ENTERPRISE AND IT COMPUTING (ENITS)
OFFENBURG UNIVERSITY OF APPLIED SCIENCES

SOFTWARE SECURITY

WINTER SEMESTER 2022/2023

RUST SAFETY REPORT

LECTURER : SCHAAD, ANDREAS

PROF. DR. PIL. M.SC

AUTHOR :

ARINN DANISH BIN ABDULLAH (191050-01)

TABLE OF CONTENT

DEFAULT BEHAVIOUR OF VARIABLES IN RUST EXAMPLE

Variables are immutable by default. This is just one of many suggestions Rust makes to help us build our code in a way that makes use of the safety and simple concurrency it provides. However, we may still make our variables changeable or mutable. When a variable is immutable, we cannot modify the value once it has been bound to a name.

Here is an example on how to examine or check the immutability error:

```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

The above code should display an error message:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` due to previous error
```

This example demonstrates how the compiler can be used to discover programming errors. The compiler errors indicate that our software isn't yet capable of performing the task we've given it. According to the error message, the reason for the error is that we cannot assign to the immutable variable "x" more than once because you attempted to do so.

For the reason of trying to alter an immutable value can result in defects, it's crucial that we obtain compile-time errors when we try to do so. It's possible that the first section of our code won't function as intended if it runs under the assumption that a value won't ever change and another component of our code modifies that value. When the second piece of code only seldom modifies the value, it can be challenging to identify the root cause of this type of error. We don't need to keep track of values ourselves since the Rust compiler ensures that when we say the values won't change, they actually will not change. Our code is thus easier to reason through.

However, mutability has its beneficial uses and can make writing code more convenient. Variables are immutable by default, but we may change that by adding the keyword “mut” before the variable name. By expressing that other portions of the code will be changing the value of this variable, the addition of the keyword “mut” also communicates purpose to potential code readers.

Here is the code example explaining the above scenario:

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

And the result will end up like this:

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

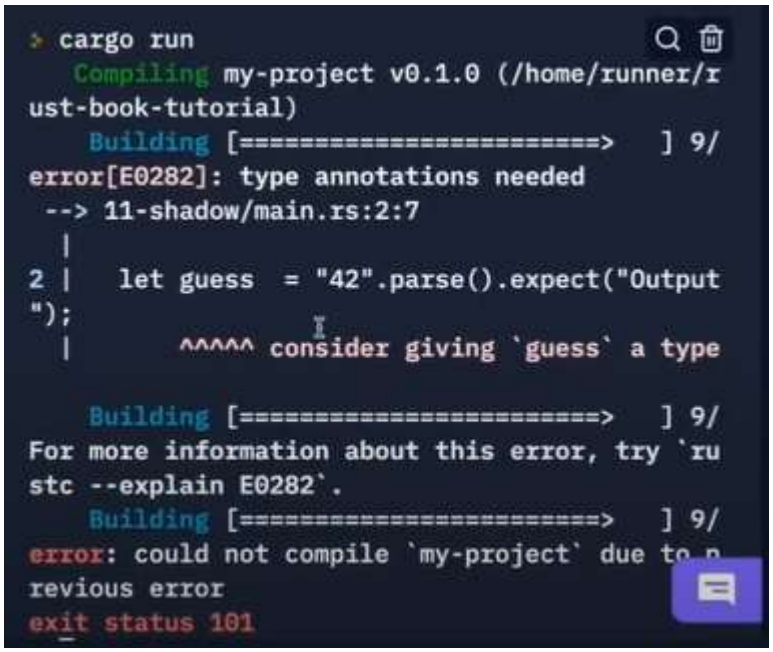
When the keyword “mut” is used, we are permitted to alter the value bound to x from 5 to 6. It is ultimately up to us to choose whether or not to use mutability and will depend on what we feel is the clearest in that specific circumstance.

STATIC TYPING IN RUST EXAMPLE

Rust is currently a statically typed language, which implies that during compilation, all of the variables' types must be known. What transpires then when a variable can have more than one value at a given time during runtime? Here includes the example and explanation to that:

```
1 ▼ fn main() {
2       let guess  = "42".parse().expect("Output");
```

This straightforward example shows the string 42 is being parsed by us. Herein lies the ambiguity. Both a text and an integer can be used to pass the number 42. Now, either of the values may be used. This might only result in two errors or the program crashes if we don't make this explicit in our code. The type to which 42 must be transferred is not stated directly in the statement.



As we can see, this ambiguity causes the application to crash. Now that it is stated explicitly to consider giving the case a type, let's do so and check if the program will run successfully.

```rust
fn main() {
    let guess: u32 =
"42".parse().expect("Output");
```

```
    Building [=========================>   ] 9/
warning: unused variable: `guess`
 --> 11-shadow/main.rs:2:7
  |
2 |   let guess: String  = "42".parse().expect
("Output");
  |       ^^^^^ help: if this is intentional,
prefix it with an underscore: `_guess`
  |
  = note: `#[warn(unused_variables)]` on by de
fault

    Building [=========================>   ] 9/
warning: `my-project` (bin "episode-11-shadow"
) generated 1 warning
    Finished dev [unoptimized + debuginfo] tar
get(s) in 0.33s
     Running `target/debug/episode-11-shadow
```

Let's start by giving it a type of string and a 32-bit unsigned type. And it works well. So, we must restrict the options and eliminate any ambiguity by giving a variable a type when it has more than one possible value at runtime. And in doing so, we may successfully run the program and ensure that there are no ambiguities during runtime.

HOW TO HANDLE OVERFLOWS (INT) IN RUST (IN RELEASE MODE)

Consider an u8 type variable that can store values between 0 and 255. Integer overflow will happen if we attempt to modify the variable to a value outside of that range, like 256, and it can have one of two outcomes. Rust contains checks for integer overflow while we're compiling in debug mode, and if this behaviour happens, our program will panic at runtime. When a program terminates incorrectly, Rust refers to it as panicking.

Rust does not provide checks for integer overflows that result in panics when we compile in release mode with the “release” flag. Instead, Rust does two's complement wrapping if overflow happens. In other words, values greater than the type's maximum value "wrap around" to the type's minimum value. In the case of an u8, the values 256 and 257 are converted to 0 and 1, respectively. The variable will likely not have the value we expected it to have, but the program won't panic. It is an error to rely on the wrapping behaviour of integer overflow.

We can use these families of methods from the standard library for primitive numeric types to specifically tackle the potential for overflow:

- Wrap in all modes with the wrapping_* methods, such as wrapping add.
- If there is an overflow, the checked_* methods should return None.
- The overflowing_* methods should return the value and a Boolean indicating whether there was an overflow.
- Use saturating_* methods to saturate at the variable's minimum or maximum value.

OWNERSHIP CONCEPT IN RUST

In Rust, each value has a variable called the owner of the value. In Rust, each piece of data will have a corresponding owner. For instance, age is the owner of the number 30 in the syntax *let age = 30*. There can only be one owner of a data set at a time and the same memory address cannot be referenced by two variables. The variables will always point to various locations in memory.

There are three ways to transfer ownership of value:

1. Assigning value of one variable to another variable.

Memory safety is the main selling point of the Rust programming language. A stringent control over who can use what and when limits is necessary to ensure memory safety. As the example shown below:

```
fn main(){
   let v = vec![1,2,3];
   // vector v owns the object in heap

   //only a single variable owns the heap memory at any given time
   let v2 = v;
   // here two variables owns heap value,
   //two pointers to the same content is not allowed in rust

   //Rust is very smart in terms of memory access ,so it detects a race cond
   //as two variables point to same heap

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

The example above declares a vector named v. The principle behind ownership is that only one variable—either v or v2—can be bound to a given resource. Use of relocated value: 'v' error is thrown in the example above. This is as a result of v2 acquiring ownership of the resource. It denotes a transfer of ownership from v to v2 (v2=v), and the subsequent invalidation of v.

2. Passing value to a function.

   When we pass an object from the heap to a closure or function, the ownership of a value also changes. An example shown below:

```
fn main(){
   let v = vec![1,2,3];     // vector v owns the object in heap
   let v2 = v;              // moves ownership to v2
   display(v2);             // v2 is moved to display and v2 is invalidated
   println!("In main {:?}",v2);    //v2 is No longer usable here
}
fn display(v:Vec<i32>){
   println!("inside display {:?}",v);
}
```

3. Returning value from a function.

   As soon as the function has finished running, any ownership provided to it will be invalidated. Allowing the function to return the owned object to the caller is one way to get around this. An example is as shown below:

```
fn main(){
   let v = vec![1,2,3];       // vector v owns the object in heap
   let v2 = v;                // moves ownership to v2
   let v2_return = display(v2);
   println!("In main {:?}",v2_return);
}
fn display(v:Vec<i32>)->Vec<i32> {
   // returning same vector
   println!("inside display {:?}",v);
}
```

In the other hands, primitive types copy the contents of one variable to another. Therefore, there is no change in ownership. Primitive variables require fewer resources than objects, which explains why. An example is as shown below:

```
fn main(){
   let u1 = 10;
   let u2 = u1;  // u1 value copied(not moved) to u2

   println!("u1 = {}",u1);
}
```

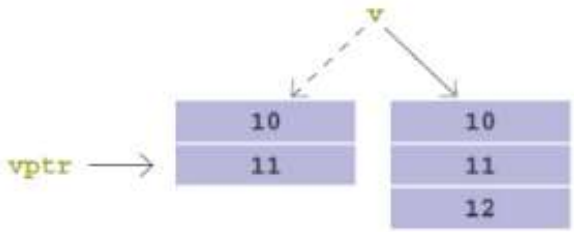
The output will be -10.

TYPICAL "USE AFTER FREE" BUG IN C++ & SOLUTION IN RUST

Let's look at some C++ code to illustrate the type of memory safety issues that frequently occur in systems programming languages. The example is as shown below:

```
1  std::vector<int> v { 10, 11 };
2  int *vptr = &v[1]; // Points *into* v.
3  v.push_back(12);
4  std::cout << *vptr; // Bug (use-after-free)
```

C++




```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // Points *into* 'v'.
3  v.push(12);
4  println!("{}", *vptr); // Compiler error
```

Rust

This program starts by producing an integer std:: vector, which is a growable array. The first two elements of v, 10 and 11, are kept in a buffer in memory. We create a pointer vptr in the second line that refers into this buffer, particularly to the location where the second element—which is now stored with the value 11—is located. Now that v and vptr both point to the same buffer's (overlapping regions), we say that the two pointers are aliasing. We add a new element to the end of v in the third line. In the buffer backing v, element 12 is added after element 11.

A new buffer is established and all the existing elements are relocated there when there is no room left for another element. What makes this instance intriguing is as vptr still refers to the previous buffer. In other words, vptr has become a dangling pointer as a result of the addition of a new element to v. Due to the fact that both pointers were aliasing, it is conceivable since any change made to pointer (v) will often also affect all of its aliases (vptr).

In the fourth line, the fact that vptr is now a hanging pointer causes an issue. Since there is a dangling pointer and we are loading from vptr, this is a use-after-free bug.

Iterator invalidation, which describes the circumstance where an iterator (often internally implemented with a reference) gets invalidated because the data structure it iterates over is changed during the iteration, is one instance of the issue, which is so widespread that it has its own name. Most frequently, it happens when one iterates over a container data structure and inadvertently calls an operation that modifies the data structure.

The call to the operation that modifies the data structure (push back in line 3 of the example above) may, in practice, be deeply nested behind several levels of abstraction. It is frequently very difficult to tell whether writing to a certain vector will invalidate references elsewhere in the program that are going to be used again later, especially when code is refactored or new features are added.

The pointer invalidation fix in Rust. The compiler in Rust detects problems like iterator invalidation and usage of null pointers statically, resulting in compile-time errors rather than run-time exceptions. Consider the Rust translation of the C++ at the bottom of example shown to see how this works. Similar to the C++ version, vptr points into the middle of the memory buffer, producing aliasing; push may reallocate the buffer, making vptr a dangling pointer; this results in a use-after-free in line 4.

All of this, however, never occurs; rather, the compiler displays the error message, "cannot borrow v as mutable more than once at a time." The key concept—the method by which Rust achieves memory safety in the presence of pointers that point into data structures—becomes apparent here; the type system upholds the rule that a reference is never both aliased and mutable at the same. This idea should be well-known in the context of concurrency, and Rust uses it to guarantee that there are no data races as well.

However, even for sequential programs, the unrestricted combination of aliasing and mutation is a prescription for catastrophe, as the example that is rejected by the Rust compiler demonstrates: Line 3 has an alias between vptr and v (v is assumed to point to all of its contents, which overlaps with vptr), and line 4 has a mutation that could result in a memory access problem.

RUST VS C++: SAFETY/SECURITY-RELATED EXAMPLE

There are many parallels between these two programming languages. However, there is one element that certainly separates them: safety. Compared to C++, Rust is a safer solution because potential errors result in code rejection. By default, Rust is secure. It strikes a balance between forbidding bad behaviour and permitting it when the loss won't be too large. In C++, a mistake could escape the review process and lead to program problems or even security flaws.

“In C, you publish your API if it's possible to use it correctly (open world). In Rust, you publish a safe API if it's impossible to use incorrectly (closed world).” This quotation amply illustrates how these two programming languages differ from one another. When it comes to safety, Rust greatly outperforms C++.

In the other hand, because it has greater safety standards and lower development costs than C++, Rust enables reaching higher levels of performance. For instance, C++ lacks automatic garbage collection tools to ensure speedier operation, which could result in a number of runtime issues. Rust is safer than C++ due to a number of fundamental features, including the fact that compilation problems rather than run-time errors can be caused by code faults. Overall. Rust enables you to write safer code at a cheaper cost of development.

In my opinion, the philosophy of C++ (and C before it) is permissive. They'll give the programmer complete freedom. Additionally, they offer a few practical primitives that help careful programmers create reliable, portable programs. The design and ideas behind Rust are also different. By default, it won't permit the programmer to do any action that appears risky. Because the language doesn't assume responsibility for catching those kinds of issues, there have been numerous security issues where C or C++ code was exposed to buffer overflows or other security risks.

However, because Rust has tougher constraints, there will be instances where the programmer can't proceed naturally, even if it is safe, and must instead compromise to find a solution that will persuade Rust that it can truly be shown safe by the application of simple rules. Rust won't let us do anything unless we do it its way. That will be the ideal compromise for some applications, but it might be a pain in the rear for others.

Additionally, instead of we learning not to make mistakes in the first place, Rust aims to prevent them. Therefore, it complains about errors at compile time, and since the compiler catches us in a safety net, we never have to learn to not make them.

REFERENCES

- https://doc.rust-lang.org/book
- https://cacm.acm.org/magazines/2021/4/251364-safe-systems-programming-in-rust/fulltext?mobile=false
- https://www.tutorialspoint.com/rust/rust_ownership.htm
- https://codilime.com/blog/rust-vs-c-safety-and-performance-in-low-level-network-programming/
- https://www.quora.com/Why-is-Rust-considered-to-be-more-secure-than-C
- https://youtu.be/YUKj_HXM6kk