An Overview of the RUST Programming Language

Meet Your Presenter!

Dr. Jim Anderson
Professor, Computer Science, University of South Florida
Chairman of FWCS Computer Society

What Will You Get Out Of Tonight's Presentation?

- Understanding of why Rust has become popular
- An introduction to the Rust programming language
- Ownership: Move, Clone, Copy
- References
- Borrowing

Image Credit: https://clipart-library.com

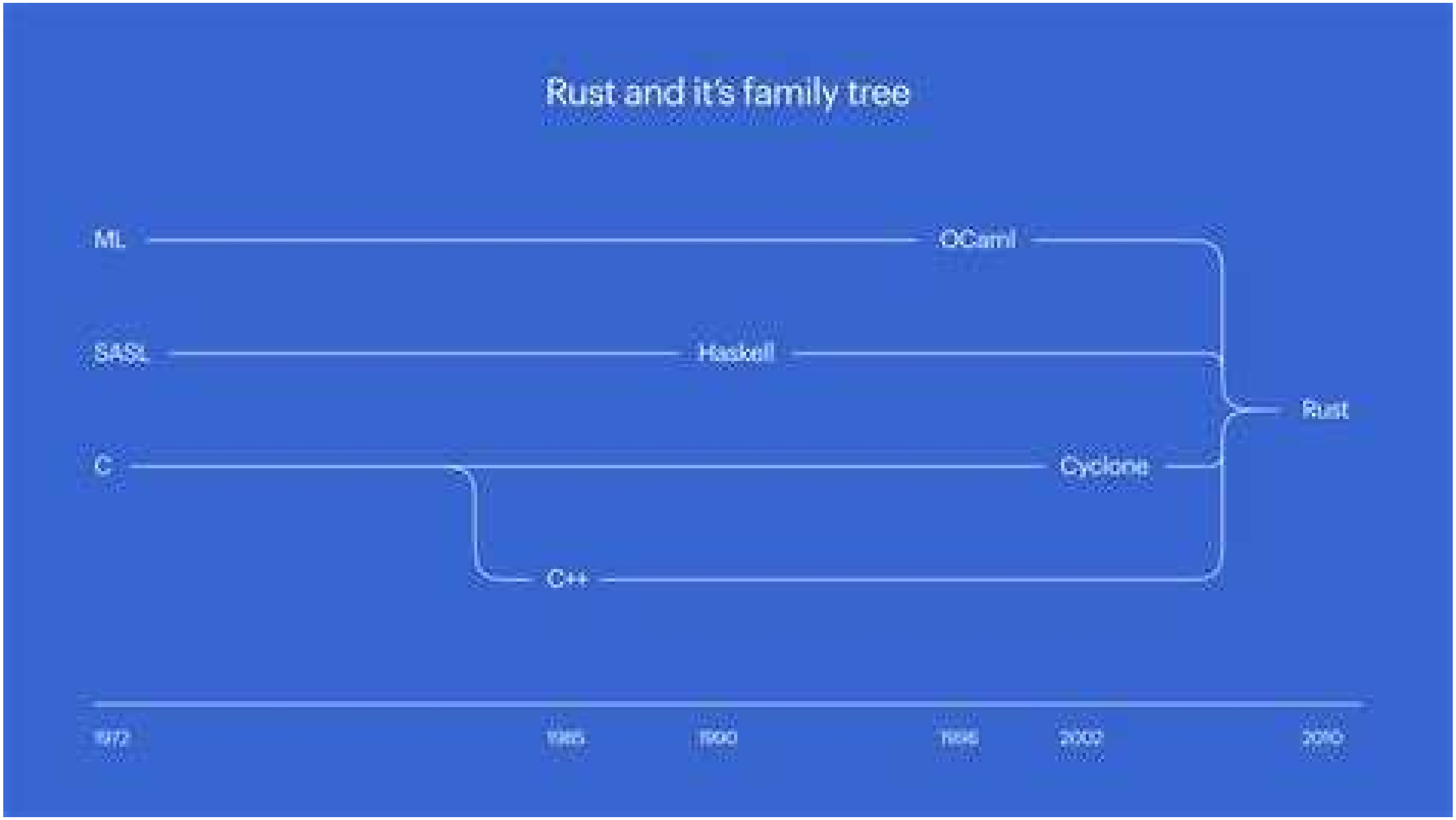
History Of Rust

- Rust started as a personal language that was developed by Graydon Hoare who worked for Mozilla in 2006.
- Rust was officially recognized as a Mozilla sponsored project in 2009 and was first publicly announced in 2010.
- The first pre-alpha version of Rust was released in January of 2012.
- The current stable version of Rust is version 1.16.
- The operating systems supported by Rust include: Linux, Windows, macOS, Android, IOS, etc.

What Is Rust?

- Graydon Hoare called Rust a "safe, concurrent, and practical language" that supports the functional and imperative paradigms.
- Rust’s syntax is comparable to that of the C++ programming language.
- Rust is free and open-source software, which means that anybody may use it for free, and the source code is openly provided so that anyone can enhance the product’s design.
- There is no such thing as direct memory management, such as calloc or malloc - Rust manages memory internally.
- Rust was developed to deliver excellent performance comparable to C and C++ while prioritizing code safety, which is the Achilles’ heel of the other two languages.

Rust Is Built On Other Langauges



Why Use Rust?

- Rust is a strongly typed programming language that prioritizes speed and safety and extraordinarily safe concurrency and memory management.
- Rust tackles two long-standing concerns for C/C++ developers: memory errors and concurrent programming.
- This is regarded as its primary advantage.
- Rust manages memory internally - trash collection is not required.
- In Rust, each reference has a lifespan, which specifies the scope for which that reference is valid.
- Over the last 12 years memory safety concerns have accounted for over 70% of all security flaws in Microsoft products, the necessity of proper memory management becomes instantly clear.

Why Use Rust?

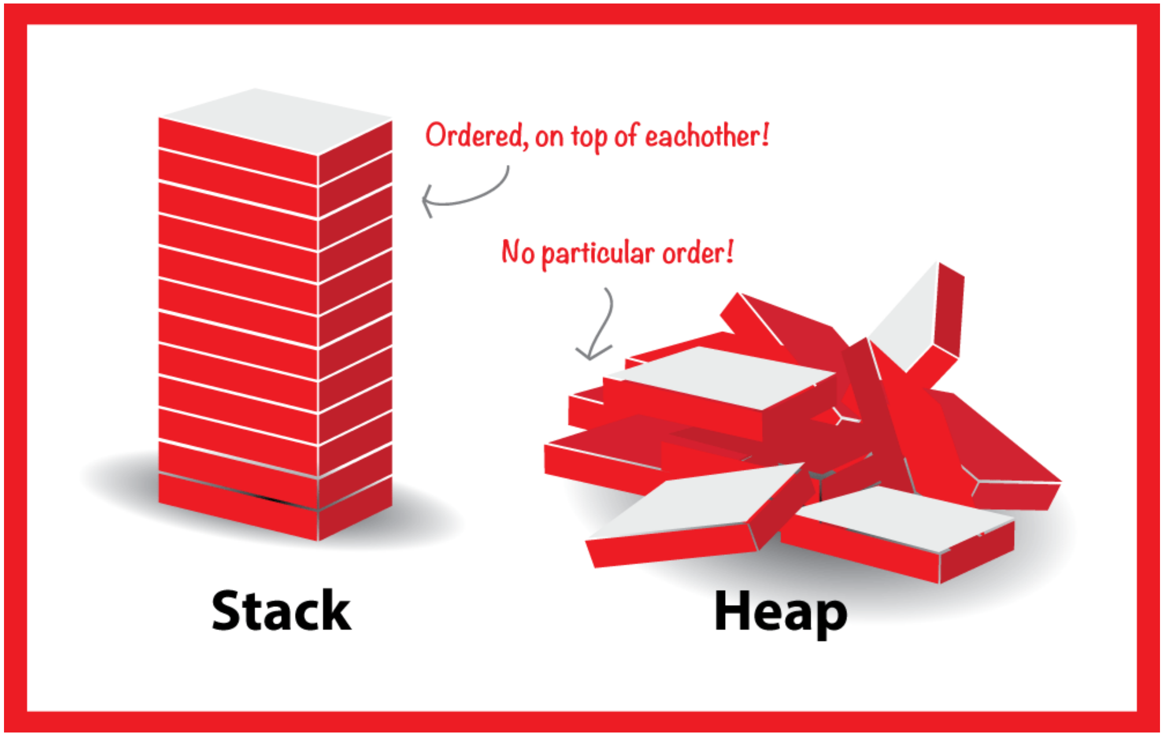
- Like C, Rust helps control low-level details as a systems programming language.
- Using Rust means that we're safe against resource leakage problems.
- Because Rust does not have an active garbage collector, other programming languages may utilize its projects as libraries via foreign-function interfaces.
- This is a good case for existing projects where high performance while preserving memory safety is crucial.
- In such cases, Rust code may replace select areas of software where speed is critical without rebuilding the entire product.

Why Use Rust – The NSA Memo

- On 11/10/22 the NSA released a memo entitled "Software Memory Safety" Cybersecurity Information Sheet
- In it they made the following statements:
  - Memory issues in software comprise a large portion of the exploitable vulnerabilities in existence.
  - NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible.
  - Memory safe languages were identified as being: C#, Go, Java®, Ruby™, Rust®, and Swift®

The Stack & The Heap

All data stored on the stack must have a known fixed size at compile time. FIFO.



Data without a known size at compile time or with a size that may change will be stored on the heap.

- Rust stores variables on either its stack or its heap
- The behavior (speed, size, etc.) is different between the two options.
- We will discuss this in detail later on.

Image Credit: https://medium.com/@Miguel_Grillo/stack-vs-heap-and-the-virtual-memory-ea075ea6e3fc

HELLO WORLD AND VARIABLES

"Hello World" In Rust

- The Rust Hello World program looks like this:

```
fn main() {
        println!("Hello, world!");
}
```

```
> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/my-project`
Hello, world!
>
```

Picking Apart "Hello World"

- **fn**
  - The fn is short for "function." In Rust (and most other programming languages), a function means "tell me some information, and I'll do some things and give you an answer."
- **main**
  - The main function is a special function: just like in C, it's where your program starts.
- **()**
  - These parentheses are the parameter list for this function. It's empty right now, meaning there are no parameters. Every left parenthesis has a matching right.
- **{ }**
  - These are called curly braces or brackets. We actually need to give the main function a body. The body lives inside these braces. The body will say what the main function actually does. Every left curly brace has a matching right.

Picking Apart "Hello World"

- **println!**
  - This is a macro. It means "print and add a new line." Macros are very similar to functions - the difference is that it ends with an exclamation point (!). Rust has a print! Macro that stays on the same line after printing.
- **("Hello, world!")**
  - This is the parameter list for the macro call. We're saying "call this macro called **println** with these parameters."
- **"Hello, world!"**
  - This is a string. Just like in C strings are a bunch of letters (or characters) put together. We put them inside the double quotes (**"**) to mark them as strings.
- **;**
  - This is a semicolon. It completes a statement. Statements do something specific. In this case, it's calling the macro.

Rust Interpolation

- Def: **<u>interpolation</u>** - the insertion of something of a different nature into something else.
- In Rust, in order to include other values in what we are printing out, we can interpolate them.
- Example:

```
fn main() {
    println!("My name is {}", "Michael");
}
```

```
> cargo run
   Blocking waiting for file lock on build directory
  Compiling my-project v0.1.0 (/home/runner/Rust-Hello-World)
   Finished dev [unoptimized + debuginfo] target(s) in 1.88s
    Running `target/debug/my-project`
My name is Michael
>
```

Interpolation

- We now pass two parameters to the **println** macro: the first string is "**My name is {}**".
- The **println** macro has special support for the **{}** braces.
- It means: see that next parameter? Stick it in here.
- This program is going to print "**My name is Michael**".
- And we separate the parameters by putting a comma.

println! Macro

- The println! macro accepts two parameters:
  1. A unique syntax {}, which acts as a placeholder
  2. The name of a variable or a constant
- The variable's value will be used to replace the placeholder.
- Example:
  println!("company rating on level 5:{}",rating_float);

ANATOMY OF RUST

Variables In Rust

- Just like in C, a variable is a named storage location that programs may access.
- A variable is a type of data structure that allows programs to store values.
- In Rust, variables are always linked with a specific data type.
- The data type dictates both the variable’s memory size and layout, the range of values stored inside that memory, and the set of operations on the variable.

Variable Naming Syntax

- When declaring a variable in Rust, the data type is optional.
- The value assigned to the variable determines the data type.
- The syntax for defining variables is as follows:
  - let variable_name = value; // no type-specified
  - let variable_name:dataType = value; //type-specified
- Example:

```
fn main() {
        let fees=35000;
        let salary:f64=45000.00;
        println!("fees is {} and salary is {}",fees,salary);
}
```

Rust Scalar Types

- A scalar type is a value that has just one value.
- For instance:
  10,
  3.14,
  ’c’
- Rust has four distinct scalar types.

1. Integer
2. Floating point
3. Booleans
4. Characters

Declaring Variables

- To declare a variable, we use the let keyword.
- Example:

```
fn main() {
    let company_string="Amazon"; // string type
    let rating_float=3.5; // float type
    let is_growing_boolean=true; // boolean type
    let icon_char='♥'; //unicode character type
    println!("company name:{}",company_string);
    println!("company rating on 5:{}",rating_float);
    println!("company is growing :{}",is_growing_boolean);
    println!("company icon:{}",icon_char);
}
```

- The data type of the variables in the example will deduce from the values assigned to them.
- Rust, for instance, will assign the string data type to the variable company string, the float data type to rating float, and so on.

Immutable

- By default, variables are immutable in Rust.
- In other words, the value of the variable cannot change once a value is bound to a variable name.
- Example:

```
fn main() {
        let fees=25_000;
        println!("fees is {} ",fees);
        fees=35_000;
        println!("fees changed is {}",fees);
}
```

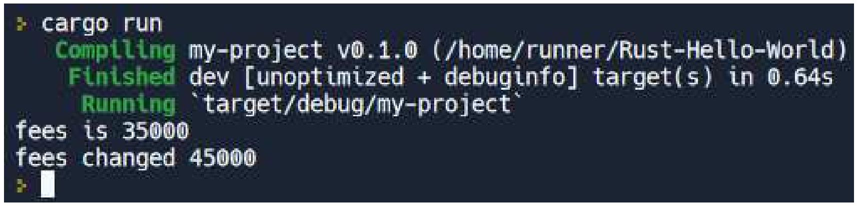
```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/Rust-Hello-World)
error[E0384]: cannot assign twice to immutable variable `fees`
 --> src/main.rs:4:3
  |
2 |   let fees=25_000;
  |       ----
  |       |
  |       first assignment to `fees`
  |       help: consider making this binding mutable: `mut fees`
3 |   println!("fees is {} ",fees);
4 |   fees=35_000;
  |   ^^^^^^^^^^^ cannot assign twice to immutable variable
```

- Note we cannot set values to the immutable variable fees twice.
- This is just one of the numerous ways Rust allows programmers to write code while benefiting from the safety and ease of concurrency

Mutable

- By default, variables are immutable.
- To make a variable changeable, prefix it with the term **mut**.
- A mutable variable's value can be changed.
- The syntax for defining a mutable variable is:
  - let **mut** variable_name = value;
  - let **mut** variable_name:dataType = value;
- Example:

```
fn main() {
        let mut fees:i32=35_000;
        println!("fees is {} ",fees);
        fees=45_000;
        println!("fees changed {}",fees);
}
```



Number Types In Rust

- There's a little bit more to numbers to talk about.
- The first thing is about ***integers*** versus ***floating point***.
- Just as in C, integers are whole numbers, like 5, -2, 0, etc.
- They are numbers that don't have a decimal point or a fractional part.
- Floating point numbers can have decimal points.

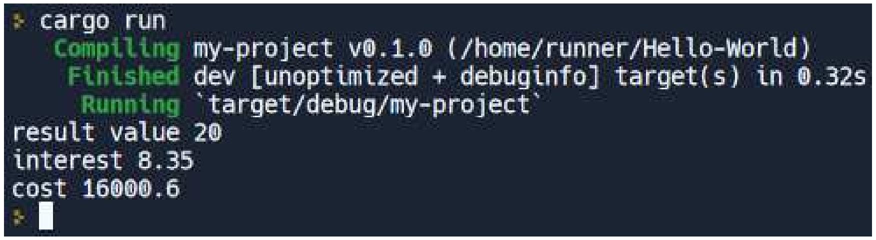
Math In Rust

- + : addition
- - : subtraction
- * : multiplication
- / : division
- % : Modulus/Remainder
- (5.0/3.0).floor() : floor division
- i32::pow(self, exp) : raise self to exp (u32) and return an integer (i32)
  f32::powi(self,exp) : raise self to exp (i32) and return a float (f32)
  f32::powf(self,exp) : raise self to exp (f32) and return a float (f32)

Float Data Type

- In Rust, float data types are categorized as f32 and f64.
- The f32 type is a single-precision float, whereas the f64 type is a double-precision float.
- The type that is used by default is f64.
- Example:

```
fn main() {
    let result=20.00;
    let interest:f32=8.35;
    let cost:f64=16000.600; // double precision
    println!("result value {}",result);
    println!("interest {}",interest);
    println!("cost {}",cost);
}
```



Printing Floats

- Just like in C, float (and integer) variables can be formatted as they are being printed.
- You have control over the size of the window that the number is printed in and the number of digits that will be displayed after the decimal point.
- You do NOT have the ability to have a comma printed after every three characters.
- Example:
  println!("The value is {0:12.2}",707.126456789);

Character Type

- Rust's character data type accepts integers, alphabets, Unicode, and special characters.
- To declare a variable of the character data type, use the char keyword.
- The char type in Rust represents a Unicode Scalar Value, which implies it may represent much more than simply ASCII.
- The Unicode Scalar Values span from U+0000 to U+D7FF [55,295] and from U+E000 [57,344] to U+10FFFF [1,114,111].
- Example:

```
fn main() {
        let special_character='@'; //default
        let alphabet:char='D';
        let emoji:char=' ';
        println!("special character {}",special_character);
        println!("alphabet {}",alphabet);
        println!("emoji {}",emoji);
}
```

Understanding Ownership:
Move, Clone, Copy

What Is Ownership In Rust?

- The primary aspect of Rust is ownership.
- Although the characteristic is simple to describe, it has deep implications for the rest of the language.
- All programs must manage how they use memory while running on a computer.
- Some languages offer **garbage collection** [i.e. Java], which searches for no longer utilized memory while the program runs; in others, the programmer must actively allocate and delete memory [i.e. C].
- Rust has a third approach: memory is controlled using an ownership system with rules that the compiler validates at compile time.
- While our software is running, none of the ownership aspects will slow it down.

Ownership Concepts

- The “owner” can modify the ownership value of a variable based on its mutability.
- The ownership of a variable can transfer to another variable.
- In Rust, ownership is just a matter of semantics.
- In addition, the ownership concept ensures safety

Rules Of Ownership

- In Rust, each value has a variable called its owner.
- At any one moment, there can only be one owner.
- When the owner exits the scope, the value is destroyed (also known as being freed).

Variable Scope

- Let's look at the scope of several variables as a first illustration of ownership.
- A scope is the range of items that are valid within a program.
- Assume we have a variable that looks something like this:
  let st=“hello”;
- The variable st refers to a literal string, the value of which is hardcoded into the program’s text.
- The variable is valid from the time it is declared until the current scope expires.

Variable Scope

- This example includes comments that indicate when the variable st is valid.

```
// st is not valid here, it's not yet declared
{
        let st="hello"; // st is valid from this point forward
        // do stuff with st
}
// this scope is now over, and st is no longer valid
```

- In other words, there are two critical time points here:

  1. It is valid when st enters the scope.
  2. It is still valid until it goes out of scope.

- The connection between the scope and when variables are valid is comparable to that of other programming languages at this stage.

How Variables and Data Interact: Move

- In Rust, several variables can interact with the same data in various ways.

- We now look at an example with an integer.
  let a=8;
  let x=a;
  let b=x;

- "Bind the value 8 to a; then make a copy of the value in x and bind it to b."

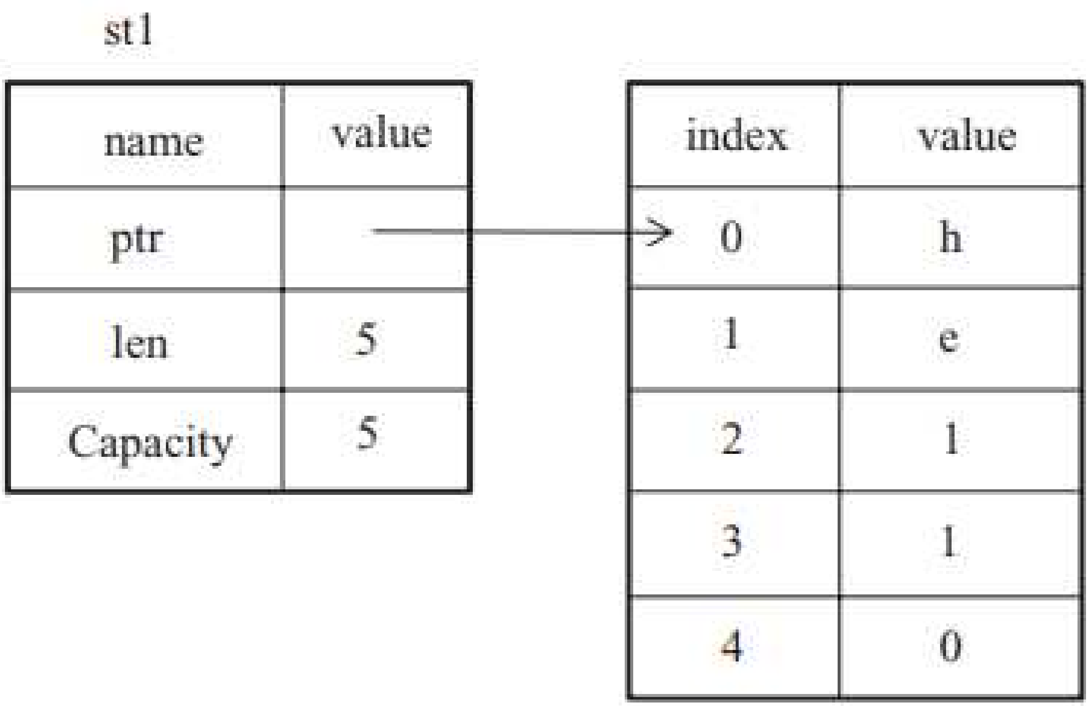
- We now have two variables, a and b, equal to 8.

- This is correct because integers are simple values with a known, defined size, and these two 8 values are placed into the stack.

- Let's have a look at the String version:
  let st1=String::from("hello");
  let st2=st1;

- This code appears to be quite similar to the preceding code, so we can conclude that the function is the same: the second line would duplicate the value in st1 and bind it to st2.

How Variables and Data Interact: Move

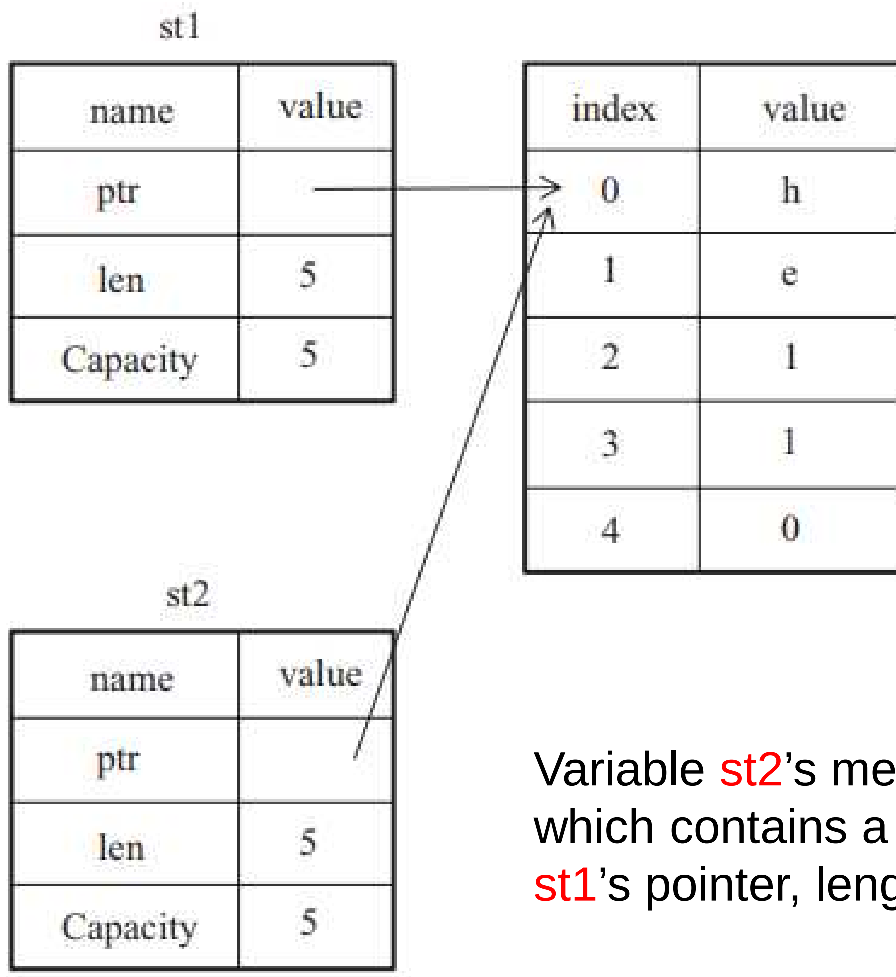


Memory representation of a String with the value “hello” linked to st1.

How Variables and Data Interact: Move

- However, this is not the case.
- The length specifies how much memory (in bytes) the String's contents presently occupy.
- The capacity is the entire amount of memory that the allocator gives the String in bytes.
- The distinction between length and capacity is essential, but not in this context, so ignore the capacity for the time being.
- When we assign st1 to st2, the String data is duplicated, which means we copy the stack's pointer, length, and capacity.

How Variables and Data Interact: Move



Variable st2's memory representation, which contains a duplicate of st1's pointer, length, and capacity.

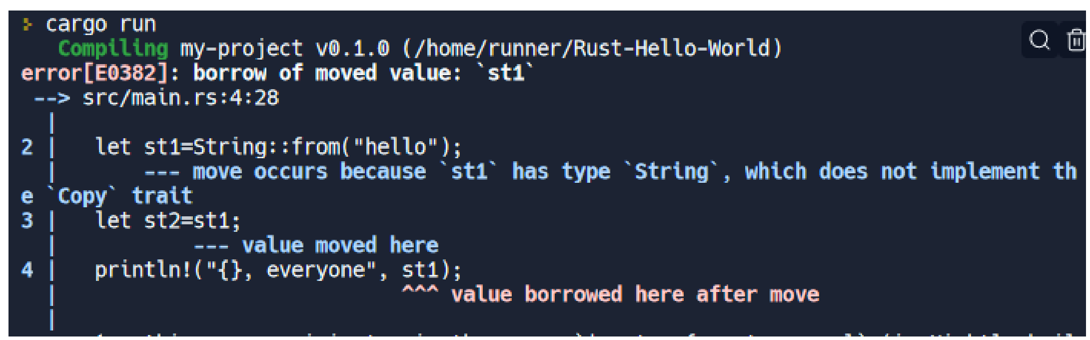
How Variables and Data Interact: Move

- We do not replicate the data on the heap to which the pointer points.
- Rust automatically executes the drop function when a variable exits scope and cleans away the heap memory for that variable.
- However, in the figure both st1 and st2 data pointers point to the same place.
- This is an issue because when st2 and st1 exit scope, they will attempt to free the same memory.
- This is referred to as a **double free mistake**.
- Memory corruption can result from freeing memory twice, leading to security vulnerabilities.

How Variables and Data Interact: Move

- There is one additional element to what occurs in this circumstance in Rust to ensure memory safety.
- Rust considers st1 invalid after letting st2 = st1.
- As a result, when st1 exits scope, Rust does not need to release anything.
- Examine what happens if we try to utilize st1 after st2 is generated; it will not work:

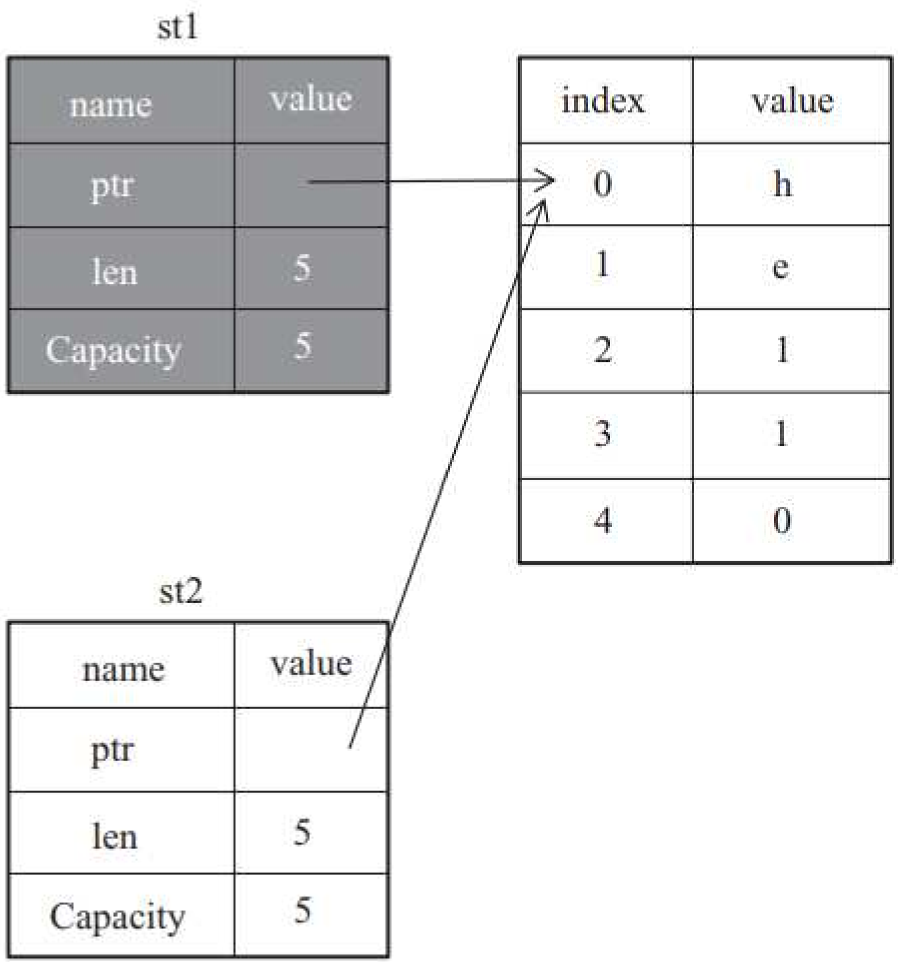
```
let st1=String::from("hello");
let st2=st1;
println!("{}, everyone", st1);
```



How Variables and Data Interact: Move

- Because Rust invalidates the first variable, it is referred to as a **move**.
- In this case, we would state that st1 was relocated to st2.
- That takes care of our issue!
- With just st2 valid, when it exits scope, it will release the memory on its own, and we're done.
- Furthermore, this implies a design choice: any automated copying may be presumed to be low cost in terms of runtime performance.

How Variables and Data Interact: Move



Memory representation after s1 has been invalidated.

Ownership And Moving

- Blocks can also be owners.
- Example:

```
fn main() {
        {
                let x: i32 = 5;
                println!("{}", x);
        }
}
```

- **main** owns that block, and the block owns the value 5.
- And values can even own other values.
- Remember that you can only have one owner for a value at a time.

No Ownership Problem: Copy

- Example:

```
fn count(apples: i32) {
        println!("You have {} apples", apples);
}
fn price(apples: i32) -> i32 {
        apples * 8
}
fn main() {
        let apples: i32 = 10;
        count(apples);
        let price = price(apples);
        println!("The apples are worth {} cents", price);
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/Rust-Hello-World)
    Finished dev [unoptimized + debuginfo] target(s) in 0.67s
     Running `target/debug/my-project`
You have 10 apples
The apples are worth 80 cents
>
```

Why Don't We Have A Problem?

- **Copy** is a trait in Rust that says "this thing is so incredibly cheap to make a copy of, that each time you try to move it, it’s fine to just make a copy and move that new copy instead."
- And i32 is an example of a type which is so cheap.
- Therefore, in our code here, count(apples) doesn’t **move** the value into count.
- Instead, it makes a **copy** of the value 10, and moves that copy into count.
- But the original 10 inside the apples variable remains unchanged.

Stack-Only Data: Copy

- The Rust annotation Copy trait may be applied to types like integers stored on the stack.
- If a type has the Copy trait, an older variable can still be used after the assignment has been performed.
- Rust will not allow us to annotate a type with the Copy trait if the type or any of its components has the Drop trait implemented.
- If we add the Copy annotation to a type that requires anything specific to happen when the value is out of scope, we will get a compile-time error.

Variables and Data Interactions: Clone

- We may use the clone method to thoroughly duplicate the String’s heap data rather than merely the stack data. [also called a "deep copy"]
- Here’s an example of how to use the clone method:

```
let st1=String::from(“hello”);
let st2=st1.clone();
println!(“st1={}, st2={}”, st1, st2);
```

- This works well and generates the behavior in the previous example, where the heap data is explicitly copied: both st1 and st2 now contain the string "hello".
- Performing a clone call might be costly to perform depending on the variable that is being duplicated.

Stack-Only Data: Copy vs Clone

- This code use integers:
  ```
  let a=8;
  let b=a;
  println!("a={}, b={}", a, b);
  ```
- However, this code appears to contradict what we have just learned: there is no call to clone, but a is still valid and was not transferred into b.
- This is because types with known sizes at build time, like integers, floats, Booleans, characters, and tuples (depending on what they contain) are wholly stored on the stack; thus, copies of the actual values are quickly produced.
- There is no reason to prevent a from being valid after we have created the variable b.
- In this case, there is no distinction between deep and shallow copying.
- Therefore, invoking clone would perform nothing more than shallow copying so that we can leave it out.

Ownership and Functions

- Passing a value to a function has semantics comparable to giving a value to a variable.
- Passing a variable to a function will cause it to move or copy much like an assignment.
- The following example has annotations indicating where variables enter and exit their scope.

Ownership and Functions

```
fn main() {
        let st=String::from("hello"); // st comes into scope
        takes_ownership(st); // st's value moves into the function... and so is no longer valid
        let a=5; // a comes into scope
        makes_copy(a); // move into the function, but a (i32) is Copy, so okay to still use afterward
}
        // Here, a goes out of scope, then st. But because st's value was moved, nothing
        // special happens.

fn takes_ownership(some_string: String) {
        //some_string comes into the scope
        println!("{}", some_string);
} // Here, some_string goes out of the scope and a `drop` is called. The backing memory is freed.

fn makes_copy(some_integer: i32) {//some_integer comes into the scope
        println!("{}", some_integer);
} // Here, some_integer goes out of the scope. Nothing special happens.
```

Return Values and Scope

- Ownership can also be transferred by returning values.
- Every time, the ownership of a variable follows the same pattern: assigning a value to another variable changes it.
- When a variable that includes heap data exits scope, the value is destroyed unless the data has been transferred to be held by another variable.
- Example:

Return Values and Scope

```
fn main() {
        let st1=gives_ownership(); // gives_ownership moves its return value into st1
        let st2=String::from(“hello”); // st2 comes into the scope
        let st3=takes_and_gives_back(st2);
        // st2 is moved into takes_and_gives_back, which also moves its return value into st3.
}
        // Here, st3  goes out of the scope and is dropped. st2 was moved, so nothing happens.
        // st1 goes out of the scope and is dropped.

fn gives_ownership() ->String { // gives_ownership will move its return the value into function that calls it
        let some_string=String::from(“yours”);  // the some_string comes into scope
        some_string        // the some_string is returned and moves out to calling function
        }

// This function takes String and returns one
fn takes_and_gives_back(a_string: String) ->String {// a_string comes into scope
        a_string // a_string is returned and moves out to the calling a function
}
```

Return Values and Scope

- Taking ownership and then restoring ownership with each function is time-consuming.
- What if we want a function to utilize a value but not own it?
- It is inconvenient because whatever we send data to a function, in addition to any data originating from the function's body that we might want to return, it must be sent back if we want to use it again.
- A tuple can be used to return many values:

```
fn main() {
        let st1=String::from("hello");
        let (st2, len)=calculate_length(st1);
        println!("length of '{}' is {}.", st2, len);
}
fn calculate_length(st: String) ->(String, usize) {
        let length=st.len(); // len() returns the length of a String
        (st, length)
}
```

References & Borrowing

References And Borrowing In Rust

- A reference is an address passed as an argument to a function.
- Borrowing is similar to when we borrow something and then return it after we are through with it.
- Borrowing and references are mutually exclusive, which means that when a reference is released, the borrowing also ends.

References - Borrow

- Example:

```
fn increase_fruit(mut numFruit: Fruit) -> Fruit {
        numFruit *= 2;
        numFruit
}
fn print_fruit(numFruit: Fruit) -> Fruit {
        println!("You have {} pieces of fruit", numFruit.apples+numFruit.bananas);
        numFruit
}
fn main() {
        let fruit = 10;
        let fruit = print_fruit(fruit);
        let fruit = increase_fruit(fruit);
        print_fruit(fruit);
}
```

Problem: we have to create another fruit variable because we have to move the value of fruit both in and out of the routine print_fruit.

Borrowed References

- The problem with this code:
  let fruit **= print_fruit**(fruit);
- We don't want to have to move the value in and back out.
- Instead, we'd like to be able to let print_fruit borrow the value we own in main, without moving it completely.
- Good news - Rust supports exactly that!
- Instead of passing print_fruit the fruit value itself, we need to pass it a **borrowed reference**.
- There's a new unary operator to learn for this: &.

Borrowed References

- It turns out that when you borrow a value of type Fruit, you don't get back a Fruit. Instead, you get a &Fruit.
- That & at the beginning of the type means "a reference to."
- In other words, & has two different but related meanings:
  - When on a value: borrow a reference to this value
  - When on a type: a reference to this type
- Right now, the type of the parameter to print_fruit is Fruit. This requires that the value be moved into print_fruit.
- Instead, let's change that so that it's a reference to a Fruit, or &Fruit:
  fn print_fruit(numFruit: **&**Fruit) **->** Fruit

Borrowed References

- Error: the only reason we were returning a Fruit in the first place was to deal with moving and ownership.
- But we don't actually need that anymore!
- So instead, let's get rid of the return value entirely:

```
fn print_fruit(numFruit: &Fruit) {
        println!("You have {} pieces of fruit",
        numFruit.apples+numFruit.bananas);
}
```

- We now replace:

```
let fruit = print_fruit(&fruit);
```

with:

```
print_fruit(&fruit);
```

Fixed Code

- Example:

```
fn increase_fruit(mut fruit: Fruit) -> Fruit {
	fruit.apples *= 2;
	fruit.bananas *= 3;
	fruit
}
fn print_fruit(numFruit: &Fruit) {
	println!("You have {} pieces of fruit", numFruit.apples+numFruit.bananas);
}
fn main() {
	let fruit = Fruit {
			apples: 10,
			bananas: 5,
	};
	print_fruit(&fruit);
	let fruit = increase_fruit(fruit);
	print_fruit(&fruit);
}
```

Mutable References

- We still want to be able to modify the fruit using the increase_fruit function.
- To make this work, we need to introduce a second kind of reference: **a mutable reference**.
- While an immutable reference is &, a mutable reference is &mut.
- It looks like:

```
fn increase_fruit(numFruit: &mut Fruit) {
        numFruit *= 2;
}
```

Mutable References

- It turns out that &Fruit and &mut Fruit are really different types.
- Therefore, we need a different operator to borrow a mutable reference than an immutable reference.
- And this operator is &mut.
- So we rewrite our function call from our original:

  let fruit **= increase_fruit**(fruit);

  updated:

  increase_fruit(**&mut** fruit);
- One final change to make it all work:

  ```
  let mut fruit = Fruit {
          apples: 10,
          bananas: 5,
  };
  ```

References

- Just like in C, every value in Rust lives somewhere in your computer's memory.
- And every place in computer memory has an address.
- It's possible to use println and the special {:p} syntax to display the address itself:

```
fn main() {
        let x: i32 = 5;
        println!("x == {}, located at {:p}", x, &x);
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/Rust-Hello-World)
    Finished dev [unoptimized + debuginfo] target(s) in 0.78s
     Running `target/debug/my-project`
x == 5, located at 0x7ffd60977afc
> 
```

References

- Just like in C, a reference can be thought of as a ***pointer***: it's an address pointing at a value that lives somewhere else.
- That's also why we use the letter p in the format string to print the address.
- When you have a variable like let y: &i32 = &x, what this means is:
  - y is an immutable variable
  - That variable holds an address
  - That address points to an i32
  - The reference is immutable, so we **can't** change the value of y
- On the other hand, let y: &mut i32 = &mut x is almost exactly the same thing, except for the last point.
- Since the reference is mutable, we **can** change the y value.

Dereferences

- Example:

```
fn main() {
        let x: i32 = 5;
        let mut y: i32 = 6;
        let z: &mut i32 = &mut y;
        z -= 1;
        assert_eq!(x, y);
        println!("Success");
}
```

- Does not work,
- The problem is that we're trying to use the -= operator on a &mut i32 value.
- The reference is really just an address, not an i32.
- We don't want to subtract 1 from an address.
- We want to subtract 1 from the value *behind* the reference.

Dereferences

- Just like in C, Rust provides another unary operator to talk about the thing behind a reference.
- It’s called the deref—short for dereference —operator, and is *.
- Example:

```
fn main() {
        let x: i32 = 5; let mut y: i32 = 6;
        let z: &mut i32 = &mut y;
        *z -= 1;
        assert_eq!(x, y);
        println!("Success");
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/Rust-Hello-World)
    Finished dev [unoptimized + debuginfo] target(s) in 0.34s
     Running `target/debug/my-project`
Success
>
```

Lifetimes Of References

- There's an important restriction on references, both mutable and immutable: they cannot live longer than the values they are referencing.

- Example:

```
fn main() {
        let x: &i32 = {
                let y = 5;
                &y
        };
        println!("x == {}", x);
}
```

- This program fails to compile.

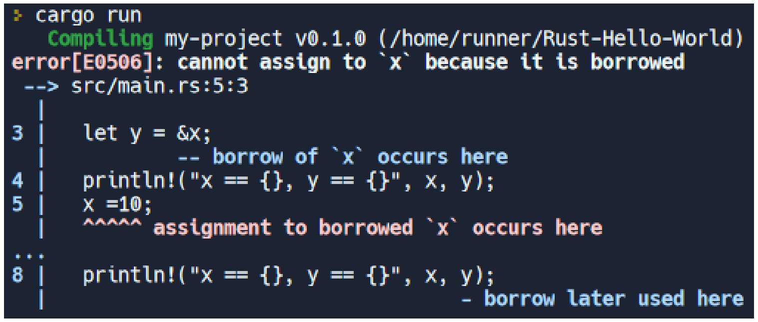
Lifetimes Of References

- The problem here is that y is *dropped* as soon as the block finishes.
- The block itself was the owner for y.
- And when an owner goes away, the value is dropped, and cannot be used anymore.
- However, we return a reference to y, which would allow us to keep using y after it's gone.
- That would be really dangerous, and so Rust doesn't let that happen.
- All values and references in Rust have a *lifetime*.
- When Rust is able to figure out the lifetime of a value, it will.

Mutating And Borrowing

- A lot of problems in software come about from things changing when you don't expect them to.
- That's why Rust defaults to having immutable variables: it's easier to think about things when they can't change.
- It means that if I have an immutable value, and I print it twice, I know it will give me the same value.
- This applies to immutable references too.
- As long as a value is borrowed, it can't be mutated:
- Example:

```
fn main() {
        let mut x = 5;
        let y = &x;
        println!("x == {}, y == {}", x, y);
        x =10;
        // I can do anything I want here...
        // And then this will produce the same output
        println!("x == {}, y == {}", x, y);
}
```



Single Mutable Reference

- Rust is picky about mutation stuff.
- We already mentioned that you can't mutate a value that's borrowed.
- This same basic logic extends to mutable references.
- If you have a mutable reference to a value, you can't mutate *or read* that value anywhere else in your program.
- We call this ***freezing***.
- Example:

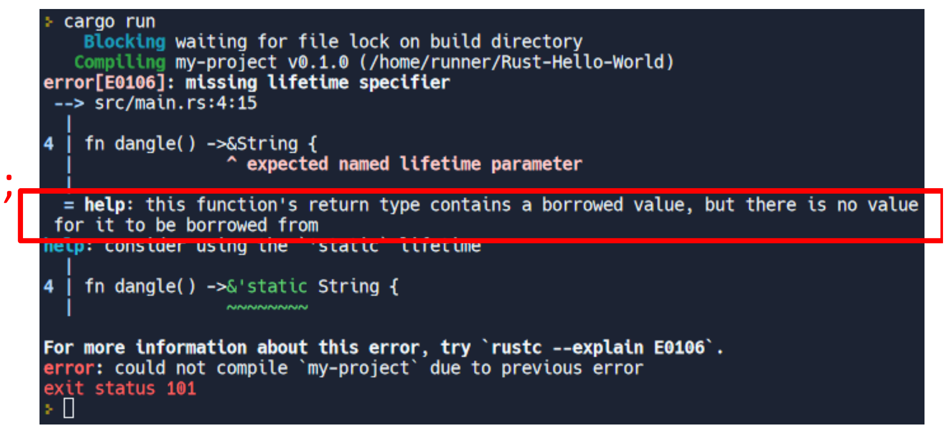
```
fn main() {
        let mut x = 5;
         let y = &mut x; // freeze
         x *= 2;
         *y *= 2; // unfreeze
         println!("x == {}", x);
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/Rust-Hello-World)
error[E0503]: cannot use `x` because it was mutably borrowed
 --> src/main.rs:3:12
  |
2 |             let y = &mut x; // freeze
  |                     ------ borrow of `x` occurs here
3 |             x *= 2;
  |             ^^^^^^ use of borrowed `x`
4 |             *y *= 2; // unfreeze
  |             ------- borrow later used here
```

Dangling References

- In pointer-based languages, it's possible to construct a dangling pointer, which refers to a place in memory that may have been passed to someone else, by releasing some memory while retaining a pointer to that region.
- In contrast, the compiler in Rust ensures that references are never dangling: if we have a reference to some data, the compiler will ensure that the data does not go out of the scope before the reference to the data does.
- Let's attempt making a dangling reference, which Rust will reject with a compile-time error:
- Example:

```
fn main() {
        let reference_to_nothing=dangle();
}
fn dangle() ->&String {
        let st=String::from("hello");
        &st
}
```



Dangling References

- Because st is generated within dangle, after dangle's code is complete, st will be deallocated.
- However, we attempted to return a reference to it.
- As a result, this reference would link to an incorrect String.
- That is not acceptable; Rust will not allow us to do so.
- The approach here is to just return the String:

```
fn no_dangle() ->String {
        let st=String::from("hello");
        st
}
```

- This works without a problem.
- Nothing has been deallocated, and ownership has been transferred.

Summary

- You are allowed to borrow references to values
- Borrowing a reference does not move ownership
- Borrowing is the preferred way to solve the "move in move out" problem with functions.
- References have their own type, and i32 is different than &i32.
- We also have mutable references such as &mut i32, which allow the values behind the reference to be changed.
- Mutable references can only be borrowed from mutable values
- References are essentially addresses for where the original value lives in memory
- If you want to operate directly on the value behind a reference, you can dereference using the * operator.

Summary

- A reference cannot outlive the value it refers to
- To avoid insanity around mutation and references, Rust has some rules you need to abide by
  - You cannot mutate a value if there is a reference to it
  - You can have multiple immutable references to a value
  - You can only have one mutable reference to a value, and then no other immutable references to it, or access the value directly
- You can create an immutable reference from a mutable reference, but not the other way around

Image Source: https://commons.wikimedia.org/wiki/File:Thats_all_folks.svg