Student Name: _____ Student ID: _____ TA: _____

Rust Lab 08 – Rust Memory Management 21/8/25

Lab 1: Stack Allocation

Learn how Rust puts memory on the stack by writing simple Rust code to do these things:

- Create a function that figures out the factorial of a number using recursion (where a function calls itself).
- During each recursive step, show the current number and where it's stored in memory.
- Add an optional extra task: Try the function with a big number (like factorial(10000)) to see if it causes a stack overflow (be careful, it might crash!).
- Explain why this happens based on how stack memory works.

_____

_____

Example result:

```
Calculating factorial(5)
Value: 5, Memory Address: 0x16fbeaf58
Calculating factorial(4)
Value: 4, Memory Address: 0x16fbeade8
Calculating factorial(3)
Value: 3, Memory Address: 0x16fbeac78
Calculating factorial(2)
Value: 2, Memory Address: 0x16fbeab08
Calculating factorial(1)
Value: 1, Memory Address: 0x16fbea998
Calculating factorial(0)
Value: 0, Memory Address: 0x16fbea828
Factorial result: 120
```

Tips: - You can use std::mem::size_of_val to check the size of stack memory if you want.

- Memory addresses usually get smaller as you go deeper into recursion, showing how the stack grows

TA Checking: _____ Time: _____

Lab 2: Static Allocation

Learn about static allocation in Rust, how it works, and its limitations by doing the following:

- Create a few static variables (like u32, i32, f64) using uppercase names with underscores (e.g., MY_NUMBER).
- In the main function, print the values and memory addresses of these variables.
- Write another function that also prints the addresses to show they stay the same (unlike stack memory).
- Try changing a static variable (comment out this code and note why it fails for safety reasons).
- Explain the downsides: Why doesn't static allocation work for recursion? Why are static variables usually unchangeable?

_____

_____

What to Expect:

```
Static A: 3, Address: 0x55555575a000
Static B: -1000000, Address: 0x55555575a004
From another function:
Static A: 3, Address: 0x55555575a000
Static B: -1000000, Address: 0x55555575a004
```

Tips: - Use the static keyword and specify the variable type (e.g., static A: u32 = 3;).

- Memory addresses should be identical in both functions, showing static memory is fixed.

TA Checking: _____ Time: _____

Student Name: _____ Student ID: _____ TA: _____

Lab 3: Basic Heap Allocation with Box

Get to know how heap allocation works using Box in Rust and compare it to references in a simple way:

- Make a basic struct, like Point with x and y as numbers (e.g., f64).
- Create one Point on the stack and another on the heap using Box<Point>.
- Use std::mem::size_of_val to check the size of each on the stack (the Box should be small, like a pointer).
- Use * to access and print the values inside the Box.
- Try double indirection: Make a Box<Box<Point>> and figure out how to get its values.
- Compare references and Box: Write two functions—one that borrows a Point with &Point and one that takes ownership with Box<Point>. Show how Box moves ownership while & just borrows it.

_____

_____

Expected output:

```
Point on stack size: 16 bytes
Boxed point on stack size: 8 bytes (pointer)
Dereferenced boxed point: (0.0, 0.0)
Double boxed point size: 8 bytes
```

Hints: - Box<T> puts data on the heap when the size isn't known ahead of time or for things that change.

- Use * to get the values out of a Box, just like with references.

- Box acts a bit like & because it has Deref, but it owns the data instead of just borrowing it.

TA Checking: _____ Time: _____

Lab 4: Building a Stack with a Linked List Using Box

Create a stack that works like a last-in, first-out (LIFO) system using a linked list, where each part is stored on the heap with Box. This shows how the heap can handle flexible, recursive types that would be too big for the stack.

1. Setting Up the Structure:
   - Make an enum called Node with two options: Cons(i32, Box<Node>) for a node with a number and a link to the next node, and Nil for the end of the list.
   - Create a struct called BoxedStack with a field top: Box<Node> to hold the top of the stack.
2. Adding Functions:
   - new() -> Self: Make a new, empty stack with top set to Box::new(Nil).
   - push(&mut self, value: i32): Add a new number to the top by making a new Cons node.
   - pop(&mut self) -> Option<i32>: Take off and return the top number, then update the top.
   - peek(&self) -> Option<&i32>: Look at the top number without removing it.
   - is_empty(&self) -> bool: Check if the stack is empty by seeing if top is Nil.
   - print_stack(&self): Print the stack from top to bottom using recursion.
3. Adding a Trait:
   - Add the Debug trait to BoxedStack so you can easily print its contents.
4. Extra Task:
   - Compare this to a stack stored on the stack (if you can) and explain why Box is necessary for recursive types like this.

How to Use It: Follow a similar pattern to the original Lab 2, but show the linked list output, like Stack contents: 30 -> 20 -> 10 -> Nil.

What to Expect:

```
Pushing 10 onto the stack.
Pushing 20 onto the stack.
Pushing 30 onto the stack.
Stack contents: 30 -> 20 -> 10 -> Nil
Top of the stack: 30
Popped 30 from the stack.
Stack contents: 20 -> 10 -> Nil
Popped 20 from the stack.
Stack contents: 10 -> Nil
Popped 10 from the stack.
Stack contents: Nil
Is the stack empty? True
```

Student Name: _____ Student ID: _____ TA: _____

Tips: - Use an enum for Node with Box to allow recursion without making the size too big.
- For print_stack, create a recursive helper function to go through the list.
- The heap lets the stack grow as needed, unlike the stack's fixed limit.
- The original stack used a single heap block with Vec; this linked version practices using pointer

TA Checking: _____ Time: _____