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);
```

This won't compile in Rust, and will throw the following error:

```
error[E0502]: cannot borrow `array` as immutable because it is also
borrowed as mutable
 --> src/main.rs:4:14
  |
3 |   let a = &mut array[0..2];
  |               ----- mutable borrow occurs here
4 |   let b = &array[2..4];
  |               ^^^^^ immutable borrow occurs here
5 |   a.copy_from_slice(b);
6 | }
  | - mutable borrow ends here
```

This error signifies the restrictions imposed by the borrow checker. Multiple immutable references are fine. One mutable reference is fine. Different references to different objects are fine. However, you can't simultaneously have a mutable and an immutable reference to the same object because this is possibly unsafe and can lead to memory corruption.

Not only does this restriction prevent possible human errors, but it in fact enables the compiler to perform some optimizations that are normally not possible in the presence of aliasing. The compiler is then free to use registers more aggressively, avoiding redundant memory access and leading to increased performance.

Option Types instead of Null Pointers

Another common issue related to pointers and references is null pointer dereferencing. Tony Hoare calls the invention of the null pointer value his billion-dollar mistake, and an increasing number of languages are including mechanisms to prevent it (for example, Nullable types in Java and C#, *std::optional* type since C++17).

Rust uses the same approach as C++ for references: they always point to an existing object. There are no null references and hence no possible issues with them. However, smart pointers aren't references and may not point to objects; there are also cases when you might like to pass a reference to an object and make no reference.

Instead of using nullable pointers, Rust has the Option type. This type has two constructors:

Some (*value*) – to declare some value

None – to declare the absence of a value

None is functionally equivalent to a null pointer (and in fact has the same representation), while *Some* carries a value (for example, a reference to some object).

The main advantage of *Option* before pointers is that it's not possible to accidentally dereference *None*, and thus null pointer dereferencing errors are eliminated. To use the value stored in *Option*, you need to use safe access patterns:

```
match option_value {
    Some(value) => {
        // use the contained value
    }
    None => {
        // handle absence of value
    }
}

if let Some(value) = option_value {
    // use the contained value
}
let value = option_value.unwrap(); // throws an exception if
option_value is None
```

Every use of *Option* acts as a clear marker, so that no object may be present above, as it requires handling in both cases. Furthermore, the *Option* type has many utility methods that make it more convenient to use:

```
// Falling back to a default value:
let foo_enabled: bool = configuration.foo_enabled.unwrap_or(true);

// Applying a conversion if the Option contains a value
// or leaving it None if Option is None:
let maybe_length: Option<usize> = maybe_string.map(|s| s.len());

// Options can be compared for equality and ordering (given that
// the wrapped values can be compared).
let maybe_number_a = Some(1);
let maybe_number_b = Some(9);
let maybe_number_c = None;
assert_eq!(maybe_number_a < maybe_number_b, true);
assert_eq!(maybe_number_a < maybe_number_c, false); // None is less than
Some
assert_eq!(maybe_number_c < maybe_number_b, true);
assert_eq!(maybe_number_a != maybe_number_c, true);
```

No Uninitialized Variables

Another possible issue with so-called plain old types in C++ is usage of uninitialized variables. Rust requires variables to be initialized before they are used. Most commonly, this is done when variables are declared:

```
let array_of_ten_zeros = [0; 10];
```

But it's also possible to first declare a variable and then initialize it later:

```
let mut x;
// x is left truly uninitialized here; the assembly
// will not contain any actual memory assignment

loop {
    if something_happens() {
        x = 1;
        println!(“{}”, x); // This is okay because x is initialized now
    }

    println!(“{}”, x);  // But this line will cause a compilation error
                        // because x may still not be initialized here

    if some_condition() {
        x = 2;
        break;
    }
    if another_condition() {
        x = 3;
        break;
    }
}

// The compiler knows that it is not possible to exit the loop
// without initializing x, so this is totally safe:
println!(“{}”, x);
```

Whatever you do, keep in mind that with Rust if you forget to initialize a variable and then accidentally use a garbage value, you’ll get a compilation error. All structure fields must be initialized at construction time as well:

```
let foo = Foo {
    bar: 5,
    baz: 10,
};
```

If a field is added to the structure at some point later than all existing constructors, it will generate compilation errors that must be fixed.

Threads without Data Races

During early development of Rust, it was discovered that the borrow checker (responsible for general memory safety) is also capable of preventing *data races* between threads. In order for a data race to occur, three conditions must simultaneously hold:

- two or more threads are concurrently accessing a memory location
- at least one thread is writing there
- at least one thread isn't synchronized

The borrow checker ensures that at any point in time a memory location has either any number of read-only references or exactly one writable reference, thus preventing the first and second conditions from occurring together.

However, while references are the most common way to read and modify memory, Rust has other options for this, so the borrow checker isn't a silver bullet. Furthermore, risks to thread safety aren't limited to data races. Rust has no special magic to completely prevent *general data races*, so deadlocks and poor usage of synchronization primitives are still possible and certain knowledge is still required to avoid them.

The Rust compiler prevents the most common issues with concurrency that are allowed by less safe programming languages like C or C++, but it doesn't require garbage collection or some background, under-the-hood threads and synchronization to achieve this. The standard library includes

many tools for safe usage of multiple concurrency paradigms. There are the following tools:

- message passing
- shared state
- lock-free
- purely functional

Passing Messages with Channels

Channels are used to transfer messages between threads. Ownership of messages sent over a channel is transferred to the receiver thread, so you can send pointers between threads without fear of a possible race condition occurring later. Rust's channels enforce thread isolation.

Here's an example of channel usage:

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

// First create a channel consisting of two parts: the sending half (tx)
// and the receiving half (rx). The sending half can be duplicated and
// shared among many threads.
let (tx, rx) = channel();

// Then spawn 10 threads, each with its own sending handle.
// Each thread will post a unique number into the channel.
for i in 0..10 {
    let tx = tx.clone();
    thread::spawn(move || {
        tx.send(i).unwrap();
    });
}

// Now in the main thread we'll receive the expected ten numbers from
// our worker threads. The numbers may arrive in any arbitrary order,
// but all of them will be read safely.
for _ in 0..10 {
    let j = rx.recv().unwrap();
    assert!(0 <= j && j < 10);
}
```

An important thing here is that messages are actually *moved* into the channel, so it's not possible to use them after sending. Only the receiver thread can re-acquire ownership over the message later.

Safe State Sharing with Locks

Another more traditional way to deal with concurrency is to use a passive shared state for communication between threads. However, this approach requires proper synchronization, which is notoriously hard to do correctly: it's very easy to forget to acquire a lock or to mutate the wrong data while holding the correct lock. Many languages even go as far as removing support for this low-level concurrency style altogether in favor of higher-level approaches (like channels). Rust's approach is dictated by the following thoughts:

1. Shared state concurrency is an essential and totally valid programming style. It's needed for system code to maximize performance and as a building block for high-level concurrency primitives.
2. Most of the time, problems arise when a state is shared *accidentally*.

So Rust provides tools for using shared state concurrency in a safe but direct way.

Threads in Rust are naturally isolated from each other via ownership, which can be transferred only at safe points like during thread creation or via safe interfaces like channels. A thread can mutate data only when it has a mutable reference to the data. In single-threaded programs, safe usage of references is enforced statically by the borrow checker.

Multi-threaded programs must use locks to provide the same mutual exclusion guarantees dynamically. Rust's ownership and borrowing system allows locks to have an API that's impossible to use unsafely. The principle "lock data, not code" is enforced in Rust.

The **Mutex** type in Rust is actually a *generic* over a type T data structure protected by the lock. When a Mutex is created, data is transferred *into* the mutex, giving up direct access to it:

```
// Let's write a generic thread-safe stack using
// a synchronized vector as its backing storage:
struct ThreadSafeStack<T> {
   elements: Mutex<Vec<T>>,
}

impl<T> ThreadSafeStack<T> {
    fn new() -> ThreadSafeStack<T> {
        ThreadSafeStack {
            // The vector we created is moved into the mutex,
            // so we cannot access it directly anymore
            elements: Mutex::new(Vec::new()),
        }
    }
}
```

In order to get safe access to the data protected by the mutex, we need to use the lock() method, which blocks access until the mutex is acquired. This method returns a value: an instance of **MutexGuard<T>** which is a RAII-style guard. The guard will automatically release the mutex when it's destroyed, so there's no unlock() method:

```
impl<T> ThreadSafeStack<T> {
  // Note that the push() method takes a non-mutable reference to
  // “self,” allowing multiple references to exist in different threads:
    fn push(&self, value: T) {
        let mut elements = self.elements.lock();
        // After acquiring the lock, the mutex will remain locked until
        // this method returns, preventing any race conditions.

        // You can safely access the underlying data and perform any
        // actions you need to with it:
        elements.deref_mut().push(value);
    }
}
```

The key idea here is that lifetimes of any references to data protected by a mutex are tied to the lifetime of the corresponding MutexGuard, and the lock is held for the whole lifetime of the MutexGuard. In this way, *Rust enforces locking discipline*: lock-protected data can only be accessed when holding the lock.

Consider a typical mistake when using mutexes: a dangling reference to protected data remains alive after the mutex is unlocked. For example, you may want to add the following method to ThreadSafeStack:

```
    impl<T> ThreadSafeStack<T> {
    // Peek into the stack, returning a reference to the top element.
    // If the stack is empty then return None.
    fn peek(&self) -> Option<&T> {
        let elements = self.elements.lock();
        // Now we can access the stack data.

        // Handle the case of empty stack.
        if elements.is_empty() {
            return None;
        }

        // And if we have at least one element then return a reference
        // to the last one in the vector:
        return Some(&elements[elements.len() - 1]);
    }
}
```

However, the Rust compiler won't allow this. It sees that the reference you're trying to return will be alive longer than the time for which the lock is held, and will tell you exactly what is wrong with the code:

```
    error[E0597]: `elements` does not live long enough
  --> src/main.rs:45:22
   |
45 |         return Some(&elements[elements.len() - 1]);
   |                      ^^^^^^^^ does not live long enough
46 |  }
   |  - borrowed value only lives until here
   |
```

Indeed, if this were allowed, the reference you got may suddenly be invalidated when some other thread popped the top element off the stack. A possible race condition has been swiftly averted at compilation time, saving you an hour or two of careful debugging of a weird behavior that occurs only on Tuesdays.

Trait-Based Generics

Generics are a way of generalizing types and functionalities to broader cases. They're extremely useful for reducing code duplication in many ways, but can call for rather involving syntax. Namely, generics require great care in specifying over which types they are actually considered valid. The simplest and most common use of generics is for type parameters.

Rust's generics are similar to C++ templates in both syntax and usage. For example, the generic Rust data structure implementing a dynamically sized array is called **Vec<T>**. It's a vector specialized at compile time to

contain instances of any type T. Here's how it's defined in the standard library:

```
    // Definitions of generic structure type look like this
pub struct Vec<T> {
    buf: RawVec<T>, // the buffer representation
    len: usize,   // used size of the vector
}

pub struct RawVec<T, A: Alloc = Heap> {
    ptr: Unique<T>, // pointer to the actual buffer array of T
    cap: usize,   // allocated size of the buffer
    a: A,         // the allocator
}
```

A generic function max() that takes two arguments of any type T can be declared like this:

```
fn max<T>(a: T, b: T) -> T { /* ··· */ }
```

However, this definition is incorrect because we can't in fact apply max() to *any* type. The maximum function is defined only for values which have some defined ordering between them (the one used by comparison operators like < and >=). This is where Rust generics are different from C++ templates.

In C++, requirements for template parameters are implicit:

```
template <typename T>
T max(T a, T b)
{
    return a < b ? b : a;
}
```

This function will compile only when used with types that actually define the 'operator<(), and will cause a compilation error otherwise:

```
    test.cpp: In instantiation of ‘T max(T, T) [with T = Person]’:
test.cpp:19:46:   required from here
test.cpp:11:14: error: no match for ‘operator<’ (operand types are
‘Person’ and ‘Person’)
    return a < b ? b : a;
           ~~^~~
```

This error may be sufficient in simple cases like max() where type requirements and the source of the error are obvious, but with more complex functions using more than one required method, you can be quickly overwhelmed by large amounts of seemingly unrelated errors about obscure missing methods.

Rust makes generic type requirements explicit by using *traits*:

```
fn max<T: Ord>(a: T, b: T) -> T {
    if a < b { b } else { a }
}
```

Now max() can be used with any type T that *implements* the Ord trait. This trait defines the ordering essential for the implementation of max() and makes it possible to use the comparison operators. Explicit requirements enable the compiler to generate much more friendly error messages when a function is accidentally provided with arguments of an incorrect type:

```
  error[E0277]: the trait bound `Person: std::cmp::Ord` is not satisfied
  --> src/main.rs:16:19
   |
16 |  let bigger_person = max(person1, person2);
   |       ^^^ the trait `std::cmp::Ord` is not implemented for `Person`
   |
   = note: required by `max`
```

Traits Define Type Interfaces

Traits are Rust’s way of defining interfaces. They describe what methods must be implemented by a type in order to satisfy the trait. For example,

the Ord trait requires a single method cmp() that compares this value to another and returns the ordering between them:

```
    pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

Traits may be derived from other traits. The Ord trait is derived from the traits *PartialOrd* (specifying partial ordering) and *Eq* (specifying equivalence relation). Thus, Ord may be implemented only for types that implement both the PartialOrd and Eq traits. Methods of parent traits are inherited by child trait implementations, so for example any Ord type can be compared with the == operator provided by the Eq trait.

Traits may also contain common implementations of methods provided to all concrete implementations of the trait. For example, the PartialOrd trait provides the implementation of the lt() method. This is the method actually called when the < comparison operator is used in code:

```
    pub trait PartialOrd<Rhs: ?Sized = Self>: PartialEq<Rhs> {
    // Method defining ordering between this value and some other value,
    // possibly returning None if such ordering is not defined.
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    // The lt() method will be automatically implemented like this
    // for all types that implement the partial_cmp() method:
    #[inline]
    fn lt(&self, other: &Rhs) -> bool {
        match self.partial_cmp(other) {
            Some(Less) => true,
            _ => false,
        }
    }
}
```

Aside from methods, traits can only contain type definitions. Like in Java or C#, traits define only interface requirements, not the data layout of concrete implementations of the interface.

Traits Implement Polymorphism

Together with generics, traits provide **static (compile-time) polymorphism**. The data layout for generic data structures and the actual implementation of generic methods and functions is selected during compilation time based on the known types of values. The resulting machine code is as efficient as it would be in case of manual specialization. Generics are a *zero-cost* abstraction mechanism.

But sometimes, you want to have generic code that acts differently based on real runtime value types. Rust implements **dynamic (runtime) polymorphism** via so-called *trait objects*.

For example, here's how the *Abstract Factory pattern* looks in Rust:

First we need to define the interfaces:

```
// Interfaces of the produced products
trait ProductA {
    fn do_foo(&mut self);
}

trait ProductB {
    fn do_bar(&mut self);
}

// Interface of the abstract product factory
trait ProductFactory {
    fn make_product_a(&self) -> Box<ProductA>;
    fn make_product_b(&self) -> Box<ProductB>;
}
```

Note that the factory produces *Boxes* with products. A Box is Rust's equivalent of std::unique_ptr in C++. It denotes an object allocated on the heap and can contain *trait objects* that are actually pointers to the concrete implementation of a trait.

Then we define the concrete implementations of products and the factory:

```
// Example of concrete implementation of products
struct ConcreteProductA;
struct ConcreteProductB;

// Implement trait ProductA for struct ConcreteProductA
impl ProductA for ConcreteProductA {
    fn do_foo(&mut self) {
        println!("ConcreteProductA doing foo");
    }
}

// Implement trait ProductB for struct ConcreteProductB
impl ProductB for ConcreteProductB {
    fn do_bar(&mut self) {
        println!("ConcreteProductB doing bar");
    }
}

// Example of concrete factory
struct ConcreteFactory;

// Implement trait ProductFactory for struct ConcreteFactory,
// creating some concrete products when asked
impl ProductFactory for ConcreteFactory {
    fn make_product_a(&self) -> Box<ProductA> {
        Box::new(ConcreteProductA)
    }

    fn make_product_b(&self) -> Box<ProductB> {
        Box::new(ConcreteProductB)
    }
}
```

Note how trait implementations are separated from declared structures. This is why Rust has *traits*, not interfaces. A trait can be implemented for a type not only by the module that declares the type but from anywhere else in the program. This opens up many possibilities for extending the behavior of library types if the provided interfaces don't meet your needs.

Finally, we can implement abstract algorithms using the abstract factory to make abstract products and operate on them:

```
    fn make_and_use_some_stuff(factory: &ProductFactory) {
    let mut a: Box<ProductA> = factory.make_product_a();
    let mut b: Box<ProductB> = factory.make_product_b();

    a.do_foo();
    b.do_bar();
}
```

Here, the function *make_and_use_some_stuff*() doesn't know the concrete types of the factory and products involved in computation. It operates only on trait objects. All method calls involve *dynamic dispatch* on the *virtual method tables* stored inside the trait objects. The function isn't generic and only one implementation of it exists in the program.

Traits May be Implemented Automatically

Some traits may be automatically derived and implemented by the compiler for user-defined data structures. For example, the **PartialEq** trait that defines the == comparison operator can be automatically implemented for any data structure provided that all its fields implement PartialEq too:

```
    #[derive(PartialEq)]
struct Person {
    name: String,
```

```
    age: u32,
}
```

The #[derive] attribute can be used with the following traits:

- Comparison traits: **Eq, PartialEq, Ord, PartialOrd**
- **Clone**, used to create a copy of a value via reference to it
- **Copy**, which gives the type *copy semantics* instead of *move semantics* by default
- **Hash**, used by many containers to compute a hash value of elements
- **Default**, used for creating empty instances in a consistent way
- **Zero**, defined for zero-initialization of numeric types
- **Debug**, a trait used by the {:?} formatting string in debugging output

Pattern Matching

Similar to C++, Rust has enumeration types:

```
enum Month {
    January, February, March, April, May, June, July,
    August, September, October, November, December,
}
```

It also has a multiple-choice construction to operate on them:

```
match month {
    Month::December | Month::January | Month::February
        => println!("It's winter!"),
    Month::March | Month::April | Month::May
        => println!("It's spring!"),
    Month::June | Month::July | Month::August
        => println!("It's summer!"),
    Month::September | Month::October | Month::November
        => println!("It's autumn!"),
}
```

However, **match** has more features than a simple **switch**. The most crucial difference is that matching must be *exhaustive*: the match clause must handle all possible values of the expressions being matched. This eliminates a typical error in which switch statements break when an enumeration is extended later with new values. Of course, there's also a default catch-all option that matches any value:

```
match number {
    0..9 => println!("small number"),
    10..100 if number % 2 == 0 => {
        println!("big even number");
    }
    _ => println!("some other number"),
}
```

Another important feature of Rust enumerations is that they can carry values, implementing *discriminated unions* safely.

```
enum Color {
    Red, Green, Blue,
    RGB(u32, u32, u32),
    CMYK(u32, u32, u32, u32),
}
```

Pattern matching can be used to match against possible options and extract values stored in a union:

```
match some_color {
    Color::Red => println("Pure red"),
    Color::Green => println("Pure green"),
    Color::Blue => println("Pure blue"),
    Color::RGB(red, green, blue) => {
        println("Red: {}, green: {}, blue: {}", red, green, blue);
    }
    Color::CMYK(cyan, magenta, yellow, black) => {
        println("Cyan: {}, magenta: {}, yellow: {}, black: {}",
            cyan, magenta, yellow, black);
    }
}
```

Unlike C and C++ unions, Rust makes it impossible to choose an incorrect branch when unpacking a union.

Type Inference

Rust uses a *static type system*, which means that types of variables, function arguments, structure fields, and so on must be known at compile time; the compiler will check that correct types are used everywhere. However, Rust also uses *type inference*, which allows the compiler to automatically deduce types based on how variables are used.

This is very convenient because you no longer need to explicitly state types, which in some cases may be cumbersome (or impossible) to write. The **auto** keyword in C++ serves the same purpose:

```
std::vector<std::map<std::string, std::vector<Object>>> some_map;

// Iterator types can easily become a mess:
for (const auto &it : some_map)
{
    /* ... */
}

// Lambda functions can only be used with auto;
// their exact type cannot be expressed in C++
auto compare_by_cost = [](const Foo &lhs, const Foo &rhs) {
    return a.cost < b.cost
};
```

However, Rust also considers future uses of a variable to deduce its type – not only the initializer – allowing programmers to write code like this:

```
let v = 10;    // v's type is some integer (based on the constant),
               // but the exact type (i32, u8, etc.) is not yet known

let mut vec = Vec::new(); // vec's type is some Vec<T>, where T may be
anything

vec.push(v);   // after this line, the compiler knows that T == v's type

let s = v + vec.len();  // vec.len() returns "usize", so this must be the
type
              // of v (as another addend) and s (as a sum), and vec
              // is now also known to have type Vec<usize>

println!("{}: {:?}", s, vec); // prints 11: [10]
```

Rust uses the widely known and thoroughly researched *Hindley-Milner inference algorithm*. This algorithm is most commonly used in functional programming languages. It can handle global type inference (inferring all types in an entire program, even the types of function arguments, returns, structure fields, etc.) But global type inference can be slow in large projects and can cause types to change with unrelated changes in the code base. Thus, Rust uses inference only for local variables. You must explicitly write types for arguments and structure fields. This strikes a good balance between expressibility, speed, and robustness. Types also make good documentation for functions, methods, and structures.

Minimal Runtime

Runtime is the language support library that's embedded into every program and provides essential features to the Rust programming language. The Java Virtual Machine can be thought of as the runtime of the Java language, for example, as it provides features like class loading

and garbage collection. The size and complexity of the runtime contributes significantly to start-up and runtime overhead. For example, the JVM requires a non-negligible amount of time to load classes, warm up the JIT compiler, collect garbage, and so on.

Rust doesn't have any garbage collection, virtual machine bytecode interpreter, or lightweight thread scheduler running in background. The code you write is exactly what's executed by the CPU. Some parts of the Rust standard library can be considered the "runtime," providing support for heap allocation, backtraces, stack unwinding, and stack overflow guards. The standard library also has some minor amount of global initialization code, similar to the initialization code of a C runtime library that sets up the stack, calls global constructors, and so on before control is transferred to the main() function. (You can compile Rust programs without the standard library if you don't need it, thus avoiding this overhead.)

In short, Rust *can* be used for really low-level work like bare-metal programming, device drivers, and operating system kernels:

- Rust Embedded (https://github.com/rust-embedded)
- rust.ko (https://github.com/tsgates/rust.ko)
- Windows KMD (https://github.com/pravic/winapi-kmd-rs)
- Redox OS (https://www.redox-os.org/)

Furthermore, the absence of a complex runtime simplifies embedding Rust modules into programs written in other languages. For example, you can

easily write JNI code for Java or extensions for dynamic languages like Python, Ruby, or Lua.

Efficient C Bindings

There's more than one programming language in the world, so it's not surprising that you might want to use libraries written in languages other than Rust. Conventionally, libraries provide a C API because C is a ubiquitous language, the common denominator of programming languages. Rust is able to easily communicate with C APIs, without any overhead, and use its ownership system to provide significantly stronger safety guarantees for them.

Calling C from Rust

Let's look at a simple example. Consider the following C library for adding numbers (here we take it easy and use a regular for loop, but we could do something clever with AVX instructions):

```
/**
 * Sum some numbers.
 *
 * @param numbers [in] pointer to the numbers to be summed
 *                     must not be NULL and must point to at least
 *                     `count` elements
 * @param count [in]    number of numbers to be summed
 *
 * @returns sum of the provided numbers.
 */
int sum_numbers(const int *numbers, size_t count)
{
    int sum = 0;

    for (size_t i = 0; i < count; i++)
    {
```

```
        sum += numbers[i];
    }

    return sum;
}
```

Note that some parts of this function API are described formally by the argument types, but some things are only specified in the documentation. For example, we can only infer that we can't pass NULL for a *numbers* argument and that there must be at least *count* numbers available. And only the common sense of a C programmer tells us that the function won't call free() for the *numbers* array.

Here's how we can call this function from Rust:

```
extern crate libc;

extern {
    fn sum_numbers(numbers: *const libc::c_int, count: libc::size_t)
        -> libc::c_int;
}

fn main() {
    let array = [1, 2, 3, 4, 5];
    let sum = unsafe { sum_numbers(array.as_ptr(), array.len()) };
    println!("Sum: {}", sum); // ===> prints "15"
}
```

As you can see, there's *no syntactical overhead* in calling an external function written in C (other than spelling out the prototype of the function). It's just like calling a native Rust function. If you look at the generated assembly code, you can see that this function call has *no runtime overhead* as well:

```
leaq        32(%rsp), %rdi
movl  $5, %esi
callq       sum_numbers@PLT
movl  %eax, 12(%rsp)
```

There's no hidden boxing and unboxing, re-allocating of the array, obligatory safety checks, or other things. We see exactly the same machine code that a C compiler would have generated for the same library function call.

The Libc Crate and Unsafe Blocks

However, there are some details in the above code that require further explanation – first of all, the **libc crate**. This is a wrapper library that provides types and functions of the C standard library to Rust. Here you can find all the usual types, constants, and functions:

- libc::c_uint (unsigned int type)
- libc::stat (struct stat structure)
- libc::pthread_mutex_t (pthread_mutex_t typedef)
- libc::open (open(2) system call)
- libc::reboot (reboot(2) system call)
- libc::EINVAL (EINVAL constant)
- libc::SIGSEGV (SIGSEGV constant)
- and many more, depending on the platform you compile on

Not only can you use "normal" C libraries via the Rust Foreign Function Interface – you can also readily use the system API via **libc crate**.

Another catch lies in the **unsafe** block:

```
let sum = unsafe { sum_numbers(array.as_ptr(), array.len()) };
```

As the sum_numbers() function is *external*, it doesn't automatically provide the degree of safety provided by native Rust functions. For example, Rust

will allow you to pass a NULL pointer as the first argument and this will cause *an undefined behavior* (just as it would in C). The function call isn't safe, so it must be wrapped in an *unsafe block* which effectively says "Compiler, you have my word that this function call is safe. I have verified that the arguments are okay, that the function won't compromise Rust safety guarantees, and that it won't cause undefined behavior."

Just as in C, the programmer is ultimately responsible for guaranteeing that the program doesn't cause undefined behavior. The difference here is that with C you must manually do this at all times, in all parts of the code, for every library you use. On the other hand, in Rust *you must manually verify safety only inside unsafe blocks*. All other Rust code (outside unsafe blocks) is automatically safe, as routinely verified by the Rust compiler.

Herein lies the power of Rust: you can provide safe wrappers for unsafe code and thus avoid tedious, manual safety verifications in the consumer code. For example, the sum_numbers function can be wrapped like this:

```
fn sum_numbers(numbers: &[libc::c_int]) -> libc::c_int {
    // This is safe because Rust slices are always non-NULL
    // and are guaranteed to be long enough
    unsafe { sum_numbers(numbers.as_ptr(), numbers.len()) }
}
```

Now the external function has a safe interface. It can be readily used by idiomatic Rust code without unsafe blocks. Callers of the function don't need to be aware of the actual safety requirements of its native C implementation. And it's still as fast as the original!

Beyond Primitive Types

Aside from primitive types like libc::c_int and pointers, Rust can use other C types as well.

Rust structs can be made compatible with C structs via a #[repr] annotation:

```
#[repr(C)]
struct UUID
{
    time_low:  u32,
    time_mid:  u16,
    time_high: u16,
    sequence:  u16,
    node:   [u8; 6],
};
```

```
struct UUID
{
    uint32_t time_low;
    uint16_t time_mid;
    uint16_t time_high;
    uint16_t sequence;
    uint8_t node[6];
};
```

Such structures can be passed by value or by pointer to C code, as they'll have the same memory layout as their C counterparts used by a C compiler. (Obviously, the fields can only have types that C can understand.)

C unions can also be directly represented in Rust:

```
  union TypePun
{
    f: f32,
    i: i32,
};
```

```
  union TypePun
{
    float f;
    int   i;
};
```

As in C, unions in Rust are *untagged*. That is, they don't store the runtime type of the value inside them. The programmer is responsible for accessing union fields correctly. The compiler can't check this automatically, so Rust unions require an explicit **unsafe** block when accessing their fields both for reading and writing.

Simple enumerations are also compatible with C:

```
enum Options
{
    ONE = 0,
    TWO,
    THREE,
}
```

```
enum Options
{
    ONE,
    TWO,
    THREE,
};
```

However, you can't use advanced features of Rust enum types when calling C code. For instance, you can't directly pass Option<T> or Result<T> values to C.

Rust functions can be converted into C function pointers given that the argument types are actually compatible and the C ABI is used:

```
fn launch_native_thread() {
    let name = "Ferris";
// We're going to launch a native thread via pthread_create() from libc.
// This is an external function, so calling it is unsafe in Rust (think
// about exception boundaries, for example).
    unsafe {
        let mut thread = 0;
        libc::pthread_create(&mut thread, // out-argument for pthread_t
            ptr::null(),                  // in-argument of thread_attr_t
            thread_body,                  // thread body (as a C callback)
            mem::transmute(&name)     // thread argument (requires a cast)
        );
        libc::pthread_join(thread, ptr::null_mut());
    }
}

// Here's our thread body with C ABI written in Rust
extern "C" fn thread_body(arg: *mut libc::c_void) -> *mut libc::c_void {
   // We need to cast the argument back to the original reference to &str.
   // This is unsafe (from the Rust compiler's point of view), but we know
   // what kind of data we have put into this void*
    let name: &&str = unsafe { mem::transmute(arg) };
    println!("Hello {} from Rust thread!", name);
    return ptr::null_mut();
}
```

Calling Rust from C

Native Rust functions and types can be made available to C code just as easily as you can call C from Rust. Let's reverse the example with the sum_numbers() function and implement it in Rust instead:

```
#[no_mangle]
pub extern "C" fn sum_numbers(numbers: *const libc::c_int, count:
libc::size_t)
    -> libc::c_int
{
// Convert the C pointer-to-array into a native Rust slice of an array.
    // This is not safe per se because the "numbers" pointer may be NULL
    // and the "count" value may not match the actual array length.
    //
    // As with C, we'll require the caller of this function to ensure
    // that these safety requirements are observed and will not check
    // them explicitly here.
```

```
    let rust_slice = unsafe { from_raw_parts(numbers, count) };

    // Rust slice types already have a handy method for summing their
    // elements. Let's use it here.
    return rust_slice.sum();
}
```

And that's it. The #[no_mangle] attribute prevents symbol mangling (so that the function is exported with the exact name "sum_numbers"). The **extern** directive specifies that the function should have the C ABI instead of the native Rust ABI. With this, any C program can link to a library written in Rust and can easily use our function:

```
// Declare the function prototype for C
int sum_numbers(const int *numbers, size_t count);

int main()
{
    int numbers[] = { 1, 2, 3, 4, 5 };
    int sum = sum_numbers(numbers, 5);
    printf("Sum is %d\n", sum);
}
```

Calling a Rust library in C is as easy as calling a native C library. There are no required conversions, no Rust VM context needs to be initialized and passed as an additional argument, and there's no overhead aside from the regular function call.

Rust vs. C++ Comparison

Rust is syntactically similar to C++, but it provides increased speed and better memory safety.

In order to explain why Rust is a safer and faster language than C++, we decided to create a Rust vs C++ comparison chart that clearly shows the differences between these two languages.

For better comparison, we've chosen features that reveal the key similarities and differences between these two languages.

- ***Zero-cost abstraction***

Issue	C++	Rust
Preferring code duplication to abstraction due to high cost of virtual method calls	Zero-cost abstraction mechanisms allow you to avoid runtime costs when possible.	Zero-cost abstraction mechanisms allow you to avoid runtime costs when possible.

- ***Move semantics***

Issue	C++	Rust
Move constructors may leave objects in invalid and unspecified states and cause use-after-move errors	Move constructors are suggested to leave the source object in a valid state (yet the object shouldn't be used in correct programs).	A built-in static analyzer disallows use of objects after they have been moved.
	Use-after-move errors are detected at runtime using a special sentinel state.	The compiler can rely on this built-in analyzer for optimization.
	External static code analyzers can spot use-after-move errors at compile time.	

- ***Smart pointers vs. null pointers***

Issue	C++	Rust
Use-after-free, double-free bugs, dangling pointers	Smart pointers and references are preferred to raw pointers.	Smart pointers and references are preferred to raw pointers.
	Manual code review can spot use of raw pointers where smart pointers would suffice.	Raw pointers can only be used inside unsafe blocks, which can automatically be found by tools.
Null dereferencing errors	References are preferred to pointers and cannot be null.	References are preferred to pointers and cannot be null.
	Null dereferencing is still possible even for smart pointers, but is declared as undefined behavior and should never appear.	Null references can be emulated by Option types, which require explicit null checks before use.
	Compilers assume that undefined behavior never happens, don't produce warnings, and use this for optimization (sometimes with fatal consequences for security).	Smart pointers return Optional references and therefore require explicit checks as well.
	External static code analyzers can spot possible errors at compile time.	Raw pointers can be null, but they can only be used inside unsafe blocks. Unsafe blocks need to be carefully reviewed, but they can be found and marked automatically.

- ***Internal buffer***

Issue	C++	Rust
Buffer overflow errors	Explicitly coded wrapper classes enforce range checks.	All slice types enforce runtime range checks.
	Debugging builds of the STL can perform range checks in standard containers.	Range checks are avoided by most common idioms (e.g. range-based for iterators).

- ***Data races***

Issue	C++	Rust
Data races (unsafe concurrent modification of data)	Good programming discipline, knowledge, and careful review are required to avoid concurrency errors.	The built-in borrow checker and Rust reference model detect and prohibit possible data races at compile time.
	External static code analyzers can spot some errors at compile time.	A novel locking API makes it impossible to misuse mutexes unsafely (though still allowing *incorrect* usage).
	External code sanitizers can spot some errors at runtime.	

- ***Object initialization***

Issue	C++	Rust
Uninitialized variables	Constructors of user-defined types are recommended to initialize all object fields.	All variables must be explicitly initialized before use (checked by the compiler).
	Primitive types still have undefined values when not initialized explicitly.	All types have defined default values that can be chosen instead of explicit initialization types.
	External static code analyzers can spot uninitialized variables.	

- ***Static (compile-time) polymorphism***

Issue	C++	Rust
Static interfaces for static polymorphism	*Concepts* should provide this feature directly, but they've been in development since 2015 and are only scheduled for standardization in C++20.	Traits provide a unified way of specifying both static and dynamic interfaces.
	Virtual functions and abstract classes may be used to declare interfaces.	Static polymorphism is guaranteed to be resolved at compile time.
	Virtual function calls may be optimized by particular compilers in known cases.	

- ***Adding new traits***

Issue	C++	Rust
Extending externally defined classes with new methods	Adding new methods normally requires inheritance, which can be inconvenient.	Traits can be added to and implemented for any class in any module at a later point.
	Unified function call syntax could be used to emulate extension methods (if it gets into C++20).	Modules restrict visibility of available methods.

- ***Standard library***

Issue	C++	Rust
Legacy design of utility types heavily used by standard library	Structured types like std::pair, std::tuple and std::variant can replace ad-hoc structures.	Built-in composable structured types: tuples, structures, enumerations.
	These types have inconvenient interfaces (though C++17 improves this).	Pattern matching allows convenient use of structured types like tuples and enumerations.
	Most of the standard library doesn't use structured types.	The standard library fully embraces available pattern matching to provide easy-to-use interfaces.

- ***Branches in switch statements***

Issue	C++	Rust
Forgetting to handle all possible branches in switch statements	Code review and external static code analyzers can spot switch statements that don't cover all possible branches.	The compiler checks that match expressions explicitly handle all possible values for an expression.

- ***Typing of variables***

Issue	C++	Rust
Complex variable types become tedious to type manually	The *auto* and *decltype* keywords provide limited type inference (for expressions).	Local type inference (for a function body) allows you to explicitly specify types less frequently.
	Lambda functions still require manual type specifications, but this is improving with C++17.	Function declarations still require explicit types which ensures good readability of code.

- ***Runtime environment***

Issue	C++	Rust
Embedded and bare-metal programming have high restrictions on runtime environment	The C++ runtime is already fairly minimal, as it directly compiles to machine code and doesn't use garbage collection.	The Rust runtime is already fairly minimal as it directly compiles to machine code and doesn't use garbage collection.
	C++ programs can be built without the standard library with disabled exceptions and dynamic type information, etc.	Rust programs can be built without the standard library with disabled range checks, etc.

- ***Using libraries written in other languages***

Issue	C++	Rust
Using existing libraries written in C and other languages	C libraries are immediately usable by C++ programs.	C libraries require Rust-specific header declarations.
	Libraries in languages other than C++ require wrappers.	Libraries in languages other than Rust require wrappers.
	Exporting a C interface requires only a simple *extern* declaration.	Exporting a C interface requires only a simple *extern* declaration.
	There's no overhead in calling C functions from C++ or calling C++ functions from C.	There's no overhead in calling C functions from Rust or calling Rust functions from C.

You have no doubt noticed that both languages use zero-cost abstractions and move semantics. They also both have smart pointers, no garbage collection, and other similarities.

In contrast to C++, Rust has a built-in static analyzer but no uninitialized variables.

Rust avoids possible data races, informs about undefined behavior, and allows null raw pointers inside unsafe blocks. The Rust language also has other distinctive features that allow programmers to achieve better safety and performance of their software.

This Rust Programming Language Tutorial is based on the experience of Apriorit team who uses Rust for software development along with other programming languages.

This Tutorial is intended for information purposes only. Any trademarks and brands are property of their respective owners and used for identification purposes only.

About Apriorit Inc.

Apriorit Inc. is a software development service provider headquartered in the Dover, DE, US, with several development centers in Eastern Europe. With over 350 professionals, we bring high-quality services on software consulting, research, and development to software vendors and IT companies worldwide.

Apriorit's main specialties are cybersecurity and data management projects, where system programming, driver and kernel level development, research and reversing matter. The company has an independent web platform development department focusing on building cloud platforms for business.

Apriorit team will be glad to contribute to your software engineering projects.

Clutch
FIRMS THAT DELIVER

Find us on Clutch.co

For more information please contact:

Apriorit Inc.

Headquarters:
8 The Green
Suite #7106, Dover, DE, 19901, US
Phone:
202-780-9339

E-mail: info@apriorit.com
www.apriorit.com