Chapter 1. Basics of Rust Concurrency

Long before multi-core processors were commonplace, operating systems allowed for a single computer to run many programs concurrently. This is achieved by rapidly switching between processes, allowing each to repeatedly make a little bit of progress, one by one. Nowadays, virtually all our computers and even our phones and watches have processors with multiple cores, which can truly execute multiple processes in parallel.

Operating systems isolate processes from each other as much as possible, allowing a program to do its thing while completely unaware of what any other processes are doing. For example, a process cannot normally access the memory of another process, or communicate with it in any way, without asking the operating system's kernel first.

However, a program can spawn extra *threads of execution*, as part of the same *process*. Threads within the same process are not isolated from each other. Threads share memory and can interact with each other through that memory.

This chapter will explain how threads are spawned in Rust, and all the basic concepts around them, such as how to safely share data between multiple threads. The concepts explained in this chapter are foundational to the rest of the book.

> *If you're already familiar with these parts of Rust, feel free to skip ahead. However, before you continue to the next chapters, make sure you have a good understanding of threads, interior mutability, `Send` and `Sync`, and know what a mutex, a condition variable, and thread parking are.*

Threads in Rust

Every program starts with exactly one thread: the main thread. This thread will execute your `main` function and can be used to spawn more threads if necessary.

In Rust, new threads are spawned using the `std::thread::spawn` function from the standard library. It takes a single argument: the function the new thread will execute. The thread stops once this function returns.

Let's take a look at an example:

```
use std::thread;

fn main() {
    thread::spawn(f);
    thread::spawn(f);

    println!("Hello from the main thread.");
}
```

```
fn f() {
    println!("Hello from another thread!");

    let id = thread::current().id();
    println!("This is my thread id: {id:?}");
}
```

We spawn two threads that will both execute `f` as their main function. Both of these threads will print a message and show their *thread id*, while the main thread will also print its own message.

Thread ID

The Rust standard library assigns every thread a unique identifier. This identifier is accessible through `Thread::id()` and is of the type `ThreadId`. There's not much you can do with a `ThreadId` other than copying it around and checking for equality. There is no guarantee that these IDs will be assigned consecutively, only that they will be different for each thread.

If you run our example program above several times, you might notice the output varies between runs. This is the output I got on my machine during one particular run:

```
Hello from the main thread.
Hello from another thread!
This is my thread id:
```

Surprisingly, part of the output seems to be missing.

What happened here is that the main thread finished executing the `main` function before the newly spawned threads finished executing their functions.

Returning from `main` will exit the entire program, even if other threads are still running.

In this particular example, one of the newly spawned threads had just enough time to get to halfway through the second message, before the program was shut down by the main thread.

If we want to make sure the threads are finished before we return from `main`, we can wait for them by *joining* them. To do so, we have to use the `JoinHandle` returned by the `spawn` function:

```
fn main() {
    let t1 = thread::spawn(f);
    let t2 = thread::spawn(f);

    println!("Hello from the main thread.");

    t1.join().unwrap();
    t2.join().unwrap();
}
```

The `.join()` method waits until the thread has finished executing and returns a `std::thread::Result`. If the thread did not successfully finish its function because it panicked, this will contain the panic message. We could attempt to handle that situation, or just call `.unwrap()` to panic when joining a panicked thread.

Running this version of our program will no longer result in truncated output:

```
Hello from the main thread.
Hello from another thread!
This is my thread id: ThreadId(3)
Hello from another thread!
This is my thread id: ThreadId(2)
```

The only thing that still changes between runs is the order in which the messages are printed:

```
Hello from the main thread.
Hello from another thread!
Hello from another thread!
This is my thread id: ThreadId(2)
This is my thread id: ThreadId(3)
```

Output Locking

The `println` macro uses `std::io::Stdout::lock()` to make sure its output does not get interrupted. A `println!()` expression will wait until any concurrently running one is finished before writing any output. If this was not the case, we could've gotten more interleaved output such as:

```
Hello fromHello from another thread!
 another This is my threthreadHello fromthread id: ThreadId!
( the main thread.
2)This is my thread
id: ThreadId(3)
```

Rather than passing the name of a function to `std::thread::spawn`, as in our example above, it's far more common to pass it a *closure*. This allows us to capture values to move into the new thread:

```
let numbers = vec![1, 2, 3];

thread::spawn(move || {
    for n in numbers {
        println!("{n}");
    }
}).join().unwrap();
```

Here, ownership of `numbers` is transferred to the newly spawned thread, since we used a `move` closure. If we had not used the `move` keyword, the closure would have captured `numbers` by reference. This would have resulted in a compiler error, since the new thread might outlive that variable.

Since a thread might run until the very end of the program's execution, the `spawn` function has a `'static` lifetime bound on its argument type. In other words, it only accepts functions that may be kept around forever. A closure capturing a local variable by reference may not be kept around forever, since that reference would become invalid the moment the local variable ceases to exist.

Getting a value back out of the thread is done by returning it from the closure. This return value can be obtained from the `Result` returned by the `join` method:

```
let numbers = Vec::from_iter(0..=1000);

let t = thread::spawn(move || {
    let len = numbers.len();
    let sum = numbers.into_iter().sum::<usize>();
    sum / len  ①
});

let average = t.join().unwrap(); ②

println!("average: {average}");
```

Here, the value returned by the thread's closure (①) is sent back to the main thread through the `join` method (②).

If `numbers` had been empty, the thread would've panicked while trying to divide by zero (①), and `join` would've returned that panic message instead, causing the main thread to panic too because of `unwrap` (②).

Thread Builder

The `std::thread::spawn` function is actually just a convenient shorthand for `std::thread::Builder::new().spawn().unwrap()`.

A `std::thread::Builder` allows you to set some settings for the new thread before spawning it. You can use it to configure the stack size for the new thread and to give the new thread a name. The name of a thread is available through `std::thread::current().name()`, will be used in panic messages, and will be visible in monitoring and debugging tools on most platforms.

Additionally, `Builder`'s `spawn` function returns an `std::io::Result`, allowing you to handle situations where spawning a new thread fails. This might happen if the operating system runs out of memory, or if resource limits have been applied to your program. The `std::thread::spawn` function simply panics if it is unable to spawn a new thread.

Scoped Threads

If we know for sure that a spawned thread will definitely not outlive a certain scope, that thread could safely borrow things that do not live forever, such as local variables, as long as they outlive that scope.

The Rust standard library provides the `std::thread::scope` function to spawn such *scoped threads*. It allows us to spawn threads that cannot outlive the scope of the closure we pass to that function, making it possible to safely borrow local variables.

How it works is best shown with an example:

```
let numbers = vec![1, 2, 3];

thread::scope(|s| { ①
    s.spawn(|| { ②
        println!("length: {}", numbers.len());
    });
    s.spawn(|| { ②
        for n in &numbers {
            println!("{n}");
        }
    });
}); ③
```

① We call the `std::thread::scope` function with a closure. Our closure is directly executed and gets an argument, `s`, representing the scope.

② We use `s` to spawn threads. The closures can borrow local variables like `numbers`.

③ When the scope ends, all threads that haven't been joined yet are automatically joined.

This pattern guarantees that none of the threads spawned in the scope can outlive the scope. Because of that, this scoped `spawn` method does not have a `'static` bound on its argument type, allowing us to reference anything as long as it outlives the scope, such as `numbers`.

In the example above, both of the new threads are concurrently accessing `numbers`. This is fine, because neither of them (nor the main thread) modifies it. If we were to change the first thread to modify `numbers`, as shown below, the compiler wouldn't allow us to spawn another thread that also uses `numbers`:

```
let mut numbers = vec![1, 2, 3];

thread::scope(|s| {
    s.spawn(|| {
        numbers.push(1);
    });
    s.spawn(|| {
        numbers.push(2); // Error!
    });
});
```

The exact error message depends on the version of the Rust compiler, since it's often improved to produce better diagnostics, but attempting to compile the code above will result in something like this:

```
error[E0499]: cannot borrow `numbers` as mutable more than once at a time
 --> example.rs:7:13
  |
4 |     s.spawn(|| {
  |             -- first mutable borrow occurs here
5 |         numbers.push(1);
  |         ------- first borrow occurs due to use of `numbers` in closure
  |
7 |     s.spawn(|| {
  |             ^^ second mutable borrow occurs here
8 |         numbers.push(2);
  |         ------- second borrow occurs due to use of `numbers` in closure
```

The Leakpocalypse

Before Rust 1.0, the standard library had a function named `std::thread::scoped` that would directly spawn a thread, just like `std::thread::spawn`. It allowed non-`'static` captures, because instead of a `JoinHandle`, it returned a `JoinGuard` which joined the thread when dropped. Any borrowed data only needed to outlive this `JoinGuard`. This seemed safe, as long as the `JoinGuard` got dropped at some point.

Just before the release of Rust 1.0, it slowly became clear that it's not possible to guarantee that something will be dropped. There are many ways, such as creating a cycle of reference-counted nodes, that make it possible to forget about something, or *leak* it, without dropping it.

Eventually, in what some people refer to as "The Leakpocalypse," the conclusion was made that the design of a (safe) interface cannot rely on the assumption that objects will always be dropped at the end of their lifetime. Leaking an object might reasonably result in leaking more objects (e.g., leaking a `Vec` will also leak its elements), but it may not result in undefined behavior. Because of this conclusion, `std::thread::scoped` was no longer deemed safe and was removed from the standard library. Additionally, `std::mem::forget` was upgraded from an `unsafe` function to a *safe* function, to emphasize that forgetting (or leaking) is always a possibility.

Only much later, in Rust 1.63, a new `std::thread::scope` function was added with a new design that does not rely on `Drop` for correctness.

Shared Ownership and Reference Counting

So far we've looked at transferring ownership of a value to a thread using a `move` closure ("Threads in Rust") and borrowing data from longer-living parent threads ("Scoped Threads"). When sharing data between two threads where neither thread is guaranteed to outlive the other, neither of them can be the owner of that data. Any data shared between them will need to live as long as the longest living thread.

Statics

There are several ways to create something that's not owned by a single thread. The simplest one is a `static` value, which is "owned" by the entire program, instead of an individual thread. In the following example, both threads can access `X`, but neither of them owns it:

```
static X: [i32; 3] = [1, 2, 3];

thread::spawn(|| dbg!(&X));
thread::spawn(|| dbg!(&X));
```

A `static` item has a constant initializer, is never dropped, and already exists before the main function of the program even starts. Every thread can borrow it, since it's guaranteed to always exist.

Leaking

Another way to share ownership is by *leaking* an allocation. Using `Box::leak`, one can release ownership of a `Box`, promising to never drop it. From that point on, the `Box` will live forever, without an owner, allowing it to be borrowed by any thread for as long as the program runs.

```
let x: &'static [i32; 3] = Box::leak(Box::new([1, 2, 3]));

thread::spawn(move || dbg!(x));
thread::spawn(move || dbg!(x));
```

The `move` closure might make it look like we're moving ownership into the threads, but a closer look at the type of `x` reveals that we're only giving the threads a *reference* to the data.

🌈
> *References are `Copy`, meaning that when you "move" them, the original still exists, just like with an integer or boolean.*

Note how the `'static` lifetime doesn't mean that the value lived since the start of the program, but only that it lives to the end of the program. The past is simply not relevant.

The downside of leaking a `Box` is that we're *leaking memory*. We allocate something, but never drop and deallocate it. This can be fine if it happens only a limited number of times. But if we keep doing this, the program will slowly run out of memory.

Reference Counting

To make sure that shared data gets dropped and deallocated, we can't completely give up its ownership. Instead, we can *share ownership*. By keeping track of the number of owners, we can make sure the value is dropped only when there are no owners left.

The Rust standard library provides this functionality through the `std::rc::Rc` type, short for "reference counted." It is very similar to a `Box`, except cloning it will not allocate anything new, but instead increment a counter stored next to the contained value. Both the original and cloned `Rc` will refer to the same allocation; they *share ownership*.

```
use std::rc::Rc;

let a = Rc::new([1, 2, 3]);
let b = a.clone();

assert_eq!(a.as_ptr(), b.as_ptr()); // Same allocation!
```

Dropping an `Rc` will decrement the counter. Only the last `Rc`, which will see the counter drop to zero, will be the one dropping and deallocating the contained data.

If we were to try to send an `Rc` to another thread, however, we would run into the following compiler error:

```
error[E0277]: `Rc` cannot be sent between threads safely
    |
8   |     thread::spawn(move || dbg!(b));
    |                   ^^^^^^^^^^^^^^^
```

As it turns out, `Rc` is not *thread safe* (more on that in "Thread Safety: Send and Sync"). If multiple threads had an `Rc` to the same allocation, they might try to modify the reference counter at the same time, which can give unpredictable results.

Instead, we can use `std::sync::Arc`, which stands for "atomically reference counted." It's identical to `Rc`, except it guarantees that modifications to the reference counter are indivisible *atomic* operations, making it safe to use it with multiple threads. (More on that in Chapter 2.)

```
use std::sync::Arc;

let a = Arc::new([1, 2, 3]); ①
let b = a.clone(); ②

thread::spawn(move || dbg!(a)); ③
thread::spawn(move || dbg!(b)); ③
```

① We put an array in a new allocation together with a reference counter, which starts at one.

② Cloning the `Arc` increments the reference count to two and provides us with a second `Arc` to the same allocation.

③ Both threads get their own `Arc` through which they can access the shared array. Both decrement the reference counter when they drop their `Arc`. The last thread to drop its `Arc` will see the counter drop to zero and will be the one to drop and deallocate the array.

Naming Clones

Having to give every clone of an `Arc` a different name can quickly make the code quite cluttered and hard to follow. While every clone of an `Arc` is a separate object, each clone represents the same shared value, which is not well reflected by naming each one differently.

Rust allows (and encourages) you to *shadow* variables by defining a new variable with the same name. If you do that in the same scope, the original variable cannot be named anymore. But by opening a new scope, a statement like `let a = a.clone();` can be used to reuse the same name within that scope, while leaving the original variable available outside the scope.

By wrapping a closure in a new scope (with `{}`), we can clone variables before moving them into the closure, without having to rename them.

```
let a = Arc::new([1, 2, 3]);

let b = a.clone();

thread::spawn(move || {
    dbg!(b);
});

dbg!(a);
```

The clone of the `Arc` lives in the same scope. Each thread gets its own clone with a different name.

```
let a = Arc::new([1, 2, 3]);

thread::spawn({
    let a = a.clone();
    move || {
        dbg!(a);
    }
});

dbg!(a);
```

The clone of the `Arc` lives in a different scope. We can use the same name in each thread.

Because ownership is shared, reference counting pointers (`Rc<T>` and `Arc<T>`) have the same restrictions as shared references (`&T`). They do not give you mutable access to their contained value, since the value might be borrowed by other code at the same time.

For example, if we were to try to sort the slice of integers in an `Arc<[i32]>`, the compiler would stop us from doing so, telling us that we're not allowed to mutate the data:

```
error[E0596]: cannot borrow data in an `Arc` as mutable
  |
6 |     a.sort();
  |     ^^^^^^^^
```

Borrowing and Data Races

In Rust, values can be borrowed in two ways:

Immutable borrowing

Borrowing something with `&` gives an *immutable reference*. Such a reference can be copied. Access to the data it references is shared between all copies of such a reference. As the name implies, the compiler doesn’t normally allow you to *mutate* something through such a reference, since that might affect other code that’s currently borrowing the same data.

Mutable borrowing

Borrowing something with `&mut` gives a *mutable reference*. A mutable borrow guarantees it’s the only active borrow of that data. This ensures that mutating the data will not change anything that other code is currently looking at.

These two concepts together fully prevent *data races*: situations where one thread is mutating data while another is concurrently accessing it. Data races are generally *undefined behavior*, which means the compiler does not need to take these situations into account. It will simply assume they do not happen.

To clarify what that means, let’s take a look at an example where the compiler can make a useful assumption using the borrowing rules:

```
fn f(a: &i32, b: &mut i32) {
    let before = *a;
    *b += 1;
    let after = *a;
    if before != after {
        x(); // never happens
    }
}
```

Here, we get an immutable reference to an integer, and store the value of the integer both before and after incrementing the integer that `b` refers to. The compiler is free to assume that the fundamental rules about borrowing and data races are upheld, which means that `b` can’t possibly refer to the same integer as `a` does. In fact, nothing in the entire program can mutably borrow the integer that `a` refers to as long as `a` is borrowing it. Therefore, the compiler can easily conclude that `*a` will not change and the condition of the `if` statement will never be true, and can completely remove the call to `x` from the program as an optimization.

It’s impossible to write a Rust program that breaks the compiler’s assumptions, other than by using an `unsafe` block to disable some of the compiler’s safety checks.

Undefined Behavior

Languages like C, C++, and Rust have a set of rules that need to be followed to avoid something called *undefined behavior*. For example, one of Rust’s rules is that there may never be more than one mutable reference to any object.

In Rust, it’s only possible to break any of these rules when using `unsafe` code. "Unsafe" doesn’t mean that the code is incorrect or never safe to use, but rather that the compiler is not validating for you that the code is safe. If the code does violate these rules, it is called *unsound*.

The compiler is allowed to assume, without checking, that these rules are never broken. When broken, this results in something called *undefined behavior*, which we need to avoid at all costs. If we allow the compiler to make an assumption that is not actually true, it can easily result in more wrong conclusions about different parts of your code, affecting your whole program.

As a concrete example, let’s take a look at a small snippet that uses the `get_unchecked` method on a slice:

```
let a = [123, 456, 789];
let b = unsafe { a.get_unchecked(index) };
```

The `get_unchecked` method gives us an element of the slice given its index, just like `a[index]`, but allows the compiler to assume the index is always within bounds, without any checks.

This means that in this code snippet, because `a` is of length 3, the compiler may assume that `index` is less than three. It’s up to us to make sure its assumption holds.

If we break this assumption, for example if we run this with `index` equal to 3, anything might happen. It might result in reading from memory whatever was stored in the bytes right after `a`. It might cause the program to crash. It might end up executing some entirely unrelated part of the program. It can cause all kinds of havoc.

Perhaps surprisingly, undefined behavior can even "travel back in time," causing problems in code that precedes it. To understand how that can happen, imagine we had a `match` statement before our previous snippet, as follows:

```
match index {
    0 => x(),
    1 => y(),
    _ => z(index),
}

let a = [123, 456, 789];
let b = unsafe { a.get_unchecked(index) };
```

Because of the unsafe code, the compiler is allowed to assume `index` is only ever 0, 1, or 2. It may logically conclude that the last arm of our `match` statement will only ever match a 2, and thus that `z` is only ever called as `z(2)`. That conclusion might be used not only to optimize the `match`, but also to optimize `z` itself. This can include throwing out unused parts of the code.

If we execute this with an `index` of `3`, our program might attempt to execute parts that have been optimized away, resulting in completely unpredictable behavior, long before we get to the `unsafe` block on the last line. Just like that, undefined behavior can propagate through a whole program, both backwards and forwards, in often very unexpected ways.

When calling any `unsafe` function, read its documentation carefully and make sure you fully understand its *safety requirements*: the assumptions you need to uphold, as the caller, to avoid undefined behavior.

Interior Mutability

The borrowing rules as introduced in the previous section are simple, but can be quite limiting—especially when multiple threads are involved. Following these rules makes communication between threads extremely limited and almost impossible, since no data that's accessible by multiple threads can be mutated.

Luckily, there is an escape hatch: *interior mutability*. A data type with interior mutability slightly bends the borrowing rules. Under certain conditions, those types can allow mutation through an "immutable" reference.

In "Reference Counting", we've already seen one subtle example involving interior mutability. Both `Rc` and `Arc` mutate a reference counter, even though there might be multiple clones all using the same reference counter.

As soon as interior mutable types are involved, calling a reference "immutable" or "mutable" becomes confusing and inaccurate, since some things can be mutated through both. The more accurate terms are "shared" and "exclusive": a *shared reference* (`&T`) can be copied and shared with others, while an *exclusive reference* (`&mut T`) guarantees it's the only *exclusive borrowing* of that `T`. For most types, shared references do not allow mutation, but there are exceptions. Since in this book we will mostly be working with these exceptions, we'll use the more accurate terms in the rest of this book.

> 🔥 *Keep in mind that interior mutability only bends the rules of shared borrowing to allow mutation when shared. It does not change anything about exclusive borrowing. Exclusive borrowing still guarantees that there are no other active borrows. Unsafe code that results in more than one active exclusive reference to something always invokes undefined behavior, regardless of interior mutability.*

Let's take a look at a few types with interior mutability and how they can allow mutation through shared references without causing undefined behavior.

Cell

A `std::cell::Cell<T>` simply wraps a `T`, but allows mutations through a shared reference. To avoid undefined behavior, it only allows you to copy the value out (if `T` is `Copy`), or replace it with another value as a whole. In addition, it can only be used within a single thread.

Let's take a look at an example similar to the one in the previous section, but this time using `Cell<i32>` instead of `i32`:

```
use std::cell::Cell;

fn f(a: &Cell<i32>, b: &Cell<i32>) {
    let before = a.get();
    b.set(b.get() + 1);
    let after = a.get();
    if before != after {
        x(); // might happen
    }
}
```

Unlike last time, it is now possible for the `if` condition to be true. Because a `Cell<i32>` has interior mutability, the compiler can no longer assume its value won't change as long as we have a shared reference to it. Both `a` and `b` might refer to the same value, such that mutating through `b` might affect `a` as well. It may still assume, however, that no other threads are accessing the cells concurrently.

The restrictions on a `Cell` are not always easy to work with. Since it can't directly let us borrow the value it holds, we need to move a value out (leaving something in its place), modify it, then put it back, to mutate its contents:

```
fn f(v: &Cell<Vec<i32>>) {
    let mut v2 = v.take(); // Replaces the contents of the Cell with an empty Vec
    v2.push(1);
    v.set(v2); // Put the modified Vec back
}
```

RefCell

Unlike a regular `Cell`, a `std::cell::RefCell` does allow you to borrow its contents, at a small runtime cost. A `RefCell<T>` does not only hold a `T`, but also holds a counter that keeps track of any outstanding borrows. If you try to borrow it while it is already mutably borrowed (or vice-versa), it will panic, which avoids undefined behavior. Just like a `Cell`, a `RefCell` can only be used within a single thread.

Borrowing the contents of `RefCell` is done by calling `borrow` or `borrow_mut`:

```
use std::cell::RefCell;

fn f(v: &RefCell<Vec<i32>>) {
    v.borrow_mut().push(1); // We can modify the `Vec` directly.
}
```

While `Cell` and `RefCell` can be very useful, they become rather useless when we need to do something with multiple threads. So let's move on to the types that are relevant for concurrency.

Mutex and RwLock

An `RwLock` or *reader-writer lock* is the concurrent version of a `RefCell`. An `RwLock<T>` holds a `T` and tracks any outstanding borrows. However, unlike a `RefCell`, it does not panic on conflicting borrows. Instead, it blocks the current thread—putting it to sleep—while waiting for conflicting borrows to disappear. We'll just have to patiently wait for our turn with the data, after the other threads are done with it.

Borrowing the contents of an `RwLock` is called *locking*. By *locking* it we temporarily block concurrent conflicting borrows, allowing us to borrow it without causing data races.

A `Mutex` is very similar, but conceptually slightly simpler. Instead of keeping track of the number of shared and exclusive borrows like an `RwLock`, it only allows exclusive borrows.

We'll go more into detail on these types in "Locking: Mutexes and RwLocks".

Atomics

The atomic types represent the concurrent version of a `Cell`, and are the main topic of Chapters 2 and 3. Like a `Cell`, they avoid undefined behavior by making us copy values in and out as a whole, without letting us borrow the contents directly.

Unlike a `Cell`, though, they cannot be of arbitrary size. Because of this, there is no generic `Atomic<T>` type for any `T`, but there are only specific atomic types such as `AtomicU32` and `AtomicPtr<T>`. Which ones are available depends on the platform, since they require support from the processor to avoid data races. (We'll dive into that in Chapter 7.)

Since they are so limited in size, atomics often don't directly contain the information that needs to be shared between threads. Instead, they are often used as a tool to make it possible to share other—often bigger—things between threads. When atomics are used to say something about other data, things can get surprisingly complicated.

UnsafeCell

An `UnsafeCell` is the primitive building block for interior mutability.

An `UnsafeCell<T>` wraps a `T`, but does not come with any conditions or restrictions to avoid undefined behavior. Instead, its `get()` method just gives a raw pointer to the value it wraps, which can only be meaningfully used in `unsafe` blocks. It leaves it up to the user to use it in a way that does not cause any undefined behavior.

Most commonly, an `UnsafeCell` is not used directly, but wrapped in another type that provides safety through a limited interface, such as `Cell` or `Mutex`. All types with interior mutability—including all types discussed above—are built on top of `UnsafeCell`.

Thread Safety: Send and Sync

In this chapter, we've seen several types that are not *thread safe*, types that can only be used on a single thread, such as `Rc`, `Cell`, and others. Since that restriction is needed to avoid undefined behavior, it's something the compiler needs to understand and check for you, so you can use these types without having to use `unsafe` blocks.

The language uses two special traits to keep track of which types can be safely used across threads:

Send

A type is Send if it can be sent to another thread. In other words, if ownership of a value of that type can be transferred to another thread. For example, Arc<i32> is Send, but Rc<i32> is not.

Sync

A type is Sync if it can be shared with another thread. In other words, a type T is Sync if and only if a shared reference to that type, &T, is Send. For example, an i32 is Sync, but a Cell<i32> is not. (A Cell<i32> is Send, however.)

All primitive types such as i32, bool, and str are both Send and Sync.

Both of these traits are *auto traits*, which means that they are automatically implemented for your types based on their fields. A struct with fields that are all Send and Sync, is itself also Send and Sync.

The way to opt out of either of these is to add a field to your type that does not implement the trait. For that purpose, the special std::marker::PhantomData<T> type often comes in handy. That type is treated by the compiler as a T, except it doesn't actually exist at runtime. It's a zero-sized type, taking no space.

Let's take a look at the following struct:

```
use std::marker::PhantomData;

struct X {
    handle: i32,
    _not_sync: PhantomData<Cell<()>>,
}
```

In this example, X would be both Send and Sync if handle was its only field. However, we added a zero-sized PhantomData<Cell<()>> field, which is treated as if it were a Cell<()>. Since a Cell<()> is not Sync, neither is X. It is still Send, however, since all its fields implement Send.

Raw pointers (*const T and *mut T) are neither Send nor Sync, since the compiler doesn't know much about what they represent.

The way to opt in to either of the traits is the same as with any other trait; use an impl block to implement the trait for your type:

```
struct X {
    p: *mut i32,
}

unsafe impl Send for X {}
unsafe impl Sync for X {}
```

Note how implementing these traits requires the `unsafe` keyword, since the compiler cannot check for you if it's correct. It's a promise you make to the compiler, which it will just have to trust.

If you try to move something into another thread which is not `Send`, the compiler will politely stop you from doing that. Here is a small example to demonstrate that:

```
fn main() {
    let a = Rc::new(123);
    thread::spawn(move || { // Error!
        dbg!(a);
    });
}
```

Here, we try to send an `Rc<i32>` to a new thread, but `Rc<i32>`, unlike `Arc<i32>`, does not implement `Send`.

If we try to compile the example above, we're faced with an error that looks something like this:

```
error[E0277]: `Rc<i32>` cannot be sent between threads safely
   --> src/main.rs:3:5
    |
3   |     thread::spawn(move || {
    |     ^^^^^^^^^^^^^ `Rc<i32>` cannot be sent between threads safely
    |
    = help: within `[closure]`, the trait `Send` is not implemented for `Rc<i32>`
note: required because it's used within this closure
   --> src/main.rs:3:19
    |
3   |     thread::spawn(move || {
    |                   ^^^^^^^
note: required by a bound in `spawn`
```

The `thread::spawn` function requires its argument to be `Send`, and a closure is only `Send` if all of its captures are. If we try to capture something that's not `Send`, our mistake is caught, protecting us from undefined behavior.

Locking: Mutexes and RwLocks

The most commonly used tool for sharing (mutable) data between threads is a *mutex*, which is short for "mutual exclusion." The job of a mutex is to ensure threads have exclusive access to some data by temporarily blocking other threads that try to access it at the same time.

Conceptually, a mutex has only two states: locked and unlocked. When a thread locks an unlocked mutex, the mutex is marked as locked and the thread can immediately continue. When a thread then attempts to lock an already locked mutex, that operation will *block*. The thread is put to sleep while it waits for the mutex to be unlocked. Unlocking is only possible on a locked mutex, and should be done by the same thread that locked it. If other threads are waiting to lock the mutex, unlocking will cause one of those threads to be woken up, so it can try to lock the mutex again and continue its course.

Protecting data with a mutex is simply the agreement between all threads that they will only access the data when they have the mutex locked. That way, no two threads can ever access that data concurrently and cause a data race.

Rust’s Mutex

The Rust standard library provides this functionality through `std::sync::Mutex<T>`. It is generic over a type `T`, which is the type of the data the mutex is protecting. By making this `T` part of the mutex, the data can only be accessed through the mutex, allowing for a safe interface that can guarantee all threads will uphold the agreement.

To ensure a locked mutex can only be unlocked by the thread that locked it, it does not have an `unlock()` method. Instead, its `lock()` method returns a special type called a `MutexGuard`. This guard represents the guarantee that we have locked the mutex. It behaves like an exclusive reference through the `DerefMut` trait, giving us exclusive access to the data the mutex protects. Unlocking the mutex is done by dropping the guard. When we drop the guard, we give up our ability to access the data, and the `Drop` implementation of the guard will unlock the mutex.

Let’s take a look at an example to see a mutex in practice:

```
use std::sync::Mutex;

fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                    *guard += 1;
                }
            });
        }
    });
    assert_eq!(n.into_inner().unwrap(), 1000);
}
```

Here, we have a `Mutex<i32>`, a mutex protecting an integer, and we spawn ten threads to each increment the integer one hundred times. Each thread will first lock the mutex to obtain a `MutexGuard`, and then use that `guard` to access the integer and modify it. The `guard` is implicitly dropped right after, when that variable goes out of scope.

After the threads are done, we can safely remove the protection from the integer through `into_inner()`. The `into_inner` method takes ownership of the mutex, which guarantees that nothing else can have a reference to the mutex anymore, making locking unnecessary.

Even though the increments happen in steps of one, a thread observing the integer would only ever see multiples of 100, since it can only look at the integer when the mutex is unlocked. Effectively, thanks to the mutex, the one hundred increments together are now a single indivisible—atomic—operation.

To clearly see the effect of the mutex, we can make each thread wait a second before unlocking the mutex:

```
use std::time::Duration;

fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                    *guard += 1;
                }
                thread::sleep(Duration::from_secs(1)); // New!
            });
        }
    });
    assert_eq!(n.into_inner().unwrap(), 1000);
}
```

When you run the program now, you will see that it takes about 10 seconds to complete. Each thread only waits for one second, but the mutex ensures that only one thread at a time can do so.

If we drop the guard—and therefore unlock the mutex—before sleeping one second, we will see it happen in parallel instead:

```
fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                    *guard += 1;
                }
                drop(guard); // New: drop the guard before sleeping!
                thread::sleep(Duration::from_secs(1));
            });
        }
    });
    assert_eq!(n.into_inner().unwrap(), 1000);
}
```

With this change, this program takes only about one second, since now the 10 threads can execute their one-second sleep at the same time. This shows the importance of keeping the amount of time a mutex is locked as short as possible. Keeping a mutex locked longer than necessary can completely nullify any benefits of parallelism, effectively forcing everything to happen serially instead.

Lock Poisoning

The `unwrap()` calls in the examples above relate to *lock poisoning*.

A `Mutex` in Rust gets marked as *poisoned* when a thread panics while holding the lock. When that happens, the `Mutex` will no longer be locked, but calling its `lock` method will result in an `Err` to indicate it has been poisoned.

This is a mechanism to protect against leaving the data that's protected by a mutex in an inconsistent state. In our example above, if a thread would panic after incrementing the integer fewer than 100 times, the mutex would unlock and the integer would be left in an unexpected state where it is no longer a multiple of 100, possibly breaking assumptions made by other threads. Automatically marking the mutex as poisoned in that case forces the user to handle this possibility.

Calling `lock()` on a poisoned mutex still locks the mutex. The `Err` returned by `lock()` contains the `MutexGuard`, allowing us to correct an inconsistent state if necessary.

While lock poisoning might seem like a powerful mechanism, recovering from a potentially inconsistent state is not often done in practice. Most code either disregards poison or uses `unwrap()` to panic if the lock was poisoned, effectively propagating panics to all users of the mutex.

Lifetime of the MutexGuard

While it's convenient that implicitly dropping a guard unlocks the mutex, it can sometimes lead to subtle surprises. If we assign the guard a name with a `let` statement (as in our examples above), it's relatively straightforward to see when it will be dropped, since local variables are dropped at the end of the scope they are defined in. Still, not explicitly dropping a guard might lead to keeping the mutex locked for longer than necessary, as demonstrated in the examples above.

Using a guard *without* assigning it a name is also possible, and can be very convenient at times. Since a `MutexGuard` behaves like an exclusive reference to the protected data, we can directly use it without assigning a name to the guard first. For example, if you have a `Mutex<Vec<i32>>`, you can lock the mutex, push an item into the `Vec`, and unlock the mutex again, in a single statement:

```
list.lock().unwrap().push(1);
```

Any temporaries produced within a larger expression, such as the guard returned by `lock()`, will be dropped at the end of the statement. While this might seem obvious and reasonable, it leads to a common pitfall that usually involves a `match`, `if let`, or `while let` statement. Here is an example that runs into this pitfall:

```
if let Some(item) = list.lock().unwrap().pop() {
    process_item(item);
}
```

If our intention was to lock the list, pop an item, unlock the list, and *then* process the item after the list is unlocked, we made a subtle but important mistake here. The temporary guard is not dropped until the end of the entire `if let` statement, meaning we needlessly hold on to the lock while processing the item.

Perhaps surprisingly, this does not happen for a similar `if` statement, such as in this example:

```
if list.lock().unwrap().pop() == Some(1) {
    do_something();
}
```

Here, the temporary guard does get dropped before the body of the `if` statement is executed. The reason is that the condition of a regular `if` statement is always a plain boolean, which cannot borrow anything. There is no reason to extend the lifetime of temporaries from the condition to the end of the statement. For an `if let` statement, however, that might not be the case. If we had used `front()` rather than `pop()`, for example, `item` would be borrowing from the list, making it necessary to keep the guard around. Since the borrow checker is only really a check and does *not* influence when or in what order things are dropped, the same happens when we use `pop()`, even though that wouldn't have been necessary.

We can avoid this by moving the pop operation to a separate `let` statement. Then the `guard` is dropped at the end of that statement, before the `if let`:

```
let item = list.lock().unwrap().pop();
if let Some(item) = item {
    process_item(item);
}
```

Reader-Writer Lock

A mutex is only concerned with exclusive access. The `MutexGuard` will provide us an exclusive reference (`&mut T`) to the protected data, even if we only wanted to look at the data and a shared reference (`&T`) would have sufficed.

A reader-writer lock is a slightly more complicated version of a mutex that understands the difference between exclusive and shared access, and can provide either. It has three states: unlocked, locked by a single *writer* (for exclusive access), and locked by any number of *readers* (for shared access). It is commonly used for data that is often read by multiple threads, but only updated once in a while.

The Rust standard library provides this lock through the `std::sync::RwLock<T>` type. It works similarly to the standard `Mutex`, except its interface is mostly split in two parts. Instead of a single `lock()` method, it has a `read()` and `write()` method for locking as either a reader or a writer. It comes with two guard types, one for readers and one for writers: `RwLockReadGuard` and `RwLockWriteGuard`. The former only implements `Deref` to behave like a shared reference to the protected data, while the latter also implements `DerefMut` to behave like an exclusive reference.

It is effectively the multi-threaded version of `RefCell`, dynamically tracking the number of references to ensure the borrow rules are upheld.

Both `Mutex<T>` and `RwLock<T>` require `T` to be `Send`, because they can be used to send a `T` to another thread. An `RwLock<T>` additionally requires `T` to also implement `Sync`, because it allows multiple threads to hold a shared reference (`&T`) to the protected data. (Strictly speaking, you can create a lock for a `T` that doesn't fulfill these requirements, but you wouldn't be able to share it between threads as the lock itself won't implement `Sync`.)

The Rust standard library provides only one general purpose `RwLock` type, but its implementation depends on the operating system. There are many subtle variations between reader-writer lock implementations. Most implementations will block new readers when there is a writer waiting, even when the lock is already read-locked. This is done to prevent *writer starvation*, a situation where many readers collectively keep the lock from ever unlocking, never allowing any writer to update the data.

Mutexes in Other Languages

Rust's standard `Mutex` and `RwLock` types look a bit different than those you find in other languages like C or C++.

The biggest difference is that Rust's `Mutex<T>` *contains* the data it is protecting. In C++, for example, `std::mutex` does not contain the data it protects, nor does it even know what it is protecting. This means that it is the responsibility of the user to remember which data is protected and by which mutex, and ensure the right mutex is locked every time "protected" data is accessed. This is useful to keep in mind when reading code involving mutexes in other languages, or when communicating with programmers who are not familiar with Rust. A Rust programmer might talk about "the data inside the mutex," or say things like "wrap it in a mutex," which can be confusing to those only familiar with mutexes in other languages.

If you really need a stand-alone mutex that doesn't contain anything, for example to protect some external hardware, you can use `Mutex<()>`. But even in a case like that, you are probably better off defining a (possibly zero-sized) type to interface with that hardware and wrapping that in a `Mutex` instead. That way, you are still forced to lock the mutex before you can interact with the hardware.

Waiting: Parking and Condition Variables

When data is mutated by multiple threads, there are many situations where they would need to wait for some event, for some condition about the data to become true. For example, if we have a mutex protecting a `Vec`, we might want to wait until it contains anything.

While a mutex does allow threads to wait until it becomes unlocked, it does not provide functionality for waiting for any other conditions. If a mutex was all we had, we'd have to keep locking the mutex to repeatedly check if there's anything in the `Vec` yet.

Thread Parking

One way to wait for a notification from another thread is called *thread parking*. A thread can *park* itself, which puts it to sleep, stopping it from consuming any CPU cycles. Another thread can then *unpark* the parked thread, waking it up from its nap.

Thread parking is available through the `std::thread::park()` function. For unparking, you call the `unpark()` method on a `Thread` object representing the thread that you want to unpark. Such an object can be obtained from the join handle returned by `spawn`, or by the thread itself through `std::thread::current()`.

Let's dive into an example that uses a mutex to share a queue between two threads. In the following example, a newly spawned thread will consume items from the queue, while the main thread will insert a new item into the queue every second. Thread parking is used to make the consuming thread wait when the queue is empty.

```
use std::collections::VecDeque;

fn main() {
    let queue = Mutex::new(VecDeque::new());

    thread::scope(|s| {
        // Consuming thread
        let t = s.spawn(|| loop {
            let item = queue.lock().unwrap().pop_front();
            if let Some(item) = item {
                dbg!(item);
            } else {
                thread::park();
            }
        });

        // Producing thread
        for i in 0.. {
            queue.lock().unwrap().push_back(i);
            t.thread().unpark();
            thread::sleep(Duration::from_secs(1));
        }
    });
}
```

The consuming thread runs an infinite loop in which it pops items out of the queue to display them using the `dbg` macro. When the queue is empty, it stops and goes to sleep using the `park()` function. If it gets unparked, the `park()` call returns, and the `loop` continues, popping items from the queue again until it is empty. And so on.

The producing thread produces a new number every second by pushing it into the queue. Every time it adds an item, it uses the `unpark()` method on the `Thread` object that refers to the consuming thread to unpark it. That way, the consuming thread gets woken up to process the new element.

An important observation to make here is that this program would still be theoretically correct, although inefficient, if we remove parking. This is important, because `park()` does not guarantee that it will only return because of a matching `unpark()`. While somewhat rare, it might have *spurious wake-ups*. Our example deals with that just fine, because the consuming thread will lock the queue, see that it is empty, and directly unlock it and park itself again.

An important property of thread parking is that a call to `unpark()` *before* the thread parks itself does not get lost. The request to unpark is still recorded, and the next time the thread tries to park itself, it clears that request and directly continues without actually going to sleep. To see why that is critical for correct operation, let's go through a possible ordering of the steps executed by both threads:

1. The consuming thread—let's call it C—locks the queue.
2. C tries to pop an item from the queue, but it is empty, resulting in `None`.
3. C unlocks the queue.
4. The producing thread, which we'll call P, locks the queue.
5. P pushes a new item onto the queue.
6. P unlocks the queue again.
7. P calls `unpark()` to notify C that there are new items.
8. C calls `park()` to go to sleep, to wait for more items.

While there is most likely only a very brief moment between releasing the queue in step 3 and parking in step 8, steps 4 through 7 could potentially happen in that moment before the thread parks itself. If `unpark()` would do nothing if the thread wasn't parked, the notification would be lost. The consuming thread would still be waiting, even if there were an item in the queue. Thanks to unpark requests getting saved for a future call to `park()`, we don't have to worry about this.

However, unpark requests don't stack up. Calling `unpark()` two times and then calling `park()` two times afterwards still results in the thread going to sleep. The first `park()` clears the request and returns directly, but the second one goes to sleep as usual.

This means that in our example above it's important that we only park the thread if we've seen the queue is empty, rather than park it after every processed item. While it's extremely unlikely to happen in this example because of the huge (one second) sleep, it's possible for multiple `unpark()` calls to wake up only a single `park()` call.

Unfortunately, this does mean that if `unpark()` is called right after `park()` returns, but before the queue gets locked and emptied out, the `unpark()` call was unnecessary but still causes the next `park()` call to instantly return. This results in the (empty) queue getting locked and unlocked an extra time. While this doesn't affect the correctness of the program, it does affect its efficiency and performance.

This mechanism works well for simple situations like in our example, but quickly breaks down when things get more complicated. For example, if we had multiple consumer threads taking items from the same queue, the producer thread would have no way of knowing which of the consumers is actually waiting and should be woken up. The producer will have to know exactly when a consumer is waiting, and what condition it is waiting for.

Condition Variables

Condition variables are a more commonly used option for waiting for something to happen to data protected by a mutex. They have two basic operations: *wait* and *notify*. Threads can wait on a condition variable, after which they can be woken up when another thread notifies that same condition variable. Multiple threads can wait on the same condition variable, and notifications can either be sent to one waiting thread, or to all of them.

This means that we can create a condition variable for specific events or conditions we're interested in, such as the queue being non-empty, and wait on that condition. Any thread that causes that event or condition to happen then notifies the condition variable, without having to know which or how many threads are interested in that notification.

To avoid the issue of missing notifications in the brief moment between unlocking a mutex and waiting for a condition variable, condition variables provide a way to *atomically* unlock the mutex and start waiting. This means there is simply no possible moment for notifications to get lost.

The Rust standard library provides a condition variable as `std::sync::Condvar`. Its `wait` method takes a `MutexGuard` that proves we've locked the mutex. It first unlocks the mutex and goes to sleep. Later, when woken up, it relocks the mutex and returns a new `MutexGuard` (which proves that the mutex is locked again).

It has two notify functions: `notify_one` to wake up just one waiting thread (if any), and `notify_all` to wake them all up.

Let's modify the example we used for thread parking to use `Condvar` instead:

```
use std::sync::Condvar;

let queue = Mutex::new(VecDeque::new());
let not_empty = Condvar::new();

thread::scope(|s| {
    s.spawn(|| {
        loop {
            let mut q = queue.lock().unwrap();
            let item = loop {
                if let Some(item) = q.pop_front() {
                    break item;
                } else {
                    q = not_empty.wait(q).unwrap();
                }
            };
            drop(q);
            dbg!(item);
        }
    });

    for i in 0.. {
        queue.lock().unwrap().push_back(i);
        not_empty.notify_one();
        thread::sleep(Duration::from_secs(1));
    }
});
```

We had to change a few things:

- We now not only have a `Mutex` containing the queue, but also a `Condvar` to communicate the "not empty" condition.
- We no longer need to know which thread to wake up, so we don't store the return value from `spawn` anymore. Instead, we notify the consumer through the condition variable with the `notify_one` method.
- Unlocking, waiting, and relocking is all done by the `wait` method. We had to restructure the control flow a bit to be able to pass the guard to the `wait` method, while still dropping it before processing an item.

Now we can spawn as many consuming threads as we like, and even spawn more later, without having to change anything. The condition variable takes care of delivering the notifications to whichever thread is interested.

If we had a more complicated system with threads that are interested in different conditions, we could define a `Condvar` for each condition. For example, we could define one to indicate the queue is non-empty and another one to indicate it is empty. Then each thread would wait for whichever condition is relevant to what they are doing.

Normally, a `Condvar` is only ever used together with a single `Mutex`. If two threads try to concurrently `wait` on a condition variable using two different mutexes, it might cause a panic.

A downside of a `Condvar` is that it only works when used together with a `Mutex`, but for most use cases that is perfectly fine, as that's exactly what's already used to protect the data anyway.

Both `thread::park()` and `Condvar::wait()` also have a variant with a time limit: `thread::park_timeout()` and `Condvar::wait_timeout()`. These take a `Duration` as an extra argument, which is the time after which it should give up waiting for a notification and unconditionally wake up.

Summary

- Multiple threads can run concurrently within the same program and can be spawned at any time.
- When the main thread ends, the entire program ends.
- Data races are undefined behavior, which is fully prevented (in safe code) by Rust's type system.
- Data that is `Send` can be sent to other threads, and data that is `Sync` can be shared between threads.
- Regular threads might run as long as the program does, and thus can only borrow `'static` data such as statics and leaked allocations.
- Reference counting (`Arc`) can be used to share ownership to make sure data lives as long as at least one thread is using it.
- Scoped threads are useful to limit the lifetime of a thread to alllow it to borrow non-`'static` data, such as local variables.
- `&T` is a *shared* reference. `&mut T` is an *exclusive* reference. Regular types do not allow mutation through a shared reference.
- Some types have interior mutability, thanks to `UnsafeCell`, which allows for mutation through shared references.
- `Cell` and `RefCell` are the standard types for single-threaded interior mutability. Atomics, `Mutex`, and `RwLock` are their multi-threaded equivalents.
- `Cell` and atomics only allow replacing the value as a whole, while `RefCell`, `Mutex`, and `RwLock` allow you to mutate the value directly by dynamically enforcing access rules.
- Thread parking can be a convenient way to wait for some condition.
- When a condition is about data protected by a `Mutex`, using a `Condvar` is more convenient, and can be more efficient, than thread parking.

Next: Chapter 2. Atomics