Rust Programming Language

Jeff East

Introduction

Rust is an open source, 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 website 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 been expended to make it easy to get started, both in terms of installation (Getting started - Rust Programming Language (rust-lang.org)) and a great book for the beginner (The Rust Programming Language - The Rust Programming Language (rust-lang.org)). There's also reference material for the standard library (std - Rust (rust-lang.org)).

Stack Overflow (Stack Overflow - Where Developers Learn, Share, & Build Careers) has a great deal of how-to discussion on various Rust topics, like 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 can combine these resources to answer most questions. They can also go directly to the forums to ask new questions. The entire experience is like that of .Net.

Language

Rust as a language is somewhat like C++ or C# -- it has a module hierarchy model, functions, named variables, data structures and enumerations, interfaces (called traits), flow control statements, closures (similar to C# lambda expressions), a macro facility, etc. It differs from both C++ and C# in that it does not support class hierarchies or exceptions. Unlike C#, Rust is not garbage collected – structures are explicitly allocated and immediately freed when no longer referenced (for example, a variable moves out of scope).

Data Types

Rust has three fundamental data abstractions:

- Atoms

Atoms are fundamental types. Examples are integer (e.g., i32, i64, u32, u64, usize), floating point (e.g., f32, f64), Boolean and character. Note that strings are not atomic – they are vectors of Unicode characters. Since Unicode characters vary in length, Rust does not allow string characters to be accessed through 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.

- 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 include an integer argument value when instantiating an instance of TokenType::Int(i32).

Rust includes several grouping constructs:

- Tuples – like structures, but the members are not named, instead they are accessed by ordinal position.
- Arrays – arrays are classical vectors, where each element shares the same datatype.

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

Functions, Methods, and Traits

Structures and enumerations can optionally be implemented. An implementation consists of one or more functions whose name is qualified by the structure or enumeration name (for example, `String::new()`, `String::from(&str)` and `String::clone(&self)`). Specify `&self` as the first parameter to create a function like a C++ or C# method. Otherwise, the function is simply accessed through the datatype's name space. There are no formal constructors in Rust.

Functions can also be created independent of datatype implementations. Such functions are owned by the module containing them, accessed through the module name.

Rust supports an abstraction called a trait, which is analogous to a C# interface. It names one or more functions which must be implemented by any datatype which implements the trait. Traits are useful as formal parameter types – enabling the actual parameter to be any object implementing the trait. Traits are also useful for restricting generic parameters.

There are a number of traits defined either as part of the language (for example, std::ops::Drop) or standard library (for example, fmt::Display and fmt::Debug). A destructor is associated with a structure by implementing the `std::ops::Drop::drop(&mut self)` method.

Flow Control

Rust implements 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 C#'s than C++ -- for example, it can sequence through members of a collection by running an enumerator.

Rust also includes a couple of unique flow control statements:

- Match & If let

  These statements are generalizations of the switch statement in C# -- you specify a test 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 are commonly 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, which 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 is shorthand for:

```
match function_call(…) {
    Ok(value) => value,
    Err(e) => return Err(e),
}
```

For example:

```
let v = function_call(…)?;
```

This says "If function_call succeeds, assign its success value to v, and if function_call fails, return its error value as the Result<S,E> value of this function."

This makes the code more concise and easier to read. The Rust compiler complains if a result is not used, which makes it easy to ensure functions terminate on failure, propagating the error back to their 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 list:

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

Option<T> is frequently used in conjunction with the match and if-let statements 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.
    a. A datum always has exactly one owner.
    b. It is deleted when it has no owner (for example, execution continues past the scope of a variable).
    c. Mutable data can only be modified by its owner.
    d. 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 must 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. This means that an alternative to simple references must be used. Examples include indexing and smart pointers.

Smart Pointers

The combination of ownership, mutability, and lifetimes caused the Rust community to introduce alternatives to references called *smart pointers*. The standard library includes several smart pointers. The two I've found most useful are:

- Ref-counted pointer

  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. The implementation wraps the interior datum with a temporary object, which uses destructors to automatically manage the ref-count.

  Rc<T> is for single-threaded access and Arc<T> for multi-threaded.

- Interior mutability of members, enforced at runtime

  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. This essentially bypasses the compile-time check for mutability and allows shared borrowing instead of moves.

  RefCell<T> is single-threaded and Mutex<T> supports multiple threads.

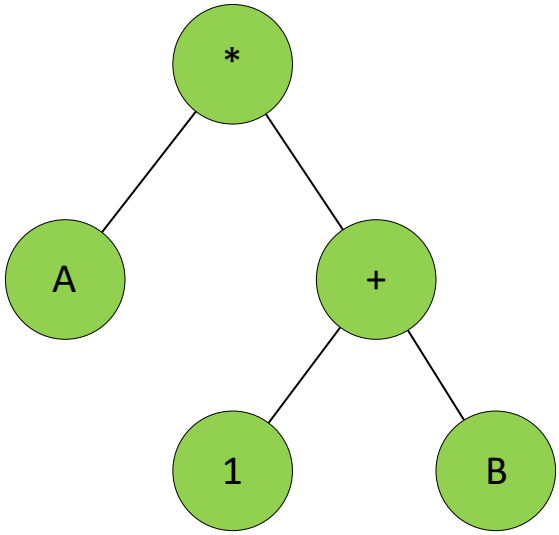
The single-thread implementations (Rc<T> and RefCell<T>) are fast, but the multi-thread implementations (Arc<T>, Mutex<T>) suffer from the fundamental performance pitfalls of multi-processor synchronization primitives. The compiler includes thread-local storage features, so it is possible to write efficient 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, or cloning structures more than in C++ or C#).

In my opinion, these are practical, inelegant solutions to restrictive language concepts enforced by the compiler. That is not to say that I have a better idea – merely to state that 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 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 may be 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 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 must 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.

I found I had to be careful to avoid overuse of cloning data to get around lifetime and ownership issues. Careful thought usually resulted in a better solution that fits with Rust, but programmers should be wary of inefficiencies introduced by excessive data structure creation and copying.