22T3 Practise Q2

In this activity, you will be building a small text searching system. It should search a large string for sentences that contain a particular search term. Another function will then look through all the search results to determine how often each sentence was found.

```
// main.rs {
use std::fs;
use std::env;
use std::io::{self, BufRead};
use std::error::Error;
use std::collections::HashMap;

// NOTE: You *may not* change the names or types of the members of this struct.
//       You may only add lifetime-relevant syntax.
pub struct SearchResult<'a, 'b> {
    pub matches: Vec<&'a str>,
    pub contains: &'b str
}

/// Returns a [`SearchResult`] struct, where the matches vec is
/// a vector of every sentence that contains `contains`.
///
/// A sentence is defined as a slice of an `&str` which is the first
/// character of the string, or the first non-space character after
/// a full-stop (`.`), all the way until the last non-space character
/// before a full-stop or the end of the string.
///
/// For example, In the string "Hello. I am Tom . Goodbye", the three
/// sentences are "Hello", "I am Tom" and "Goodbye"
fn find_sentences_containing<'a, 'b>(text: &'a str, contains: &'b str) ->
SearchResult<'a, 'b> {
    let mut search_result = SearchResult {
        matches: Vec::new(),
        contains,
    };
    search_result.contains = contains;
    let sentences = text.split(".").collect::<Vec<&str>>();
    //let re = Regex::new(r"([^.]*\.[^A-Z]*|[^.]+\z)").unwrap();
    //let sentences = re.find_iter(text).map(|mat|
mat.as_str().trim()).collect::<Vec<&str>>();
    for sentence in sentences {
        if sentence.contains(contains) {
            search_result.matches.push(sentence.trim());
        }
    }

    return search_result
}

/// Given a vec of [`SearchResult`]s, return a hashmap, which lists how many
/// time each sentence occured in the search results.
```

```
fn count_sentence_matches<'a>(searches: Vec<SearchResult<'a, '_>>) -> HashMap<&'a
str, i32> {
    let mut hashmap_result: HashMap<&str, i32> = HashMap::new();
    for search_result in searches {
        for sentence in search_result.matches {
            if hashmap_result.contains_key(sentence) {
                let sentence_count = hashmap_result.get_mut(sentence).unwrap();
                *sentence_count += 1;
            } else {
                hashmap_result.insert(sentence, 1);
            }
        }
    }

    hashmap_result
}

/////////// DO NOT CHANGE BELOW HERE ///////////

fn main() -> Result<(), Box<dyn Error>> {
    let args: Vec<String> = env::args().collect();
    //let file_path = &args[1];
    let file_path =
r"C:\Users\kevin\Desktop\lab10\prac_q2\test_data\test_data.txt".to_string();
    let text = fs::read_to_string(file_path)?;

    let mut sentence_matches = {
        let mut found = vec![];

        let stdin = io::stdin();
        //let matches = stdin.lock().lines().map(|l|
l.unwrap()).collect::<Vec<_>>();
        let matches = vec!["there".to_string(), "very".to_string(),
"prove".to_string(), "the universe".to_string()];
        for line in matches.iter() {
            let search_result = find_sentences_containing(&text, line);
            println!("Found {} results for '{}'.", search_result.matches.len(),
search_result.contains);
            found.push(search_result);
        }

        count_sentence_matches(found).into_iter().collect::<Vec<_>>()
    };
    sentence_matches.sort();

    for (key, value) in sentence_matches {
        println!("'{}' occured {} times.", key, value);
    }

    Ok(())
}
```

// main.rs EOF }

22T3 Practise Q3

In this question, your task is to complete two functions, and make them generic: zip_tuple and unzip_tuple. Right now, the zip_tuple function takes a Vec<Coordinate> and returns a tuple: (Vec<i32>, Vec<i32>). The unzip_tuple function performs the inverse of this.
This code currently does not compile, because q3_lib (i.e. lib.rs) does not know what the type of Coordinate is. Rather than telling the functions what type Coordinate is, in this exercise we will make the functions generic, such that it works for both q3_a (i.e. main_1.rs) and q3_b (i.e. main_2.rs). This is to say, tuple_unzip should work for any Vec<T> such that T implements Into into a 2-tuple of any 2 types, and tuple_zip should work for any Vec<(T, U)> such that (T, U) implements Into into any type.
Once you have modified your function signatures for tuple_unzip and tuple_zip, you should find that the only concrete type appearing within the signature is Vec. In other words, the functions should work for any type which can be created from a 2-tuple and which can be converted into a 2-tuple.

```
// lib.rs {
pub fn tuple_unzip<T, U, V>(items: Vec<T>) -> (Vec<U>, Vec<V>)
where
    T: Into<(U, V)>,
{
    let mut vec1: Vec<U> = Vec::new();
    let mut vec2: Vec<V> = Vec::new();
    for item in items {
        let (u, v) = item.into();
        vec1.push(u);
        vec2.push(v);
    }
    (vec1, vec2)
}

pub fn tuple_zip<T, U, V>(items: (Vec<U>, Vec<V>)) -> Vec<T>
where
    (U, V): Into<T>,
{
    let mut vec_output = Vec::new();
    for (u, v) in items.0.into_iter().zip(items.1.into_iter()) {
        vec_output.push((u, v).into());
    }
    vec_output
}
// end lib.rs }
```

22T3 Practise Q4.2

Emily is designing a function that has different possibilities for the value it may return. She is currently deciding what kind of type she should use to represent this property of her function.
She has narrowed down three possible options:

- ***An enum***

- ***A trait object***
- ***A generic type (as fn foo(...) -> impl Trait)***

For each of her possible options, explain one possible advantage and one possible disadvantage of that particular choice.

- Enums offer a clear and explicit means to represent multiple return values, potentially of different types. However, enum variants are predetermined at compile time, necessitating code modification for new scenarios.
- Trait objects provide runtime polymorphism with dynamic dispatch, where the specific implementation is determined at runtime based on the concrete type of the trait object. This incurs a runtime overhead due to the need for vtables.
- Using generics with impl Trait offers compile-time polymorphism with static dispatch, eliminating the runtime overhead. However, the disadvantage is that the concrete type must be known at compile time, thus limiting flexibility.

22T3 Practise Q5.3

While reviewing someone's code, you find the following type: Box<dyn Fn() -> i32 + Send>.
Explain what the + Send means in the code above?
Explain one reason you might need to mark a type as Send, and what restrictions apply when writing a closure that must be Send.

- The + Send indicates that the type is threadsafe, meaning ownership of the value type can be transferred from one thread to another without violating Rust's ownership and borrowing rules. This ensures concurrency can be executed without data races or undefined behavior.
- One reason you might need to mark a type as Send is to facilitate concurrent execution by allowing values of that type to be safely transferred between threads.
- If a closure captures variables by value and is marked as Send, then the captured values must be Send.
- If a closure captures variables by reference and is marked as Send, then the captured values must be both Send and Sync. This is necessary because shared data might be accessed simultaneously, which is ensured by the Sync trait, allowing multiple threads to read it simultaneously.

22T3 Practise Q5.4

Your friend tells you they don't need the standard library's channels, since they've implemented their own alternative with the following code:

```rust
use std::collections::VecDeque;
use std::sync::Mutex;
use std::sync::Arc;
use std::thread;

#[derive(Clone, Debug)]
struct MyChannel<T> {
    internals: Arc<Mutex<VecDeque<T>>>
}

impl<T> MyChannel<T> {
    fn new() -> MyChannel<T> {
        MyChannel {
            internals: Arc::new(Mutex::new(VecDeque::new()))
        }
    }
    fn send(&mut self, value: T) {
        let mut internals = self.internals.lock().unwrap();
        internals.push_front(value);
    }

    fn try_recv(&mut self) -> Option<T> {
        let mut internals = self.internals.lock().unwrap();
        internals.pop_back()
    }
}

fn main() {
    let mut sender = MyChannel::<i32>::new();
    let mut receiver = sender.clone();
    sender.send(5);
    thread::spawn(move || {
        println!("{:?}", receiver.try_recv())
    }).join().unwrap();
}
```
Identify a use-case where this implementation would not be sufficient, but the standard library's channel would be.
Furthermore, explain why this is the case.
The provided MyChannel implementation shares a single VecDeque guarded by a Mutex across all sender threads. This results in contention and reduced concurrency when multiple sender threads attempt to send values concurrently. Conversely, the standard library's MPSC channel enables multiple senders to transmit messages concurrently to a single receiver without contention, owing to its internal synchronization and buffering mechanisms.

22T3 Practise Q6
The "Read Copy Update" pattern is a common way of working with data when many sources need to be able to access data, but also to update it. It allows a user to access a value whenever it's needed, achieving this by never guaranteeing that the data is always the latest copy. In other words, there will always be something, but it might be slightly old. In some cases, this trade-off is one that's worth making.
In this task, you will be implementing a small RCU data-structure. You should ensure that:
- ***Multiple threads are able to access a given piece of data.***
- ***Threads can pass a closure to the type which updates the data.***
- ***When created, the RCU type starts at generation 0. Every time it is updated, that counter is increased by one.***

```
// lib.rs {
use std::sync::{RwLock, Arc, atomic::{AtomicUsize, Ordering}};

pub struct RCUType<T> {
    data: Arc<RwLock<Arc<T>>>,
```

```rust
    generation: Arc<AtomicUsize>,
}

impl<T> RCUType<T> {
    /// Creates a new `RCUType` with a given value.
    pub fn new(value: T) -> RCUType<T> {
        RCUType {
            data: Arc::new(RwLock::new(Arc::new(value))),
            generation: Arc::new(AtomicUsize::new(0)),
        }
    }

    /// Will call the closure `updater`, passing the current
    /// value of the type; allowing the user to return a new
    /// value for this to store.
    pub fn update(&self, updater: impl FnOnce(&T) -> T) {
        let data_clone = Arc::clone(&self.data);
        let mut data_clone_write = data_clone.write().unwrap();
        let current_arc = Arc::clone(&*data_clone_write);
        let new_value = updater(&*current_arc);
        *data_clone_write = Arc::new(new_value);

        let generation_clone = Arc::clone(&self.generation);
        generation_clone.fetch_add(1, Ordering::SeqCst);
    }

    /// Returns an atomically reference counted smart-pointer
    /// to the most recent copy of data this function has.
    pub fn get(&self) -> Arc<T> {
        let data_clone = Arc::clone(&self.data);
        let data_clone_read = data_clone.read().unwrap();
        let current_arc = Arc::clone(&*data_clone_read);
        current_arc
    }

    /// Return the number of times that the RCUType has been updated.
    pub fn get_generation(&self) -> usize {
        let generation_clone = Arc::clone(&self.generation);
        generation_clone.load(Ordering::SeqCst)
    }
}

impl<T> Clone for RCUType<T> {
    fn clone(&self) -> Self {
        Self {
            data: self.data.clone(),
            generation: self.generation.clone(),
        }
    }
}
// end lib.rs }
```

```
let current_arc = Arc::clone(&*data_clone_write);
```

- The RwLock is dereferenced with * (couldn’t call deref). The & is needed to prevent the inner Arc<T> from being consumed when passed as an argument into Arc::clone. For some reason, there is no compiler error when the * is omitted. This can be explained by deref coercion. **A reference to a type implementing Deref is automatically converted to a reference to its Deref target if required by the context.** That is: (1) data_clone_write is a RwLockWriteGuard<Arc<T>>, (2) RwLockWriteGuard implements Deref with the target type Arc<T>, (3) when Arc::clone(&data_clone_write) is called, Rust sees that Arc::clone expects a &Arc<T>, (4) Rust automatically applies deref coercion to convert &RwLockWriteGuard<Arc<T>> to &Arc<T> using the Deref implementation.

Ordering::SeqCst

- Ordering::SeqCst ensures a globally consistent order of operations. If one thread sees the counter increment from 1 to 2 to 3, all other threads will observe the increments in the same order, preventing any thread from seeing the increments out of sequence, such as jumping from 1 to 3. This ensures that every thread observes atomic modifications in the same sequence. However, the interleaving of threads remains random and depends on thread scheduling.

22T3 Practise Q7.2

Hannah writes a Rust program that intends to call some C code directly through FFI. Her C function has the following prototype:

```
int array_sum(int *array, int array_size);
```

and the following implementation:

```
int array_sum(int *array, int array_size) {
	int sum = 0;
	for (int i = 0; i < array_size; i++) { sum += array[i]; }
	return sum;
}
```

Note that you can assume that this C code is written entirely correctly, and the below extern "C" block is an accurate translation of the C interface.

Her Rust code is currently written as follows:

```
use std::ffi::c_int;

#[link(name = "c_array")]
extern "C" {
    fn array_sum(array: *mut c_int, array_size: c_int) -> c_int;
}

fn test_data() -> (*mut c_int, c_int) {
    let size = 10;
    let array = vec![6991; size].as_mut_ptr();
    (array, size as c_int)
}

fn main() {
    let sum = {
        let (array, size) = test_data();

        // Debug print:
        let message = format!("Calling C function with array of size: {size}");
        println!("{message}");

        unsafe { array_sum(array, size) }
    };

    println!("C says the sum was: {sum}");
}
```
She expects that if she runs her code, it should print that the C code summed to 69910. To her surprise, she runs the program and finds the following:
```
6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/ffi`
Calling C function with array of size: 10
C says the sum was: -2039199222
```
Hannah correctly concludes that there must be a problem with her Rust code.
Identify the issue that is causing the program to misbehave.
Describe a practical solution Hannah could use to fix the bug.
Explain why Rust wasn't able to catch this issue at compile-time.

To fix the issue, Hannah should ensure that the Vec remains in scope and is not deallocated while the C code is still using the data. This can be achieved by changing how the array is passed to the C function:
```
use std::ffi::c_int;

#[link(name = "c_array")]
extern "C" {
    fn array_sum(array: *const c_int, array_size: c_int) -> c_int;
}

fn test_data() -> (Vec<c_int>, *const c_int, c_int) {
    let size = 10;
    let mut array = vec![6991; size];
    let array_ptr = array.as_mut_ptr();
    (array, array_ptr, size as c_int)
}

fn main() {
    let (array, array_ptr, size) = test_data();

    // Debug print:
    let message = format!("Calling C function with array of size: {size}");
    println!("{message}");

    let sum = unsafe { array_sum(array_ptr, size) };

    println!("C says the sum was: {sum}");
}
```
Changes made:

- Modified array_sum function to take a *const c_int to reflect that the array data is not modified.
- Altered test_data() to return the Vec<c_int> to ensure it stays in scope as long as needed.
- Stored the Vec<c_int> in the main function and passed its pointer to the C function, ensuring the Vec lives throughout the necessary duration.

Rust's safety guarantees do not cover operations involving raw pointers (*mut T or *const T). In unsafe blocks, the Rust compiler relies on the programmer to ensure memory safety, effectively bypassing the usual compile-time checks that prevent such issues.

22T3 Exam Q2

Differences between two rolls of people

You will be given two string references, called left and right. On each line is the name of a person. For every person on the left roll, you should return either DiffResult::LeftOnly or DiffResult::Both containing a string reference to that line, depending on whether they're in the right list also. For every person on the right roll, you should return DiffResult::RightOnly containing a reference to that line if they are not in the left roll. You can assume that the people on any single roll are unique. That is, you won't see two of the same person on the left roll, nor two of the same person on the right roll. Before returning, you should sort your list of differences alphabetically by their names. Note that since DiffResult derives PartialOrd + Ord, you should simply be able to call .sort() on your final Vec.

```rust
// Solution {
#[derive(Debug, PartialOrd, Ord, PartialEq, Eq)]
pub enum DiffResult<'a, 'b> {
    LeftOnly(&'a str),
    RightOnly(&'b str),
    Both(&'a str)
}

pub fn compare_rolls<'a, 'b>(left_roll: &'a str, right_roll: &'b str) ->
Vec<DiffResult<'a, 'b>> {
    let left_rolls = left_roll.split("\n").collect::<Vec<&str>>();
    let mut right_rolls = right_roll.split("\n").collect::<Vec<&str>>();

    let mut output: Vec<DiffResult> = Vec::new();
    for left_item in left_rolls {
        if right_rolls.contains(&left_item) {
            output.push(DiffResult::Both(left_item));
            let remove_idx = right_rolls.iter().position(|item| *item ==
left_item);
            right_rolls.remove(remove_idx.unwrap());

        } else {
            output.push(DiffResult::LeftOnly(left_item));
        }
    }

    for right_item in right_rolls {
        output.push(DiffResult::RightOnly(right_item));
    }

    output.sort();
    output
}
// }

/*
// Tutor solution
#[derive(Debug, PartialOrd, Ord, PartialEq, Eq)]
pub enum DiffResult<'a, 'b> {
    LeftOnly(&'a str),
    RightOnly(&'b str),
    Both(&'a str)
}

pub fn compare_rolls<'a, 'b>(left_roll: &'a str, right_roll: &'b str) ->
Vec<DiffResult<'a, 'b>> {
    let left_lines = left_roll.lines().collect::<Vec<_>>();
    let right_lines = right_roll.lines().collect::<Vec<_>>();
    let mut results = Vec::new();
    for line in &left_lines {
        if right_lines.contains(&line) {
            results.push(DiffResult::Both(line))
        } else {
            results.push(DiffResult::LeftOnly(line))
        }
    }
    for line in &right_lines {
        if !left_lines.contains(&line) {
            results.push(DiffResult::RightOnly(line))
        }
    }

    results.sort();
    return results;
}
*/

fn get_string_until_empty_line() -> String {
    std::io::stdin()
        .lines()
        .map(|l| l.unwrap())
        .filter(|l| !l.starts_with('#'))
        .map(|l| format!("{l}\n"))
        .take_while(|l| !l.trim().is_empty())
        .collect()
}

/*
// Test 1
fn main() {
    let tutors_start_of_term = get_string_until_empty_line();
    let start_of_term_only = {
        let tutors_end_of_term = get_string_until_empty_line();
        let comparison = compare_rolls(&tutors_start_of_term,
&tutors_end_of_term);

        comparison
            .into_iter()
            .filter_map(|c| {
                if let DiffResult::LeftOnly(l) = c {
                    Some(l)
                } else {
                    None
                }
            })
            .collect::<Vec<_>>()
    };

    for student in start_of_term_only {
        println!("Left Only: {}", student);
    }
}
*/

// Test 2
fn main() {
    let tutors_start_of_term = get_string_until_empty_line();
    let tutors_end_of_term = get_string_until_empty_line();

    let end_of_term_only = {
        let comparison = compare_rolls(&tutors_start_of_term,
&tutors_end_of_term);

        let right_onlys = comparison
            .into_iter()
            .filter_map(|c| {
                if let DiffResult::RightOnly(l) = c {
                    Some(l)
                } else {
                    None
                }
            })
            .collect::<Vec<_>>();

        drop(tutors_start_of_term);

        right_onlys
    };

    for student in end_of_term_only {
        println!("Right Only: {}", student);
    }
}
```

22T3 Exam Q3

DBMap generic struct

A "DBMap" type will wrap a Vec of tuples, of the form (key, value). Currently, the type only works for tuples of the form (i32, &'static str), however you will need to modify this so it works for any tuples where the key is Eq. Your task is to make this struct generic over all valid types for both its keys and values, then to implement a method on this data-structure.

The method you will implement is called merge. It should take ownership of two DBMaps with the same type of key, and then return a new DBMap with its values being tuples. To describe the algorithm merge uses, we will call one of the DBMaps self, and one of them other.

To create the new DBMap, you should iterate over each element in self. We will call these key and value You should then try to find the first element in other with an equal key. We will call that other_value. You should insert (key, (value, Some(other_value))) into the new DBMap. If you cannot find a matching value in other, you should insert (key, (value, None)) into the new DBMap.

Your implementation should be generic, such that the key is any type which supports equality checking; and the value is any type.

```
/*
// Non-generic
pub struct DBMap {
    pub data: Vec<(i32, &'static str)>
}
```

```
impl DBMap {
    pub fn merge(self, mut other: DBMap) -> DBMap {
        todo!()
    }
}
*/

pub struct DBMap<T, U>
where T: Eq,
{
    pub data: Vec<(T, U)>
}

impl<T, U> DBMap<T, U>
where
    T: Eq,
{
    pub fn merge<V>(mut self, mut other: DBMap<T, V>) -> DBMap<T, (U,
Option<V>)> {

        let mut output: DBMap<T, (U, Option<V>)> = DBMap {
            data: Vec::new(),
        };
        while !self.data.is_empty() {
            let mut found = false;
            for i in 0..other.data.len() {
                if self.data[0].0 == other.data[i].0 {
                    let removed_self = self.data.remove(0);
                    let removed_other = other.data.remove(i);
                    let entry: (T, (U, Option<V>)) = (removed_self.0,
(removed_self.1, Some(removed_other.1)));

                    output.data.push(entry);
                    found = true;
                    break;
                }
            }
            if !found {
                let removed_self = self.data.remove(0);
                let entry: (T, (U, Option<V>)) = (removed_self.0,
(removed_self.1, None));
```

```
                output.data.push(entry);
            }
        }

        output
    }
}

/*
// Test 1
// Non-generic
fn main() {
    let names = DBMap {
        data: vec![
            (1, "Max Verstappen"),
            (3, "Daniel Riccardo"),
            (4, "Lando Norris"),
            (5, "Sebastian Vettel"),
            (6, "Nicholas Latifi"),
            (7, "Kimi Räikkönen"),
            (9, "Nikita Mazepin"),
            (11, "Sergio Pérez")
        ]
    };

    let teams = DBMap {
        data: vec![
            (1, "Red Bull Racing"),
            (4, "McLaren"),
            (81, "McLaren"),
            (10, "Alpine"),
            (11, "Red Bull Racing"),
        ]
    };

    let merged = names.merge(teams);

    for (key, (val, other_val)) in merged.data {
        println!("#{key}: {val} ({})", other_val.unwrap_or("None"));
    }

}
```

```rust
*/

// Test 2
#[derive(Debug)]
enum FormulaOneTeams {
    RedBullRacing,
    McLaren,
    Alpine
}

fn main() {
    let names = DBMap {
        data: vec![
            (1, ("Max Verstappen", "NED")),
            (3, ("Daniel Riccardo", "AUS")),
            (4, ("Lando Norris", "GBR")),
            (5, ("Sebastian Vettel", "GER")),
            (6, ("Nicholas Latifi", "CAN")),
            (7, ("Kimi Räikkönen", "FIN")),
            (9, ("Nikita Mazepin", "RAF")),
            (11, ("Sergio Pérez", "MEX"))
        ]
    };

    let teams = DBMap {
        data: vec![
            (1, FormulaOneTeams::RedBullRacing),
            (4, FormulaOneTeams::McLaren),
            (81, FormulaOneTeams::McLaren),
            (10, FormulaOneTeams::Alpine),
            (11, FormulaOneTeams::RedBullRacing),
        ]
    };

    let merged = names.merge(teams);

    for (key, (val, other_val)) in merged.data {
        let other_val = other_val.map(|f|
format!("{f:?}")).unwrap_or(String::from("None"));
        println!("#{key}: {val:?} ({other_val})");
    }
}
```

22T3 Exam Q4.1

Therese is writing a library to help sell her car. The library defines a Car trait as follows:

```rust
trait Car {
    fn get_price(&self) -> u32;
}
```

Users of the library will create their own structs which represent specific cars by implementing the Car trait. As seen in the trait definition, all Cars have a price.
Therese wants to write a function to total the cost of all the cars in a slice. As she wants this to work for any models of Car, she considers two potential approaches:

```rust
fn get_total_price<C: Car>(cars: &[C]) -> u32;
```

and

```rust
fn get_total_price(cars: &[Box<dyn Car>]) -> u32;
```

Explain the difference between the two approaches.
The difference between the two approaches lies in how they handle ownership and polymorphism. The first approach fn get_total_price<C: Car>(cars: &[C]) -> u32; uses generics and allows users to pass a slice of any type that implements the Car trait directly.
The second approach fn get_total_price(cars: &[Box<dyn Car>]) -> u32; uses trait objects (Box<dyn Car>) and allows users to pass a slice of trait objects that implement the Car trait.
Give one reason why Therese might choose each approach.
Generic approach (fn get_total_price<C: Car>(cars: &[C]) -> u32;):
Therese might choose this approach if she wants to optimize for performance and avoid the overhead of dynamic dispatch. With generics, the compiler can perform monomorphization, generating specialized code for each concrete type, resulting in potentially faster execution.
Trait object approach (fn get_total_price(cars: &[Box<dyn Car>]) -> u32;):
Therese might choose this approach if she needs to work with different types of cars at runtime and wants flexibility in accepting any type that implements the Car trait. Trait objects allow for dynamic dispatch, enabling runtime polymorphism and late binding, which can be useful for scenarios where the concrete types of cars are determined dynamically or when dealing with heterogeneous collections of cars.

22T3 Exam Q4.3

Dynamic size of trait objects indeterminable at compile time

```rust
trait IntoRollCall {
    fn into_roll_call(self) -> String;
}


struct UnswStudent {
    name: String,
    zid: u32,
}

impl IntoRollCall for UnswStudent {
    fn into_roll_call(self) -> String {
        let Self { name, zid } = self;
        format!("{name} z{zid}")
    }
}

/* Error
fn call_roll(students: Vec<Box<dyn IntoRollCall>>) {
    for student in students.into_iter() {
        println!("{}", student.into_roll_call());
    }
}
*/
// Solution {
fn call_roll<T: IntoRollCall>(students: Vec<Box<T>>) {
    for student in students.into_iter() {
        println!("{}", student.into_roll_call());
    }
}
// }

fn main() {
    call_roll(vec![
        Box::new(UnswStudent { name: String::from("Alice"),   zid: 5000000 }),
        Box::new(UnswStudent { name: String::from("Bertie"),  zid: 5000001 }),
        Box::new(UnswStudent { name: String::from("Candice"), zid: 5000002 }),
    ]);
}
```

22T3 Exam Q5.1

MutexGuard is not Send

```
use std::thread;
use std::sync::{Arc, Mutex};

fn main() {
    /* Error
    let mutex: Mutex<i32> = Mutex::new(0);

    thread::scope(|scope| {
        for _ in 0..3 {
            let mut i = mutex.lock().unwrap();
            scope.spawn(move || {
                *i += 1;
            });
        }
    });

    println!("{}", *mutex.lock().unwrap());
    */

    // Solution {
    let arc_mutex: Arc<Mutex<i32>> = Arc::new(Mutex::new(0));

    thread::scope(|scope| {
        for _ in 0..3 {
            let arc_mutex_clone: Arc<Mutex<i32>> = Arc::clone(&arc_mutex);
            scope.spawn(move || {
                let mut arc_mutex_clone_locked =
arc_mutex_clone.lock().unwrap();
                *arc_mutex_clone_locked += 1;
            });
        }
    });

    println!("{}", *arc_mutex.lock().unwrap());
    // }
}
```

22T3 Exam Q5.3

Assorted thread::spawn errors

```
use std::thread;
use std::sync::{Mutex, Arc};
use std::time::Duration;

fn main() {
    let mutex: Arc<Mutex<Vec<i32>>> = Arc::new(Mutex::new(Vec::new()));

    /*
    // Error. Threads are not joined, push and increment are not in same
scope.
    for _ in 0..3 {
        let mutex_clone = mutex.clone();
        thread::spawn(move || {
            {
                // Push 1 to the end of the vec...
                let mut vector = mutex_clone.lock().unwrap();
                vector.push(1);
            }

            {
                // ... then increment that element by 1.
                let mut vector = mutex_clone.lock().unwrap();
                let index = vector.len() - 1;
                vector[index] += 1;
            }
        });
    }

    // NOTE: this code makes it work better for some reason???
    for _ in 0..2000 {}
    */

    /*
    // Error. Push and increment are not in same scope.
    thread::scope(|scope| {
        for _ in 0..10 {
            let mutex_clone = mutex.clone();
            scope.spawn(move || {
                {
                    let mut vector = mutex_clone.lock().unwrap();
                    vector.push(1);
                }
                thread::sleep(Duration::from_millis(1000));
                {
                    let mut vector = mutex_clone.lock().unwrap();
                    let index = vector.len() - 1;
                    vector[index] += 1;
                }
            });
        }
    });
    */

    /*
    // Error. Mutex locked twice in same scope results in deadlock.
    thread::scope(|scope| {
        for _ in 0..10 {
            let mutex_clone = mutex.clone();
            scope.spawn(move || {
                let mut vector = mutex_clone.lock().unwrap();
                vector.push(1);
                let mut vector = mutex_clone.lock().unwrap();
                let index = vector.len() - 1;
                vector[index] += 1;
            });
        }
    });
    */

    thread::scope(|scope| {
        for _ in 0..10 {
            let mutex_clone = mutex.clone();
            scope.spawn(move || {
                let mut vector = mutex_clone.lock().unwrap();
                vector.push(1);
                thread::sleep(Duration::from_millis(1000));
                let index = vector.len() - 1;
                vector[index] += 1;
            });
        }
    });

    println!("{:?}", *mutex.lock().unwrap());
}
```

22T3 Exam Q6

Parallelise code without rayon

```
pub fn get_factors(mut num: u128) -> Vec<u128> {
    let mut factors = Vec::new();
    for i in 2..=num {
        while num % i == 0 {
            factors.push(i);
            num = num / i;
        }
        if num == 1 {
            break;
        }
    }

    factors
}

pub fn get_common_factors(list1: &Vec<u128>, list2: &Vec<u128>) -> Vec<u128> {
    let mut list2 = list2.clone();
    let mut factors = Vec::new();

    for elem in list1 {
        let index = list2.iter().position(|x| *x == *elem);
        if let Some(i) = index {
            factors.push(*elem);
            list2.swap_remove(i);
        }
    }

    factors
}

/*
// Non-concurrent
fn main() {
    use std::collections::HashMap;

    let args: Vec<u128> = std::env::args().skip(1).map(|x|
x.parse().unwrap()).collect();

    let mut factors: HashMap<u128, Vec<u128>> = HashMap::new();
    for arg in args {
        factors.insert(arg, get_factors(arg));
    }

    let mut common_factors: HashMap<(u128, u128), Vec<u128>> = HashMap::new();
    for (val1, facs1) in &factors {
        for (val2, facs2) in &factors {
            if val1 > val2 {
                common_factors.insert((*val1, *val2),
get_common_factors(&facs1, &facs2));
            }
        }
    }

    let mut keys = common_factors.keys().collect::<Vec<_>>();
    keys.sort();

    for k in keys {
        println!("{k:?}: {:?}", common_factors[k]);
    }
}
*/

/*
// Solution
fn main() {
    use std::collections::HashMap;
    use std::thread;
    use std::sync::{Arc, Mutex};

    //let args: Vec<u128> = std::env::args().skip(1).map(|x|
x.parse().unwrap()).collect();
    let args: Vec<u128> = vec![10, 20, 30, 40, 50, 60];

    let factors: Arc<Mutex<HashMap<u128, Vec<u128>>>> =
Arc::new(Mutex::new(HashMap::new()));
    let mut handles = Vec::new();
    for arg in args {
        let factors_clone = Arc::clone(&factors);
        let handle = thread::spawn(move || {
            let get_factors_output = get_factors(arg.clone());
            println!("arg = {}", arg.clone());
            for i in 0..(&get_factors_output).len() {
                println!("get_factors_output[{}] = {}", i,
get_factors_output[i]);
            }
            let mut factors_clone_locked = factors_clone.lock().unwrap();
            (*factors_clone_locked).insert(arg.clone(), get_factors_output);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    let factors = Arc::try_unwrap(factors).unwrap().into_inner().unwrap();
    let common_factors: Arc<Mutex<HashMap<(u128, u128), Vec<u128>>>> =
Arc::new(Mutex::new(HashMap::new()));
    let mut handles2 = Vec::new();

    for (val1, facs1) in &factors {
        for (val2, facs2) in &factors {
            let common_factors_clone = Arc::clone(&common_factors);
            let val1_clone = *val1;
            let val2_clone = *val2;
            let facs1_clone = facs1.clone();
            let facs2_clone = facs2.clone();
            let handle2 = thread::spawn(move || {
                if &val1_clone > &val2_clone {
                    let get_common_factors_output =
get_common_factors(&facs1_clone, &facs2_clone);
                    let mut common_factors_clone_locked =
common_factors_clone.lock().unwrap();
                    (*common_factors_clone_locked).insert((val1_clone,
val2_clone), get_common_factors_output);
                }
            });
            handles2.push(handle2);
        }
    }

    for handle in handles2 {
        handle.join().unwrap();
    }

    let common_factors =
Arc::try_unwrap(common_factors).unwrap().into_inner().unwrap();
    let mut keys = common_factors.keys().collect::<Vec<_>>();
    keys.sort();
    for k in keys {
        println!("{k:?}: {:?}", common_factors[k]);
    }
}
*/

// Tutor solution
fn main() {
    use std::collections::HashMap;
    use std::thread::{scope, spawn};

    //let args: Vec<u128> = std::env::args().skip(1).map(|x|
x.parse().unwrap()).collect();
    let args: Vec<u128> = vec![10, 20, 30, 40, 50, 60];

    let factors: HashMap<u128, Vec<u128>> = args
        .clone()
        .into_iter()
        .map(|a| spawn(move || (a, get_factors(a))))
        .collect::<Vec<_>>()
        .into_iter()
        .map(|a| a.join().unwrap())
        .collect::<HashMap<_, _>>();

    let mut factor_pairs = vec![];
    for val1 in &args {
        for val2 in &args {
            if val1 > val2 {
                factor_pairs.push((*val1, *val2));
            }
        }
    }

    scope(|s| {
        let common_factors: HashMap<(u128, u128), Vec<u128>> = factor_pairs
            .iter()
            .map(|(a, b)| s.spawn(|| ((*a, *b),
get_common_factors(&factors[a], &factors[b]))))
            .collect::<Vec<_>>()
            .into_iter()
            .map(|a| a.join().unwrap())
```

```rust
            .collect::<HashMap<_, _>>();

        let mut keys = common_factors.keys().collect::<Vec<_>>();
        keys.sort();
        for k in keys {
            println!("{k:?}: {:?}", common_factors[k]);
        }
    });
}
```

22T3 Exam Q7.2

Custom implementation of Mutex

The type T is not bounded by thread safety traits, particularly Send. This allows safe code to be written to produce undefined behaviour. By wrapping the standard library implementation of Rc in the custom MyMutex, the inner value of the Mutex can be cloned, thereby existing independently of MyMutex. This negates the safety mechanism of the MutexGuard, allowing access to the inner value from multiple locations in the code. The undefined behaviour generated in this case is an infinite loop that clones the independently accessed inner Rc, leading to unsoundness.

```rust
use std::{cell::UnsafeCell, sync::Mutex, ops::{Deref, DerefMut}};

pub struct MyMutex<T> {
    data: UnsafeCell<T>,
    is_locked: Mutex<bool>,
}

impl<T> MyMutex<T> {
    pub fn new(data: T) -> Self {
        Self {
            data: UnsafeCell::new(data),
            is_locked: Mutex::new(false),
        }
    }

    pub fn lock<'lock>(&'lock self) -> MyGuard<'lock, T> {
        loop {
            let mut is_locked = self.is_locked.lock().unwrap();

            if !*is_locked {
                // we now hold the lock!
                *is_locked = true;
                return MyGuard { mutex: self };
            }
        }
    }
}

// Safety: Mutexes are designed to be used on multiple threads,
//         so we can send them to other threads
//         and share them with other threads.
unsafe impl<T> Send for MyMutex<T> {}
unsafe impl<T> Sync for MyMutex<T> {}

pub struct MyGuard<'lock, T> {
    mutex: &'lock MyMutex<T>,
}

impl<T> Deref for MyGuard<'_, T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        // Safety: We hold the lock until we are dropped,
        //         so we have exclusive access to the data.
        //         The shared borrow of the data is tracked through
        //         the shared borrow of self (elided lifetime).
        unsafe { &*self.mutex.data.get() }
    }
}

impl<T> DerefMut for MyGuard<'_, T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        // Safety: We hold the lock until we are dropped,
        //         so we have exclusive access to the data.
        //         The exclusive borrow of the data is tracked through
        //         the exclusive borrow of self (elided lifetime).
        unsafe { &mut *self.mutex.data.get() }
    }
}

impl<T> Drop for MyGuard<'_, T> {
    fn drop(&mut self) {
        *self.mutex.is_locked.lock().unwrap() = false;
    }
}

/*
// Error. Subtle unsoundness not detected by Miri.
fn main() {
    use std::thread;

    const N_THREADS:    u64 = 20;
    const N_INCREMENTS: u64 = 1000;
    const EXPECTED:     u64 = N_THREADS * N_INCREMENTS;

    let my_mutex: MyMutex<u64> = MyMutex::new(0);

    thread::scope(|scope| {
        for _ in 0..N_THREADS {
            scope.spawn(|| {
                for _ in 0..N_INCREMENTS {
                    *my_mutex.lock() += 1;
                }
            });
        }
    });

    let final_value = *my_mutex.lock();
    println!("Final value: {final_value} (expected {EXPECTED})");
}
*/

fn main() {
    use std::thread;
    use std::rc::Rc;

    const N_THREADS: u64 = 20;
    const N_INCREMENTS: u64 = 1000;

    let my_mutex: MyMutex<Rc<u64>> = MyMutex::new(Rc::new(0));

    thread::scope(|scope| {
        for i in 0..N_THREADS {
            scope.spawn(|| {
                for j in 0..N_INCREMENTS {
                    let rc = {
                        my_mutex.lock().clone()
                    };

                    let mut rcs = Vec::new();
                    loop {
                        rcs.push(rc.clone());
                    }
                }
            });
        }
    });
}
```

23T1 Exam Q4.1

Signature of std::thread::spawn

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
{
      ...
}
```

```rust
use std::thread;

fn main() {
    /* Error
    let the_string = String::from("Hello, World!");

    let handle_1 = thread::spawn(|| {
        print_string(&the_string);
    });
    let handle_2 = thread::spawn(|| {
        print_string(&the_string);
    });
    */

    // Solution
    let the_string = String::from("Hello, World!");
    let the_string_clone = the_string.clone();

    let handle_1 = thread::spawn(move || {
        print_string(&the_string);
    });
    let handle_2 = thread::spawn(move || {
        print_string(&the_string_clone);
    });

    /* Solution 2
    let the_string = "Hello, World!";

    let handle_1 = thread::spawn(move || {
        print_string(&the_string);
    });
    let handle_2 = thread::spawn(move || {
        print_string(&the_string);
    });
    */

    handle_1.join().unwrap();
    handle_2.join().unwrap();
}

fn print_string(string: &str) {
    println!("{string}");
}
```

23T1 Exam Q4.3

TODO! Rayon source code.

Signature of std::iter::Iterator::map

```
pub trait Iterator {
      type Item;

      fn map<B, F>(self, f: F) -> Map<Self, F>
      where
          F: FnMut(Self::Item) -> B,
      {
            ...
      }
}
```

Signature of rayon::iter::ParallelIterator::map

```
pub trait ParallelIterator: Send {
      type Item: Send;

      fn map<F, R>(self, map_op: F) -> Map<Self, F>
      where
            F: Fn(Self::Item) -> R + Sync + Send,
            R: Send,
      {
            ...
      }
}
```

4.3a) Justify the change in type bounds to the type Item associated type parameter.

In the rayon::iter::ParallelIterator::map, the Item associated type must implement the Send trait. This is necessary because rayon performs parallel computations, which means the Item may be moved between threads. The Send trait ensures that the Item can be safely transferred across thread boundaries, which is essential for parallel execution.

4.3b) Justify the change in type bounds involving Send and Sync within fn map.
In rayon, the function closure map_op must implement both Send and Sync traits. The Send trait ensures that the closure itself can be safely passed to a different thread. The Sync trait ensures that the closure can be safely shared and accessed from multiple threads simultaneously. This is crucial for parallel execution, where the same closure might be applied to different elements concurrently across multiple threads.

4.3c) Justify the change in function trait (FnMut to Fn) for the provided closure to map.
In std::iter::Iterator::map, the function closure f is defined as an FnMut, which means it allows for mutable access and modification of its captured variables. In contrast, rayon::iter::ParallelIterator::map uses an Fn closure, which is stateless and can be safely called from multiple threads without requiring mutable access. This is important for parallel execution, where a closure may be applied concurrently, and making it an Fn prevents potential race conditions by not allowing internal state modification.

23T1 Exam Q5
Basic in-memory cache generics
```
/*
// Non-generic
use std::{collections::HashMap, rc::Rc};

pub struct Cache {
    calculator: fn(&String) -> String,
    cache_map: HashMap<String, Rc<String>>,
}

impl Cache {
    pub fn new(calculator: fn(&String) -> String) -> Self {
        Cache {
            calculator,
            cache_map: HashMap::new(),
        }
    }

```

```rust
    pub fn get(&mut self, key: String) -> Rc<String> {
        if let Some(value) = self.cache_map.get(&key) {
            Rc::clone(value)
        } else {
            let value = Rc::new((self.calculator)(&key));
            self.cache_map.insert(key, Rc::clone(&value));

            value
        }
    }
}
*/

// Solution {
use std::{collections::HashMap, hash::Hash, rc::Rc};

pub struct Cache<T, U, F>
where
    F: FnMut(&T) -> U,
    T: Hash + Eq,
{
    calculator: F,
    cache_map: HashMap<T, Rc<U>>,
}

impl<T, U, F> Cache<T, U, F>
where
    F: FnMut(&T) -> U,
    T: Hash + Eq,
{
    pub fn new(calculator: F) -> Self {
        Cache {
            calculator,
            cache_map: HashMap::new(),
        }
    }

    pub fn get(&mut self, key: T) -> Rc<U> {
        if let Some(value) = self.cache_map.get(&key) {
            Rc::clone(value)
        } else {
            let value = Rc::new((self.calculator)(&key));
            self.cache_map.insert(key, Rc::clone(&value));

            value
        }
    }
}
// }

/*
// Test 1
// Non-generic case
fn main() {
    fn string_to_string(x: &String) -> String {
        println!("Function called!");

        x.chars().rev().collect()
    }

    let mut cache = Cache::new(string_to_string);

    let foo = String::from("foo");
    let bar = String::from("bar");
    let baz = String::from("baz");

    println!("cache.get(foo) = {:?}", cache.get(foo.clone()));
    println!("cache.get(bar) = {:?}", cache.get(bar.clone()));
    println!("cache.get(baz) = {:?}", cache.get(baz.clone()));
    println!("cache.get(foo) = {:?}", cache.get(foo));
    println!("cache.get(baz) = {:?}", cache.get(baz));
    println!("cache.get(bar) = {:?}", cache.get(bar));
}
*/

/*
// Test 2
// closure_easy
```

```
fn main() {
    let mut num = 42;
    let mut cache = Cache::new(|key| {
        println!("Closure called!");

        format!("{key} {num}")
    });

    let foo = "foo";
    let bar = "bar";
    let baz = "baz";

    println!("cache.get(\"foo\") = {:?}", cache.get(foo));
    println!("cache.get(\"bar\") = {:?}", cache.get(bar));
    println!("cache.get(\"foo\") = {:?}", cache.get(foo));
    println!("cache.get(\"baz\") = {:?}", cache.get(baz));
    println!("cache.get(\"foo\") = {:?}", cache.get(foo));
    println!("cache.get(\"bar\") = {:?}", cache.get(bar));
    println!("cache.get(\"baz\") = {:?}", cache.get(baz));
}
*/

/*
// Test 3
// closure_hard
fn main() {
    let mut num = 0;
    let mut cache = Cache::new(|key| {
        println!("Closure called!");

        let value = format!("{key} {num}");
        num += 1;

        value
    });

    let foo = "foo";
    let bar = "bar";
    let baz = "baz";
    println!("cache.get(\"foo\") = {:?}", cache.get(foo));
    println!("cache.get(\"bar\") = {:?}", cache.get(bar));
    println!("cache.get(\"foo\") = {:?}", cache.get(foo));
    println!("cache.get(\"baz\") = {:?}", cache.get(baz));
    println!("cache.get(\"foo\") = {:?}", cache.get(foo));
    println!("cache.get(\"bar\") = {:?}", cache.get(bar));
    println!("cache.get(\"baz\") = {:?}", cache.get(baz));
}
*/

// Test 4
// fn_ptr_generic
fn main() {
    fn int_to_string(x: &i32) -> String {
        println!("Function called!");
        x.to_string()
    }

    let mut cache = Cache::new(int_to_string);
    println!("cache.get(&1) = {:?}", cache.get(1));
    println!("cache.get(&2) = {:?}", cache.get(2));
    println!("cache.get(&3) = {:?}", cache.get(3));
    println!("cache.get(&42) = {:?}", cache.get(42));
    println!("cache.get(&1) = {:?}", cache.get(1));
    println!("cache.get(&2) = {:?}", cache.get(2));
    println!("cache.get(&3) = {:?}", cache.get(3));
    println!("cache.get(&42) = {:?}", cache.get(42));
}
```

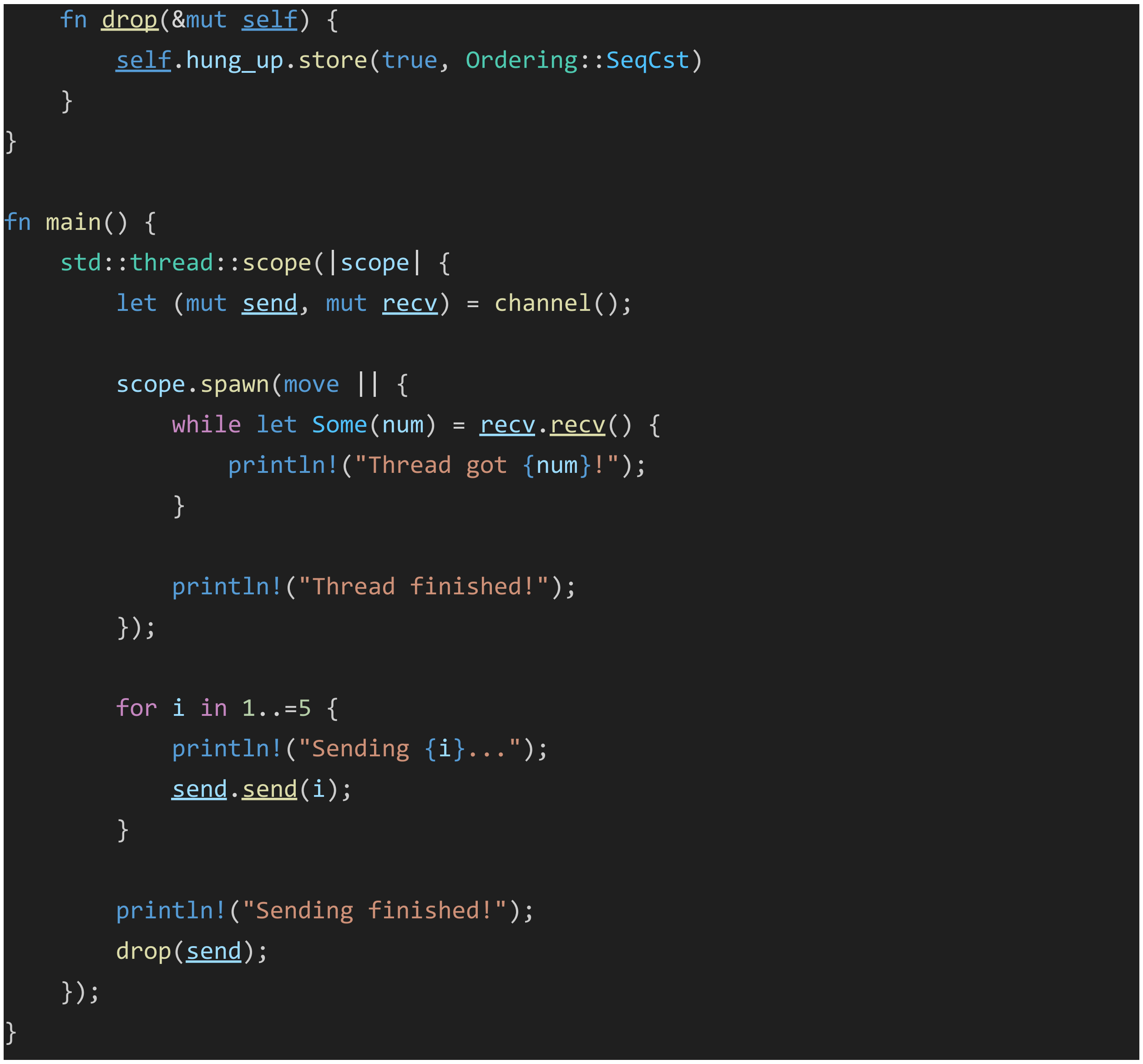
23T1 Exam Q6.2

Single-producer single-consumer channel with buffer size bounded to 1

The channel is initialized with a shared buffer and a "hung up" flag. Both the sender and receiver access these via raw pointers. This buffer holds an Option<i32>, and the "hung up" flag indicates whether the sender has been dropped. After sending all integers, the main thread explicitly drops the sender. This sets the hung_up flag to true, signaling that the sender is no longer available to provide new values. However, the receiver may not have read the final value yet. When the sender is dropped, the receiver's recv() method recognizes the "hung up" state and exits, missing any unprocessed values.

```
/* Error
pub fn channel() -> (Sender<i32>, Receiver<i32>) {
    let buffer  = Box::into_raw(Box::new(None::<i32>));
    let hung_up = Box::into_raw(Box::new(false));

    let sender   = Sender   { buffer, hung_up };
    let receiver = Receiver { buffer, hung_up };

    (sender, receiver)
}

pub struct Sender<T> {
    buffer:  *mut Option<T>,
    hung_up: *mut bool,
}

pub struct Receiver<T> {
    buffer:  *mut Option<T>,
    hung_up: *mut bool,
}

impl<T> Sender<T> {
    pub fn send(&mut self, value: T) -> Option<()> {
        if unsafe { *self.hung_up } {
            return None;
        }

        // wait until the channel is empty...
        loop {
            let value = unsafe { std::ptr::read(self.buffer) };
            if value.is_none() { break; }
            std::mem::forget(value);
        }

        // send the value into the shared buffer
        unsafe { std::ptr::write(self.buffer, Some(value)); }
        Some(())
    }
}

impl<T> Receiver<T> {
    pub fn recv(&mut self) -> Option<T> {
        loop {
            if unsafe { *self.hung_up } {
                return None;
            }

            // wait until the value exists...
            if let Some(value) = unsafe { std::ptr::read(self.buffer) } {
                // clear the channel for the next message
                unsafe { std::ptr::write(self.buffer, None); }

                return Some(value);
            }
        }
    }
}

unsafe impl<T: Send> Send for Sender<T> {}
unsafe impl<T> Send for Receiver<T> {}

impl<T> Drop for Sender<T> {
    fn drop(&mut self) {
        unsafe { *self.hung_up = true; }
    }
}

impl<T> Drop for Receiver<T> {
    fn drop(&mut self) {
        unsafe { *self.hung_up = true; }
    }
}
*/

// Solution
```

```rust
use std::sync::atomic::AtomicBool;
use std::sync::atomic::Ordering;
use std::sync::Arc;

pub fn channel() -> (Sender<i32>, Receiver<i32>) {
    let buffer  = Box::into_raw(Box::new(None::<i32>));
    let hung_up = Arc::new(AtomicBool::new(false));

    let sender = Sender {
        buffer,
        hung_up: Arc::clone(&hung_up),
    };
    let receiver = Receiver {
        buffer,
        hung_up,
    };

    (sender, receiver)
}

pub struct Sender<T> {
    buffer:  *mut Option<T>,
    hung_up: Arc<AtomicBool>,
}

pub struct Receiver<T> {
    buffer:  *mut Option<T>,
    hung_up: Arc<AtomicBool>,
}

impl<T> Sender<T> {
    pub fn send(&mut self, value: T) -> Option<()> {
        if self.hung_up.load(Ordering::SeqCst) {
            return None;
        }

        // wait until the channel is empty...
        loop {
            let value = unsafe { std::ptr::read(self.buffer) };
            if value.is_none() { break; }
            std::mem::forget(value);
        }

        // send the value into the shared buffer
        unsafe { std::ptr::write(self.buffer, Some(value)); }

        Some(())
    }
}

impl<T> Receiver<T> {
    pub fn recv(&mut self) -> Option<T> {
        loop {
            if self.hung_up.load(Ordering::SeqCst) {
                return None;
            }

            // wait until the value exists...
            if let Some(value) = unsafe { std::ptr::read(self.buffer) } {
                // clear the channel for the next message
                unsafe { std::ptr::write(self.buffer, None); }

                return Some(value);
            }
        }
    }
}

unsafe impl<T: Send> Send for Sender<T> {}
unsafe impl<T> Send for Receiver<T> {}

impl<T> Drop for Sender<T> {
    fn drop(&mut self) {
        self.hung_up.store(true, Ordering::SeqCst)
    }
}

impl<T> Drop for Receiver<T> {
```



```rust
    fn drop(&mut self) {
        self.hung_up.store(true, Ordering::SeqCst)
    }
}

fn main() {
    std::thread::scope(|scope| {
        let (mut send, mut recv) = channel();

        scope.spawn(move || {
            while let Some(num) = recv.recv() {
                println!("Thread got {num}!");
            }

            println!("Thread finished!");
        });

        for i in 1..=5 {
            println!("Sending {i}...");
            send.send(i);
        }

        println!("Sending finished!");
        drop(send);
    });
}
```

23T3 Exam Q2

Returned tuples must have the same lifetimes for its elements

```rust
pub fn sort_references<'a>(a: &'a i32, b: &'a i32) -> (&'a i32, &'a i32) {
    if *a > *b {
        (b, a)
    } else {
        (a, b)
    }
}
```

23T3 Exam Q3.3

Extension trait implementation

Helper function that permits the shorthand of collecting an Iterator into a Vec, and then sorting that resulting Vec.

```rust
use std::cmp::Ord;

trait CollectSortedVec<T> {
    fn collect_sorted_vec(self) -> Vec<T>;
}

impl<I, T> CollectSortedVec<T> for I
where
    I: Iterator<Item = T>,
    T: Ord,
{
    fn collect_sorted_vec(self) -> Vec<T> {
        let mut vec: Vec<T> = self.collect(); // Collect the iterator into a Vec
        vec.sort(); // Sort the Vec in place
        vec
    }
}

fn main() {
    let vec = [3, 1, 2].into_iter()
        .map(|x| x * 2)
        .collect_sorted_vec();

    assert_eq!(vec, vec![2, 4, 6]);
}
```

23T3 Exam Q4.2

Blocking call in thread::scope

The program hangs because thread::scope ensures all threads spawned within it are joined before returning to the main thread's execution. The spawned thread is blocked on recv.recv(), awaiting a message indefinitely, preventing further execution.

```rust
use std::{sync::mpsc::channel, thread};

fn main() {
    let (send, recv) = channel();

    /* Error
    thread::scope(|scope| {
        scope.spawn(move || {
            let item = recv.recv().unwrap();
            println!("Received {item} on thread!");
        });
    });
    println!("Unreachable.");
    send.send("hello").unwrap();
    */

    // Solution
    thread::scope(|scope| {
        scope.spawn(move || {
            let item = recv.recv().unwrap();
            println!("Received {item} on thread!");
        });
        send.send("hello").unwrap();
    });
}
```

23T3 Exam Q5

Generic "group-add" function

Group-add is a fictional operation that takes a sequence of elements, and a grouping function which classifies each element into some particular group. It then produces a HashMap of groups to items, where items of the same group are summed up.

For example, the sequence [1, 2, 3, 4, 5, 6, 7] with the grouping function |x| x % 3 should produce a HashMap with the key-value pairs:
0 => 9,
1 => 12,
2 => 7,
because the sequence is grouped into [1, 2, 0, 1, 2, 0, 1] based on their remainders. Note that it is the items that are then summed up:
0 => 3 + 6,
1 => 1 + 4 + 7,
2 => 2 + 5,

```rust
/*
// Non-generic implementation of group_add_items
use std::collections::HashMap;

pub fn group_add_items(items: Vec<i32>, grouper: fn(&i32) -> i32) -> HashMap<i32, i32> {
    let mut groupings = HashMap::new();

    for item in items {
        let group = grouper(&item);
        let current_group = groupings.entry(group)
            .or_default();
        *current_group += item;
    }

    groupings
}
*/

// Solution {
use std::collections::HashMap;
use std::hash::Hash;
use std::ops::AddAssign;

pub fn group_add_items<Item, Group, S, F>(items: S, mut grouper: F) -> HashMap<Group, Item>
where
    F: FnMut(&Item) -> Group,
    Item: AddAssign + Default,
    Group: Hash + Eq,
    S: IntoIterator<Item = Item>,
{
    let mut groupings = HashMap::new();

    for item in items {
        let group = grouper(&item);
        let current_group = groupings.entry(group)
            .or_default();
        *current_group += item;
    }

    groupings
}
// }

/*
// Test 1
// Non-generic
fn main() {
    let items = vec![1, 2, 3, 4, 5, 6, 7];

    let map = group_add_items(items, |x| x % 3);
    assert_eq!(
        map,
        HashMap::from([
            (0, 9),
            (1, 12),
            (2, 7),
        ]),
    );
}
*/

/*
// Test 2
// main_odd_even
fn main() {
    let items = vec![1, 2, 3, 4, 5, 6, 7];
    let map = group_add_items(items, |x| x % 2 == 0);
    assert_eq!(
        map,
        HashMap::from([
            (false, 1 + 3 + 5 + 7),
            (true,  2 + 4 + 6),
        ]),
    );
}
*/

/*
// Test 3
// main_duration
use std::collections::LinkedList;
use std::time::Duration;

#[derive(PartialEq, Eq, Hash)]
enum DurationType {
    Short,
    Medium,
    Long,
}

impl From<&Duration> for DurationType {
    fn from(duration: &Duration) -> Self {
        use DurationType::*;

        if duration < &Duration::from_secs(1) {
            Short
        } else if duration < &Duration::from_secs(60) {
            Medium
        } else {
            Long
        }
    }
}

fn main() {
```

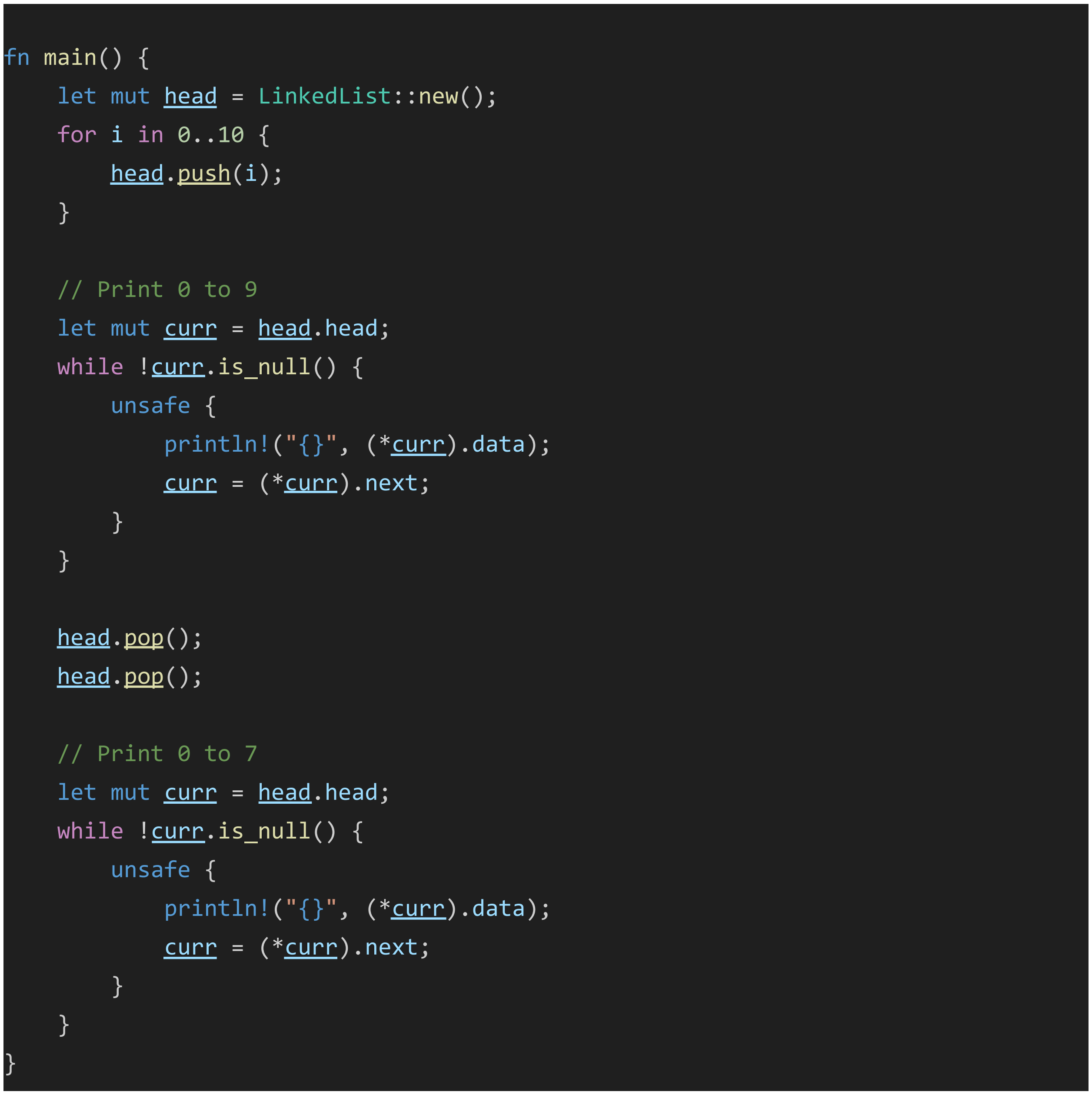
```rust
    let items = LinkedList::from([
        Duration::from_secs(42),
        Duration::from_millis(500),
        Duration::from_secs(5 * 60),
        Duration::from_secs(59) + Duration::from_millis(999),
        Duration::from_secs(120),
    ]);

    let map: HashMap<DurationType, _> = group_add_items(items, |duration|
duration.into());
    assert_eq!(
        map.get(&DurationType::Short),
        Some(&Duration::from_millis(500)),
    );
    assert_eq!(
        map.get(&DurationType::Medium),
        Some(&(Duration::from_secs(42 + 59) + Duration::from_millis(999))),
    );
    assert_eq!(
        map.get(&DurationType::Long),
        Some(&Duration::from_secs(5 * 60 + 120)),
    );
}
*/

// Test 4
// main_naughty_nice
use std::borrow::Cow;
use Verdict::*;

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
enum Verdict {
    Naughty,
    Nice,
}

impl Verdict {
    fn flip(&mut self) {
        *self = match self {
            Naughty => Nice,
            Nice => Naughty,
        }
    }
}

fn main() {
    let staff = [
        Cow::from("Zac\n"),
        Cow::from("Tom\n"),
        Cow::from("Andrew\n"),
        Cow::from("Shrey\n"),
    ];

    let mut verdict = Nice;

    let map = group_add_items(staff, |_| {
        verdict.flip();
        verdict
    });

    assert_eq!(
        map,
        HashMap::from([
            (Naughty, Cow::from("Zac\nAndrew\n")),
            (Nice, Cow::from("Tom\nShrey\n")),
        ]),
    );
}
```

23T3 Exam Q6.2

Linked list code review

```
use std::ptr::{self, null_mut};

struct LinkedList<T> {
    head: *mut Node<T>,
    tail: *mut Node<T>,
```

```rust
}

struct Node<T> {
    data: T,
    next: *mut Node<T>,
}

impl<T> LinkedList<T> {
    fn new() -> Self {
        Self {
            head: ptr::null_mut(),
            tail: ptr::null_mut(),
        }
    }

    fn push(&mut self, data: T) {
        /* Error. Memory not allocated on heap, node gets dropped when out of
scope.
        let node = &mut Node {
            data,
            next: null_mut(),
        } as *mut Node<T>;
        */
        // Solution {
        let node_alloc = Box::new(Node { data, next: null_mut() });
        let node = Box::into_raw(node_alloc);
        // }

        if self.head.is_null() {
            self.head = node;
            self.tail = node;
        } else {
            let curr_tail = unsafe { &mut *self.tail };
            curr_tail.next = node;

            self.tail = node;
        }
    }
    fn pop(&mut self) -> Option<T> {
        unsafe {
            if self.tail.is_null() {
                None
            } else if self.head == self.tail {
                /* Error. Memory is not deallocated.
                let data = ptr::read(self.head).data;
                */
                // Solution {
                let data = ptr::read(self.head).data;
                let _ = Box::from_raw(self.head);
                // }

                self.head = null_mut();
                self.tail = null_mut();

                Some(data)
            } else {
                let mut curr = self.head;

                while ptr::read(curr).next != self.tail {
                    curr = ptr::read(curr).next;
                }
                /* Error. Memory is not deallocated.
                let data = ptr::read(self.tail).data;
                ptr::read(curr).next = null_mut();
                */
                // Solution {
                let data = ptr::read(self.tail).data;
                let _ = Box::from_raw(self.tail);
                ptr::write(&mut (*curr).next, null_mut());
                self.tail = curr;
                // }

                Some(data)
            }
        }
    }
}
```



```rust
fn main() {
    let mut head = LinkedList::new();
    for i in 0..10 {
        head.push(i);
    }

    // Print 0 to 9
    let mut curr = head.head;
    while !curr.is_null() {
        unsafe {
            println!("{}", (*curr).data);
            curr = (*curr).next;
        }
    }

    head.pop();
    head.pop();

    // Print 0 to 7
    let mut curr = head.head;
    while !curr.is_null() {
        unsafe {
            println!("{}", (*curr).data);
            curr = (*curr).next;
        }
    }
}
```