Rust Lab 09 — TA Guide

Lab A — Sizes, Alignment & #[repr]

Completed Code:

```
use std::mem::{size_of, align_of, size_of_val};

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 }

fn main() {
    println!("S1 size={} align={}", size_of::<S1>(), align_of::<S1>());
    println!("S2 size={} align={}", size_of::<S2>(), align_of::<S2>());
    println!("S3 size={} align={}", size_of::<S3>(), align_of::<S3>());
    let s1 = S1 { a:1, b:2, c:3 };
    println!("size_of_val(S1)={}", size_of_val(&s1));
}
```

Expected Results & TA Notes

- On 64-bit: S1 size=24 align=8; S2 size=24 align=8; S3 size=16 align=8.
- Correct padding explanation: S1 has 14 bytes of padding total (7 after `a`, 7 after `c`).
- #[repr(C)] on S2 prevents field reordering; S3 shows tighter packing by ordering largest-first.
- size_of_val(&s1) equals size_of::<S1>() for fixed-size types.

Lab B — Enums & Niche Optimization

Completed Code:

```
use std::mem::size_of;
use core::num::NonZeroUsize;

fn main() {
    println!("&u8                   : {}", size_of::<&u8>());
    println!("Option<&u8>           : {}", size_of::<Option<&u8>>());
    println!("usize                 : {}", size_of::<usize>());
    println!("Option<usize>         : {}", size_of::<Option<usize>>());
    println!("NonZeroUsize          : {}", size_of::<NonZeroUsize>());
    println!("Option<NonZeroUsize>: {}", size_of::<Option<NonZeroUsize>>());
}
```

Expected Results & TA Notes

- Expect: &u8=8, Option<&u8>=8; usize=8, Option<usize>=16; NonZeroUsize=8, Option<NonZeroUsize>=8.
- Why: null pointer is a niche for Option<&u8>; zero is a niche for Option<NonZeroUsize>.
- Contrast: Option<u8> is larger than u8 (no niche), unlike references/nonzero types.

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

Completed Code (prints when capacity actually changes):

```
fn main() {
    println!("--- Lab C1: Vec::new() push 1..=40 ---");
    let mut v: Vec<i32> = Vec::new();
    let mut prev_cap = v.capacity();
    for i in 1..=40 {
        v.push(i);
        if v.capacity() != prev_cap {
            println!("len={} cap={} ptr={:p}", v.len(), v.capacity(), v.as_ptr());
            prev_cap = v.capacity();
        }
    }

    println!("\n--- Lab C2: Vec::with_capacity(32) push 1..=40 ---");
    let mut v2: Vec<i32> = Vec::with_capacity(32);
    let mut prev_cap2 = v2.capacity();
    for i in 1..=40 {
        v2.push(i);
        if v2.capacity() != prev_cap2 {
            println!("len={} cap={} ptr={:p}", v2.len(), v2.capacity(), v2.as_ptr());
            prev_cap2 = v2.capacity();
        }
    }

    println!("\nBefore shrink_to_fit: len={}, cap={}", v2.len(), v2.capacity());
    v2.shrink_to_fit();
    println!("After shrink_to_fit:  len={}, cap={}", v2.len(), v2.capacity());

    println!("\n--- Lab C4: Boxed slice ---");
    let v3 = vec![1, 2, 3, 4, 5];
    let b: Box<[i32]> = v3.into_boxed_slice();
    println!("Boxed slice length = {}", b.len());
}
```

Expected Results & TA Notes

- Capacity growth typically doubles: 0→4→8→16→32→64 (addresses change due to ASLR).
- Vec::with_capacity(32) avoids reallocations until pushing the 33rd element (cap likely jumps to 64).
- shrink_to_fit is a hint: cap may remain 64 or shrink closer to len (both acceptable). Students should note non-determinism.
- Boxed slice yields an immutable, tight buffer (no over-allocation).

Lab D — Bytes & Endianness (Safe vs Unsafe)

Completed Code (uses #[repr(C)] to stabilize layout for the unsafe view):

```
use std::mem;

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

```
fn main() {
    println!("--- Lab D1: Endianness ---");
    let x: u32 = 0x11223344;
    println!("to_ne_bytes: {:02X?}", x.to_ne_bytes());
    println!("to_be_bytes: {:02X?}", x.to_be_bytes());
    println!("to_le_bytes: {:02X?}", x.to_le_bytes());

    println!("\n--- Lab D2: Struct bytes ---");
    let d = Data { i: 0x11223344, c: 'A', b: true };
    println!("size_of::<Data>() = {}", mem::size_of::<Data>());

    // (a) Safe per-field copy
    let mut safe_bytes = Vec::new();
    safe_bytes.extend_from_slice(&d.i.to_ne_bytes());          // 4
    safe_bytes.extend_from_slice(&(d.c as u32).to_ne_bytes()); // 4
    safe_bytes.push(d.b as u8);                                // 1
    println!("Safe bytes (per-field) = {:02X?}", safe_bytes);

    // (b) Unsafe zero-copy (layout-dependent)
    let raw: &[u8] = unsafe {
        std::slice::from_raw_parts((&d as *const Data) as *const u8,
mem::size_of::<Data>())
    };
    println!("Raw bytes (unsafe)     = {:02X?}", raw);
}
```

Expected Results & TA Notes

- Endianness: on little-endian, to_ne_bytes == to_le_bytes = [44,33,22,11], to_be_bytes = [11,22,33,44].
- size_of::<Data>() commonly 12 (padding after bool).
- Safe path contains only the explicit field bytes (no padding).
- Unsafe raw view includes padding; TA must ensure students explain risks: layout depends on repr, target, compiler.
- Add note: If #[repr(C)] is removed, raw layout may change between builds/platforms.

Lab E — Strings & UTF-8 (Bytes, Chars, Graphemes)

Completed Code (consistent example: "Rust 🚀 café á").

Add to Cargo.toml: unicode-segmentation = "1.10"

```
use unicode_segmentation::UnicodeSegmentation;

fn main() {
    let s = "Rust 🚀 café á";

    println!("--- Lab E1: Counts ---");
    println!("bytes = {}", s.as_bytes().len());
    println!("chars = {}", s.chars().count());
    println!("graphemes = {}", s.graphemes(true).count());

    println!("\n--- Lab E2: Slicing ---");
    let valid = &s[0..5]; // "Rust "
    println!("Valid slice [0..5] = {:?}", valid);
    // Invalid: this would panic because it cuts inside the 4-byte emoji:
    // let invalid = &s[0..7];
    // println!("Invalid slice [0..7] = {:?}", invalid);

    println!("\n--- Lab E3: Graphemes ---");
    for (i, g) in s.graphemes(true).enumerate() {
        println!("Grapheme {}: {}", i, g);
    }
}
```

Expected Results & TA Notes

- Counts for this exact string: bytes=18, chars=13, graphemes=13.
- Valid slice [0..5] = "Rust "; invalid [0..7] (if attempted) would panic: index not at char boundary.
- TA should confirm students explain when to use bytes (I/O, hashing), chars (scalar values), graphemes (UI).

Extension — Tiny Box<T> Stack

Reference Solution (runnable):

```
use std::fmt;

enum Node {
    Cons(i32, Box<Node>),
    Nil,
}

struct BoxedStack {
    top: Box<Node>,
}

impl BoxedStack {
    fn new() -> Self {
        Self { top: Box::new(Node::Nil) }
    }
    fn is_empty(&self) -> bool { matches!(*self.top, Node::Nil) }
    fn push(&mut self, v: i32) {
        let old = std::mem::replace(&mut self.top, Box::new(Node::Nil));
        self.top = Box::new(Node::Cons(v, old));
    }
    fn pop(&mut self) -> Option<i32> {
        match std::mem::replace(&mut self.top, Box::new(Node::Nil)) {
            box Node::Nil => { self.top = Box::new(Node::Nil); None }
            box Node::Cons(v, next) => { self.top = next; Some(v) }
        }
    }
    fn peek(&self) -> Option<&i32> {
        match self.top.as_ref() { Node::Cons(v, _) => Some(v), Node::Nil => None }
    }
}

impl fmt::Debug for BoxedStack {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        fn walk(n: &Node, out: &mut Vec<i32>) {
            match n { Node::Cons(v, next) => { out.push(*v); walk(next, out) }, Node::Nil
=> {} }
        }
        let mut vals = Vec::new(); walk(&self.top, &mut vals);
        write!(f, "Stack(top → {:?} → Nil)", vals)
    }
}

fn main() {
    let mut stack = BoxedStack::new();
    stack.push(1); stack.push(2); stack.push(3);
    println!("Stack: {:?}", stack);
    println!("Peek:  {:?}", stack.peek());
    println!("Pop:   {:?}", stack.pop());
    println!("Pop:   {:?}", stack.pop());
    println!("Empty? {}", stack.is_empty());
    println!("Pop:   {:?}", stack.pop());
    println!("Empty? {}", stack.is_empty());
}
```

Expected Behavior & TA Notes

- After pushes, top is 3; peek=Some(3). Pop sequence: 3, 2, 1, then None; ends empty.
- Assessment checklist: uses Box to enable recursion; ownership moves via std::mem::replace; O(1) push/pop/peek; discussion: Vec is typically preferred for stacks due to cache locality & fewer allocations.

Common Mistakes & Troubleshooting

- Lab C: printing only when len==cap misses realloc events; track capacity changes instead (fixed here).
- Lab C: expecting shrink_to_fit to always reduce capacity — it's a hint; both outcomes acceptable.
- Lab D: forgetting layout dependence; if #[repr(C)] is removed, raw bytes may change across builds/targets.
- Lab E: mixing strings in examples — use exactly "Rust 🚀 café á" to match counts and slices.
- Extension: trying recursive enum without Box — won't compile (infinite size).

To TA:

Please note in paper if you notice that student using Generative AI – by write down "AI" on top of the sheet.