Alexander Semaan
Mark Clancy
Group 17 - CSS

Project 3 - Rust

Variable Bindings

variables1.rs

```rust
fn main() {
    let x = 5;
    println!("x has the value {}", x);
}
```

The variable x was not declared using “let”. So all we had to do was add “let” before x.

variables2.rs

```rust
fn main() {
    let x:u32=10;
    if x == 10 {
        println!("Ten!");
    } else {
        println!("Not ten!");
    }
}
```

The variable x did was not assigned a value nor a type. To solve this, you can either assign an integer value to x, or assign a data type and value which is more accurate. If you assign a data type that is not an integer to x, the code will not compile, which is part of Rust’s type safety measure, to check data type compatibility. In this case x needs to be an integer, signed or unsigned and big enough to fit the value required. In this case I chose an unsigned integer just to test out something other than just i32.

variables3.rs

```rust
fn main() {
    let mut x = 3;
    println!("Number {}", x);
    x = 5;
    println!("Number {}", x);
}
```

The variable x is by default immutable, so when a value is assigned to it, this value cannot be changed. In order to make it mutable, we need to add “mut” right after declaring x with “let”.

variables4.rs

```rust
fn main() {
    let x: i32 = 32;
    println!("Number {}", x);
}
```

Alexander Semaan
Mark Clancy
Group 17 - CSS

X was not assigned a value, which doesn't allow the code to compile because there is nothing to print, and this is a common mistake that can cause bugs, but Rust helps with this.

Functions

functions1.rs

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

The call_me() function was not declared, so the code could not possibly compile.

functions2.rs

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

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

When assigning a parameter to a function, you need to declare what type this parameter is or else, you cannot pass this function any parameters. In this case we need an integer so we added "i32" to num.

functions3.rs

```
fn main() {
    call_me(5);
}

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

The function called in main, "call_me()" did not take any parameters, so the code would not compile. We had to add a parameter to the function call in main, in this case any integer less than 32 bits works.

functions4.rs

Alexander Semaan
Mark Clancy
Group 17 - CSS

```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
}
```

The sale_price function was missing the return data type, so the code did not compile because Rust cannot assume the return data type for functions. It needs to be explicitly stated.

functions5.rs

```rust
fn main() {
    let answer = square(3);
    println!("The answer is {}", answer);
}

fn square(num: i32) -> i32 {
    num * num
}
```

In the function square, we want to have write an expression and not a statement. An expression returns a value whereas a statement doesn't. All we must do is remove the ";" at the end of the statement to make it an expression.

Primitive Types

primitives_types1.rs

Alexander Semaan
Mark Clancy
Group 17 - CSS

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

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

    let is_evening = true;
    if is_evening {
        println!("Good evening!");
    }
}
```

primitives_types2.rs

```rust
fn main() {
    // Characters (`char`)

    let my_first_initial = 'C';
    if my_first_initial.is_alphabetic() {
        println!("Alphabetical!");
    } else if my_first_initial.is_numeric() {
        println!("Numerical!");
    } else {
        println!("Neither alphabetic nor numeric!");
    }

    let your_character = '♪';
    if your_character.is_alphabetic() {
        println!("Alphabetical!");
    } else if your_character.is_numeric() {
        println!("Numerical!");
    } else {
        println!("Neither alphabetic nor numeric!");
    }
}
```

primitives_type3.rs

Alexander Semaan
Mark Clancy
Group 17 - CSS

```rust
fn main() {
    let a = ["Hi"; 100];

    if a.len() >= 100 {
        println!("Wow, that's a big array!");
    } else {
        println!("Meh, I eat arrays like that for breakfast.");
    }
}
```

primitives_type4.rs

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];

    let nice_slice = [a[1], a[2], a[3]];

    if nice_slice == [2, 3, 4] {
        println!("Nice slice!");
    } else {
        println!("Not quite what I was expecting... I see: {:?}", nice_slice);
    }
}
```

primitives_type5.rs

```rust
fn main() {
    let cat = ("Furry McFurson", 3.5);
    let (name, age) = cat;

    println!("{} is {} years old.", name, age);
}
```

primitives_type6.rs

```rust
fn main() {
    let numbers = (1, 2, 3);
    println!("The second number is {}", numbers.1);
}
```

Strings

strings1.rs

Alexander Semaan
Mark Clancy
Group 17 - CSS

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

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

strings2.rs

```rust
fn main() {
    let word = String::from("green"); // Try not changing this line :)
    if is_a_color_word(&word) {
        println!("That is a color word I know!");
    } else {
        println!("That is not a color word I know.");
    }
}

fn is_a_color_word(attempt: &str) -> bool {
    attempt == "green" || attempt == "blue" || attempt == "red"
}
```

strings3.rs

```rust
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());
}
```

Being able to make the difference of when to use Strings and when to use String Slices is very important, and being able to convert from one type to the other helps a great deal.

Move Semantics

move_semantics1.rs

Alexander Semaan
Mark Clancy
Group 17 - CSS

```rust
pub fn main() {
    let vec0 = Vec::new();

    let mut vec1 = fill_vec(vec0);

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

    vec1.push(88);

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

}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
    let mut vec = vec;

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

Vec1 is immutable so it is not possible to change its value using the push function unless we make it a mutable object/variable.

move_semantics2.rs

1.

Alexander Semaan
Mark Clancy
Group 17 - CSS

```rust
pub fn main() {
    let vec0: Vec<i32> = Vec::new();
    let vec2 = Vec::new();

    let mut vec1 = fill_vec(vec2);

    // Do not change the following line!
    println!("{} has length {} content `{:?}`", "vec0", vec0.len(), vec0);

    vec1.push(88);

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

}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
    let mut vec = vec;

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

2.

```rust
pub fn main() {
    let vec0 = Vec::new();

    let mut vec1 = fill_vec(&vec0);

    // Do not change the following line!
    println!("{} has length {} content `{:?}`", "vec0", vec0.len(), vec0);

    vec1.push(88);

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

}

fn fill_vec(vec: &Vec<i32>) -> Vec<i32> {
    let mut vec = vec.clone();

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

3.

Alexander Semaan
Mark Clancy
Group 17 - CSS

```rust
pub fn main() {
    let mut vec0 = Vec::new();

    fill_vec(&mut vec0);

    // Do not change the following line!
    println!("{} has length {} content `{:?}`", "vec0", vec0.len(), vec0);

}

fn fill_vec(vec: &mut Vec<i32>) -> Vec<i32> {

    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec.to_vec()
}
```

move_semantics3.rs

```rust
pub fn main() {
    let vec0 = Vec::new();

    let mut vec1 = fill_vec(vec0);

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

    vec1.push(88);

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

}

fn fill_vec(mut vec: Vec<i32>) -> Vec<i32> {
    vec.push(22);
    vec.push(44);
    vec.push(66);

    vec
}
```

move_semantics4.rs

Alexander Semaan
Mark Clancy
Group 17 - CSS

```rust
pub fn main() {

    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);

    vec
}
```

All move semantics exercises show us the importance of knowing when to make objects mutable and in some cases we need to know where an object's scope ends and how to be able to borrow certain objects in order to keep the program flowing properly.

Threads

threads1.rs

Alexander Semaan
Mark Clancy
Group 17 - CSS

```rust
use std::sync::Arc;
use std::sync::Mutex;
use std::thread;
use std::time::Duration;

struct JobStatus {
    jobs_completed: u32,
}

fn main() {
    let status = Arc::new(Mutex::new(JobStatus { jobs_completed: 0 }));
    let status_shared = status.clone();
    thread::spawn(move || {
        for _ in 0..10 {
            thread::sleep(Duration::from_millis(250));
            status_shared.lock().unwrap().jobs_completed += 1;
        }
    });
    while status.lock().unwrap().jobs_completed < 10 {
        println!("waiting... ");
        thread::sleep(Duration::from_millis(500));
    }
}
```

Arc is an Atomic Reference Counted pointer that allows safe, shared access to immutable data. But we want to change the number of jobs_completed so we'll need to also use another type that will only allow one thread to mutate the data at a time. So we have to use Mutex.

To access the data in a mutex, a thread must first signal that it wants access by asking to acquire the mutex's lock. The lock is a data structure that is part of the mutex that keeps track of who currently has exclusive access to the data. Therefore, the mutex is described as guarding the data it holds via the locking system, which offers a great deal of security and we can control which thread has access and control over variables and data.

The call to lock would fail if another thread holding the lock panicked. In that case, no one would ever be able to get the lock, so we've chosen to unwrap and have this thread panic if we're in that situation.

Hence the use of lock() and unwrap() in order to be able to modify jobs_completed with no trouble.