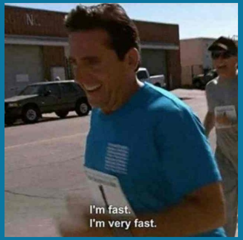
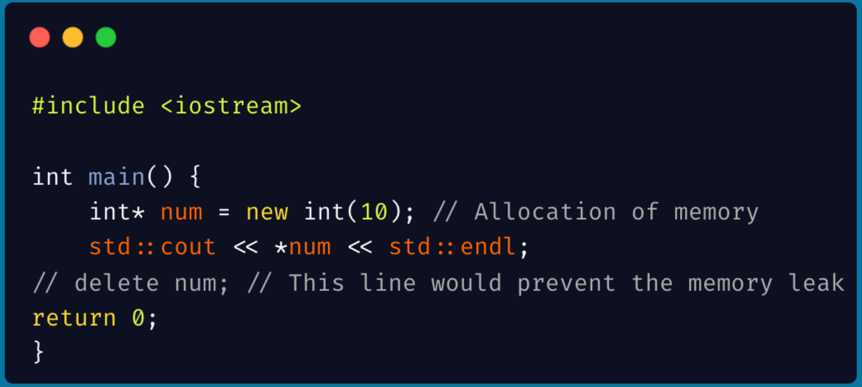


✅High Performant ✅
⛔Memory-Safe✅

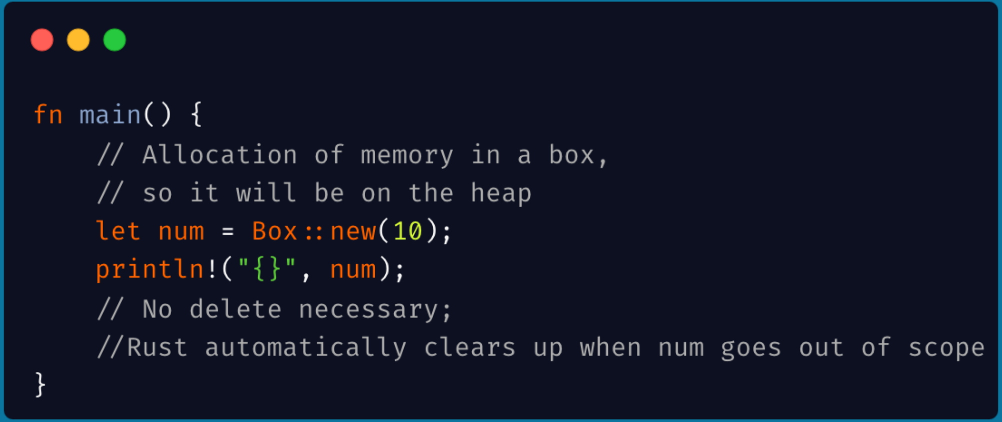
C++ has some safe patterns,
but these are not enforced.

70% of all bugs are memory leaks*



```cpp
#include <iostream>

int main() {
    int* num = new int(10); // Allocation of memory
    std::cout << *num << std::endl;
// delete num; // This line would prevent the memory leak
return 0;
}
```

Rust enforces OBRM* (AKA RAII)
„Roughly speaking the pattern is as follows: to acquire a resource, you create an object that manages it. To release the resource, you simply destroy the object, and it cleans up the resource for you. The most common "resource" this pattern manages is simply memory. Box, Rc, and basically everything in std::collections is a convenience to enable



```rust
fn main() {
    // Allocation of memory in a box,
    // so it will be on the heap
    let num = Box::new(10);
    println!("{}", num);
    // No delete necessary;
    //Rust automatically clears up when num goes out of scope
}
```

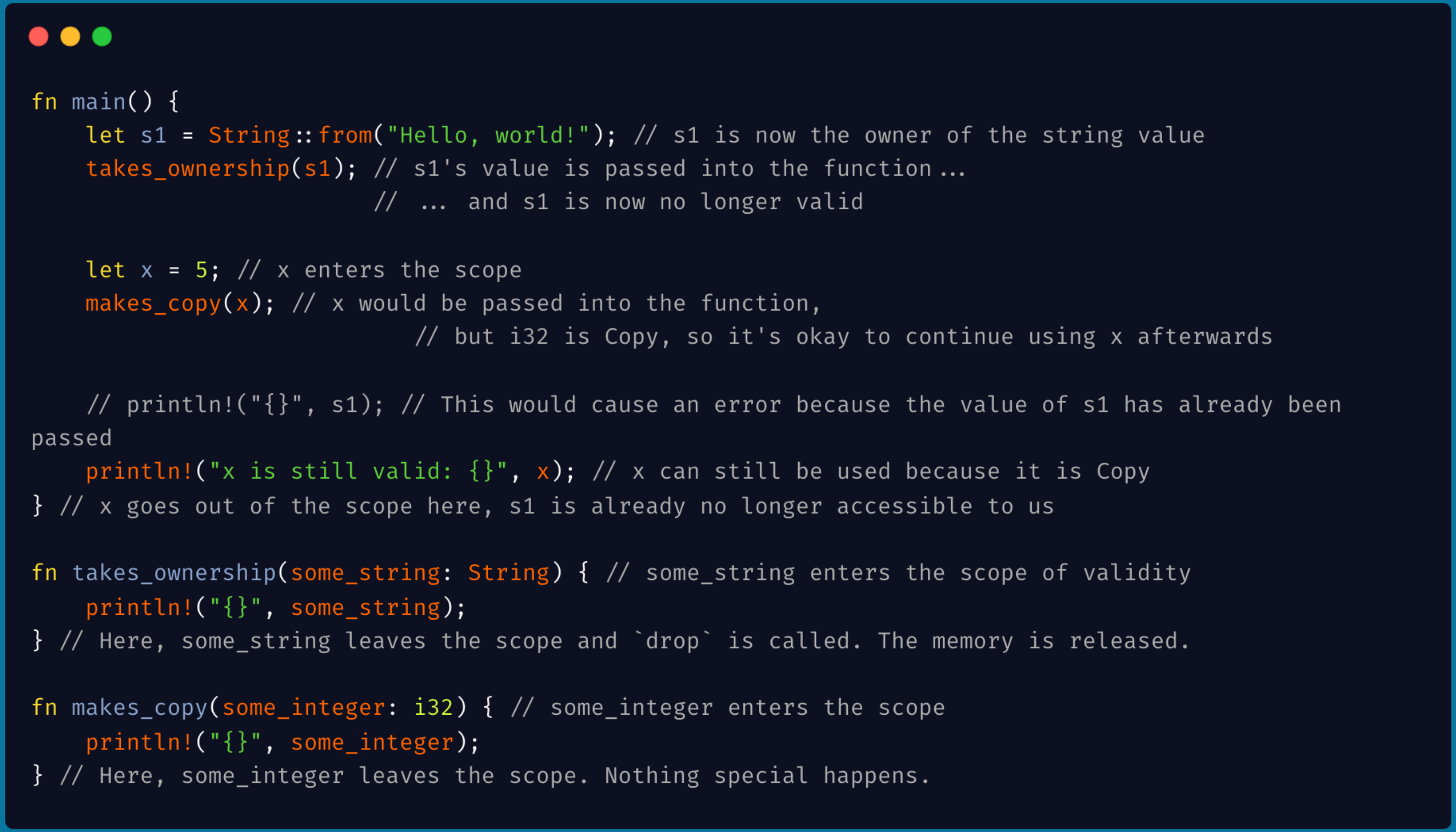
*https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

*https://doc.rust-lang.org/nomicon/obrm.html

What makes Rust memory safe?

Ownership Model

Rust uses a unique concept called ownership, which aims to prevent memory leaks and other memory errors. Each value in Rust has an owner, and there can only be one owner at a time. If the owner goes out of scope, the value is deleted.



```rust
fn main() {
    let s1 = String::from("Hello, world!"); // s1 is now the owner of the string value
    takes_ownership(s1); // s1's value is passed into the function...
                         // ... and s1 is now no longer valid

    let x = 5; // x enters the scope
    makes_copy(x); // x would be passed into the function,
                            // but i32 is Copy, so it's okay to continue using x afterwards

    // println!("{}", s1); // This would cause an error because the value of s1 has already been
passed
    println!("x is still valid: {}", x); // x can still be used because it is Copy
} // x goes out of the scope here, s1 is already no longer accessible to us

fn takes_ownership(some_string: String) { // some_string enters the scope of validity
    println!("{}", some_string);
} // Here, some_string leaves the scope and `drop` is called. The memory is released.

fn makes_copy(some_integer: i32) { // some_integer enters the scope
    println!("{}", some_integer);
} // Here, some_integer leaves the scope. Nothing special happens.
```

Why you should choose Rust for y-platform projects

Platform Support: Rust provides strong support for a wide range of platforms, including Windows, macOS, Linux, and various Unix flavors. This also extends to less common targets like various ARM platforms, WebAssembly, and embedded systems.

Zero Runtime: Rust has no runtime or garbage collector, making it a good choice for systems programming across different platforms. This lean runtime environment helps to ensure consistent performance across different systems.

Standard Library Abstractions: Rust's standard library abstracts over platform-specific details, offering a consistent interface for common functionalities like file handling, network programming, threading, and more. This simplifies development by allowing code to be written once and work across different platforms.

Cargo and Crates: Rust's package manager and build system, Cargo, along with crates.io (the Rust package registry), provide a vast ecosystem of libraries (crates) that can be easily integrated into projects. Many of these crates are designed to be cross-platform and handle platform-specific details internally, further easing the development process.

Robust Compilation: Rust's strong type system and emphasis on safety (especially around memory safety and concurrency) reduce the number of platform-specific bugs, leading to more robust applications.

Community and Ecosystem Support: The Rust community places a high value on cross-platform compatibility, and many of the most popular crates in the ecosystem are rigorously tested on multiple platforms.

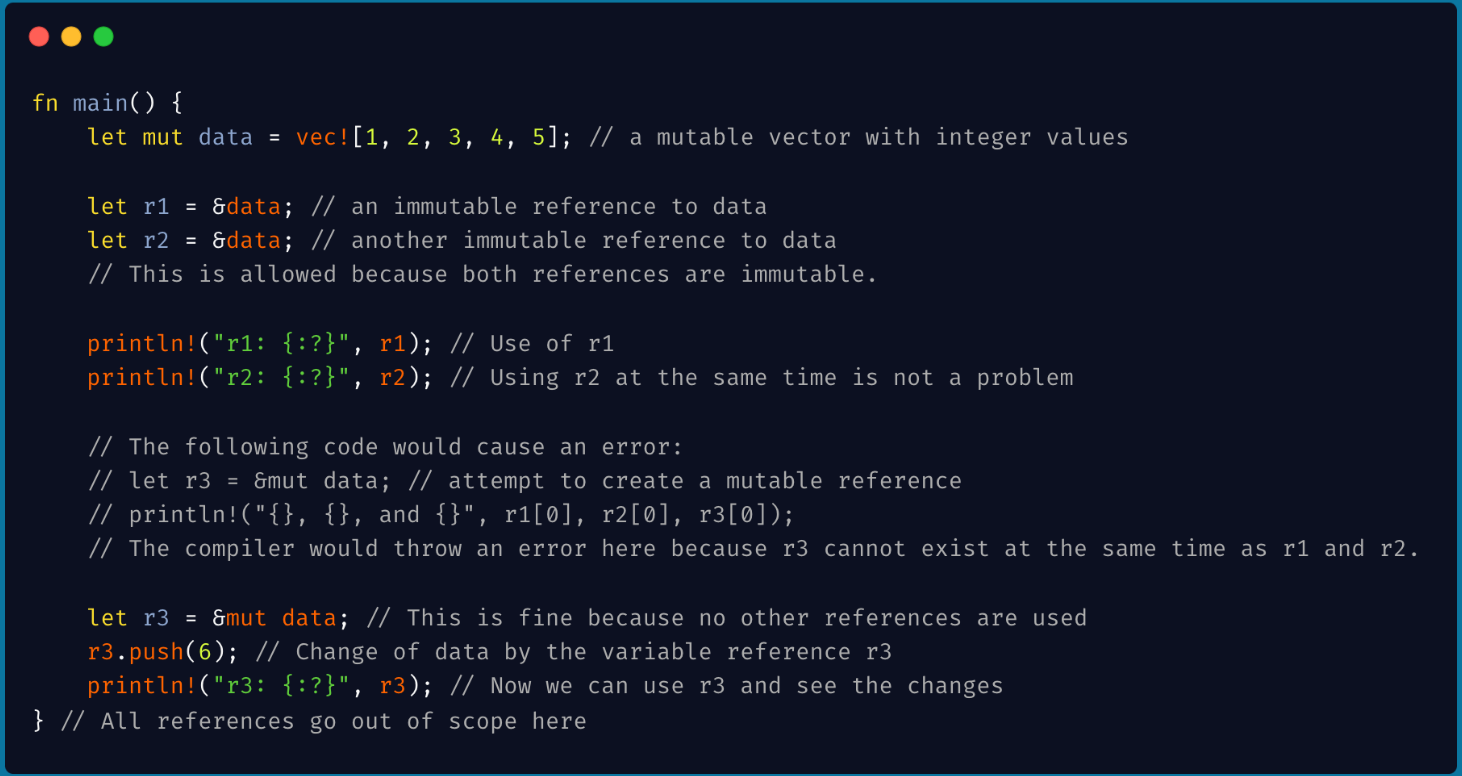
FFI (Foreign Function Interface): Rust provides an FFI to C, allowing it to interoperate with code written in C and other languages that can interface with C. This is valuable for cross-platform development, as it allows Rust to leverage existing C libraries that already target multiple platforms.

Compilation Targets: Rust supports cross-compilation, allowing developers to compile their code on one platform and target another, which is a significant advantage in cross-platform development.

What makes Rust memory safe?

Borrow Checker

Rust's compiler has a feature called Borrow Checker. This ensures that references are always valid and prevents simultaneous write access, which avoids race conditions.

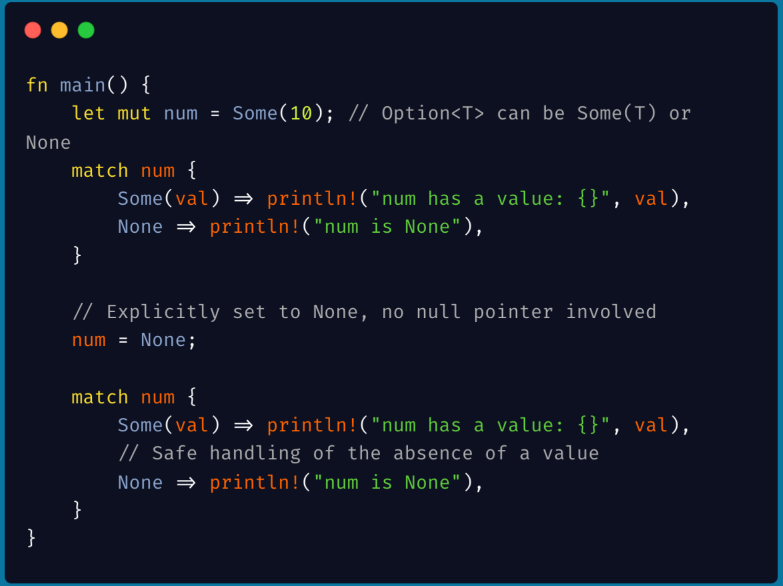


```rust
fn main() {
    let mut data = vec![1, 2, 3, 4, 5]; // a mutable vector with integer values

    let r1 = &data; // an immutable reference to data
    let r2 = &data; // another immutable reference to data
    // This is allowed because both references are immutable.

    println!("r1: {:?}", r1); // Use of r1
    println!("r2: {:?}", r2); // Using r2 at the same time is not a problem

    // The following code would cause an error:
    // let r3 = &mut data; // attempt to create a mutable reference
    // println!("{}, {}, and {}", r1[0], r2[0], r3[0]);
    // The compiler would throw an error here because r3 cannot exist at the same time as r1 and r2.

    let r3 = &mut data; // This is fine because no other references are used
    r3.push(6); // Change of data by the variable reference r3
    println!("r3: {:?}", r3); // Now we can use r3 and see the changes
} // All references go out of scope here
```

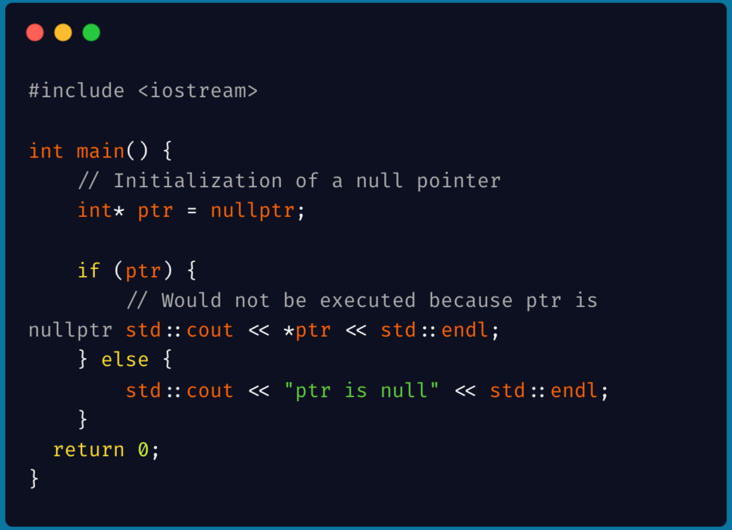
What makes Rust memory safe?

No Null pointer

Rust does not allow null pointers, which is a common source of errors in other languages. Instead, it uses the Option<T> type to safely handle the presence or absence of a value.



```
fn main() {
    let mut num = Some(10); // Option<T> can be Some(T) or
None
    match num {
        Some(val) => println!("num has a value: {}", val),
        None => println!("num is None"),
    }

    // Explicitly set to None, no null pointer involved
    num = None;

    match num {
        Some(val) => println!("num has a value: {}", val),
        // Safe handling of the absence of a value
        None => println!("num is None"),
    }
}
```

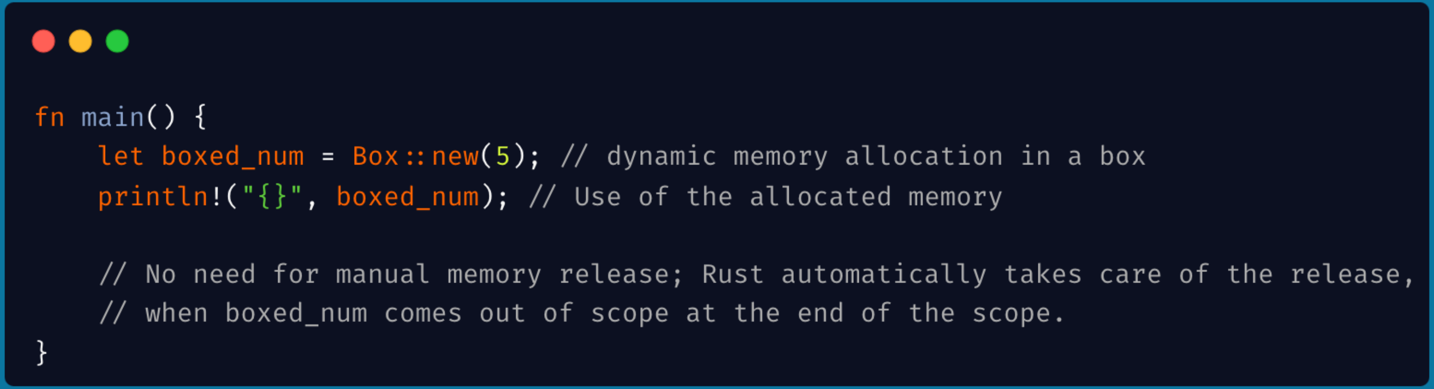


```
#include <iostream>

int main() {
    // Initialization of a null pointer
    int* ptr = nullptr;

    if (ptr) {
        // Would not be executed because ptr is
nullptr std::cout << *ptr << std::endl;
    } else {
        std::cout << "ptr is null" << std::endl;
    }
  return 0;
}
```

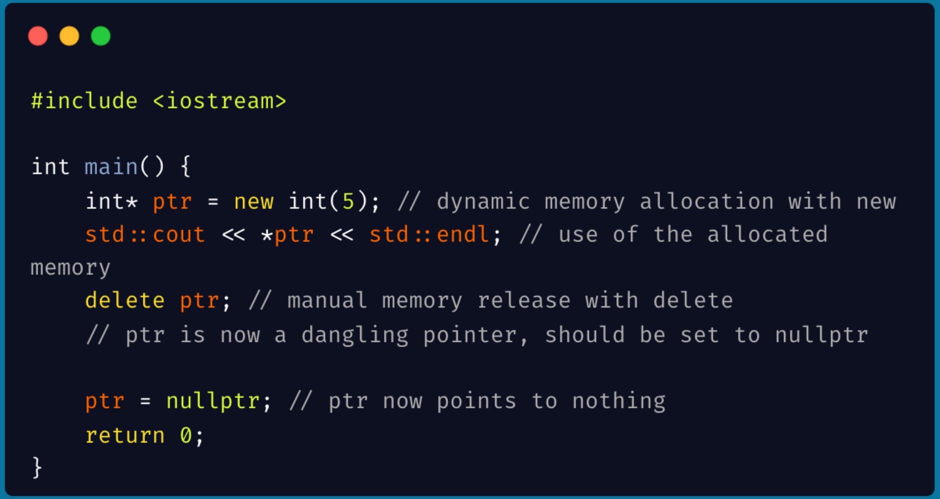
What makes Rust memory safe?

No manual memory management

Rust does not require manual memory management like malloc/free in C or new/delete in C++. The memory is managed automatically, which prevents many errors that often occur in C/C++.



```rust
fn main() {
    let boxed_num = Box::new(5); // dynamic memory allocation in a box
    println!("{}", boxed_num); // Use of the allocated memory

    // No need for manual memory release; Rust automatically takes care of the release,
    // when boxed_num comes out of scope at the end of the scope.
}
```



```cpp
#include <iostream>

int main() {
    int* ptr = new int(5); // dynamic memory allocation with new
    std::cout << *ptr << std::endl; // use of the allocated
memory
    delete ptr; // manual memory release with delete
    // ptr is now a dangling pointer, should be set to nullptr

    ptr = nullptr; // ptr now points to nothing
    return 0;
}
```

Ownership

move

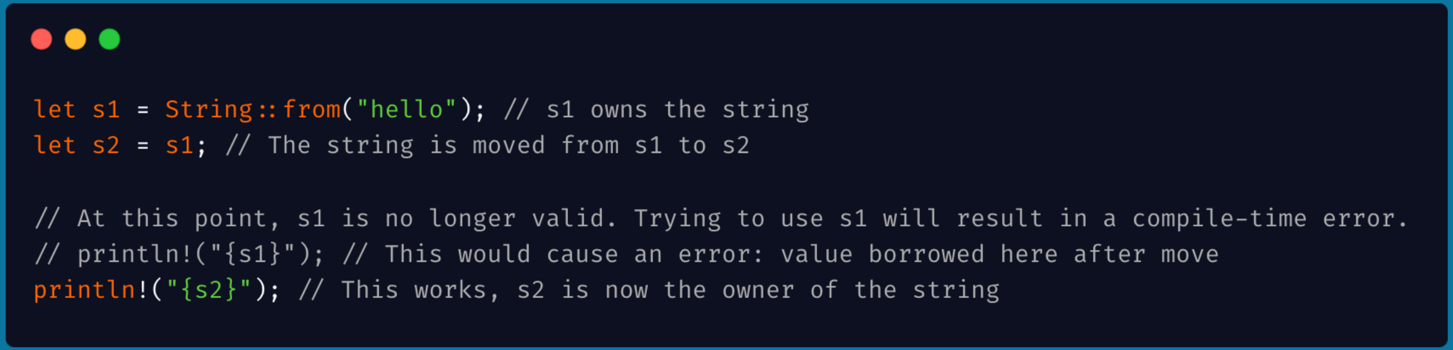
In Rust, a "move" occurs when the ownership of a value is transferred from one variable to another. This concept is crucial to Rust's system of ownership, which is a set of rules the compiler checks at compile time to ensure memory safety.

Transfer of Ownership: When you assign a value of a non-Copy type from one variable to another, the ownership of that value is moved to the new variable. After the move, the old variable can no longer be used to access the value.

Prevention of Double Free Errors: Moves are Rust's way of preventing "double free" errors. Because the original variable no longer owns the value after it's been moved, it can't accidentally deallocate the value when it goes out of scope. Only the new owner will be able to do that.

Efficient Data Management: Moves are also a performance feature. Instead of deep-copying large amounts of data when assigning from one variable to another, Rust moves the data by default, which is a very fast operation because it involves copying the pointer, length, and capacity without copying the actual data.

Compile-Time Check: The Rust compiler enforces move semantics. If you try to use a variable after its value has been moved, the compiler will throw an error and prevent the program from compiling.



```rust
let s1 = String::from("hello"); // s1 owns the string
let s2 = s1; // The string is moved from s1 to s2

// At this point, s1 is no longer valid. Trying to use s1 will result in a compile-time error.
// println!("{s1}"); // This would cause an error: value borrowed here after move
println!("{s2}"); // This works, s2 is now the owner of the string
```

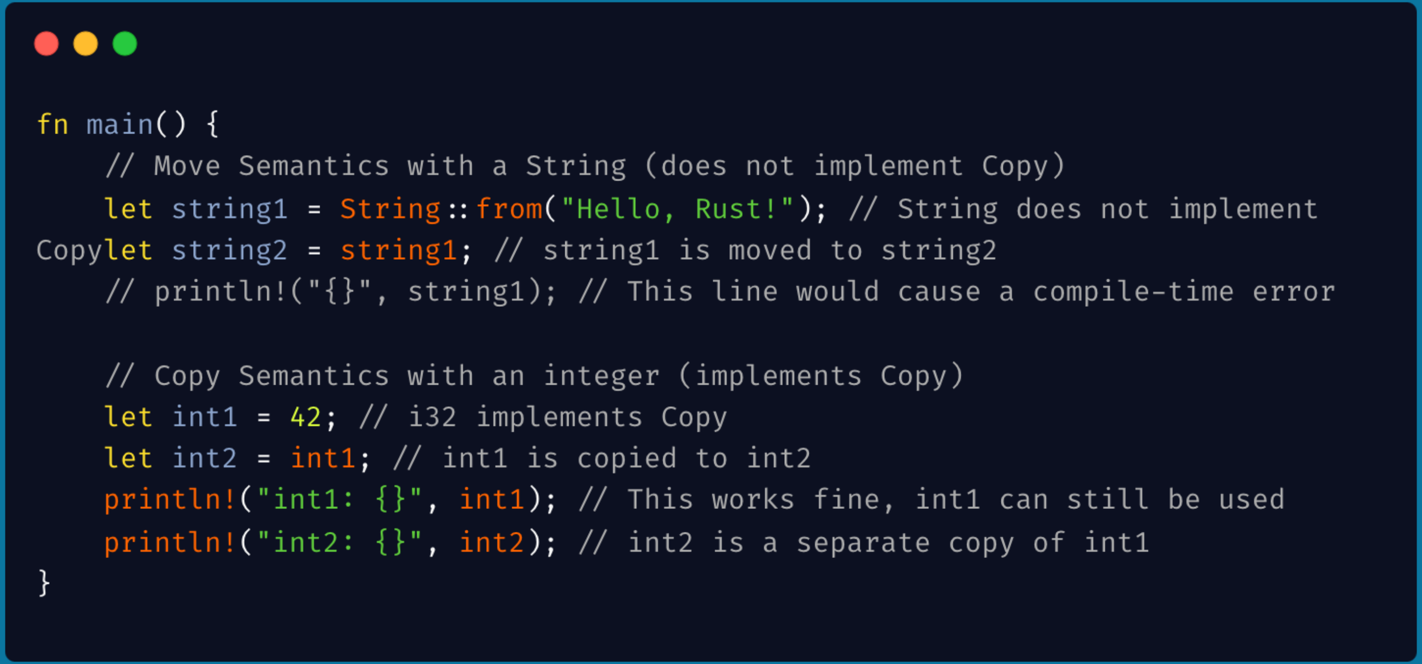
Ownership

move vs. copy

In Rust, whether a value is moved or copied depends on the trait implemented by the data type:

Move Semantics: By default, Rust uses move semantics. When you assign a value of a type to another variable, the ownership of that value is moved to the new variable. After the move, the old variable is no longer valid and cannot be used. Move semantics are used for types that implement the Drop trait, which means they require some cleanup (like freeing memory) when they go out of scope. Most complex types in Rust, like String, Vec<T>, and user-defined structs, fall into this category.

Copy Semantics: Some types in Rust implement the Copy trait. For these types, when you assign a value to another variable, a copy of the value is made, and both variables can be used independently after the assignment. The Copy trait is typically implemented by simple, scalar value types like i32, u64, f32, f64, char, bool, and tuples composed only of types that are also Copy. For example, (i32, i32) is Copy, but (i32, String) is not because String is not Copy.

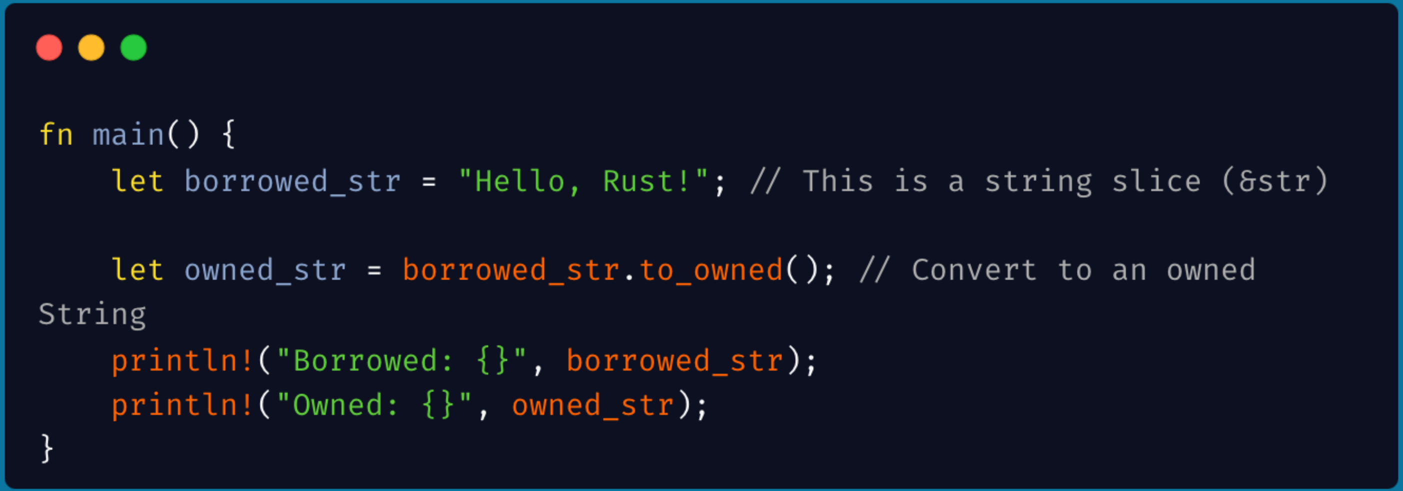


```rust
fn main() {
    // Move Semantics with a String (does not implement Copy)
    let string1 = String::from("Hello, Rust!"); // String does not implement
Copylet string2 = string1; // string1 is moved to string2
    // println!("{}", string1); // This line would cause a compile-time error

    // Copy Semantics with an integer (implements Copy)
    let int1 = 42; // i32 implements Copy
    let int2 = int1; // int1 is copied to int2
    println!("int1: {}", int1); // This works fine, int1 can still be used
    println!("int2: {}", int2); // int2 is a separate copy of int1
}
```

Ownership

to_owned()

The .to_owned() method in Rust is used to create an owned copy of a data value, most commonly with types that implement the Clone trait. It's often used with borrowed data types, like string slices (&str), to create an owned String. This method is particularly useful because it works with any type that implements the ToOwned trait, which includes common types like String, Vec, and others.



```
fn main() {
    let borrowed_str = "Hello, Rust!"; // This is a string slice (&str)

    let owned_str = borrowed_str.to_owned(); // Convert to an owned
String
    println!("Borrowed: {}", borrowed_str);
    println!("Owned: {}", owned_str);
}
```

Ownership vs Borrowing

Ownership

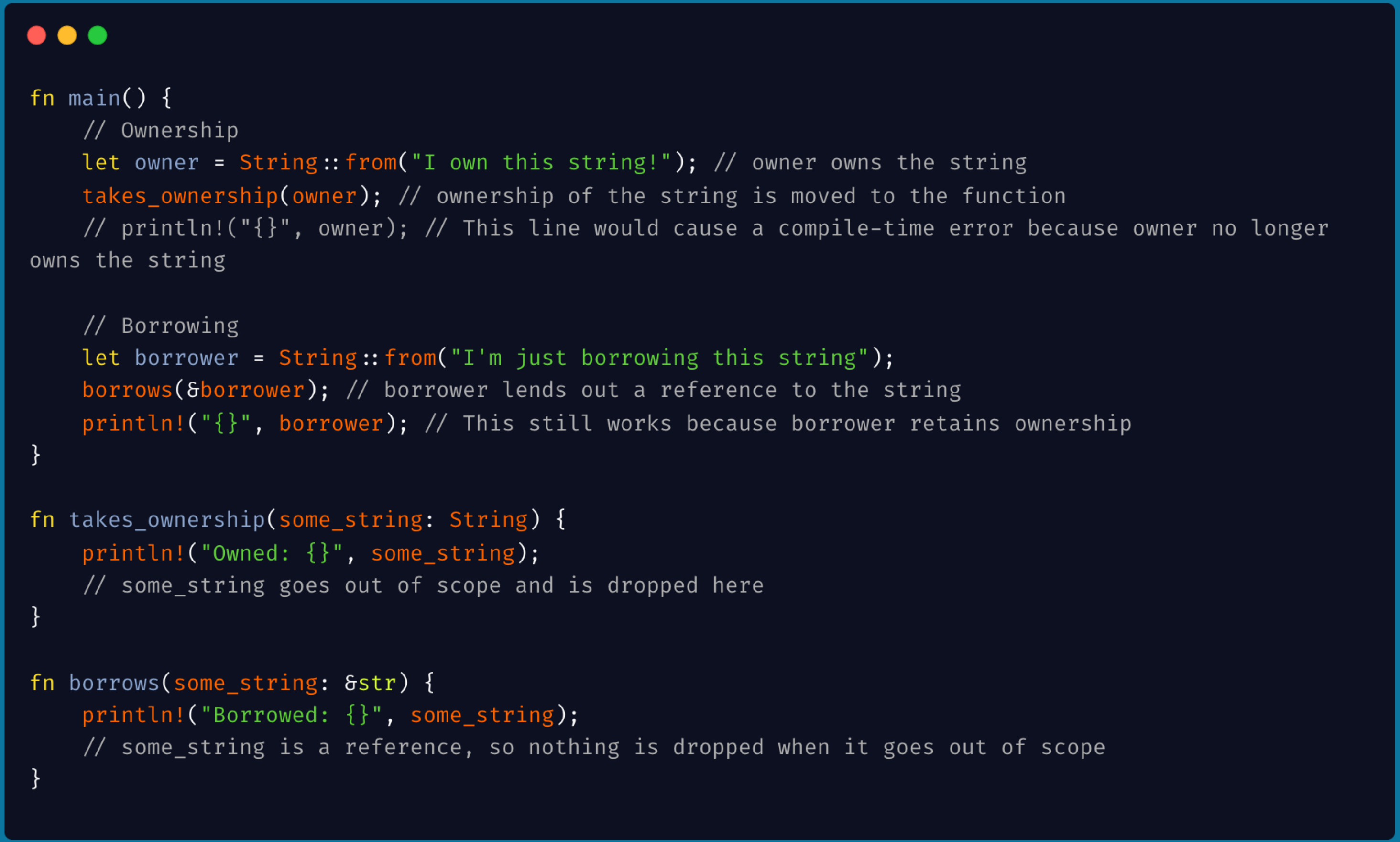
The owner variable owns the String it is assigned.

When we pass owner to the takes_ownership function, the ownership of the String is transferred to the function.

After the call to takes_ownership, owner is no longer valid since its ownership has been moved. Trying to use owner after this point would result in a compile-time error.

Borrowing

The borrower variable owns another String. When we pass &borrower to the borrows function, we are passing a **reference** to the String, **not the String itself**. This is borrowing.
After the call to borrows, borrower is still valid and can be used because only a reference to it was passed to the function, not the ownership.



```rust
fn main() {
    // Ownership
    let owner = String::from("I own this string!"); // owner owns the string
    takes_ownership(owner); // ownership of the string is moved to the function
    // println!("{}", owner); // This line would cause a compile-time error because owner no longer owns the string

    // Borrowing
    let borrower = String::from("I'm just borrowing this string");
    borrows(&borrower); // borrower lends out a reference to the string
    println!("{}", borrower); // This still works because borrower retains ownership
}

fn takes_ownership(some_string: String) {
    println!("Owned: {}", some_string);
    // some_string goes out of scope and is dropped here
}

fn borrows(some_string: &str) {
    println!("Borrowed: {}", some_string);
    // some_string is a reference, so nothing is dropped when it goes out of scope
}
```