Experiences with the

Rust Programming Language

Jeff East

# Introduction

Rust is a type-safe compiled systems programming language available on both Windows and Linux platforms. I set out to learn Rust as an exercise; this document summarizes my experiences and opinions after about six intensive months of use.

From what I’ve seen so far, Rust is targeted towards lower-level, compute-intensive high performance programs, rather than higher level user-interface oriented programs. The language, runtime, and toolset all focus more on libraries and “console” level programs than on quick-to-develop GUI tasks. That said, there are GUI libraries available for Rust, and my next task may be to explore them.

# Learning Aids & Community

The primary Rust resource is <https://www.rust-lang.org>. From here, you can find links to documentation, installation kits, community forums, the governance board and more. A great deal of effort has expended to make it easy to get started, both in terms of installation and a great book for the beginner: <https://doc.rust-lang.org/book/>. There’s also reference material for the standard library: [std - Rust (rust-lang.org)](https://doc.rust-lang.org/std/index.html).

Stack Overflow ([Stack Overflow - Where Developers Learn, Share, & Build Careers](https://stackoverflow.com/)) has a great deal of how-to discussion on various Rust topics, similar to those you find for .Net.

Bing’s AI is surprisingly accurate at providing usable code examples when you post a question to the browser.

Developers will find these resources combine to answer most questions. They can also go directly to the forums to ask new questions. The entire experience is similar to that of .Net.

# Language

Rust as a language is somewhat similar to C++ or C# -- it has a module hierarchy model, functions, named variables, data structures and enumerations, interfaces (traits), if/else, for loops, while loops, closures (similar to C# lambda expressions), a macro facility, etc. It differs from both C++ and C# in that it does not support classes, methods, inheritance or exceptions.

## Data Types

Rust has three fundamental data abstractions:

* Atoms

Atoms are fundamental types. Examples are integer (e.g., i32, i64, u32, u64), floating point (e.e., f32, f64), Boolean and character. Note that strings are not atomic – they are vectors of Unicode characters. Since Unicode characters vary in length, you cannot access the characters in a string by indexing. While this is great to encourage internationalizable code, it prohibits one of the hacks commonly used in quick one-off programming!

* Structures  
    
  Structures are very similar to C++ structures. Although Rust does not have a class model, structures can have implementations, which associate functions with the namespace of the structure. Including a parameter of type “&self” makes such functions look syntactically similar to C++ methods.   
    
  Traits are a Rust feature similar to .Net interfaces – a trait names a set of functions which any structure implementing the trait must implement. Thus it is possible to abstract a parameter to a function as implementing a trait, which allows variable different structures to be passed as an actual parameter, as long as each structure implements the trait.
* Enumerations  
    
  Rust implements a superset of the enum model in C++ and C# -- in Rust, enumerated values can include arguments. This is a very powerful feature and is widely used. An example of such an enumeration is:

enum TokenType {  
 EOS,  
 Int(i32),  
 Real(f32),  
 String(String),  
}

This example defines four enumerated values for the TokenType enumeration – one (EOS) has no value associated with it, but each of the other three has an associated value. For example, you cannot create an instance of TokenType::Int without also providing an integer (i32) value.

Rust includes the several grouping constructs:

* Tuples – similar to a structure, but the members are not named, instead they are accessed by indexing
* Arrays – arrays are classical vectors, where each element has the same datatype

A novel concept in Rust is that of a slice – a contiguous range of elements within a datatype. Slices are identified by their starting/ending indices together with a reference to the host datatype. They provide an efficient mechanism for passing data by reference.

## Flow Control

Rust has variants on common flow control mechanisms:

* If/Else  
    
  Rust’s if statement is identical in power to those of C++ and C#, although you do not include the condition in parentheses (and will get a warning if you do!).
* Looping  
    
  The looping constructs are those you would expect, including continue and break:  
  + While
  + Loop
  + For  
      
    The for statement in Rust is closer to what is in C# than C++ -- for example, it can sequence through members of a collection by running an enumerator.
* Match & If let  
    
  These statements are generalizations of the switch statement in C++ -- you specify a value (or tuple of values) and a set of patterns to be tested against. When a pattern matches, it’s corresponding statement is run. Patterns are restricted to compile time expressions.  
  Patterns can be used to pull the associated value out of an enumerated value. For example,

match token.token\_type {  
 EOS => return,  
 Int(int\_value) => numeric\_value += int\_value;  
 \_ => {},

}

The first pattern is simple – there is no associated value, just the return statement is executed.

The second pattern shows an associated value (int\_value) extracted from the enumerated value.

The final pattern is a wild-card, and matches any TokenType. The braces indicate there is no action to perform.

The match statement is one of the pleasures of using Rust – it’s a very useful tool, and necessary when using enumerated values.

## Exceptions and Generics

Rust has full support for generic structures and parameters, very similar to those in C++ and C#. However, Rust does not support try/catch/finally blocks, as do C++ and C#. Instead, it makes use of a generic Result<S,E> structure (S – success value, E – failure value) together with the ? operator. The ? operator can be used to replace:

let v = {

let r = function\_call(…);  
 if r.is\_err() {   
 return r;  
 } else {

r.unwrap();  
 }

with

let v = function\_call(…)?;

This makes the code a lot easier to read, and since the Rust compiler complains if a result is not used, makes it easy to write functions that terminate on failure and propagate the error back to the caller.

## Null

One of the interesting absences in the language is that of a null pointer – there is no null value. This was done to eliminate a common programming error (traversing a null pointer, resulting in an access violation). However, it requires a significant mind-shift if you are used to programming with nulls!

Null is replaced by another generic type: Option<T>. Option is an enumeration with two values:

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

For example, Option<T>s can be used to build a singly-linked list:

struct Element<T> {  
 pub next : Option<T>,  
 …  
}

Option<T> is frequently used in conjunction with match and if-let to test for the presence of a value and unwrap the value from the enumeration:

if let Some(v) = expression-yielding-option<T> { … } else { … }

## Mutability

Mutability refers to the ability for a variable to be assigned a new value after it has been created. By default, variables are read-only. You explicitly tag those that can be modified. Modifiable parameters passed to a function are tagged at both ends (caller and function declaration).

## Ownership

A central tenet of Rust is eliminating race conditions resulting from “unexpected” changes to shared data structures. An example of such a change is invoking a function with a modifiable passed-by-reference argument, and having that function pass the argument to a third-party function, which modifies the argument in such a way that invalidates state kept by the original caller.

This is a common programming error, and frequently is the result of code maintenance by people not familiar with undocumented assumptions made by the initial author.

To prevent such race conditions, Rust introduces and enforces several concepts which limit how data is shared:

1. Data is by default read-only, and must be explicitly be tagged as modifiable.
2. The concept of ownership is used to restrict when a datum can change and who can keep copies.
   1. Data always has exactly one owner.
   2. It is deleted when it has no owner (for example, moves out of scope).
   3. It can only be modified by the owner.
   4. It can be borrowed – any number of readers can borrow a datum, but only if there is no one who can modify it. There can only be one reference to modifiable data at a time. This includes results derived from modifiable data – this is a key restriction that differs from other languages.

These restrictions are implemented and enforced by the compiler – you simply cannot pass multiple mutable (changeable) references to a datum to multiple writers.

Rule 1 is demonstrated by the following program:

fn main() {  
    let my\_string = String::from("Hi there");  
    my\_string = my\_string + "\n";  
    println!("{my\_string}");  
}

Compiling this generates the following diagnostics:

error[E0384]: cannot assign twice to immutable variable `my\_string`  
 --> src\main.rs:3:5  
 |  
2 | let my\_string = String::from("Hi there");  
 | ---------

| |

| first assignment to `my\_string`

| help: consider making this binding mutable: `mut my\_string`

3 | my\_string = my\_string + "\n";

| ^^^^^^^^^ cannot assign twice to immutable variable

We forgot to mark the variable as changeable. Adding the “mut” keyword to the declaration clears up the error.

fn main() {

    let mut my\_string = String::from("Hi there");

    my\_string = my\_string + "\n";

    println!("{my\_string}");

}

Rule 2 is demonstrated by:

fn main() {  
    let mut my\_string = String::from("Hi there");  
    let my\_string\_ptr = &my\_string;  
    my\_string += "\n";  
    println!("{my\_string\_ptr}");  
}

which yields:

error[E0502]: cannot borrow `my\_string` as mutable because it is also borrowed as immutable

--> src\main.rs:4:5

|

3 | let my\_string\_ptr = &my\_string;

| ---------- immutable borrow occurs here

4 | my\_string += "\n";

| ^^^^^^^^^^^^^^^^^ mutable borrow occurs here

5 | println!("{my\_string\_ptr}");

| --------------- immutable borrow later used here

Ownership can be passed from one variable to another:

fn main() {  
    let my\_string = String::from("Hi there");  
    let another\_string = my\_string;  
    println!("{my\_string}");  
}

yielding:

error[E0382]: borrow of moved value: `my\_string`

--> src\main.rs:4:15

|

2 | let my\_string = String::from("Hi there");

| --------- move occurs because `my\_string` has type `String`, which does not implement the `Copy` trait

3 | let another\_string = my\_string;

| --------- value moved here

4 | println!("{my\_string}");

| ^^^^^^^^^^^ value borrowed here after move

|

= note: this error originates in the macro `$crate::format\_args\_nl` which comes from the expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)

help: consider cloning the value if the performance cost is acceptable

|

3 | let another\_string = my\_string.clone();

| ++++++++

Ownership is one of the fundamental Rust concepts that differs from other languages, and consequently is one that makes learning Rust difficult for experienced programmers.

## References

Rust passes parameters by value unless explicitly told to pass them by reference. The compiler is perfectly happy to pass gigantic values between functions without a hint of the overhead involved. Calling a function with a by-value parameter passes ownership of that value to the called function:

fn main() {  
    let my\_string = String::from("Hi there");  
    func(my\_string);  
    println!("{my\_string}");  
}  
  
fn func(\_arg: String) {  
}

yields:

error[E0382]: borrow of moved value: `my\_string`

--> src\main.rs:4:15

|

2 | let my\_string = String::from("Hi there");

| --------- move occurs because `my\_string` has type `String`, which does not implement the `Copy` trait

3 | func(my\_string);

| --------- value moved here

4 | println!("{my\_string}");

| ^^^^^^^^^^^ value borrowed here after move

|

note: consider changing this parameter type in function `func` to borrow instead if owning the value isn't necessary

--> src\main.rs:7:15

|

7 | fn func(\_arg: String) {

| ---- ^^^^^^ this parameter takes ownership of the value

| |

| in this function

= note: this error originates in the macro `$crate::format\_args\_nl` which comes from the expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)

help: consider cloning the value if the performance cost is acceptable

|

3 | func(my\_string.clone());

| ++++++++

The compiler diagnostic is suggesting a change to the function signature to pass the parameter by reference (“borrowing” the value, instead of “moving” the value):

fn main() {  
    let my\_string = String::from("Hi there");  
    func(&my\_string);  
    println!("{my\_string}");  
}  
  
fn func(\_arg: &String) {  
}

## Lifetimes

The compiler needs additional information when a member of a structure is a reference to another datum:

struct MyStruct {  
    member: &String,  
}

It wants to know how long the referenced datum will live before it is deallocated:

error[E0106]: missing lifetime specifier

--> src\main.rs:2:13

|

2 | member: &String,

| ^ expected named lifetime parameter

|

help: consider introducing a named lifetime parameter

|

1 ~ struct MyStruct<'a> {

2 ~ member: &'a String,

|

The notation ‘a is used to denote a lifetime. Notice it is passed as a generic parameter:

struct MyStruct<'a> {  
    member: &'a String,  
}

Instances of MyStruct will need to specify a value for this parameter. Depending on its usage it might be forever (‘\_), static (‘static) or an explicit parameter to the function.

The notion of lifetimes is very important, and is central to avoiding the pitfall of referencing deallocated structures. Stale pointers are a very common problem in C++. C# uses garbage collection, so stale pointers merely (!) result in multiple copies of data, perhaps leading to cohesion problems.

Nevertheless, in my opinion, lifetimes are a poorly thought out feature of the language. Once you introduce a generic lifetime to a structure, it must be propagated throughout the code wherever the structure is used. This can result in massive code changes, in part because structures included in other structures force the outer structure to also require a generic lifetime! Since slices include a reference, this reduces the utility of slices for efficiently referencing data, frequently resulting in more cloning of data to pass it by value and avoid the lifetime trap. This yields inefficient programs.

It cannot be argued that stale pointers are not a frequent source of programming errors. But the inability to limit the propagation of lifetimes throughout containing structures makes the Rust implementation clunky, in my opinion. I’ve read opinions from experienced Rust programmers in Stack Overflow suggesting that including lifetime parameters in structures should be avoided.

## Smart Pointers

The combination of these rules have caused the Rust community to introduce alternatives to references.

* Coarse grained structure mutability
* Single ownership
* Reference lifetimes

The alternatives are called smart pointers, and there are two fundamental smart pointers, each of which has a single-threaded and multi-threaded implementation:

* Ref-counted pointer.
* Mutable members, enforcing exclusive access at runtime

The Rc<T> and Arc<T> types implement ref-counts on contents, deleting them (and themself) when the count goes to zero. Very similar to the old Com/OLE reference counts. They are made palatable by using destructors to automatically manage the ref-count.

RefCell<T> and Mutex<T> are used to mark individual members of a structure as mutable. This allows the programmer to pass immutable references to the structure, yet still modify individual members.

There are single-thread and multi-thread implementations of each of these. The single-thread implementations (Rc<T> and RefCell<T>) are fast, but the multi-thread implementations (Arc<T>, Mutex<T>) suffer from the fundamental performance pitfalls of multi-processor synchronization primitives. The compiler includes thread-local storage features, so it is possible to write efficient multi-threaded code, but non-judicious usage of the multi-threaded smart pointers will result in inefficient code. If you can implement your program without using smart pointers, your code should be very efficient. But it’s hard to do for complex data structures and may require inefficient choices (for example, using indexing instead of references).

In my opinion, these are practical, inelegant solutions to untenable language concepts enforced by the compiler. That is not to say that I have a better idea – merely to state that the language features introduced to promote safe programming habits have been found to be too restrictive in some real-life, complex programs.

## Example Parse Tree

Going through the process to implement a simple parse tree shows some of the differences developers will see between Rust and other languages.

Consider a simple parse tree, consisting of operation nodes. Each node has zero, one or two operands. The figure below shows a simple parse tree diagram for the expression A\*(1+B):



Our first thought is:

enum Operation {  
    Add,  
    Multiply,  
    Value(i32),  
    Variable(String),  
}  
struct Node {  
    op: Operation,  
    right: Node,  
    left: Node,  
}

But the compiler tells us:

error[E0072]: recursive type `Node` has infinite size

--> src\main.rs:5:1

|

5 | struct Node {

| ^^^^^^^^^^^

6 | op: Operation,

7 | right: Node,

| ---- recursive without indirection

|

help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle

|

7 | right: Box<Node>,

| ++++ +

This makes sense, since not all nodes have two children. We might be tempted to drop the Node structure altogether and try:

enum Operation {  
    Add(Operation,Operation),  
    Multiply(Operation, Operation),  
    Value(i32),  
    Variable(String),  
}

fn main() {  
    let tree = Operation::Multiply(Operation::Variable(String::from("A")), Operation::Add(Operation::Value(1), Operation::Variable(String::from("B"))));  
    func(&tree);  
}

fn func(tree: &Operation) {  
}

But we get the sample complaint from the compiler. So instead we try using references to Node:

enum Operation {

    Add,

    Multiply,

    Value(i32),

    Variable(String),

}

struct Node<'a> {

    op: Operation,

    right: &'a Node<'a>,

    left: &'a Node<'a>,

}

The compiler accepts this, but when we go to use it, we realize there is no such thing as a null reference, so we change it to:

enum Operation {  
    Add,  
    Multiply,  
    Value(i32),  
    Variable(String),  
}  
struct Node<'a> {  
    op: Operation,  
    right: Option<&'a Node<'a>>,  
    left: Option<&'a Node<'a>>,  
}

But when we go to use it, we discover a problem:

fn main() {  
    let tree = Node{ op: Operation::Multiply,   
        left: Some(&Node { op: Operation::Variable(String::from("A")), left: None, right: None }),  
        right: Some(&Node { op: Operation::Add,  
            left: Some(&Node { op: Operation::Value(1), left: None, right: None}),   
            right: Some(&Node { op: Operation::Variable(String::from("B")), left: None, right: None }),  
        })};  
    func(&tree);  
}

fn func(tree: &Node) {  
}

When the compiler complains:

error[E0716]: temporary value dropped while borrowed

--> src\main.rs:15:21

|

15 | left: Some(&Node { op: Operation::Variable(String::from("A")), left: None, right: None }),

| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ creates a temporary value which is freed while still in use

...

19 | })};

| - temporary value is freed at the end of this statement

20 | func(&tree);

| ----- borrow later used here

|

help: consider using a `let` binding to create a longer lived value

|

14 ~ let binding = Node { op: Operation::Variable(String::from("A")), left: None, right: None };

15 ~ let tree = Node{ op: Operation::Multiply,

16 ~ left: Some(&binding),

|

This can be fixed by following the compiler’s suggestion:

fn main() {  
    let one\_node = Node { op: Operation::Value(1), left: None, right: None};  
    let b\_node = Node { op: Operation::Variable(String::from("B")), left: None, right: None };  
    let plus\_node = Node { op: Operation::Add, left: Some(&one\_node), right: Some(&b\_node),};  
    let a\_node = Node { op: Operation::Variable(String::from("A")), left: None, right: None };  
    let tree = Node{ op: Operation::Multiply, left: Some(&a\_node), right: Some(&plus\_node)};  
    func(&tree);  
}

So a solution is to independently create the nodes before we reference them. The requirement being that the lifetimes of the nodes match or exceed the lifetimes of the references to them. This makes intuitive sense.

An alternative solution could be to encapsulate the Node structure in a smart pointer (Rc<Node>). This removes the references and explicit lifetime parameter:

struct Node {  
    op: Operation,  
    right: Option<Rc<Node>>,  
    left: Option<Rc<Node>>,  
}

fn main() {  
    let one\_node = Rc::new(Node { op: Operation::Value(1), left: None, right: None});  
    let b\_node = Rc::new(Node { op: Operation::Variable(String::from("B")), left: None, right: None });  
    let plus\_node = Rc::new(Node { op: Operation::Add, left: Some(one\_node.clone()), right: Some(b\_node.clone()),});  
    let a\_node = Rc::new(Node { op: Operation::Variable(String::from("A")), left: None, right: None });  
    let tree = Rc::new(Node{ op: Operation::Multiply, left: Some(a\_node.clone()), right: Some(plus\_node.clone())});  
    func(&tree);  
}

It is worth noting that the lifetimes of temporaries created by the compiler impact when the developer can use multiple statements (for example, using let to assign an intermediate or common result to a variable) versus one statement (temporaries are discarded at the termination of the statement they are created in). I haven’t noticed this when writing C++ or C#. Usually it’s merely a question of programming style and readability. In Rust, the lifetime rules have more control over this choice.

# Runtime Libraries

Rust includes a comprehensive set of standard runtime libraries, very similar in functionality to those in .Net. There are libraries to support file I/O, complex data structures, interprocess communication, and so on.

There is also a formal community-based set of open-source libraries, similar to Visual Studio’s Nuget. These provide a wide range of capabilities with generous licensing.

# Tool Set

## Compiler

The rust compiler is very good. It includes excellent diagnostics, including suggested solutions to what the community must have found to be common mistakes. The diagnostics are the best I have encountered in any compiler. I have not found any bugs in the compiler nor have I had it bugcheck.

The generated code seems to be adequate. I have not done much with release optimized Rust code.

## Cargo

Cargo is a general command facility which automates many common tasks in building a Rust application.

## Visual Studio Code

I used Visual Studio Code to build and debug my application. I found it to be a barely adequate experience. As an editor, it handles small programs well, but as the number of modules increases, it does little to make it easy to manage. There is no macro facility, and the editing primitives are simple.

As a debugger, Visual Studio Code still has a long way to go to match Visual Studio itself. The primary deficiencies include:

* Inability to display enumerated types that include interior values or generic structures  
    
  This is huge, because almost all the code uses Option<T> (enumeration) and Result<S,E> (generic). This means you use a lot of print statements to see what is going on.
* Inefficient stepping. You frequently have to step multiple times on a line.  
    
  This is probably related to how the compiler is generating debug information, confusing the debugger into thinking what looks like a single statement is actually multiple statements.
* No way to break on functions returning an error  
    
  This isn’t really a Visual Studio Code problem, it is a result of not having the equivalent of “break on exception” that you get with .Net. Nevertheless, it makes it tedious to track down failures.
* No integration with fmt::Debug  
    
  Rust includes a couple of traits (interfaces) which it uses to format data. One is fmt::Display, used to format data for end-users, and one is fmt::Debug, used to format data used for programmers. Visual Studio Code could make use of these to display structures.

I would like to see full Rust support in Visual Studio itself.

# Summary

Rust is a quality environment for developing system-level programs. The development experience is not as mature as for other languages on Windows (e.g., C++, C#). This may adversely impact developer productivity. Like any open-source community driven system, you have to work around issues in the near term, hoping for improvements in future releases.

The language imposes a design and implementation style unlike that of C++ and C#. It’s reminiscent of APL – you approached problems in APL differently than in other languages, and the same is true of Rust. I think experienced programmers could find Rust frustrating, because common design patterns they’ve been trained to use are not necessarily usable in Rust. You have to approach the design from a Rust perspective. That said, Rust may be a good environment for new systems-level programmers, as many of the design patterns they learn can be transported to other languages.