you don't use, you don't pay for."* And further: *"What you do use, you couldn't hand code any better."*

Abstraction is a great tool used by Rust developers to deal with complex code. Generally, abstraction comes with runtime costs because

abstracted code is less efficient than specific code. However, with clever language design and compiler optimizations, some abstractions can be made to have effectively zero runtime cost. The usual sources of these optimizations are static polymorphism (templates) and aggressive inlining, both of which Rust embraces fully.

Iterators are an example of commonly used (and thus heavily optimized) abstractions that they decouple algorithms for sequences of values from the concrete containers holding those values. Rust iterators provide many built-in *combinators* for manipulating data sequences, enabling concise expressions of a programmer's intent. Consider the following code:

```
// Here we have two sequences of data. These could be stored in vectors
// or linked lists or whatever. Here we have _slices_ (references to
// arrays):
let data1 = &[3, 1, 4, 1, 5, 9, 2, 6];
let data2 = &[2, 7, 1, 8, 2, 8, 1, 8];

// Let's compute some valuable results from them!
let numbers =
    // By iterating over the first array:
    data1.iter()              // {3,      1,    4,    ...}
    // Then zipping this iterator with an iterator over another array,
    // resulting in an iterator over pairs of numbers:
    .zip(data2.iter())        // {(3, 2), (1, 7), (4, 1), ...}
    // After that we map each pair into the product of its elements
    // via a lambda function and get an iterator over products:
    .map(|(a, b)| a * b)      // {6,      7,    4,    ...}
    // Given that, we filter some of the results with a predicate:
    .filter(|n| *n > 5)       // {6,      7,                ...}
    // And take no more than 4 of the entire sequence which is produced
    // by the iterator constructed to this point:
    .take(4)
    // Finally, we collect the results into a vector. This is
    // the point where the iteration is actually performed:
    .collect::<Vec<_>>();

// And here is what we can see if we print out the resulting vector:
println!("{:?}", numbers);  // ===> [6, 7, 8, 10]
```

Combinators use high-level concepts such as closures and lambda functions that have significant costs if compiled natively. However, due to optimizations powered by LLVM, this code compiles as efficiently as the explicit hand-coded version shown here:

```
use std::cmp::min;

let mut numbers = Vec::new();

for i in 0..min(data1.len(), data2.len()) {
    let n = data1[i] * data2[i];

    if n > 5 {
        numbers.push(n);
    }
    if numbers.len() == 4 {
        break;
    }
}
```

While this version is more explicit in what it does, the code using combinators is easier to understand and maintain. Switching the type of container where values are collected requires changes in only one line with combinators versus three in the expanded version. Adding new conditions and transformations is also less error-prone.

Iterators are Rust examples of “couldn’t hand code better” parts. *Smart pointers* are an example of the “don’t pay for what you don’t use” approach in Rust.

The C++ standard library has a ***shared_ptr*** template class that’s used to express shared ownership of an object. Internally, it uses reference

counting to keep track of an object's lifetime. An object is destroyed when its last **shared_ptr** is destroyed and the count drops to zero.

Note that objects may be shared between threads, so we need to avoid data races in reference count updates. One thread must not destroy an object while it's still in use by another thread. And two threads must not concurrently destroy the same object. Thread safety can be ensured by using *atomic operations* to update the reference counter.

However, some objects (e.g. tree nodes) may need shared ownership but may not need to be shared between threads. Atomic operations are unnecessary overhead in this case. It may be possible to implement some ***non_atomic_shared_ptr*** class, but accidentally sharing it between threads (for example, as part of some larger data structure) can lead to hard-to-track bugs. Therefore, the designers of the Standard Template Library chose not to provide a single-threaded option.

On the other hand, Rust *is* able to distinguish these use cases safely and provides two reference-counted wrappers: ***Rc*** for single-threaded use and ***Arc*** with an atomic counter. The cherry on top is the ability of the Rust compiler to ensure at compilation time that Rcs are never shared between threads (more on this later). Therefore, it's not possible to accidentally share data that isn't meant to be shared and we can be freed from the unnecessary overhead of atomic operations.

Move Semantics

C++11 has brought *move semantics* into the language. This is a source of countless optimizations and safety improvements in libraries and

programs by avoiding unnecessary copying of temporary values, enabling safe storage of non-copyable objects like mutexes in containers, and more.

Rust recognizes the success of move semantics and embraces them by default. That is, all values are in fact moved when they're assigned to a different variable:

```
let foo = Foo::new();
let bar = foo;          // the Foo is now in bar
```

The punchline here is that after the move, you generally can't use the previous location of the value (*foo* in our case) because no value remains there. But C++ doesn't make this an error. Instead, it declares foo to have an *unspecified value* (defined by the move constructor). In some cases, you can still safely use the variable (like with primitive types). In other cases, you shouldn't (like with mutexes).

Some compilers may issue a diagnostic warning if you do something wrong. But the standard doesn't require C++ compilers to do so, as use-after-move may be perfectly safe. Or it may not be and might instead lead to an *undefined behavior*. It's the programmer's responsibility to know when use-after-move breaks and to avoid writing programs that break.

On the other hand, Rust has a more advanced type system and it's *a compilation error* to use a value after it has been moved, no matter how complex the control flow or data structure:

```
error[E0382]: use of moved value: `foo`
  --> src/main.rs:13:1
   |
11 | let bar = foo;
   |  --- value moved here
12 |
13 | foo.some_method();
   | ^^^ value used here after move
   |
```

Thus, use-after-move errors aren't possible in Rust.

In fact, the Rust type system allows programmers to safely encode more use cases than they can with C++. Consider converting between various value representations. Let's say you have a string in UTF-8 and you want to convert it to a corresponding vector of bytes for further processing. You don't need the original string afterwards. In C++, the only safe option is to copy the whole string using the vector copy constructor:

```
std::string string = "Hello, world!";
std::vector<uint8_t> bytes(string.begin(), string.end());
```

However, Rust allows you to move the internal buffer of the string into a new vector, making the conversion efficient and disallowing use of the original string afterwards:

```
let string = String::from_str("Hello, world!");
let bytes = string.into_bytes();          // string may not be used now
```

Now, you may think that it's dumb to move *all* values by default. For example, when doing arithmetic we expect that we can reuse the results of intermediate calculations and that an individual constant may be used more than once in the program. Rust makes it possible to copy a value implicitly when it's assigned to a new variable, based on its type. Numbers

are an example of such copyable type, and any user-defined type can also be marked as copyable with the *#[derive(Copy)]* attribute.

Guaranteed Memory Safety

Memory safety is the most prized and advertised feature of Rust. In short, Rust guarantees the absence (or at least the detectability) of various memory-related issues:

- segmentation faults
- use-after-free and double-free bugs
- dangling pointers
- null dereferences
- unsafe concurrent modification
- buffer overflows

These issues are declared as *undefined behaviors* in C++, but programmers are mostly on their own to avoid them. On the other hand, in Rust, memory-related issues are either immediately reported as compile-time errors or if not then safety is enforced with runtime checks.

Ownership

The core innovation of Rust is *ownership and borrowing*, closely related to the notion of *object lifetime*. Every object has a lifetime: the time span in which the object is available for the program to use. There's also an owner for each object: an entity which is responsible for ending the lifetime of the object. For example, local variables are owned by the function scope. The variable (and the object it owns) dies when execution leaves the scope.

```
1   fn f() {
2     let v = Foo::new();     // ----+ v's lifetime
3                             //     |
4     /* some code */          //     |
5   }                         // <---+
```

In this case, the object **Foo** is owned by the variable **v** and will die at line 5, when function *f()* returns.

Ownership can be *transferred* by moving the object (which is performed by default when the variable is assigned or used):

```
1  fn f() {
2     let v = Foo::new();     // ----+ v's lifetime
3     {                       //     |
4           let u = v;        // <---X---+ u's lifetime
5                             //         |
6           do_something(u);  // <-------X
7     }                       //
8   }                         //
```

Initially, the variable **v** would be alive for lines 2 through 7, but its lifetime ends at line 4 where **v** is assigned to **u**. At that point we can't use **v** anymore (or a compiler error will occur). But the object **Foo** isn't dead yet; it merely has a new owner **u** that is alive for lines 4 through 6. However, at line 6 the ownership of **Foo** is transferred to the function *do_something()*. That function will destroy **Foo** as soon as it returns.

Borrowing

But what if you don't want to transfer ownership to the function? Then you need to use references to pass a pointer to an object instead:

```
1   fn f() {
2     let v = Foo::new();     // ---+ v's lifetime
3                             //    |
4     do_something(&v);       // :--|----.
5                             //    |     } v's borrowed
6     do_something_else(&v);  // :--|----'
```

```
7   }                          // <--+
```

In this case, the function is said to *borrow* the object **Foo** via references. It can access the object, but the function doesn't own it (i.e. it can't destroy it). References are objects themselves, with lifetimes of their own. In the example above, a separate reference to **v** is created for each function call, and that reference is transferred to and owned by the function call, similar to the variable **u** above.

It's expected that a reference will be alive for at least as long as the object it refers to. This notion is implicit in C++ references, but Rust makes it an explicit part of the reference type:

```
fn do_something<'a>(v: &'a Foo) {
    // something with v
}
```

The argument **v** is in fact a reference to Foo with the lifetime **'a**, where **'a** is defined by the function *do_something()* as the duration of its call.

C++ can handle simple cases like this just as well. But what if we want to return a reference? What lifetime should the reference have? Obviously, not longer than the object it refers to. However, since lifetimes aren't part of C++ reference types, the following code is syntactically correct for C++ and will compile just fine:

```
const Foo& some_call(const Foo& v)
{
    Foo w;

    /* 10 lines of complex code using v and w */

    return w; // accidentally returns w instead of v
}
```

Though this code is syntactically correct, however, it is *semantically* incorrect and has undefined behavior if the caller of *some_call()* actually uses the returned reference. Such errors may be hard to spot in casual code review and generally require an external static code analyzer to detect.

Consider the equivalent code in Rust:

```
fn some_call(v: &Foo) -> &Foo {  // ------------------+ expected
    let w = Foo::new();          // ---+ w's lifetime | lifetime
                                 //    |              | of the
    return &w;                   // <--+              | returned
}                                //                   | value
                                 // <-----------------+
```

The returned reference is expected to have the same lifetime as the argument **v**, which is expected to live longer than the function call. However, the variable **w** lives only for the duration of *some_call()*, so references to it can't live longer than that. The *borrow checker* detects this conflict and complains with a compilation error instead of letting the issue go unnoticed.

```
error[E0597]: `w` does not live long enough
  --> src/main.rs:10:13
   |
10 |  return &w;
   |             ^ does not live long enough
11 | }
   | - borrowed value only lives until here
   |
```

The compiler is able to detect this error because it tracks lifetimes explicitly and thus knows exactly how long values must live for the references to still be valid and safe to use. It's also worth noting that you

don’t have to explicitly spell out all lifetimes for all references. In many cases, like in the example above, the compiler is able to automatically infer lifetimes, freeing the programmer from the burden of manual specification.

Mutability and Aliasing

Another feature of the Rust borrow checker is *alias analysis*, which prevents unsafe memory modification. Two pointers (or references) are said to *alias* if they point to the same object. Let’s look at the following Rust example:

```
Foo c;
Foo *a = &c;
const Foo *b = &c;
```

Here, pointers **a** and **b** are aliases of the **Foo** object owned by **c**. Modifications performed via **a** will be visible when **b** is dereferenced. Usually, aliasing doesn’t cause errors, but there are some cases where it might.

Consider the *memcpy()* function. It can and is used for copying data, but it’s known to be unsafe and can cause memory corruption when applied to overlapping regions:

```
char array[5] = { 1, 2, 3, 4, 5 };
const char *a = &array[0];
      char *b = &array[2];
memcpy(a, b, 3);
```

In the sample above, the first three elements are now undefined because their values depend on the order in which *memcpy()* performs the copying:

```
{ 3, 4, 5, 4, 5 }       // if the elements are copied from left to right
{ 5, 5, 5, 4, 5 }       // if the elements are copied from right to left
```

The ultimate issue here is that the program contains two aliasing references to the same object (the array), one of which is non-constant. If such programs were syntactically incorrect then *memcpy()* (and any other function with pointer arguments as well) would always be safe to use.

Rust makes it possible by enforcing the following *rules of borrowing*:

1. At any given time, you can have *either* but not both of:
    - one mutable reference
    - any number of immutable references
2. References must always be valid.

The second rule relates to ownership, which was discussed in the previous section. The first rule is the real novelty of Rust.

It's obviously safe to have multiple aliasing pointers to the same object *if* none of them can be used to modify the object (i.e. they are constant references). If there are two mutable references, however, then modifications can conflict with each other. Also, if there is a const-reference **A** and a mutable reference **B**, then presumably the constant object as seen via **A** can in fact change if modifications are made via **B**. But it's perfectly safe if only one mutable reference to the object is allowed to exist in the program. The Rust borrow checker enforces these rules during compilation, effectively making each reference act as a read-write lock for the object.

The following is the equivalent of *memcpy()* as shown above:

```
let mut array = [1, 2, 3, 4, 5];
let a = &mut array[0..2];
let b = &   array[2..4];
a.copy_from_slice(b);
```