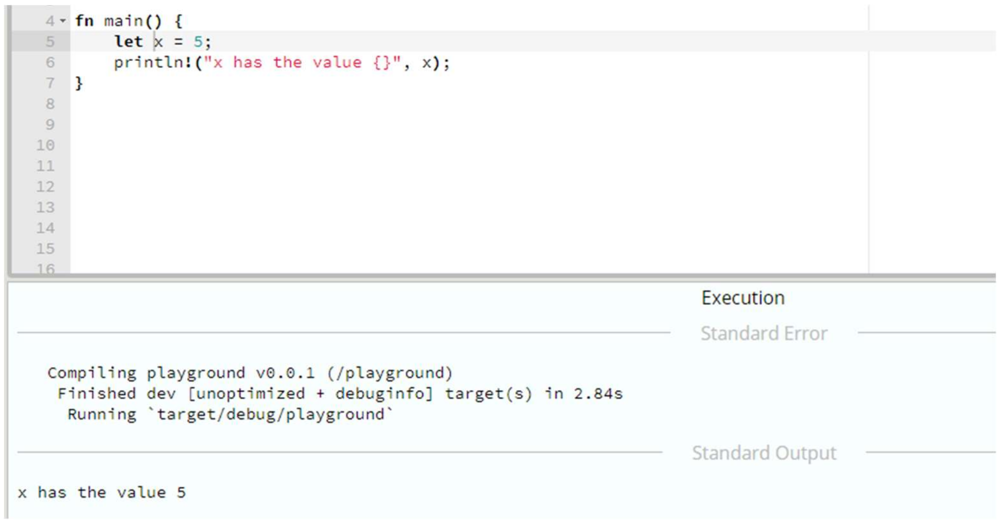
RUST AND SECURITY LAB

CY5130

1. **Variable bindings:**

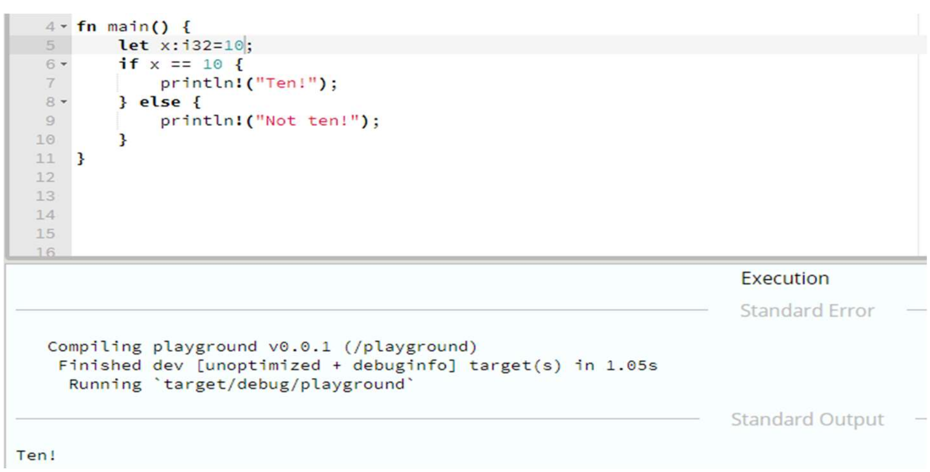
a) **variables1.rs:**

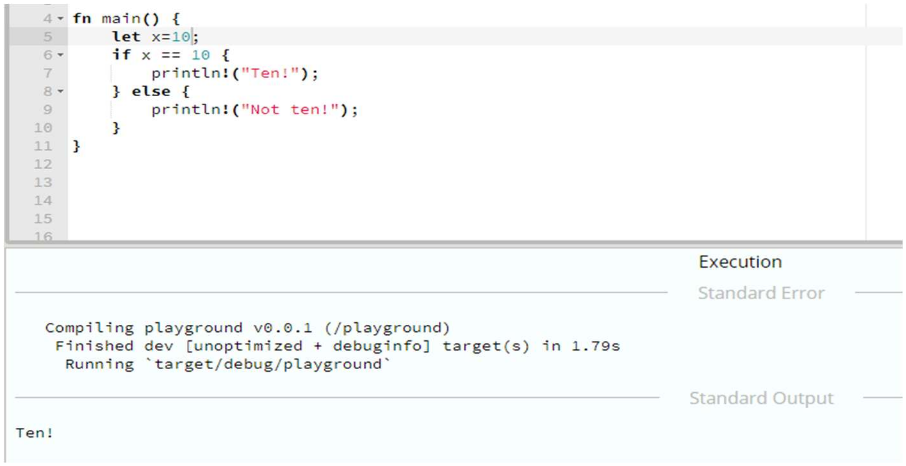
The error was variable x not found in the scope of usage. This can be debugged using the 'let' keyword. The let keyword binds variable to the current scope. Hence, rust enables usage of only those variables that belongs to its scope.



b) **variables2.rs:**

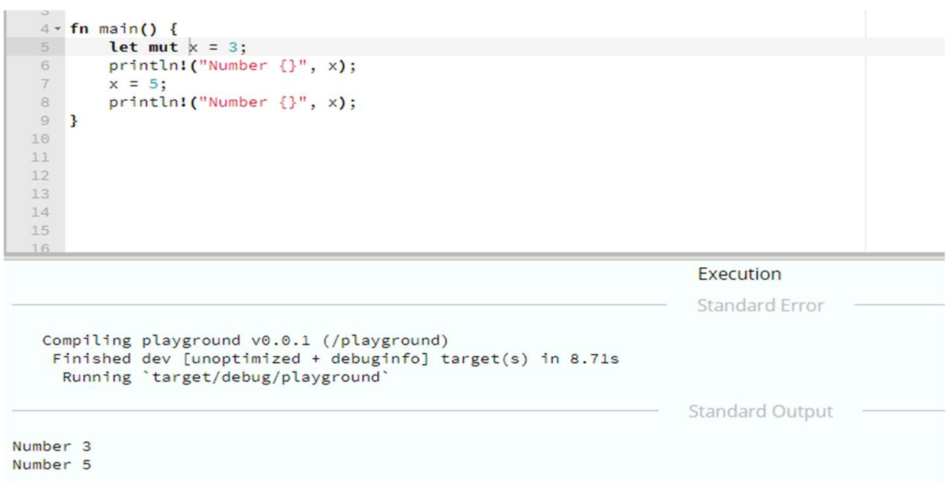
The bug here is that the variable x is uninitialized, and no type annotation is specified. Rust compiler does not allow usage of uninitialized variables (In any case where we need to use uninitialized variable then we can do so by using unsafe function/code). Initialization can be done in 2 ways: either by specifying the datatype and value matching that of 10 or just by specifying the value and making the compiler to infer the type.





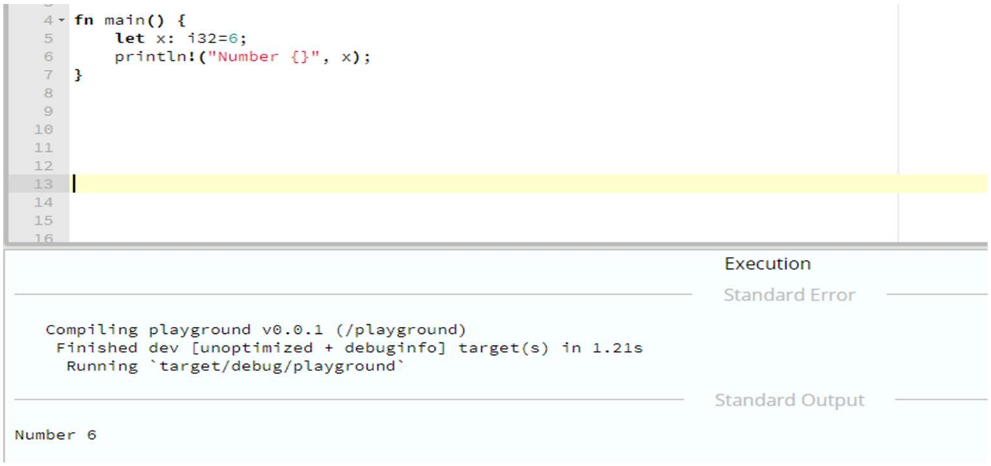
c) variables3.rs:

Variables are immutable by default in rust i.e we can't reassign values to the same variable more than once hence, making sure that we only make the variable we want, to be mutable. We can debug this by making 'x' mutable by using the keyword 'mut'.



d) variables4.rs:

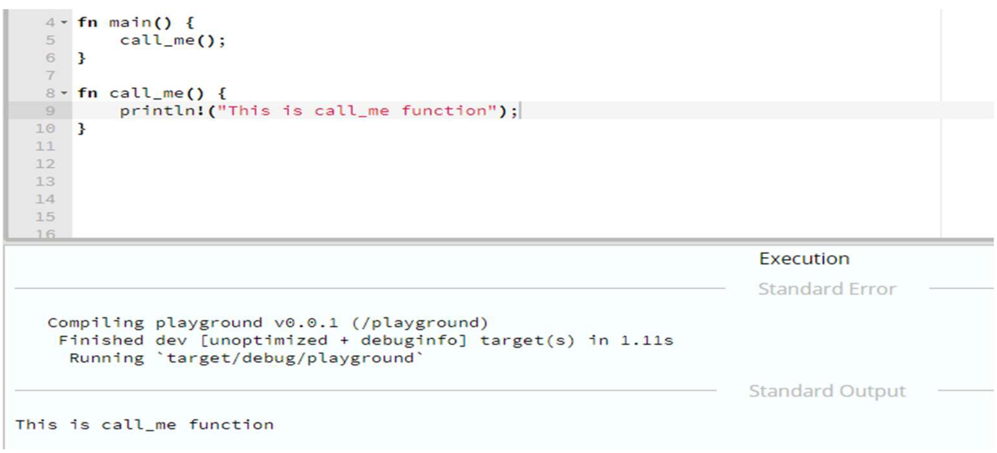
We can't print uninitialized variables. Even though the type is already specified, compiler doesn't know what value the variable holds to print. Hence, the rust compiler makes sure that no variable holds garbage value and it is always initialized unlike in languages like C where an uninitialized variable would be assigned a default value.



2. **Functions:**

   a) **functions1.rs:**

   The bug here is that the main function is invoking a call_me() which doesn't exist. This can be rectified by specifying the function declaration of call_me(). Hence rust makes sure that we don't wrongly invoke undeclared function.



   b) **functions3.rs:**

   The bug found was that the type annotation was missing for the argument passed in the call_me() function definition. To correct this, we need to mention the type of value 3 which is an integer (i32). Rust requires type annotations in function definitions to check for type errors if any.

```rust
fn main() {
    call_me(3);
}

fn call_me(num:i32) {
    for i in 0..num {
        println!("Ring! Call number {}", i + 1);
    }
}
```

```
Execution
Standard Error

   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.82s
     Running `target/debug/playground`

Standard Output

Ring! Call number 1
Ring! Call number 2
Ring! Call number 3
```

c) **functions3.rs:**

Here, the function call_me is expecting to receive an integer argument when invoked which is absent while calling it in main function. To debug, we need to specify any integer as the parameter in line 5. Hence, the rust compiler makes sure that the function is invoked with correct number of parameters and is type checked with that of function signature.

```rust
4  fn main() {
5      call_me(3);
6  }
7
8  fn call_me(num:i32) {
9      for i in 0..num {
10         println!("Ring! Call number {}", i + 1);
11     }
12 }
13
14
15
```

```
Execution
Standard Error

   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.82s
     Running `target/debug/playground`

Standard Output

Ring! Call number 1
Ring! Call number 2
Ring! Call number 3
```

d) **functions4.rs:**

If a function invoked requires returning some value, then it is necessary to mention the return type in the function signature. The error here is that the function is supposed to return an integer i.e sale price based on if the original price is odd/even but, the return type annotation is missing. By specifying the type (integer i32) it can be corrected.

```rust
fn main() {
    let original_price = 51;
    println!("Your sale price is {}", sale_price(original_price));
}

fn sale_price(price: i32) -> i32{
    if is_even(price) {
        price - 10
    } else {
        price - 3
    }
}

fn is_even(num: i32) -> bool {
    num % 2 == 0
}
```

```
Execution
Standard Error

Compiling playground v0.0.1 (/playground)
 Finished dev [unoptimized + debuginfo] target(s) in 0.74s
  Running `target/debug/playground`

Standard Output

Your sale price is 48
```

e) functions5.rs:

Here the function square() is calculating and returning the squared value of the argument passed. However, it isn't returning the calculated value. To do so, we can either explicitly add the 'return' keyword along with the multiplication instruction in line 10 or use the shorthand by omitting the semicolon in the end of line 10.

```rust
4  fn main() {
5      let answer = square(3);
6      println!("The answer is {}", answer);
7  }
8
9  fn square(num: i32) -> i32 {
10     num * num
11 }
12
13
14
15
16
```

```
Execution
Standard Error

Compiling playground v0.0.1 (/playground)
 Finished dev [unoptimized + debuginfo] target(s) in 1.13s
  Running `target/debug/playground`

Standard Output

The answer is 9
```

3. Primitive types:

a) primitivetypes1.rs:

This requires us to assign a Boolean value to the variable is_evening in line 13. The Boolean datatype consists of two values: **true** and **false**. Here I set the value to be false hence the if statement would return false.

```rust
fn main() {
    // Booleans (`bool`)

    let is_morning = true;
    if is_morning {
        println!("Good morning!");
    }

    let is_evening = false;// Finish the rest of this line like the example! Or make it be false!
    if is_evening {
        println!("Good evening!");
    }
}
```

```
Execution
Standard Error
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.74s
     Running `target/debug/playground`
Standard Output
Good morning!
```

b) primitivetypes2.rs:

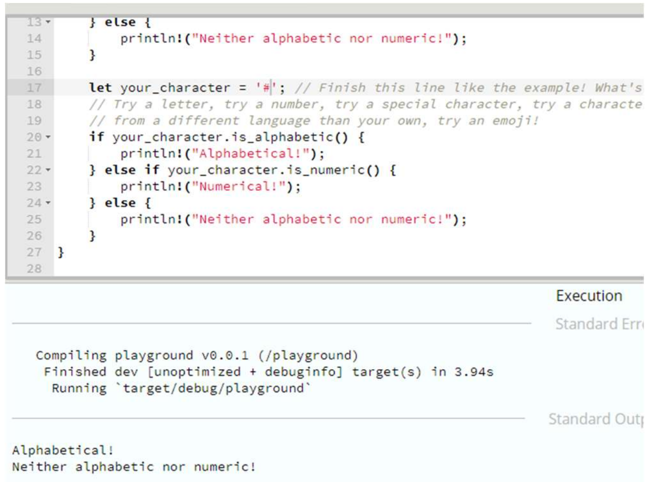
Here we are required to assign some value of character data type into the variable your_character and checking if it is an alphabet, numerical or neither.

```rust
    } else {
        println!("Neither alphabetic nor numeric!");
    }

    let your_character = 'S'; // Finish this line like the example! What's
    // Try a letter, try a number, try a special character, try a characte
    // from a different language than your own, try an emoji!
    if your_character.is_alphabetic() {
        println!("Alphabetical!");
    } else if your_character.is_numeric() {
        println!("Numerical!");
    } else {
        println!("Neither alphabetic nor numeric!");
    }
}
```

```
Execution
Standard Err
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.74s
     Running `target/debug/playground`
Standard Out
Alphabetical!
Alphabetical!
```

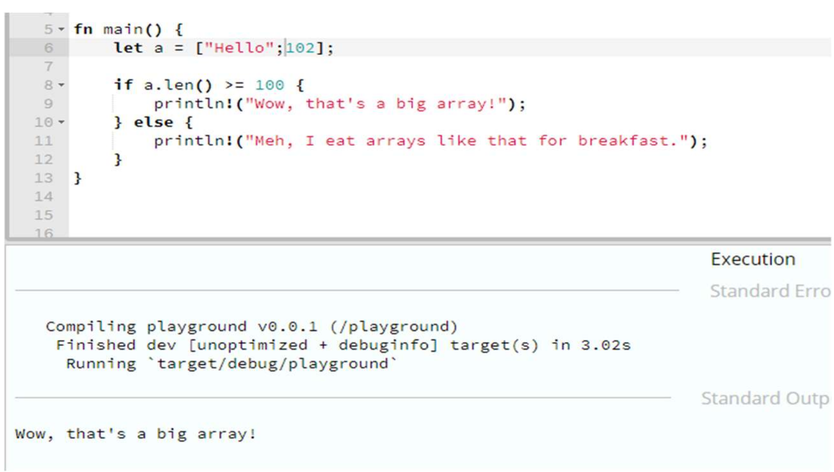
```rust
    } else {
        println!("Neither alphabetic nor numeric!");
    }

    let your_character = '3'; // Finish this line like the example! What's
    // Try a letter, try a number, try a special character, try a characte
    // from a different language than your own, try an emoji!
    if your_character.is_alphabetic() {
        println!("Alphabetical!");
    } else if your_character.is_numeric() {
        println!("Numerical!");
    } else {
        println!("Neither alphabetic nor numeric!");
    }
}
```

```
Execution
Standard Err
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.23s
     Running `target/debug/playground`
Standard Out
Alphabetical!
Numerical!
```



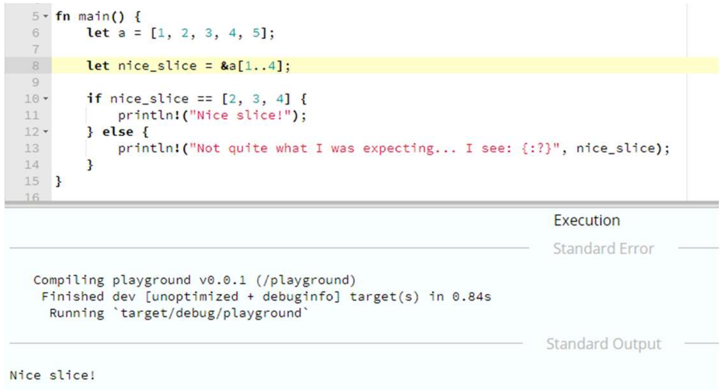
c) primitivetypes3.rs:

Here we need to assign a with an array of minimum 100 elements. This can be done using the shorthand [<value>, <#>] i.e it creates an array of size # specified where each element is the ‘value’ specified. For instance, here array a would contain 102 elements where the value of each element is “Hello”.

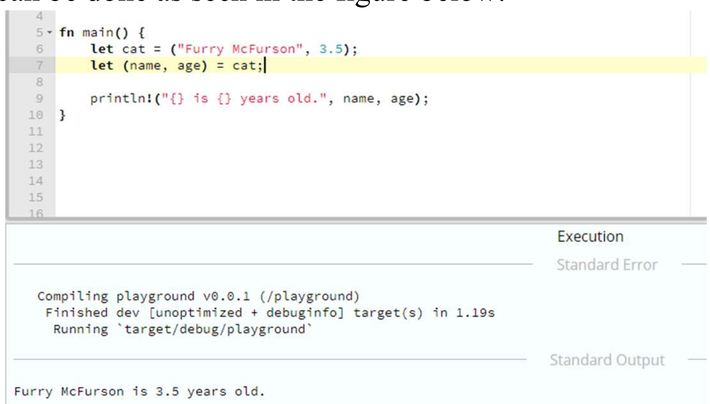


d) primitivetypes4.rs:

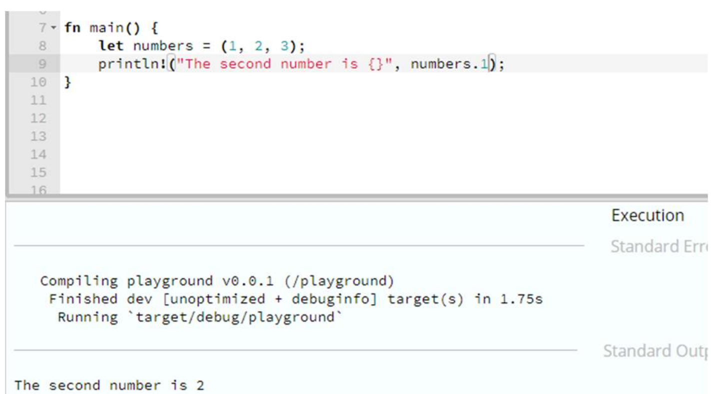
We are required to slice the array **a** such that the following if condition returns true. This can be done as seen in the below figure on line 8. Here we borrow and slice the array **a** and assign that to nice_slice variable.



e) **primitivetypes5.rs:**
We are asked to destructure the cat tuple into name, age so that we can print it in the next line. This can be done as seen in the figure below.



f) **primitivetypes6.rs:**
Here we need to print the second element of the tuple using its index. Tuple indexing starts from 0 hence, the index of second element would be 1.



4. **Strings:**
   a) **strings1.rs:**
   In this the function current_favorite_color( ) is expected to return the value blue upon invocation. However, the function signature specifies that the value “blue” is a String

object, but the function is trying to return string literal/slice. Hence the type mismatch. This can be corrected using to_String() method that converts the string literal into the String object.

```
fn main() {
    let answer = current_favorite_color();
    println!("My current favorite color is {}", answer);
}

fn current_favorite_color() -> String {
    "blue".to_string()
}
```

```
Execution
Standard Error

Compiling playground v0.0.1 (/playground)
 Finished dev [unoptimized + debuginfo] target(s) in 1.04s
  Running `target/debug/playground`

Standard Outpu

My current favorite color is blue
```

b) string2.rs:

In this we are creating a string object word with the value 'green'. However, the function is_a_color_word requires a string slice as the argument. Strings implement Deref<Target=str>, and so it inherits all str's methods. So, we can coerce String into &str and pass this as the argument i.e passing **&word** instead of **word** in line 6.

```
4  fn main() {
5      let word = String::from("green"); // Try not changing this line :)
6      if is_a_color_word(&word) {
7          println!("That is a color word I know!");
8      } else {
9          println!("That is not a color word I know.");
10     }
11 }
12
13 fn is_a_color_word(attempt: &str) -> bool {
14     attempt == "green" || attempt == "blue" || attempt == "red"
15 }
```

```
Execution
Standard Error

Compiling playground v0.0.1 (/playground)
 Finished dev [unoptimized + debuginfo] target(s) in 0.81s
  Running `target/debug/playground`

Standard Output

That is a color word I know!
```

c) string3.rs:

For this exercise we are required to specify which function to use based on the value to pass. This is done as shown in the figure below.

(The first one uses a string literal/slice, second, we convert the slice into a String object, hence string. In 3rd we are creating a String object with the value 'hi' and 4th we are creating an owned string object from a slice. 5th into() returns a String object and 6th format! is similar to that of printf in C and returns a String object created using interpolation of runtime expressions. 7th creates a new string slice, 8th also creates a string slice with trim(), 9th we are converting a slice into a String object, 10th creates and returns a String object.)

```
fn string_slice(arg: &str) { println!("{}", arg); }
fn string(arg: String) { println!("{}", arg); }

fn main() {
    string_slice("blue");
    string("red".to_string());
    string(String::from("hi"));
    string("rust is fun!".to_owned());
    string_slice("nice weather".into());
    string(format!("Interpolation {}", "Station"));
    string_slice(&String::from("abc")[0..1]);
    string_slice("  hello there ".trim());
    string("Happy Monday!".to_string().replace("Mon", "Tues"));
    string("mY sHiFt KeY iS sTiCkY".to_lowercase());
}
```

```
Execution
Standard Output

blue
red
hi
rust is fun!
nice weather
Interpolation Station
a
hello there
Happy Tuesday!
my shift key is sticky
```

5. **Move Semantics:**

   a) **move_semantics1.rs:**

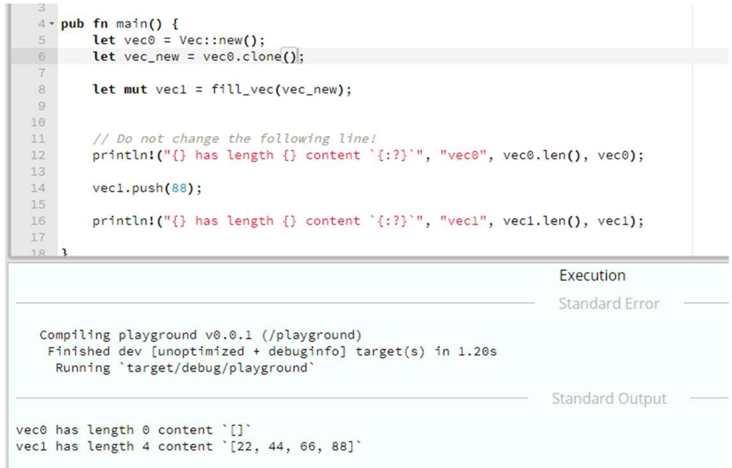
   In this example, we are borrowing a vector vec0 and assigning 3 values to vec1 through the function fill_vec. Again, in line 11, we are trying push an additional value to the vector. Since vectors are immutable by default, to make it work we need to make vec1 mutable using 'mut' keyword.

```
3
4  pub fn main() {
5      let vec0 = Vec::new();
6
7      let mut vec1 = fill_vec(vec0);
8
9      println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);
10
11     vec1.push(88);
12
13     println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);
14
15 }
16
```

```
Execution
Standard Error

   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.54s
     Running `target/debug/playground`

Standard Output

vec1 has length 3 content `[22, 44, 66]`
vec1 has length 4 content `[22, 44, 66, 88]`
```

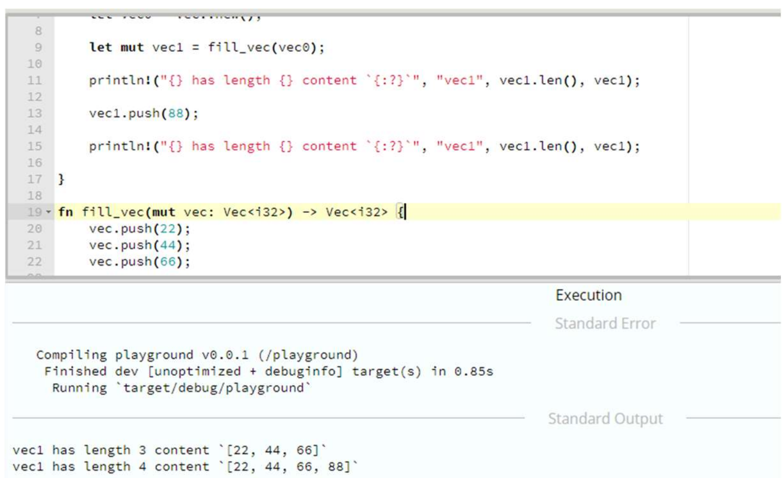
   b) **move_semantics2.rs:**

   The vector does not implement copy trait, so in line 7 the vector vec0 is moved when the function fill_vec is called and is dropped at the end of function. To make vec0 accessible even after the function call, we can create a clone of vec0 and pass that as the argument for fill_vec. Even though cloning is expensive, it is a safer option than making passing the mutable vector as the argument.



c) move_semantics3.rs:

Here in the function fill_vec we are trying to use vector vec as mutable when it is not declared as one. It can be debugged by declaring it as mutable in the function signature as seen in the figure.



d) move_semantics4.rs:

In this example, we are just refactoring the code where instead of creating a vector vec0 in main and passing it to the function fill_vec, we are creating a fresh vector inside the function and pushing some values on to it and returning this back to main(). We can do so by adding the line 21 as shown and removing the in main() where vec0 is created. The code now is much simpler by eliminating the need of creation of vector vec0.

```
    let mut vec1 = fill_vec();

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);

    vec1.push(88);

    println!("{} has length {} content `{:?}`", "vec1", vec1.len(), vec1);

}

fn fill_vec() -> Vec<i32> {

    let mut vec = Vec::new();
    vec.push(22);
    vec.push(44);
    vec.push(66);
```

```
Execution

Standard Error

   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.85s
     Running `target/debug/playground`

Standard Output

vec1 has length 3 content `[22, 44, 66]`
vec1 has length 4 content `[22, 44, 66, 88]`
```

6. **Threads:**

   The motive behind this example is to implement concurrency without resulting in a deadlock. To make this run successfully, we need to make use of mutex in addition to Arc. While Arc allows to share safe ownership of an immutable variable (JobStatus structure in our case), Mutex controls the concurrency, i.e only one thread can access the resources at a time. Hence, we can safely mutate the value of jobs_completed one at a time. This is done in line 10. In addition to that, we need to call lock on status_shared in order acquire the mutex and block other threads until it goes out of scope. If this is not implementing properly (i.e if the threads are holding onto the mutex lock while sleeping) it could easily result in a deadlock causing the IDE to timeout.

```
3  use std::time::Duration;
4
5  struct JobStatus {
6      jobs_completed: u32,
7  }
8
9  fn main() {
10     let status = Arc::new(Mutex::new(JobStatus { jobs_completed: 0 }));
11     let status_shared = status.clone();
12     thread::spawn(move || {
13         for _ in 0..10 {
14             thread::sleep(Duration::from_millis(250));
15             status_shared.lock().unwrap().jobs_completed += 1;
16         }
17     });
```

```
Execution

   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.93s
     Running `target/debug/playground`

Standard Output

waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
```