Student Name: ______________________________ ID: ________________ TA: ______________

Rust Lab 09 — Data Implementation & Memory (Revised)

Date: 3/9/2025

Introduction

In this lab you will explore how Rust manages memory at a low level, including object sizes, alignment, enums, vectors, byte representation, and UTF-8 strings. Each lab exercise includes detailed tasks and spaces for observations. Answer all questions and be ready to explain your results to the TA.

Lab A — Sizes, Alignment & #[repr]

In this exercise, you will learn how struct field ordering and alignment affect memory usage. Rust may insert padding bytes to ensure proper alignment, and different representations (`repr`) change layout guarantees.

Learning objectives:

- Use std::mem::size_of, size_of_val, and align_of.
- Observe padding and field-order effects.
- Compare #[repr(Rust)] vs #[repr(C)].

Tasks:

1) Define the following:

```
struct S1 { a: u8, b: u64, c: u8 }
#[repr(C)] struct S2 { a: u8, b: u64, c: u8 }
struct S3 { b: u64, c: u8, a: u8 }
```

2) Print size_of and align_of for each.
3) Explain why S3 uses less memory.
4) Use size_of_val(&instance) to confirm type size.

Expected outcome: Students should notice padding in S1, no field reordering in S2, and tighter packing in S3 due to ordering.

Observation Table:

S1 size=______ align=______ S2 size=______ align=______ S3 size=______ align=______

Explanation: ____

TA Check: ____

Lab B — Enums & Niche Optimization

Enums in Rust can sometimes reuse unused bit patterns, called niches, to store variant tags. This saves memory and makes Option<T> very efficient.

Learning objectives:

- Observe memory layout of Option<T>.
- Understand how references and NonZero types benefit from niche optimization.

Tasks:

1) Print size_of::<&u8>(), size_of::<Option<&u8>>(), size_of::<usize>(), size_of::<Option<usize>>(), size_of::<NonZeroUsize>(), size_of::<Option<NonZeroUsize>>().
2) Compare results and explain why Option<&u8> is the same size as &u8.

Expected outcome: Students should see that Option<&u8> has the same size as &u8 because null is reserved as None. For usize, Option<usize> might be larger.

Observation Table:

&u8=______ Option<&u8>=______ usize=______ Option<usize>=______

NonZeroUsize=______ Option<NonZeroUsize>=______

Explanation: ____

Student Name: ______________________________________ ID: _____________________ TA: __________________

TA Check: ____

Lab C — Vectors: Header vs Buffer, Growth & Preallocation

Vectors in Rust consist of a header (pointer, length, capacity) stored on the stack, and a buffer stored on the heap. When the buffer is full, Rust reallocates a larger block, usually doubling capacity.

Learning objectives:

- Distinguish length vs capacity.
- Observe buffer growth and pointer changes.
- Use preallocation and shrink_to_fit.

Tasks:

1) Create Vec::new(), push 1..=40, and print len, cap, and pointer whenever capacity changes.
2) Repeat with Vec::with_capacity(32) and push 1..=40.
3) Call shrink_to_fit and note capacity changes.
4) Convert vec![1,2,3,4,5] into a boxed slice and discuss use cases.

Expected outcome: Students should observe doubling behavior (4, 8, 16, 32...). Preallocation avoids multiple reallocations. shrink_to_fit may or may not reduce capacity.

Observation Table:

Cap changes: len=____, cap=____, ptr=0x_______ Preallocated first cap=______ growth=______
Shrink before=______ after=______
Boxed slice: when preferable? ____

TA Check: ____

Lab D — Bytes & Endianness: Safe vs Unsafe

Endianness describes byte order in memory. Rust provides safe functions to view integers as byte arrays. Unsafe methods like *from_raw_parts* let us view structs as bytes directly, but this can be risky.

Learning objectives:

- Understand endianness.
- Practice safe conversions with to_*_bytes.
- Explore unsafe zero-copy memory views.
- Discuss risks of unsafe methods.

Tasks:

1) Print to_ne_bytes, to_be_bytes, to_le_bytes for 0x11223344.
2) Define struct Data { i: u32, c: char, b: bool } and view bytes:

```
#[repr(C)]
struct Data { i: u32, c: char, b: bool }
```

(a) Safe: convert fields individually. (b) Unsafe: use from_raw_parts on the struct.

3) Compare results and explain risks of unsafe method.

Expected outcome: Students see that output depends on machine endianness. The #[repr(C)] annotation ensures consistent field ordering across compilations. Unsafe gives raw bytes including padding. They should mention risks like undefined behavior, layout instability, aliasing.

Observation Table:

ne=______ be=______ le=______ Raw bytes (unsafe)=____
Safety discussion: ____

TA Check: ____

Student Name: ____ ID: ______________________ TA: ___________________

Lab E — Strings & UTF-8

Strings in Rust are UTF-8 encoded. Indexing by bytes can cut characters, causing runtime panics. Rust enforces safety by preventing direct indexing. Use chars() or graphemes for correct iteration.

Learning objectives:

- Work with bytes, chars, and graphemes.
- Avoid invalid slicing.
- Understand graphemes for user-visible characters.

Tasks:

1) For s="Rust 🚀 café á": print number of bytes, chars, and graphemes.
2) Show valid slice, and attempt invalid slice causing panic. **Example byte indices for s="Rust 🦀 café á":**
   - Valid: &s[0..4] (gets "Rust")
   - Invalid: &s[5..6] (cuts through 🦀 emoji at byte 5, should panic)
   - Invalid: &s[10..11] (cuts through é at byte 10, should panic)
3) Extract first grapheme with graphemes(true).

Expected outcome: Students should count differences: more bytes than chars, and graphemes may differ when accents/emoji are used. They should see why slicing incorrectly can panic - the emoji 🦀 takes 4 bytes (5-8), and é takes 2 bytes (10-11 for the combined character).

Observation Table:

bytes=____ chars=____ graphemes=____ Valid slice=____ Invalid slice=____ Grapheme example=____
Explanation: ____

TA Check: ____

Extension — Tiny Box<T> Stack

Box<T> lets us allocate on the heap. Recursive data structures like linked lists must use Box to avoid infinite size at compile time. This exercise builds a simple stack using Box.

Task: Implement BoxedStack with Node enum (Cons, Nil). Functions: new, push, pop, peek, is_empty, and Debug. Compare performance with Vec-based stack.

Expected outcome: Students practice Box usage and recursive enums. They should note that Vec is more efficient, but Boxed structures are useful for linked, recursive patterns.

TA Check: ____