RUST

A PREPRINT

Louis Galluzzi, Steven Phu, louisgalluzzi@email.arizona.edu, sphu@email.arizona.edu

April 29, 2020

ABSTRACT

1 Introduction

Rust is a programming language with the primary goal of providing a strict and safe coding platform. The Rust developers created a set of rules called Ownership that would enforce users to follow strict guidelines when using variables. Rust developer's attempt at making a safe language with a new set of unique rules has been met with a largely positive reception. While Ownership is confusing to many at first, it has created a language that solves many headaches that generally plague low level languages without impacting performance.

2 History

Rust was designed in 2006 by Graydon Hoare, who was an employee of Mozilla. The project began as a hobby project, but changed when Graydon presented the prototype to his manager. In 2009, Mozilla began sponsoring the project in order to rebuild their browser stack. The main reason for the development of Rust was to make systems programming safer by solving two main issues: memory management and concurrency.

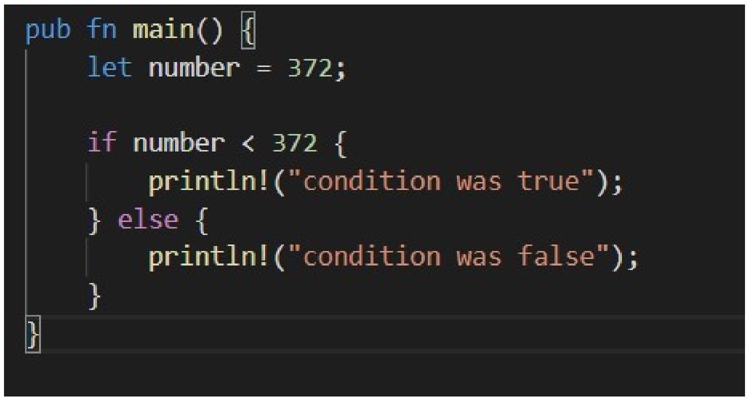
On January 2012, the pre-alpha version of Rust compiler was released. The first stable version did not come out until May 15, 2015. Rust shifted to a self-hosting compiler named rustc in 2010.

Rust began to gain traction in 2015 by winning the third most loved language in Stack Overflow. Since then it has won first place every single year. Rust is still alive with the most stable version being released March 12, 2020. The reddit subcommunity has 94,500 users in it.

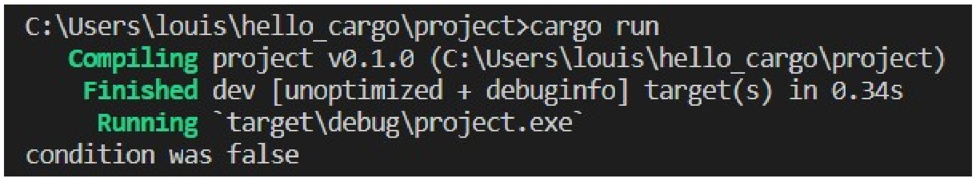
3 Control Structures

3.1 If

The "if" expression is pretty standard with the other languages. It starts with the keyword "if", which is then followed by a condition. If condition evaluates to true, the next block of code that is wrapped in curly brackets will be executed. There is an optional else expression that can be placed after the if block of code. This expression gives the program an alternative block of code that will execute if the "if" expression evaluates to false. If no else expression is provided the if block of code will be skipped. In the example picture below. We assign variable number to 372. The if expression checks if number is less than 372. Since the if expression evaluates to false, it goes to the else expression and prints out "condition was false".



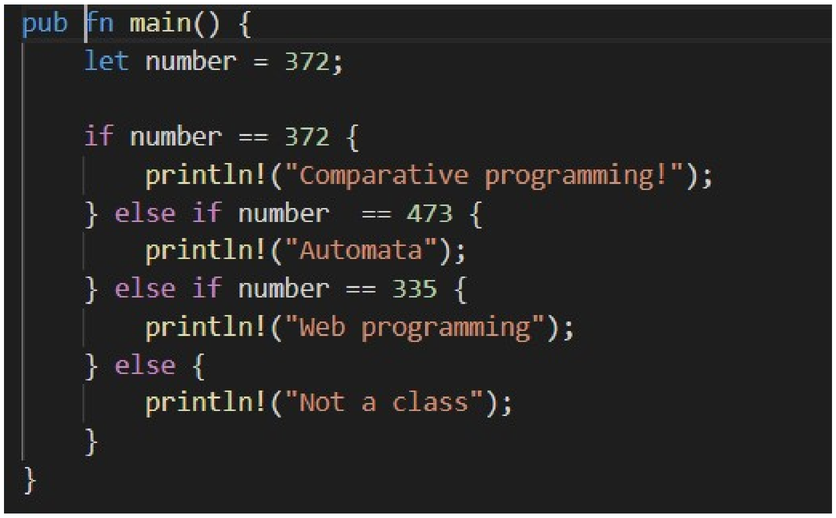
```
pub fn main() {
    let number = 372;

    if number < 372 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```



```
C:\Users\louis\hello_cargo\project>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\project)
    Finished dev [unoptimized + debuginfo] target(s) in 0.34s
     Running `target\debug\project.exe`
condition was false
```

3.2 Else If

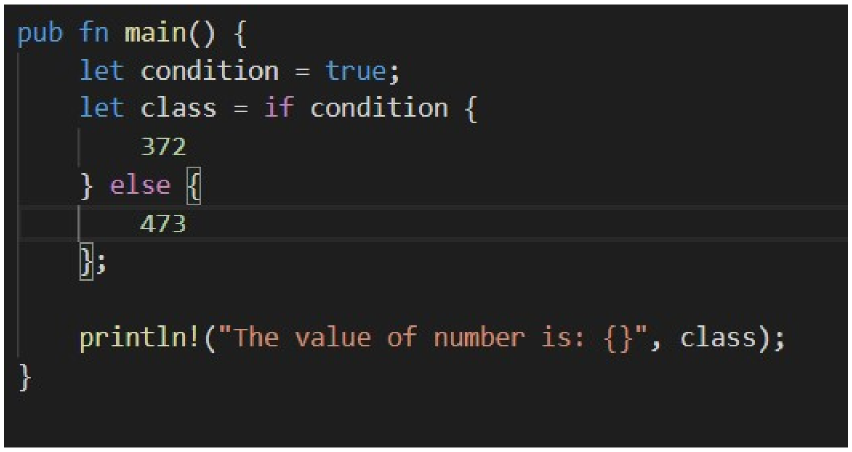
The "else if" expression is used in combination with if expression to allow multiple conditions. It begins with a regular if expression then is followed by an "else if" expression. Each "else if" expression is followed by a condition and a block of code. If the condition evaluates true then the block of code is executed. Otherwise it skips the block of code. Just like the if expression, there is an optional "else" expression.



```
pub fn main() {
    let number = 372;

    if number == 372 {
        println!("Comparative programming!");
    } else if number  == 473 {
        println!("Automata");
    } else if number == 335 {
        println!("Web programming");
    } else {
        println!("Not a class");
    }
}
```

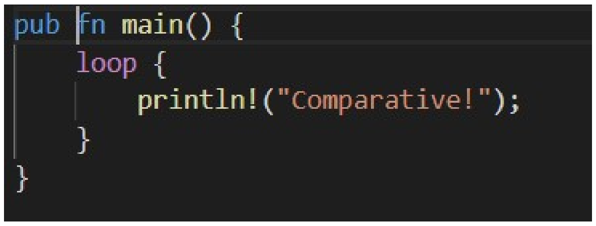
3.3 If Let

Rust has an "if let" expression that allows you to combine if and let to handle values that match a pattern while ignoring the rest. The benefit of this expression is less typing, less indentation, and less code. In our example, we set condition to be true. We set class to equal the first true condition in the if else expression. In this case, class would be assigned 372.


```
pub fn main() {
    let condition = true;
    let class = if condition {
        372
    } else {
        473
    };

    println!("The value of number is: {}", class);
}
```

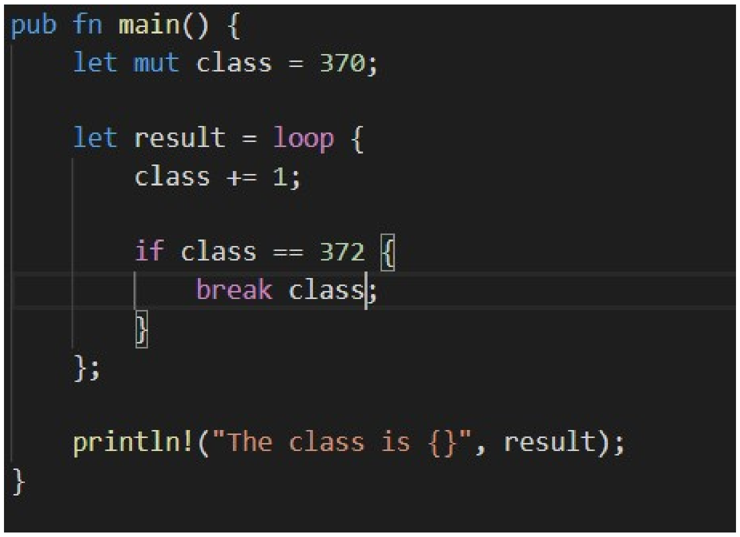
3.4 loop

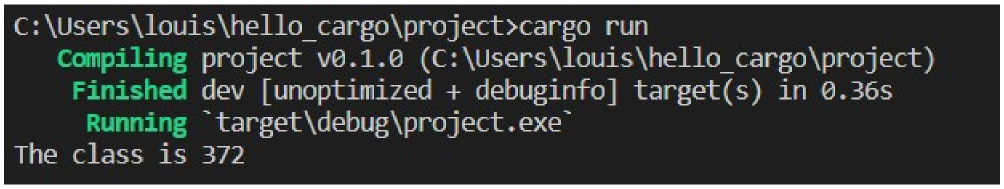
The loop expression allows Rust to execute a code infinitely until the user explicitly tells it to stop. The example below will print the statement "Comparative!" infinitely as there is no break or return.


```
pub fn main() {
    loop {
        println!("Comparative!");
    }
}
```

3.5 Return

Rust has the normal return just like any other languages where it'll return a value from a method with the return expression. Like python Rust will default return None if a return expression does not exist. Rust has another way to return a value from loops with the expression break. In order to achieve this we declare a variable and assign it to loop. Once a break condition has been met, we put the value we want returned after the break. In the example below, we have result hold the return value from the loop. Class is assigned 370 before the loop and increments by 1 for each loop. Once class hits 372, we break out and return class which will be 372. Result then gets returned 372.

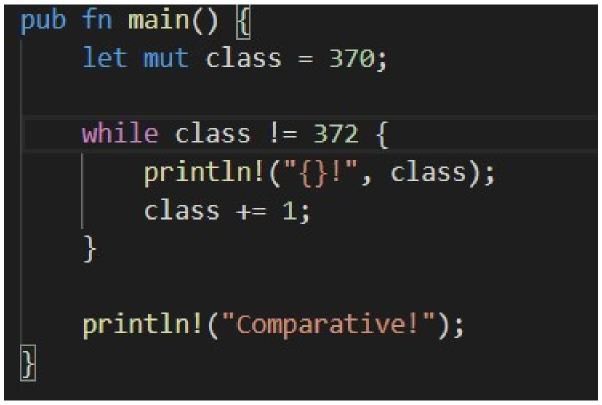

```
pub fn main() {
    let mut class = 370;

    let result = loop {
        class += 1;

        if class == 372 {
            break class;
        }
    };

    println!("The class is {}", result);
}
```



```
C:\Users\louis\hello_cargo\project>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\project)
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
     Running `target\debug\project.exe`
The class is 372
```

3.6 While

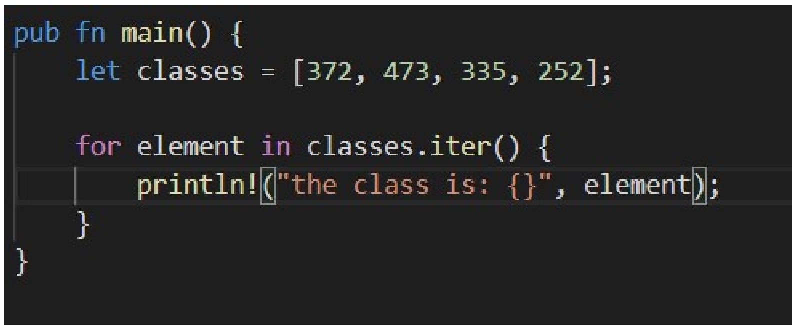
While loop in rust is your standard while loop as in many other languages. It keeps looping a block of code until the condition evaluates false. In the example below, the condition is while class is not 372. The code will print class and increment it by 1 until the condition equates to false.



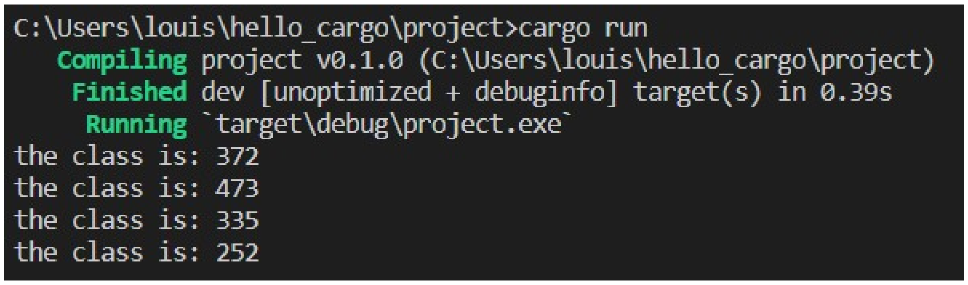
```rust
pub fn main() {
    let mut class = 370;

    while class != 372 {
        println!("{}!", class);
        class += 1;
    }

    println!("Comparative!");
}
```

3.7 For

For loops in Rust works a little bit differently compared to other systems languages. Rust has a struct called Range which can act as the range you want to iterate through. You are also able to iterate through elements in a collection with the .iter() method. In our example below, we are iterating through elements of an array called classes and print out each elements.



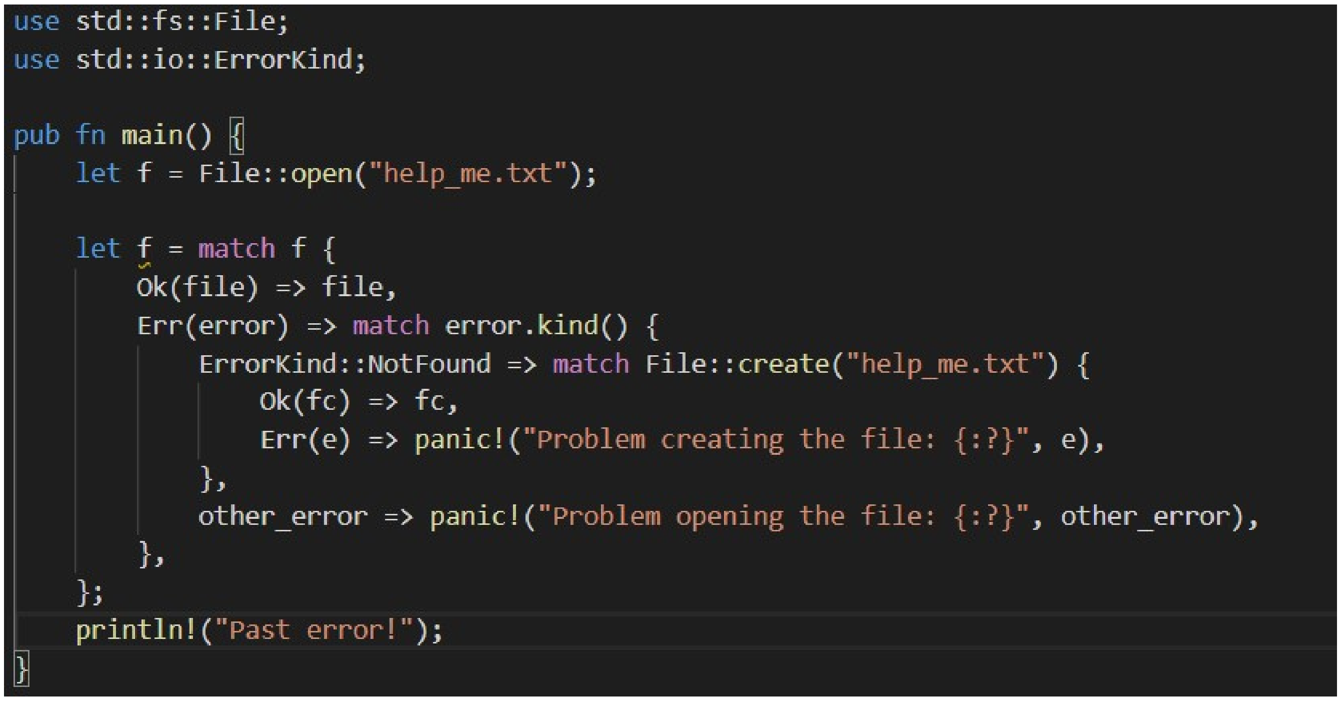
```rust
pub fn main() {
    let classes = [372, 473, 335, 252];

    for element in classes.iter() {
        println!("the class is: {}", element);
    }
}
```



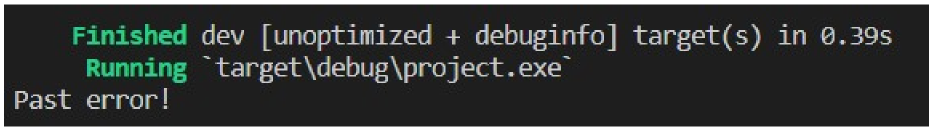
```
C:\Users\louis\hello_cargo\project>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\project)
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
     Running `target\debug\project.exe`
the class is: 372
the class is: 473
the class is: 335
the class is: 252
```

3.8 Result

Result is an enum that contains the variants Ok and Err. It is used for any recoverable errors In our example, we're trying to open a file and assign it to f. f will either get passed back an instance of Ok that contains the file handle or an instance of Err that will contain more information about the type of error that has occurred. Since we have information about the type of error that has occurred, we are able to make prediction on how to handle the situation. In our example, we're trying to open a file that does not exist. When this happens, f will contain a file not found error, which our program will then try to create the file thus preventing the program from crashing. If it does however, fail to create the file, a panic! will occur and thus shut down the code. In this case the program does not crash and we get past this section of code and to our print statement that says "Past error!".



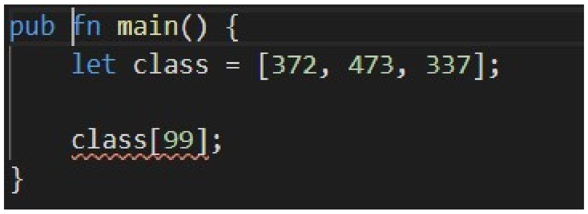
```
use std::fs::File;
use std::io::ErrorKind;

pub fn main() {
    let f = File::open("help_me.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("help_me.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => panic!("Problem opening the file: {:?}", other_error),
        },
    };
    println!("Past error!");
}
```



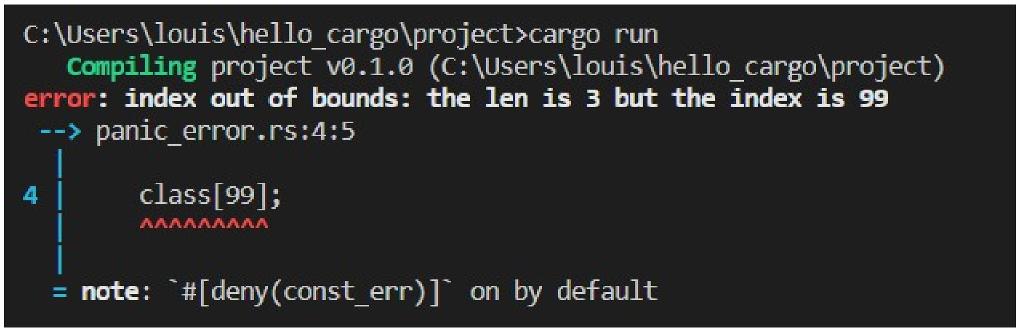
```
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
     Running `target\debug\project.exe`
Past error!
```

3.9 Panic!

Panic! is one of two error handling types of Rust. Panic! is a macro that stops the execution of a software when it encounters an unrecoverable error. Unrecoverable errors are usually symptoms of bugs. In our example we have our program try to access a memory location outside of the array. This will cause a panic! and shut down our code. It'll notify the user of the reason and the line.



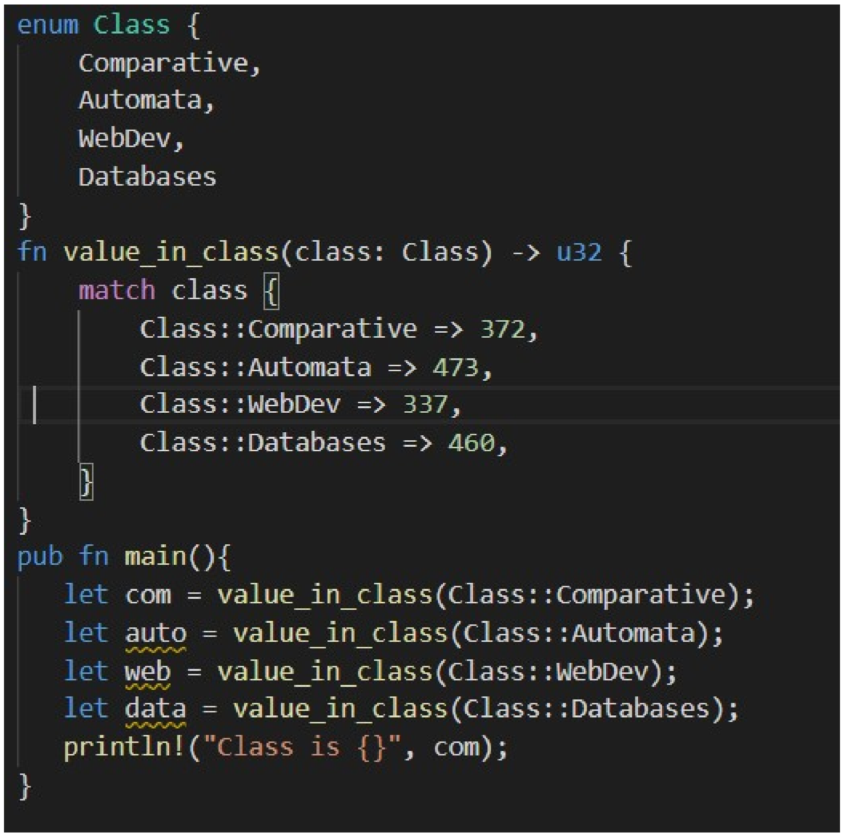
```
pub fn main() {
    let class = [372, 473, 337];

    class[99];
}
```



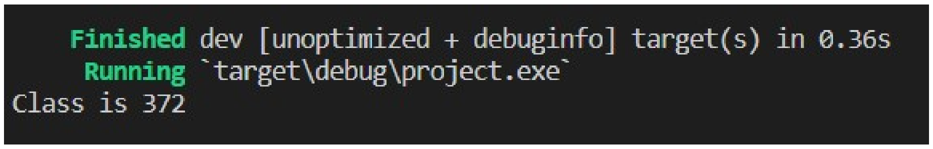
```
C:\Users\louis\hello_cargo\project>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\project)
error: index out of bounds: the len is 3 but the index is 99
 --> panic_error.rs:4:5
  |
4 |     class[99];
  |     ^^^^^^^^^
  |
  = note: `#[deny(const_err)]` on by default
```

3.10 Match

Rust contains a control flow operator called match that allows you to compare a value against a a series of patterns and then execute the code based on which pattern matches. It is very similar to pattern matching in Haskell. The example below shows an enum named Class that has some defined words. The function valueinclass takes in the enum Class type and outputs a 32 bit Int by matching the Class names to a particular 32 bit number.



```
enum Class {
    Comparative,
    Automata,
    WebDev,
    Databases
}
fn value_in_class(class: Class) -> u32 {
    match class {
        Class::Comparative => 372,
        Class::Automata => 473,
        Class::WebDev => 337,
        Class::Databases => 460,
    }
}
pub fn main(){
   let com = value_in_class(Class::Comparative);
   let auto = value_in_class(Class::Automata);
   let web = value_in_class(Class::WebDev);
   let data = value_in_class(Class::Databases);
   println!("Class is {}", com);
}
```



```
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
     Running `target\debug\project.exe`
Class is 372
```

4 Data Types

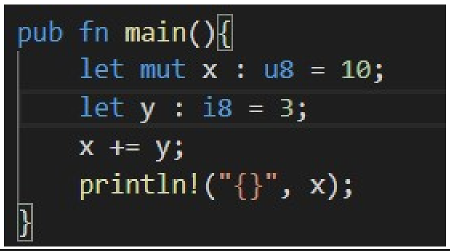
In Rust, data types are statically typed, which means that variable types are known at compile time. Data types are also strongly typed due to Rust emphasizing memory safety. Rust does both explicit and implicit variables. It is implicit by default but it allows the user to explicitly declare their variables. All data types are immutable by default, but users can change it to mutable. There are two categories of data types in Rust: Scalar types and Compound Types.

4.1 Scalar Types

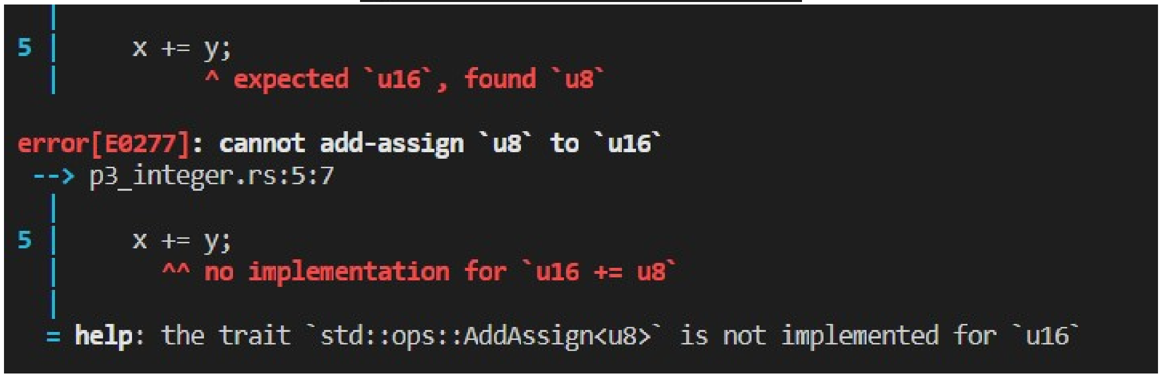
Scalar types represent a single value. Scalar types are made up of four types: Integers, Floats, Booleans, and Characters.

4.1.1 Integer

Integers are whole non fractional numbers. Rust has six different integer lengths: 8-bit, 16-bit, 32-bit, 64-bit, 128-bit, and arch. Each length support signed and unsigned versions. Signed and Unsigned integers refer to whether an integer can be negative or positive. The default integer type is 32-bit.From our testing, Rust doesn't support operations between different integer sizes. We tried adding an 8-bit integer with a 16-bit integer.

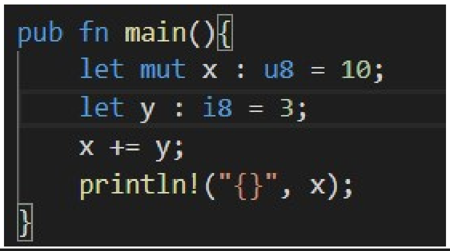


```
pub fn main(){
    let mut x : u8 = 10;
    let y : i8 = 3;
    x += y;
    println!("{}", x);
}
```

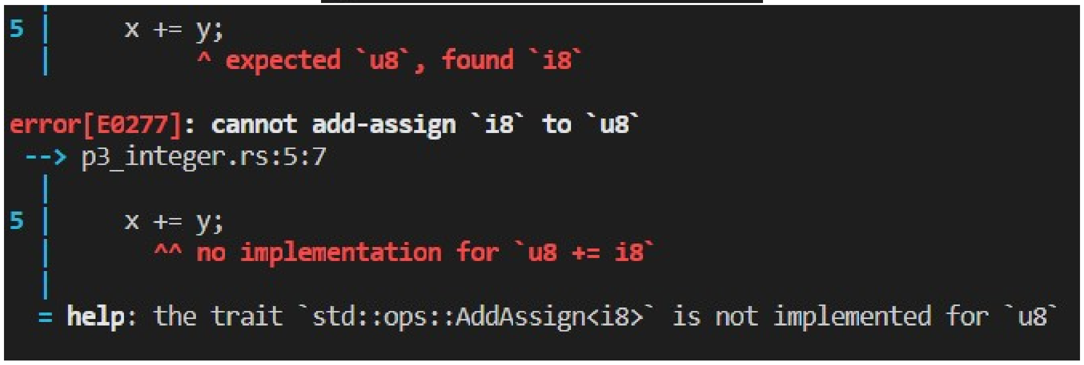


```
  |
5 |     x += y;
  |          ^ expected `u16`, found `u8`

error[E0277]: cannot add-assign `u8` to `u16`
 --> p3_integer.rs:5:7
  |
5 |     x += y;
  |       ^^ no implementation for `u16 += u8`
  |
  = help: the trait `std::ops::AddAssign<u8>` is not implemented for `u16`
```

Rust also doesn't support operations between signed and unsigned integers. In testing, we tried adding an unsigned 8-bit with a signed 8-bit.



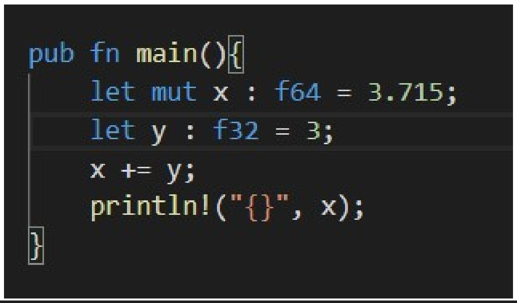
```
pub fn main(){
    let mut x : u8 = 10;
    let y : i8 = 3;
    x += y;
    println!("{}", x);
}
```



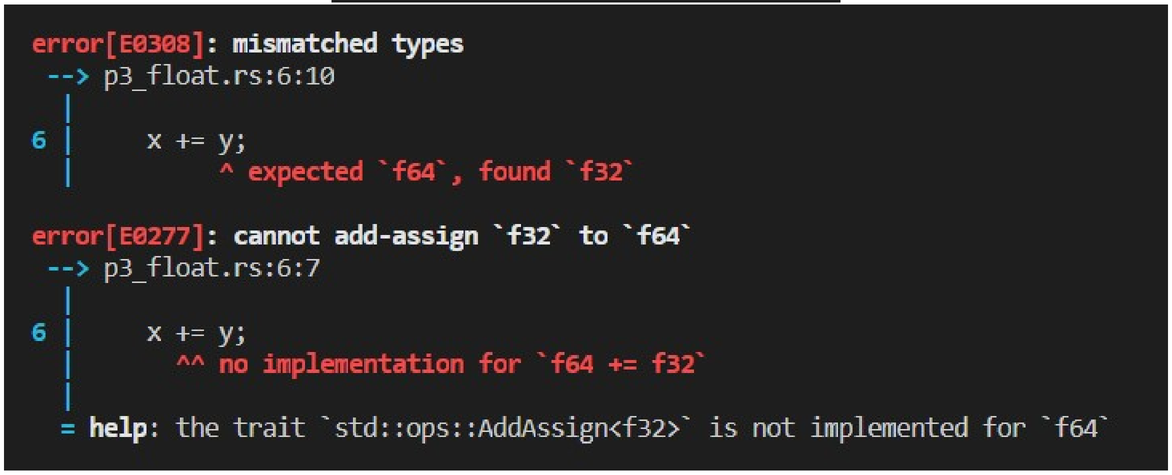
```
5 |     x += y;
  |          ^ expected `u8`, found `i8`

error[E0277]: cannot add-assign `i8` to `u8`
 --> p3_integer.rs:5:7
  |
5 |     x += y;
  |       ^^ no implementation for `u8 += i8`
  |
  = help: the trait `std::ops::AddAssign<i8>` is not implemented for `u8`
```

4.1.2 Floating-Point

Rust has two floating-point types: 64-bit and 32-bit. Floating point data types are numbers with decimal points. The default float type for Rust is 64-bit. Just like integers, Rust doesn't support operations between different float sizes.

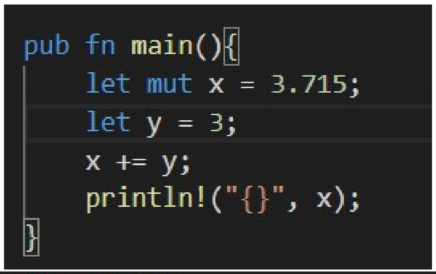


```
pub fn main(){
    let mut x : f64 = 3.715;
    let y : f32 = 3;
    x += y;
    println!("{}", x);
}
```

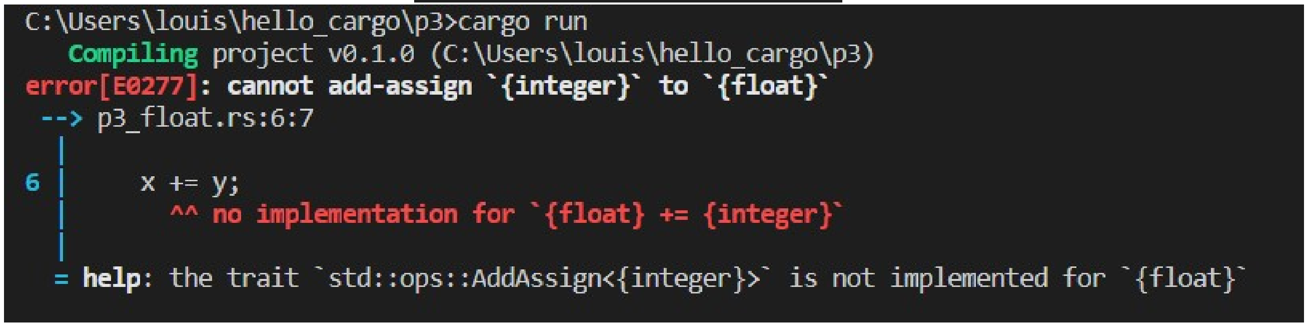


```
error[E0308]: mismatched types
 --> p3_float.rs:6:10
  |
6 |     x += y;
  |          ^ expected `f64`, found `f32`

error[E0277]: cannot add-assign `f32` to `f64`
 --> p3_float.rs:6:7
  |
6 |     x += y;
  |       ^^ no implementation for `f64 += f32`
  |
  = help: the trait `std::ops::AddAssign<f32>` is not implemented for `f64`
```

Some languages allow operators to handle floats and integers. Unfortunately Rust is not one of them. In the picture below, Rust will throw an error when adding a float and an integer together.



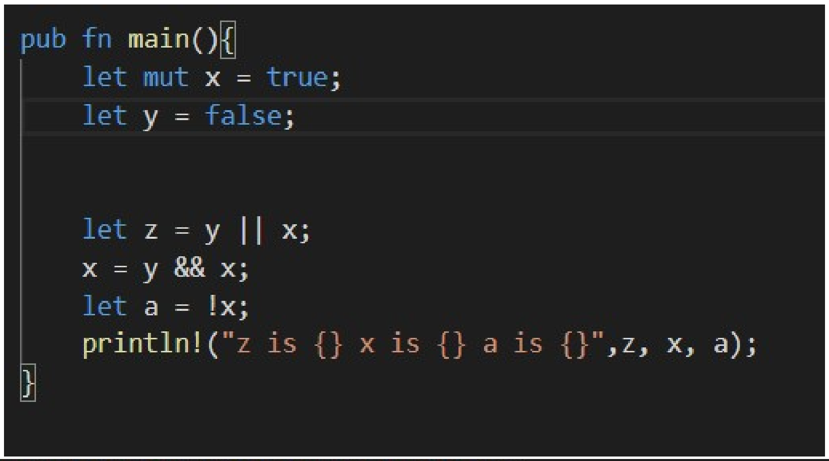
```
pub fn main(){
    let mut x = 3.715;
    let y = 3;
    x += y;
    println!("{}", x);
}
```



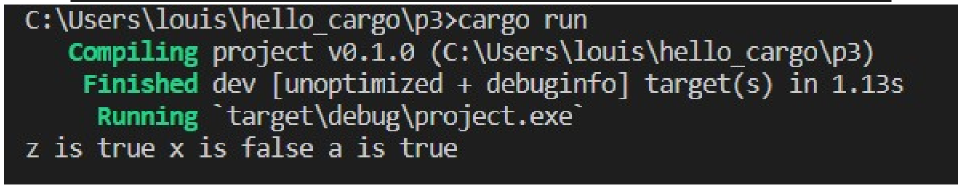
```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
error[E0277]: cannot add-assign `{integer}` to `{float}`
 --> p3_float.rs:6:7
  |
6 |     x += y;
  |       ^^ no implementation for `{float} += {integer}`
  |
  = help: the trait `std::ops::AddAssign<{integer}>` is not implemented for `{float}`
```

4.1.3 Boolean

Just like every other language, Rust has two values for booleans: true and false. Booleans are one byte in size. Rust has the typical operators that support booleans types: &&(and), !(not), and ||(or). The example below shows a simple code where x is assigned true and y is assigned false. We assign z to the OR of y and x. Since x is mutable, we assigned x to y AND x. Finally we assigned a to NOT of x.

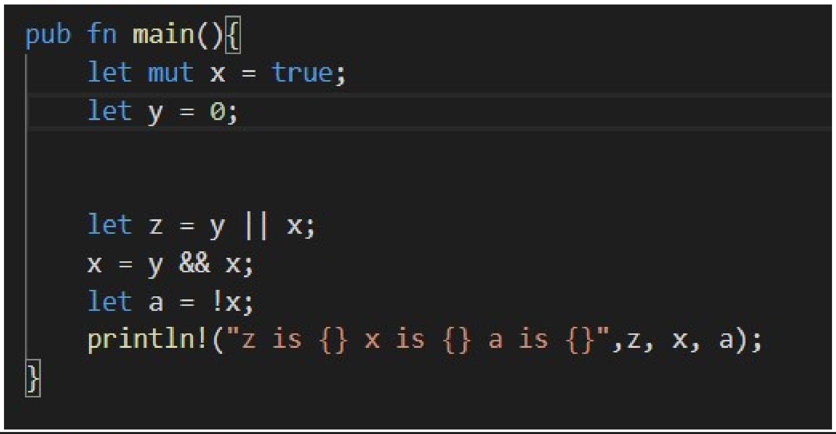


```
pub fn main(){
    let mut x = true;
    let y = false;


    let z = y || x;
    x = y && x;
    let a = !x;
    println!("z is {} x is {} a is {}",z, x, a);
}
```

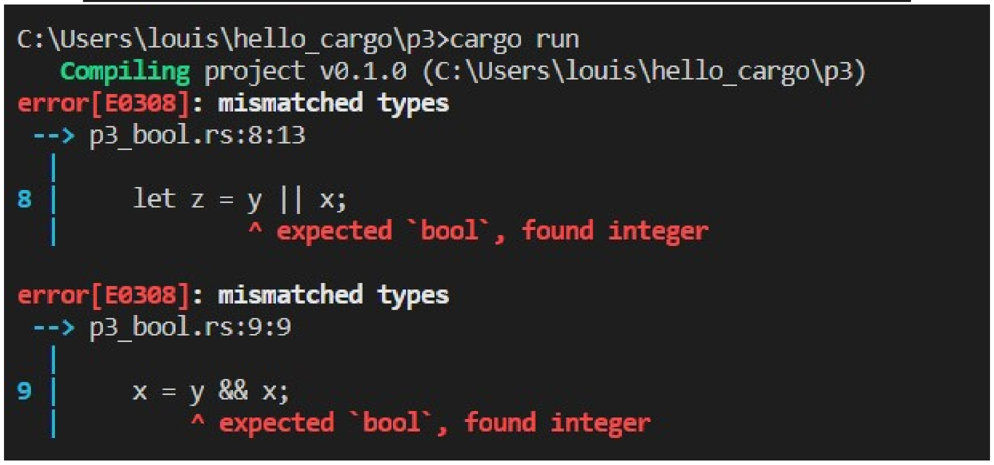


```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
    Finished dev [unoptimized + debuginfo] target(s) in 1.13s
     Running `target\debug\project.exe`
z is true x is false a is true
```

We also did a test if Rust supported 0 and 1 representation of true and false. In the snippet below, it shows that Rust does not support this feature.



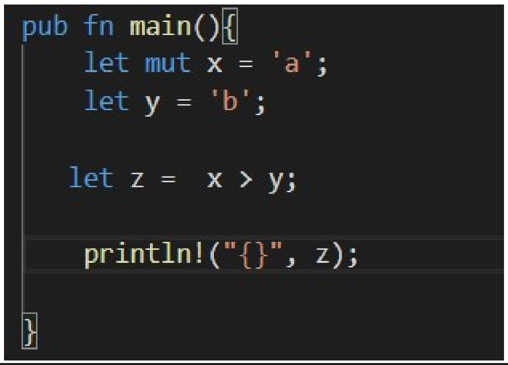
```
pub fn main(){
    let mut x = true;
    let y = 0;


    let z = y || x;
    x = y && x;
    let a = !x;
    println!("z is {} x is {} a is {}",z, x, a);
}
```



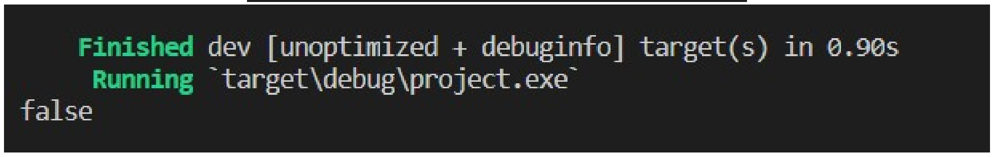
```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
error[E0308]: mismatched types
 --> p3_bool.rs:8:13
  |
8 |     let z = y || x;
  |             ^ expected `bool`, found integer

error[E0308]: mismatched types
 --> p3_bool.rs:9:9
  |
9 |     x = y && x;
  |         ^ expected `bool`, found integer
```

4.1.4 Character

Characters in Rust are 4 bytes in size and are based of Unicode instead of ASCII. This allows Rust to support other language characters and emojis too. In our code we tested if greater than operators support Characters, which they do.



```
pub fn main(){
    let mut x = 'a';
    let y = 'b';

   let z =  x > y;

    println!("{}", z);

}
```



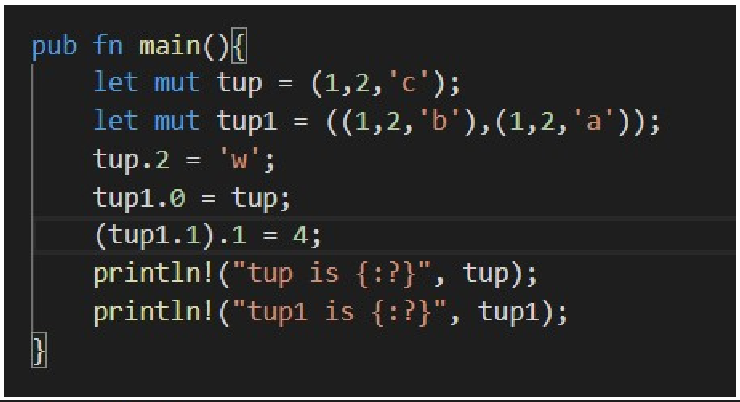
```
    Finished dev [unoptimized + debuginfo] target(s) in 0.90s
     Running `target\debug\project.exe`
false
```

4.2 Compound Types

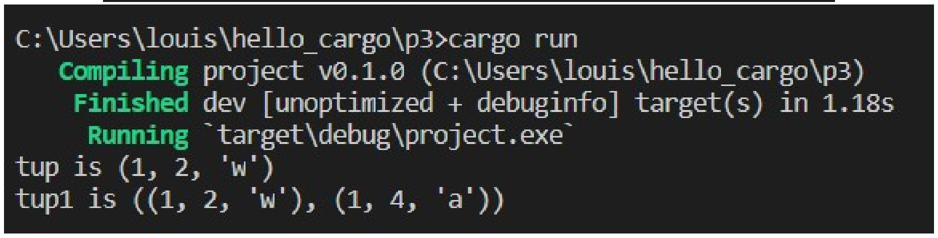
Compound Types are multiple values grouped into one type. There are two primitive compound types: tuples and arrays.

4.2.1 Tuple

Tuples in Rust are used to group different types of data into one compound type. Tuples' size are fixed, so once they're declared, it can not change. Also once a type has been assigned to that index, the type can not be changed. We can change the value however. In our example, we created a variable named tup which holds 2 integers and a character. Then we created another tuple variable named tup1, which is a tuple of tuples, that has the same type signature as tup. We changed the character element of tup index 2 to 'w' and assigned tup1's 0 index to tup.

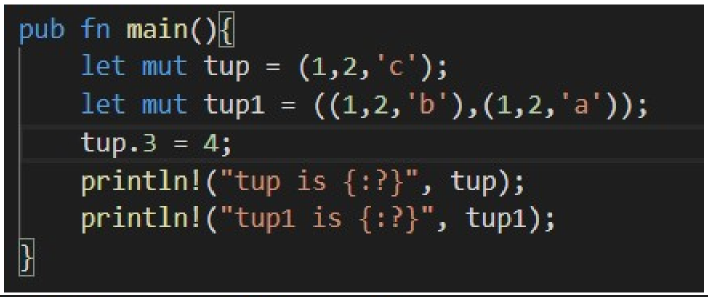


```
pub fn main(){
    let mut tup = (1,2,'c');
    let mut tup1 = ((1,2,'b'),(1,2,'a'));
    tup.2 = 'w';
    tup1.0 = tup;
    (tup1.1).1 = 4;
    println!("tup is {:?}", tup);
    println!("tup1 is {:?}", tup1);
}
```

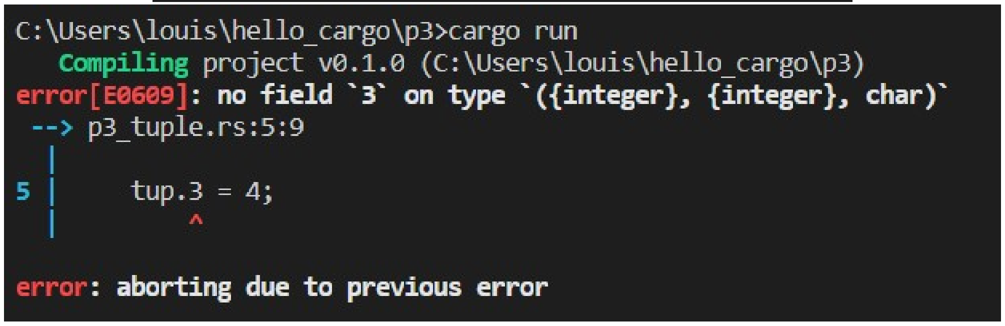


```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
    Finished dev [unoptimized + debuginfo] target(s) in 1.18s
     Running `target\debug\project.exe`
tup is (1, 2, 'w')
tup1 is ((1, 2, 'w'), (1, 4, 'a'))
```

Below is an example of tuple's immutable length. We tried accessing an index outside of tuple's original size.

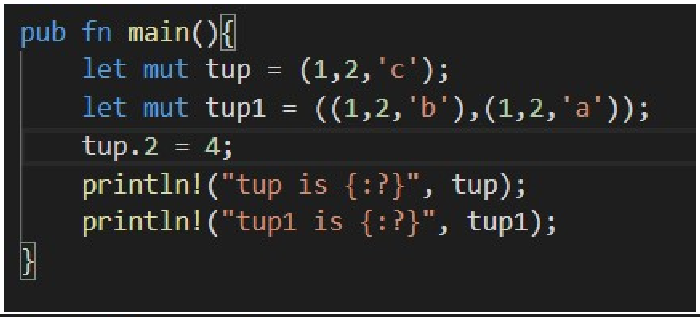


```
pub fn main(){
    let mut tup = (1,2,'c');
    let mut tup1 = ((1,2,'b'),(1,2,'a'));
    tup.3 = 4;
    println!("tup is {:?}", tup);
    println!("tup1 is {:?}", tup1);
}
```

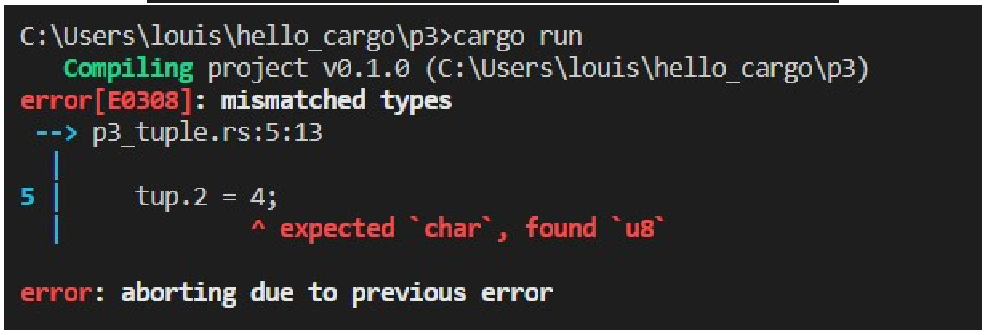


```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
error[E0609]: no field `3` on type `({integer}, {integer}, char)`
 --> p3_tuple.rs:5:9
  |
5 |     tup.3 = 4;
  |         ^

error: aborting due to previous error
```

This example shows how tuple's type signature can not be changed. We tried changing index 2 of the tuple from a character to an integer.



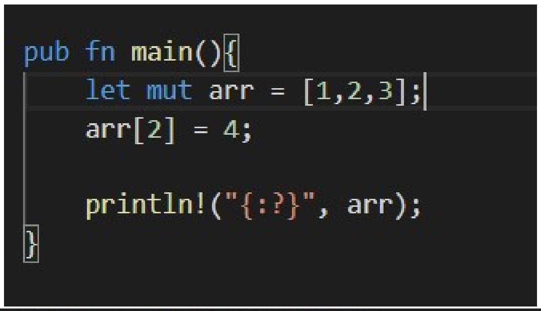
```
pub fn main(){
    let mut tup = (1,2,'c');
    let mut tup1 = ((1,2,'b'),(1,2,'a'));
    tup.2 = 4;
    println!("tup is {:?}", tup);
    println!("tup1 is {:?}", tup1);
}
```



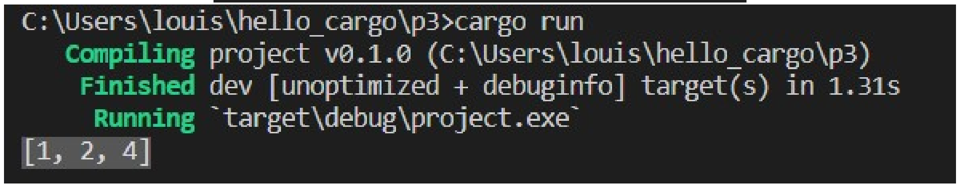
```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
error[E0308]: mismatched types
 --> p3_tuple.rs:5:13
  |
5 |     tup.2 = 4;
  |             ^ expected `char`, found `u8`

error: aborting due to previous error
```

4.2.2 Array

Arrays in Rust are very similar to it's tuple data type. The key difference between the tuple and array is that all the elements in an array must be the same data type and arrays uses the stack memory instead of heap. Below is an example of a simple array with integer elements. We can change the data in the array and print it out.

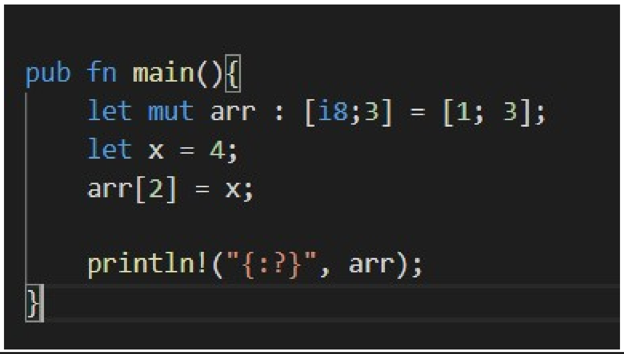


```
pub fn main(){
    let mut arr = [1,2,3];
    arr[2] = 4;

    println!("{:?}", arr);
}
```

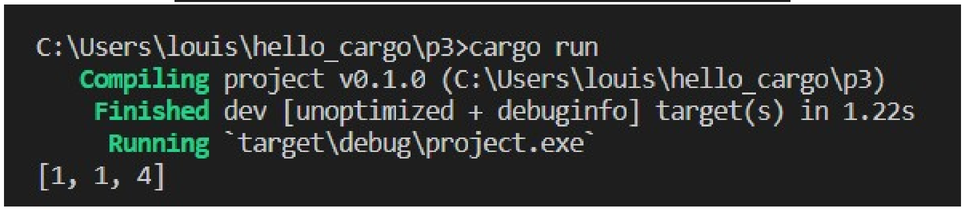


```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
    Finished dev [unoptimized + debuginfo] target(s) in 1.31s
     Running `target\debug\project.exe`
[1, 2, 4]
```

This example shows how arrays can be initialized with preset values. Arr is declared as an 8-bit integer array with a size of 3 with values of 1.

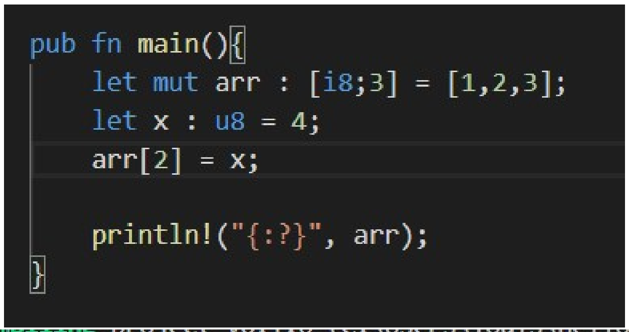


```
pub fn main(){
    let mut arr : [i8;3] = [1; 3];
    let x = 4;
    arr[2] = x;

    println!("{:?}", arr);
}
```

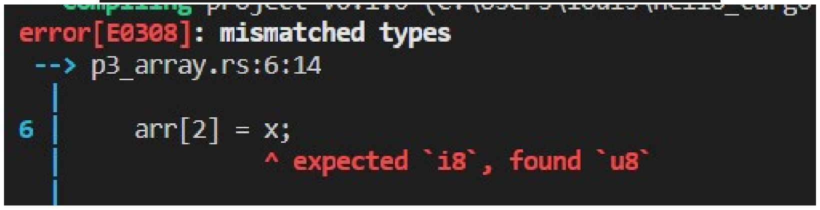


```
C:\Users\louis\hello_cargo\p3>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p3)
    Finished dev [unoptimized + debuginfo] target(s) in 1.22s
     Running `target\debug\project.exe`
[1, 1, 4]
```

This example shows the error that occurs when changing the data type of an array from an signed 8-bit integer to an unsigned 8-bit integer.



```
pub fn main(){
    let mut arr : [i8;3] = [1,2,3];
    let x : u8 = 4;
    arr[2] = x;

    println!("{:?}", arr);
}
```



```
error[E0308]: mismatched types
 --> p3_array.rs:6:14
  |
6 |     arr[2] = x;
  |              ^ expected `i8`, found `u8`
  |
```

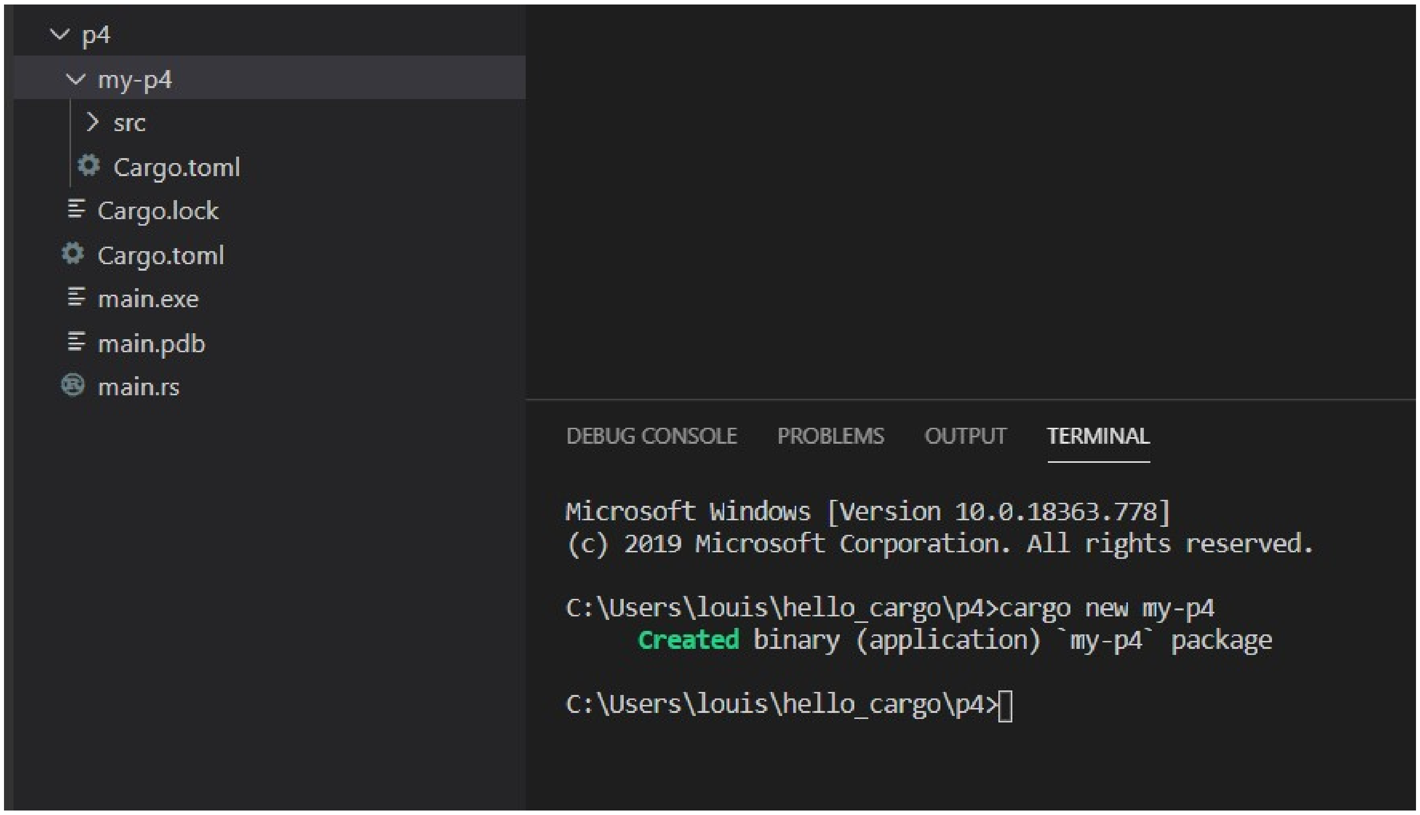
5 Subprograms

Rust has a module system that includes numerous features that allow you to manage your code's organization. These range from the privacy of data within your code to names that are in each scope. The module system is comprised of:
1. Packages: A Cargo feature that lets you build, test, and share crates
2. Crates: a tree of modules that produces a library or executable
3. Modules and Use: This lets you control the organization, scope, and privacy
4. Paths: a way of naming data types

5.1 Packages and Crates

Packages in Rust is one or more crates that provide functionality. Within the package is a Cargo.toml file that describes how to build crates. There are several rules that determine what package must contain:
1. Package must contain 0 or one library
2. May contain as many binary crates as you like
3. Must contain at least 1 crate (binary or library)

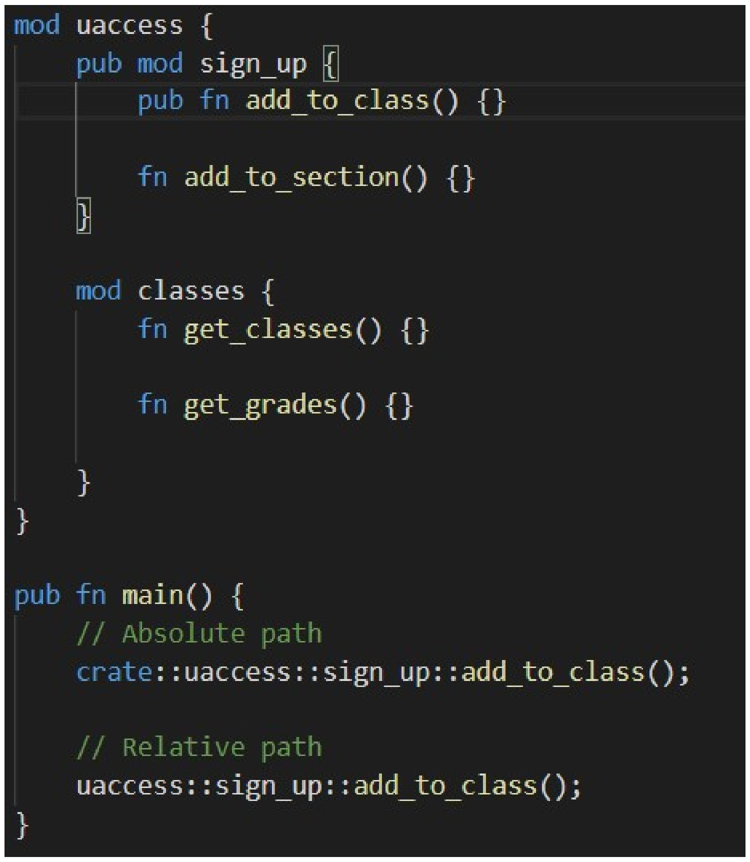
Below is an example on how to create a package



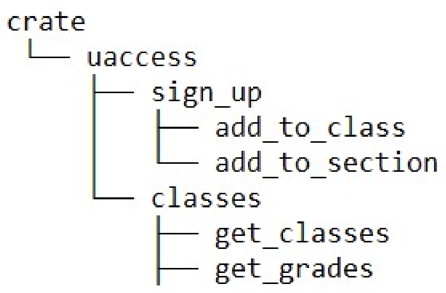
```
Microsoft Windows [Version 10.0.18363.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\louis\hello_cargo\p4>cargo new my-p4
     Created binary (application) `my-p4` package

C:\Users\louis\hello_cargo\p4>
```

5.2 Modules and Use

Modules allows us to organize codes within a crate into groups for readability and ease of use. It also controls the privacy of data(public or private). In our example below, we define the module with the mod keyword along with with name of the module(uaccess). We have another module within this module named signup and classes. Note that the module signup is public which is indicated by the pub keyword in front, while the other one is defaulted to private. Modules can hold structs, enums, constants, or in our case functions. This trait of modules allows us to group related items together and name why they're related.



```
mod uaccess {
    pub mod sign_up {
        pub fn add_to_class() {}

        fn add_to_section() {}
    }

    mod classes {
        fn get_classes() {}

        fn get_grades() {}

    }
}

pub fn main() {
    // Absolute path
    crate::uaccess::sign_up::add_to_class();

    // Relative path
    uaccess::sign_up::add_to_class();
}
```



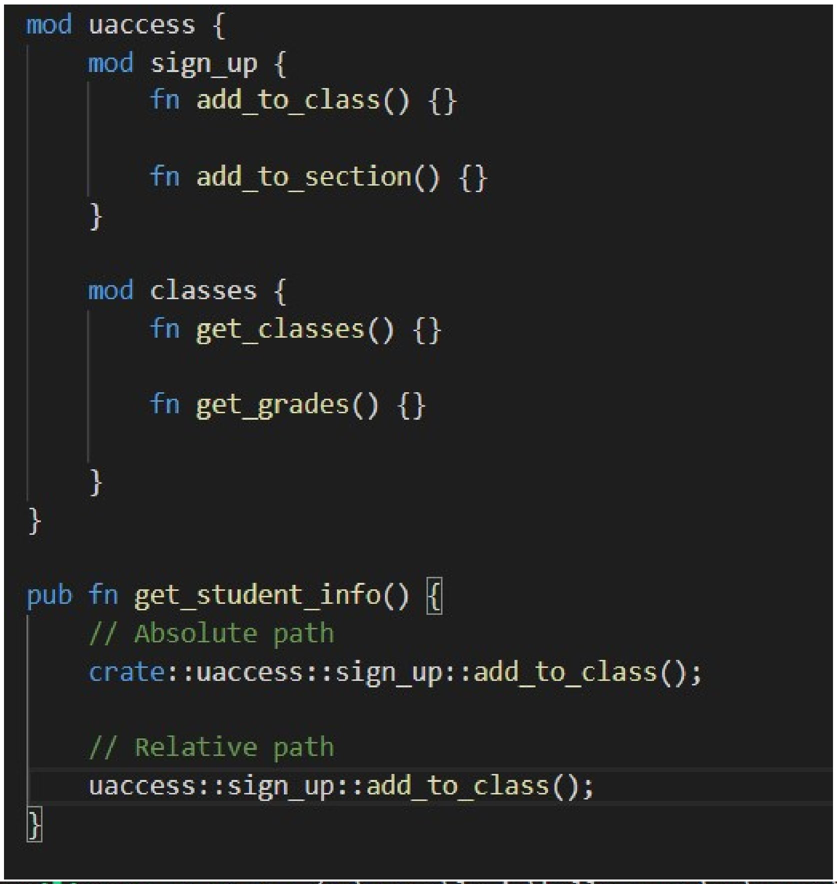
```
crate
 └── uaccess
      ├── sign_up
      │    ├── add_to_class
      │    └── add_to_section
      └── classes
           ├── get_classes
           ├── get_grades
```

Module tree of our crate example

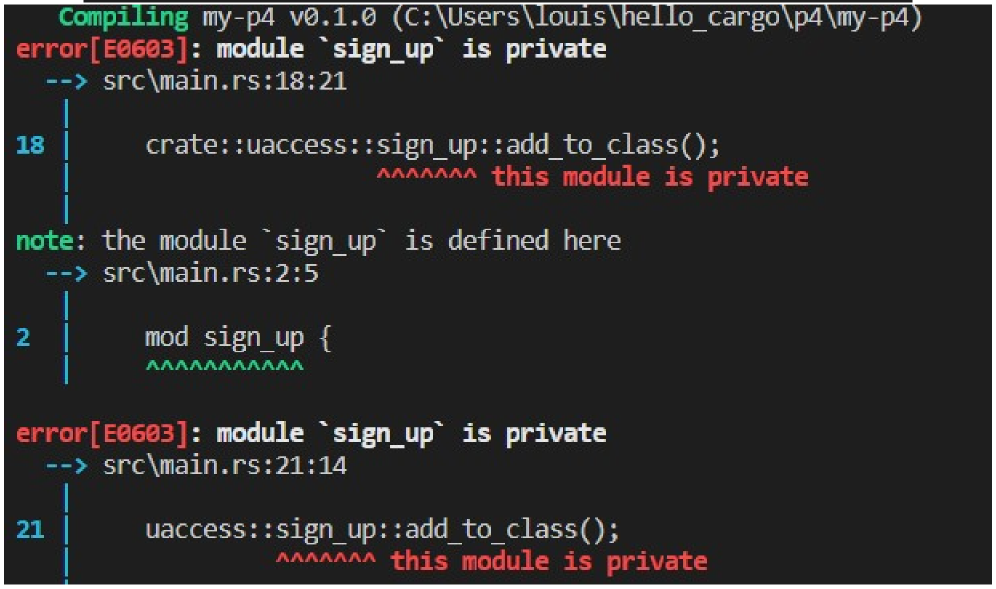
5.3 Paths

Rust uses path to navigate through a module tree. Each identifier is separated by double colons. There are two types of paths:
1. absolute path: starts from the crate root by using the name
2. relative path: starts from the current module and uses self, super, or an identifier in the current module

In our example below, we show how to use both paths to call a different module. Our function get_student_info() uses both paths to access add_to_class() function that is within signup module that is within uaccess module. "use" is not only limited to modules within the same package. It can also be used to bring in external packages. You can also bring in multiple data from a module by wrapping around the module. If you want to import everything that data in a particular module, you use the keyword "*".

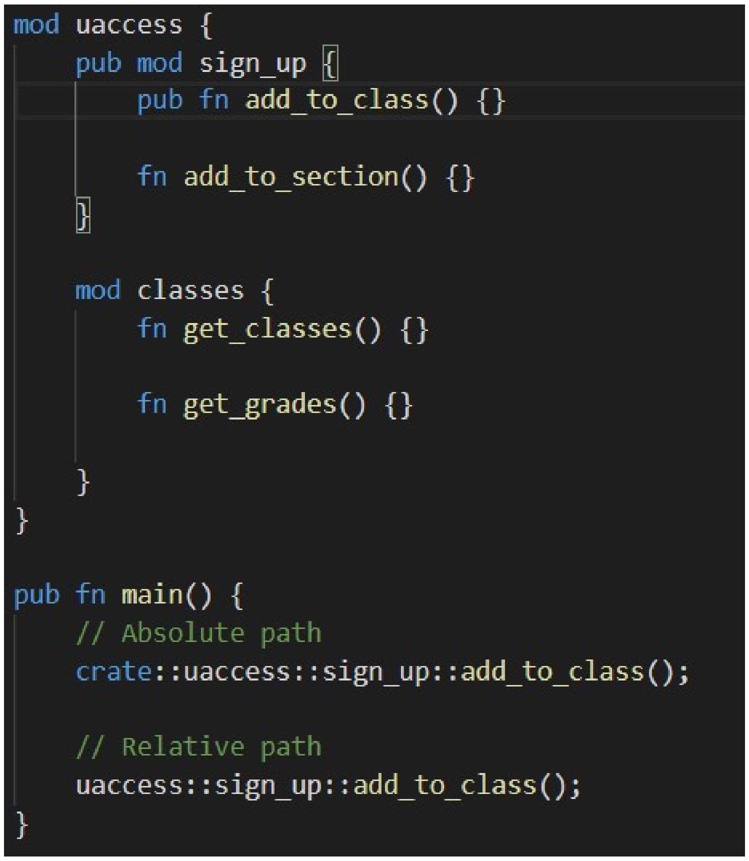


```rust
mod uaccess {
    mod sign_up {
        fn add_to_class() {}

        fn add_to_section() {}
    }

    mod classes {
        fn get_classes() {}

        fn get_grades() {}

    }
}

pub fn get_student_info() {
    // Absolute path
    crate::uaccess::sign_up::add_to_class();

    // Relative path
    uaccess::sign_up::add_to_class();
}
```



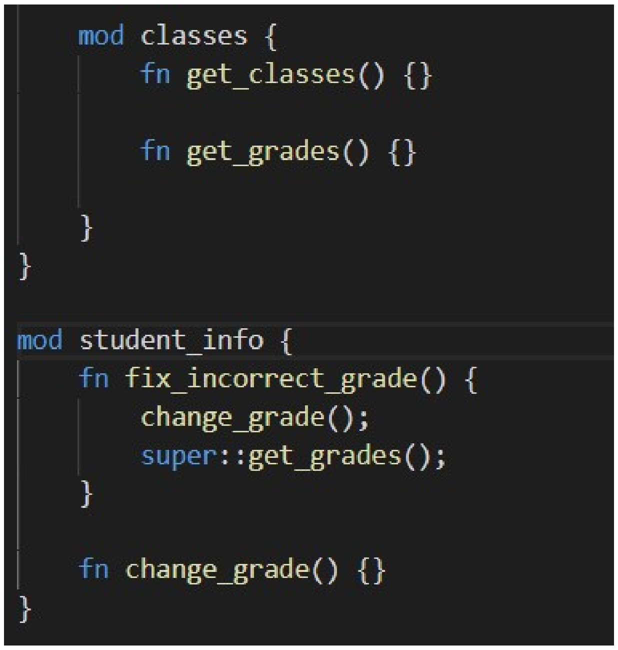
```
   Compiling my-p4 v0.1.0 (C:\Users\louis\hello_cargo\p4\my-p4)
error[E0603]: module `sign_up` is private
  --> src\main.rs:18:21
   |
18 |     crate::uaccess::sign_up::add_to_class();
   |                     ^^^^^^^ this module is private
   |
note: the module `sign_up` is defined here
  --> src\main.rs:2:5
   |
2  |     mod sign_up {
   |     ^^^^^^^^^^^

error[E0603]: module `sign_up` is private
  --> src\main.rs:21:14
   |
21 |     uaccess::sign_up::add_to_class();
   |              ^^^^^^^ this module is private
```

This however will cause an error. This is due to the module being private. In order to access uaccess module and its contents, we will have to make it public. In the image below, you can see that we have to make signup and add_to_class public in order to access it. This build-in default provides safety while coding in Rust. For example if we create a public struct, the fields are still private and must be declared public in order for them to be accessed.



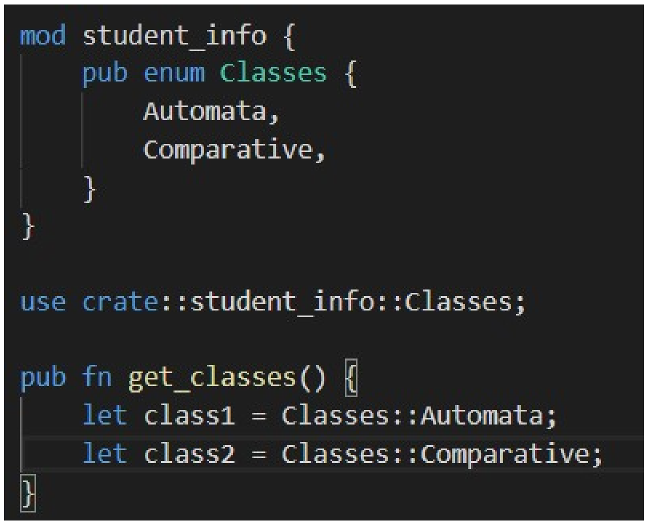
```rust
mod uaccess {
    pub mod sign_up {
        pub fn add_to_class() {}

        fn add_to_section() {}
    }

    mod classes {
        fn get_classes() {}

        fn get_grades() {}

    }
}

pub fn main() {
    // Absolute path
    crate::uaccess::sign_up::add_to_class();

    // Relative path
    uaccess::sign_up::add_to_class();
}
```

We can also create relative paths that begin with the parent module by using the keyword super at the start of the path.



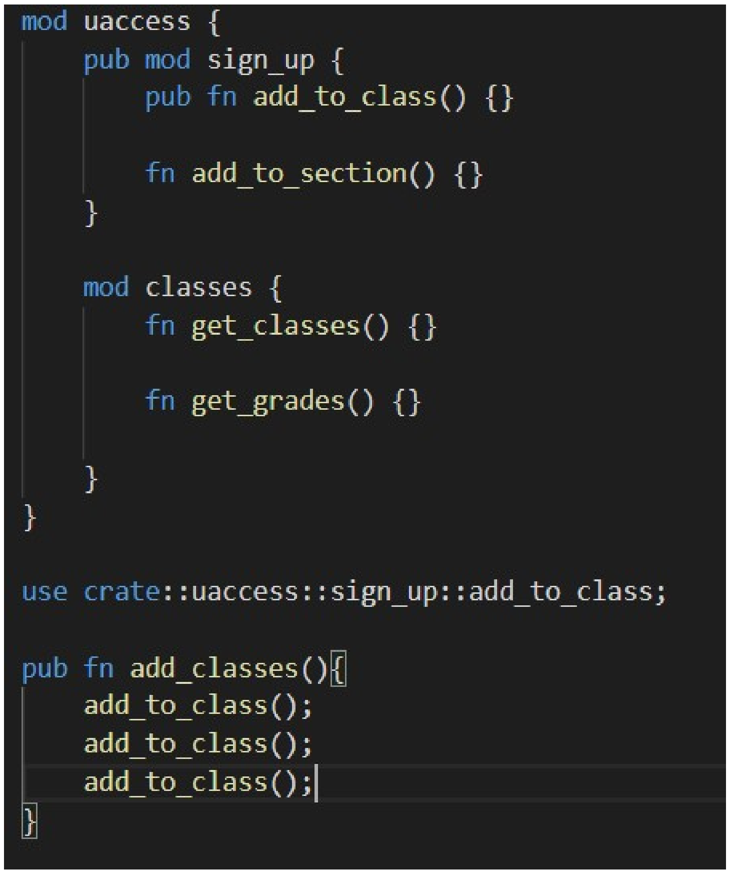
```rust
    mod classes {
        fn get_classes() {}

        fn get_grades() {}

    }
}

mod student_info {
    fn fix_incorrect_grade() {
        change_grade();
        super::get_grades();
    }

    fn change_grade() {}
}
```

Rust has a neat little feature for lazy coders. Writing out the paths can be long and repetitive. Rust allows "use" to call the items paths and treat it as a local variable. Adding "use" along with a path in a scope is similar to creating a symbolic link in the file system. In our example we have use crate::student_info::Classes, Classes is now a valid name in the scope. Now we can call the public enum Classes within student_info and assign it to class1 and class2.



```rust
mod student_info {
    pub enum Classes {
        Automata,
        Comparative,
    }
}

use crate::student_info::Classes;

pub fn get_classes() {
    let class1 = Classes::Automata;
    let class2 = Classes::Comparative;
}
```

We are also able to create an unidiomatic path by calling all the way to the function in the "use". This however makes it ambiguous where the function is defined. In our example below, we have use crate::uaccess::sign_up::add_to_class which allows us to just call add_to_class() function by itself. The idiomatic way would be to "use::uaccess:sign_up;" Then we would have to call sign_up::add_to_class();. The idiomatic has become more of the conventional way.



```rust
mod uaccess {
    pub mod sign_up {
        pub fn add_to_class() {}

        fn add_to_section() {}
    }

    mod classes {
        fn get_classes() {}

        fn get_grades() {}

    }
}

use crate::uaccess::sign_up::add_to_class;

pub fn add_classes(){
    add_to_class();
    add_to_class();
    add_to_class();
}
```

Rust has another feature that helps with situation where if we're using two different functions with the same name from different modules. We use the keyword "as" after the declaration to assign it a different name.

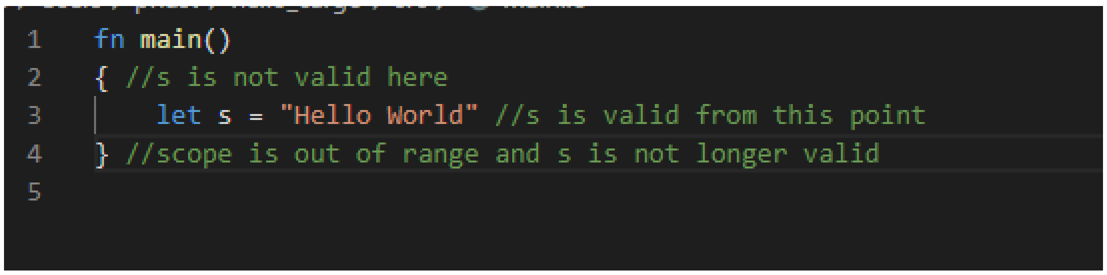
6 Ownership

The most unique feature that Rust has is ownership. Ownership is the feature that allows Rust to guarantee memory safety without a garbage collector. Every single language has some way to manage memory while running. Some use a garbage collector to look for memory no longer used, others have the user allocate and free memory. Rust uses ownership to manage memory with set of rules that compiler checks at compile time. The beauty of ownership is that it does not affect performance unlike garbage collection and it solves the safety issue of having users manually allocate/free memory. In our examples, we'll be using string data types. This is because Rust has two types of String: literal and type. Literal strings are essentially primitive type where the pointer to the value is stored on stack and the value is stored in read only memory and is immutable. The String type can be compared to complex data types and is stored in the heap and are immutable by default but can be made mutable by keyword "mut". This allows us to show how ownership works with simple and complex data.

Three rules of Ownership in Rust:
1. Each value in Rust has a variable that's called its owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the values will be dropped.

6.1 Variable Scope

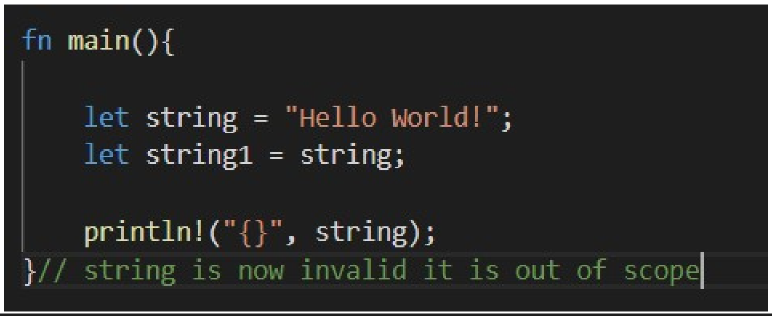
In our example below, we'll discuss how Rust differs with variable scopes compared to other languages. The image below has s be assigned a literal string. The moment s does this assignment, the variable is in scope and can be used until the end of method. After this method ends, the variable s is out of scope and can not be used. This is typical for many languages. Where Rust differs is how the memory is dealt with after the variable exits scope. Normally a garbage collector has to free the memory or the user has to manually free it. Rust instead has the memory automatically return once the variable goes out of scope by calling the drop function and cleaning up the memory for that variable.



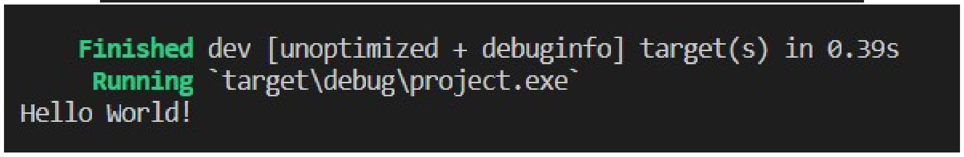
```rust
fn main()
{ //s is not valid here
    let s = "Hello World" //s is valid from this point
} //scope is out of range and s is not longer valid

```

6.2 Ways Variable and Data Interact

It is very common to have variables reference other variables in programming. There are two types of ways this is done: deep copy and shallow copy. In the example below, we have a literal variable called string that points to the value "Hello World!" in read-only memory. String1 is then assigned to it and does a deep copy of string pointer and both are stored on the stack. Since literal strings aren't affected in the same way by ownership as String types, there will be no compile issues.

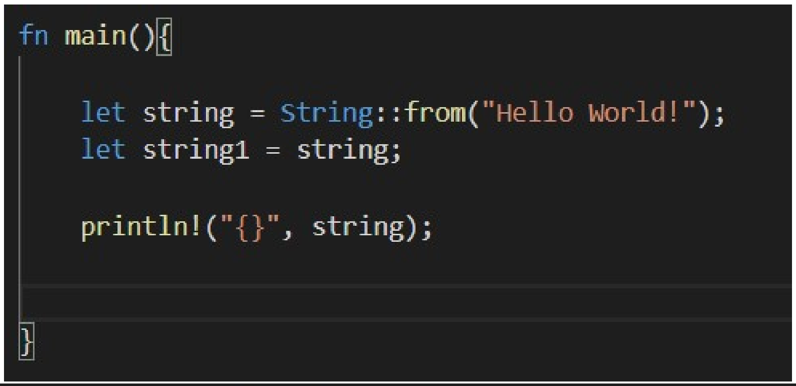


```
fn main(){

    let string = "Hello World!";
    let string1 = string;

    println!("{}", string);
}// string is now invalid it is out of scope
```

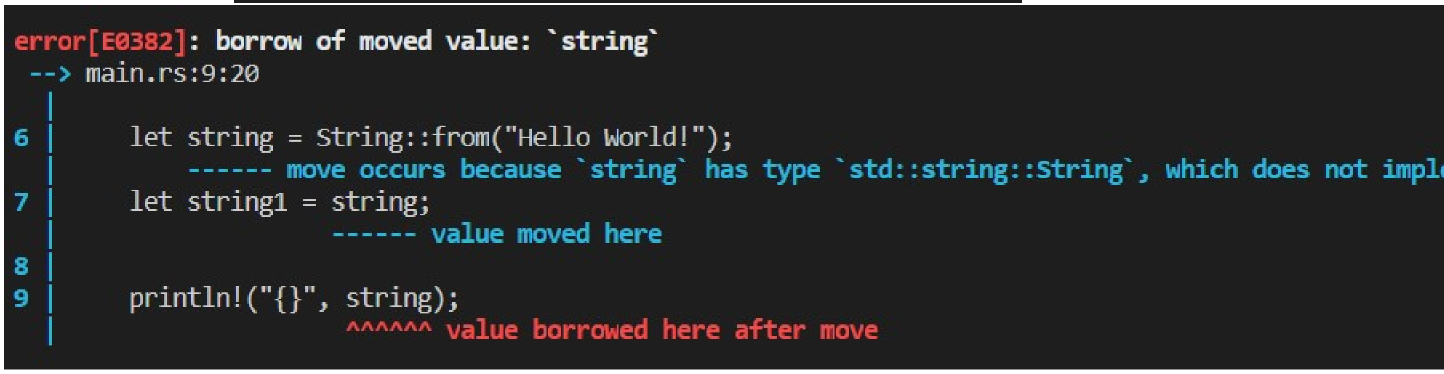


```
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
     Running `target\debug\project.exe`
Hello World!
```

In this particular example, we have a String type. We have variable string assigned the value "Hello World!" except this it is a String type. When variable string1 is assigned to variable string, instead of a deep copy, it does a shallow copy. This means that both variables reference the same memory. This is where Rust's unique ownership feature comes into play. Ownership of that memory transfers from variable string to string1 and variable string goes out of scope. This feature is designed to fix the potential issue of a double free error if the user were to try to free both string and string1. As you can see in the image, this particular example causes an error due to variable string going out of scope. The reason Rust does not implement a deep copy by default is because String types are stored on the heap and it would be very expensive. But should the user want to they could do a deep copy using the .clone() method.



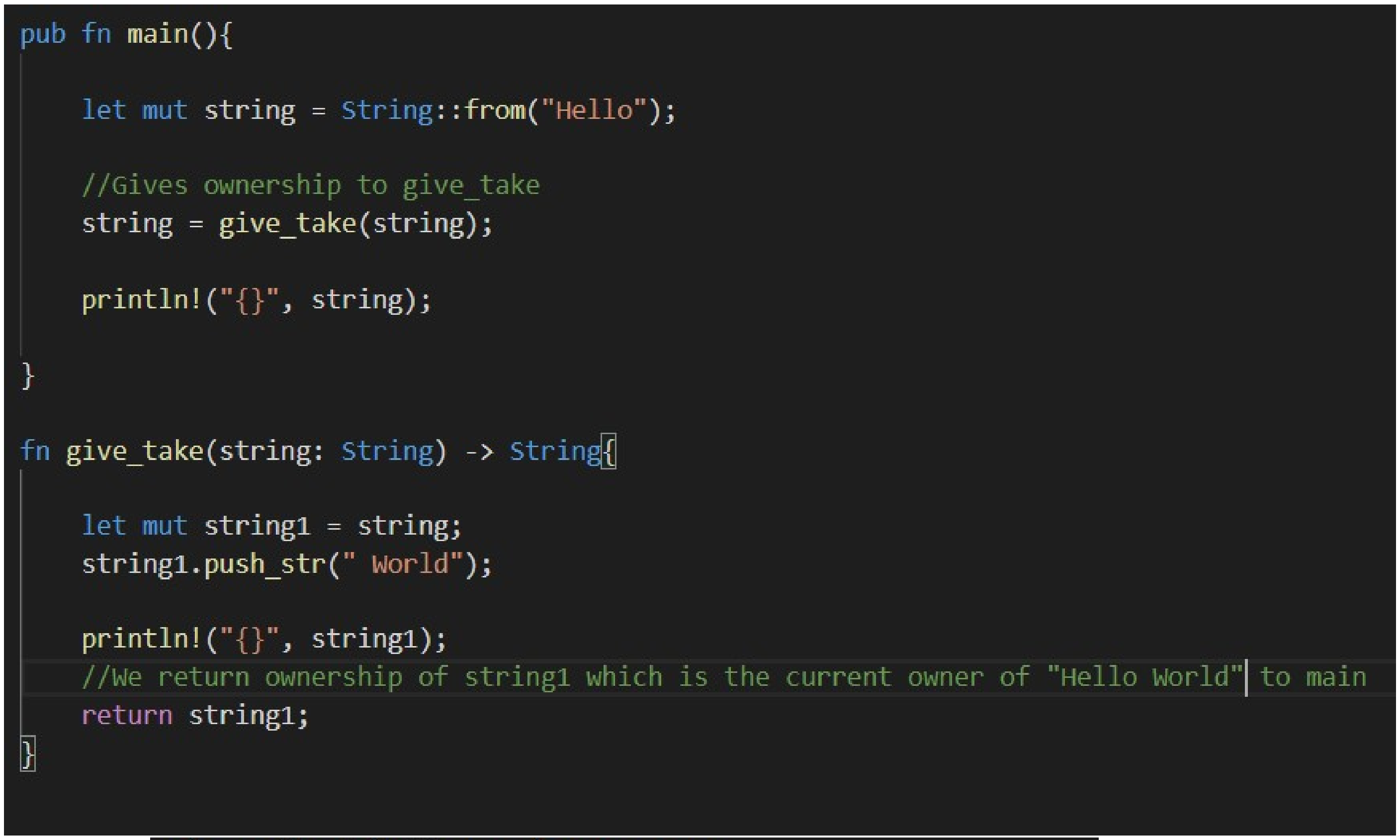
```
fn main(){

    let string = String::from("Hello World!");
    let string1 = string;

    println!("{}", string);


}
```



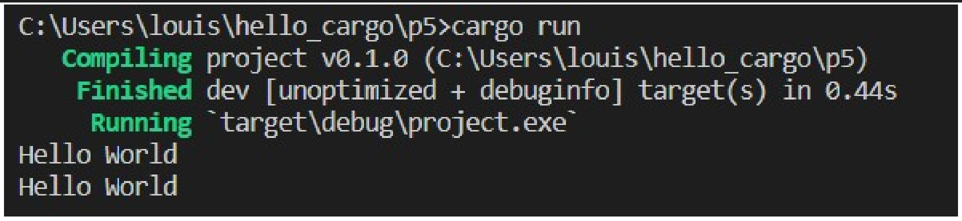
```
error[E0382]: borrow of moved value: `string`
 --> main.rs:9:20
  |
6 |     let string = String::from("Hello World!");
  |         ------ move occurs because `string` has type `std::string::String`, which does not impl
7 |     let string1 = string;
  |                   ------ value moved here
8 |
9 |     println!("{}", string);
  |                    ^^^^^^ value borrowed here after move
```

6.3 Borrowing and Returning

Passing values to functions operate the same way as assigning a value to a variable in terms of ownership. When passing a variable to another function, the original variable loses ownership and goes out of scope. Ownership is now transferred to the parameter in the new function. You can however return the ownership by returning that value in the new function and reassigning the original variable to the new function call. Below we have the variable string assign to String type "Hello". We then give the ownership of string from main to string1 in give_take function. We print string1, which shows that string1 at that point has ownership over that memory. We then return string1 and in main we reassign variable string to give_take function call. This gives variable string ownership over that memory again and is shown by the print in main.



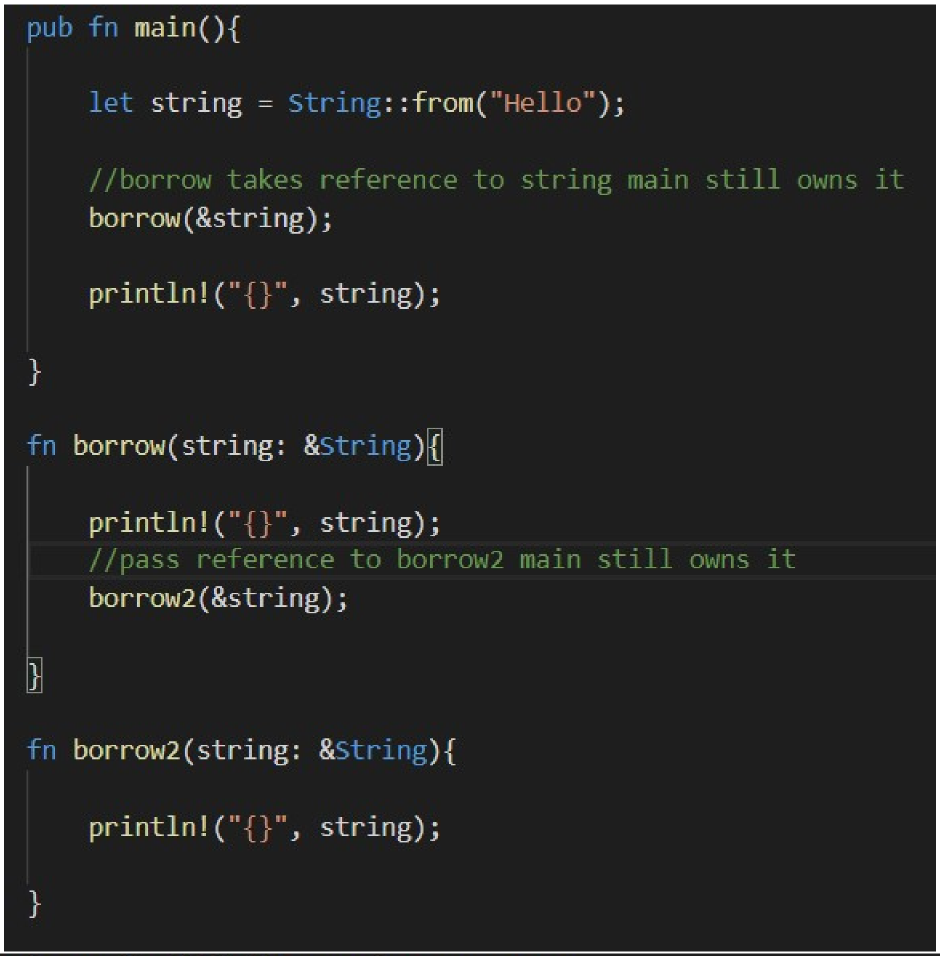
```
pub fn main(){

    let mut string = String::from("Hello");

    //Gives ownership to give_take
    string = give_take(string);

    println!("{}", string);

}

fn give_take(string: String) -> String{

    let mut string1 = string;
    string1.push_str(" World");

    println!("{}", string1);
    //We return ownership of string1 which is the current owner of "Hello World" to main
    return string1;
}
```



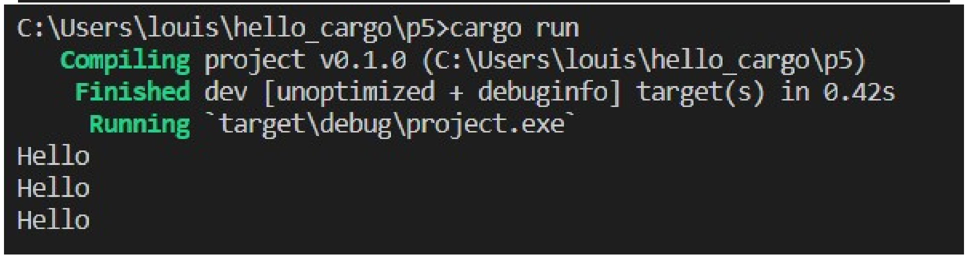
```
C:\Users\louis\hello_cargo\p5>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p5)
    Finished dev [unoptimized + debuginfo] target(s) in 0.44s
     Running `target\debug\project.exe`
Hello World
Hello World
```

6.4 References and Borrowing

The concept of passing ownership and returning ownership is a bit tedious however. Rust addresses this issue with references. This is a different form of copying by having the pointer of the new variable point to the pointer of the original variable instead of having both pointers point to the same memory. We use the keyword "" to indicate a reference. Below we have a variable string in main assign a String type "Hello". We then call borrow function and pass in a reference of variable string. In borrow we print string and call borrow2 and pass in a reference of string. In borrow2 we print the string value again. borrow2 and borrow returns and we get back into main function. Notice how we do not return the string in borrow and borrow2. Ownership throughout this whole process is still with variable string in main. This is the power of reference and allows us to avoid the tedious task of having to borrow and return ownership.

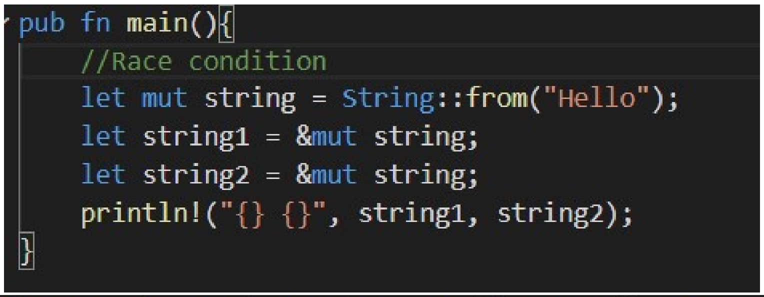


```rust
pub fn main(){

    let string = String::from("Hello");

    //borrow takes reference to string main still owns it
    borrow(&string);

    println!("{}", string);

}

fn borrow(string: &String){

    println!("{}", string);
    //pass reference to borrow2 main still owns it
    borrow2(&string);

}

fn borrow2(string: &String){

    println!("{}", string);

}
```

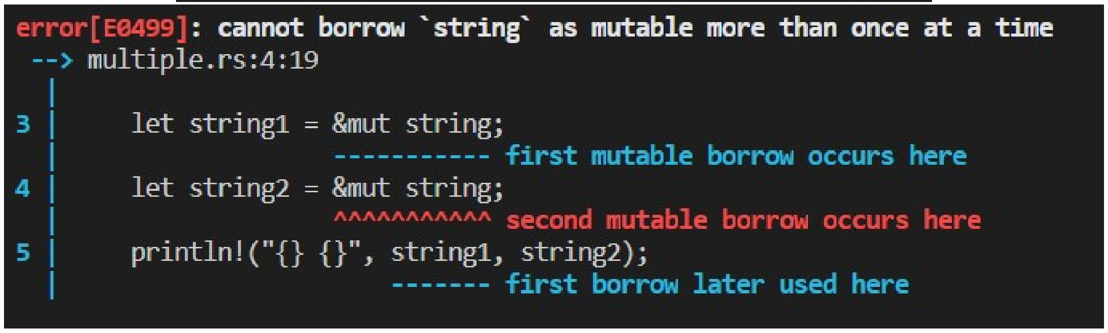


```
C:\Users\louis\hello_cargo\p5>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p5)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target\debug\project.exe`
Hello
Hello
Hello
```

It is important to note however that we are only allowed to have one mutable reference to a particular memory in the same scope. The example shows string1 and string2 both trying to have reference to variable string. This will generate an error. This restriction is to prevent a data race issue.

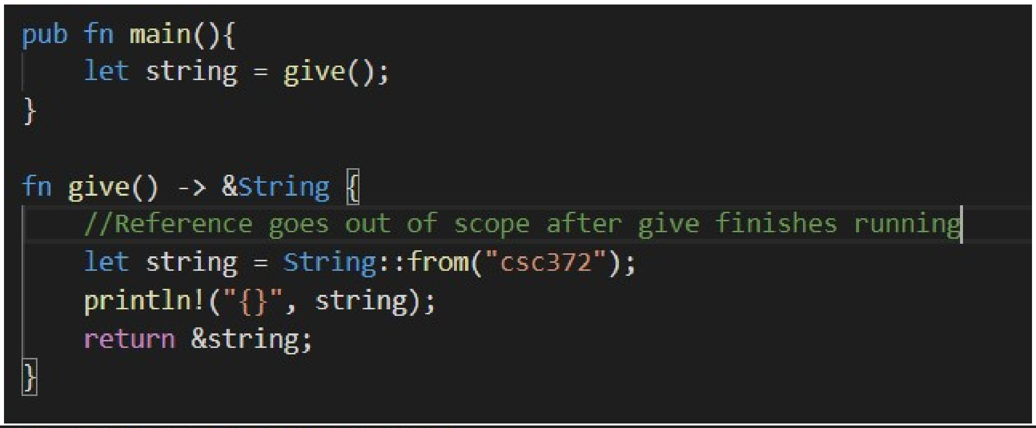


```
pub fn main(){
    //Race condition
    let mut string = String::from("Hello");
    let string1 = &mut string;
    let string2 = &mut string;
    println!("{} {}", string1, string2);
}
```

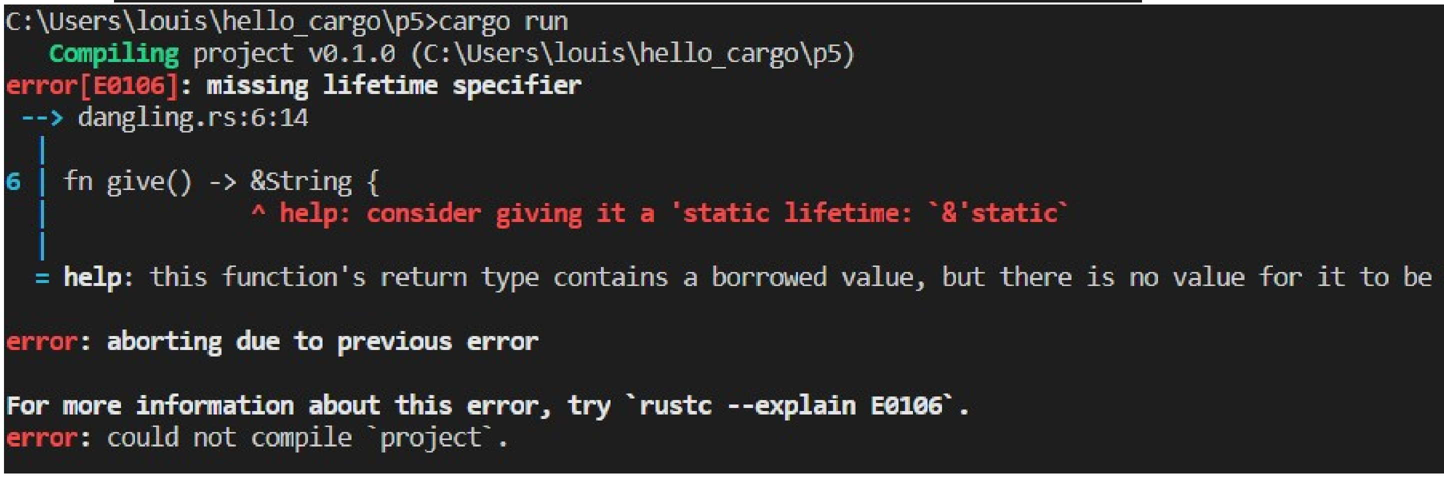


```
error[E0499]: cannot borrow `string` as mutable more than once at a time
 --> multiple.rs:4:19
  |
3 |     let string1 = &mut string;
  |                   ----------- first mutable borrow occurs here
4 |     let string2 = &mut string;
  |                   ^^^^^^^^^^^ second mutable borrow occurs here
5 |     println!("{} {}", string1, string2);
  |                       ------- first borrow later used here
```

Any languages that have pointers, have the potential problem of dangling pointers. If we were to assign variable string to function give(), and in that function we create a string variable and return the reference to it, it'll generate a dangling reference error because the original data goes out of scope which makes the reference invalid. We can resolve this issue by just returning the data directly however.



```
pub fn main(){
    let string = give();
}

fn give() -> &String {
    //Reference goes out of scope after give finishes running
    let string = String::from("csc372");
    println!("{}", string);
    return &string;
}
```



```
C:\Users\louis\hello_cargo\p5>cargo run
   Compiling project v0.1.0 (C:\Users\louis\hello_cargo\p5)
error[E0106]: missing lifetime specifier
 --> dangling.rs:6:14
  |
6 | fn give() -> &String {
  |              ^ help: consider giving it a 'static lifetime: `&'static`
  |
  = help: this function's return type contains a borrowed value, but there is no value for it to be

error: aborting due to previous error

For more information about this error, try `rustc --explain E0106`.
error: could not compile `project`.
```

The Rules of Reference for Rust
1. At any given time, you can have either 1 mutable reference or any number of immutable references.
2. References must ALWAYS be valid.

7 Summary

Rust has provided a viable alternative for a low level systems programming language through its ownership feature. Through ownership, Rust has solved many issues that low level languages come across such as concurrency, manual memory allocation/free, and data race conditions. This allows users to have the speed of low level languages without the drawback of limited safety. The popularity of Rust attests to how good of a programming language it has become with the 4 consecutive "most loved programming language" award by Stack Overflow developers. Despite ownerships initial learning curve it has proven itself to be a creative and elegant solution to problems that would leave many low level language programmers scratching their heads for hours. While its strictness can become tedious it keeps programmers from making errors that could take hours to debug.

References

[1] Rust community. Rust(programming language). Accessed: 4/6/2020.

[2] Adam Zachary Wasserman. Rust: Built to last. Accessed: 4/6/2020.

[3] with contribution Rust Community Steven Klabnik, Carol Nichols. The rust programming language. Accessed: 4/6/2020.

[4] Thomas Countz. Ownership in rust. Accessed: 4/28/2020.

[5] Jason Orendorff and Jim Blandy. *Programming Rust: Safe Systems Development*. O'Reilly Media.

[1] [2] [3] [4] [5]