Week 10 Weekly Exercises

Objectives

- Show how the final exam works.
- Prac exam from 22T3, followed by real exam from 22T3, followed by real exam from 23T1, followed by real exam from 23T3
- None of these are marked, but the real exam will be!
- The 24T1 exam will have 7 questions alike 23T1 and 23T3, instead of 8 alike 22T3

Activities To Be Completed

The following is a list of all the markedactivities available to complete this week...

The following practice activities are optional and are not **marked, or required** to be completed for the week.

- **Exam Preamble**
- **Q1**
- **Q2**
- **Q3**
- **Q4**
- **Q5**
- **Q6**
- **Q7**
- **Q8**
- **22T3 Q1: Theory (10 marks)**
- **22T3 Q2: Practical (10 marks)**
- **22T3 Q3: Practical (10 marks)**
- **22T3 Q4: Theory (10 marks)**
- **22T3 Q5: Theory (10 marks)**
- **22T3 Q6: Practical (10 marks)**
- **22T3 Q7: Theory (10 marks)**
- **22T3 Q8: Theory (10 marks)**
- **23T1 Q1: Theory (10 marks)**
- **23T1 Q2: Practical (10 marks)**
- **23T1 Q3: Theory (10 marks)**
- **23T1 Q4: Theory (10 marks)**
- **23T1 Q5: Practical (10 marks)**
- **23T1 Q6: Theory (10 marks)**
- **23T1 Q7: Theory (10 marks)**
- **23T3 Q1: Theory (10 marks)**
- **23T3 Q2: Practical (10 marks)**
- **23T3 Q3: Theory (10 marks)**
- **23T3 Q4: Theory (10 marks)**
- **23T3 Q5: Practical (10 marks)**
- **23T3 Q6: Theory (10 marks)**
- **23T3 Q7: Theory (10 marks)**

Preparation

Before attempting the weekly exercises you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this week's exercises called `lab10`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab10
$ cd lab10
$ 6991 fetch lab 10
```

Or, if you're not working on CSE, you can download the provided code as a tar file.

(OPTIONAL) EXERCISE:

Exam Preamble

Starting time: 2024-05-21 14:00:00

Finishing time: 2024-05-21 17:00:00

Time for the exam: 3 hours

This exam contains 7 questions, each of equal weight (10 marks each).

Total number of marks: 70

Total number of practical programming questions: 2 (20 marks)

Total number of theoretical programming questions: 5 (50 marks)

You should attempt all questions.

Exam Condition Summary

- This exam is "Open Book"
- Joint work is NOT permitted in this exam
- You are NOT permitted to communicate (email, phone, message, talk) with anyone during this exam, except for the COMP6991 staff via cs6991.exam@cse.unsw.edu.au
- The exam paper is confidential, sharing it during or after the exam is prohibited.
- You are NOT permitted to submit code that is not your own
- You may NOT ask for help from online sources.
- Even after you finish the exam, on the day of the exam, do NOT communicate your exam answers to anyone. Some students have extended time to complete the exam.
- Do NOT place your exam work in any location, including file sharing services such as Dropbox or GitHub, accessible to any other person.
- Your zpass should NOT be disclosed to any other person. If you have disclosed your zpass, you should change it immediately.
- **The use of AI assistants is strictly prohibited in this exam. This includes services such as Github Copilot and OpenAI ChatGPT.**

Deliberate violation of these exam conditions will be referred to Student Integrity Unit as serious misconduct, which may result in penalties up to and including a mark of 0 in COMP6991 and exclusion from UNSW.

- You are **allowed** to use any resources from the course during the exam.
- You are **allowed** to use small amounts of code (< 10 lines) of general-purpose code (not specific to the exam) obtained from a site such as Stack Overflow or other publicly available resources. You should attribute the source of this code clearly in an accompanying comment.

Exam submissions will be checked, both automatically and manually, for any occurrences of plagiarism.

By starting this exam, as a student of The University of New South Wales, you do solemnly and sincerely declare that you have not seen any part of this specific examination paper for the above course prior to attempting this exam, nor have any details of the exam's contents been communicated to you. In addition, you will not disclose to any University student any information contained in the abovementioned exam for a period of 24 hrs after the exam. Violation of this agreement is considered Academic Misconduct and penalties may apply.

For more information, read the UNSW Student Code, or contact the Course Account.

- This exam comes with starter files.
- You will be able to commence the exam and fetch the files once the exam commences.

- You may complete the exam questions using any platform you wish (VLab, VSCode, etc). You should ensure that the platform works correctly.
- You may submit your answers, using the give command provided below each question.
- You can use give to submit as many times as you wish. Only the last submission will be marked.
- Do NOT leave it to the deadline to submit your answers. Submit each question when you finish working on it.
- Please make sure that you submit all your answers at the conclusion of the exam - running the autotests does not automatically submit your code.
- Autotests are available for all practical questions to assist in your testing. You can use the command: `6991 autotest`
- Passing autotests does not guarantee any marks. Remember to do your own testing!
- No marks are awarded for commenting - but you can leave comments for the marker to make your code more legible as needed

Language Restriction

- All **practical** programming questions must be answered entirely in Rust; you may not submit code in any other programming languages.
- You are not permitted to use third-party crates other than the standard library (std).

Fit to Sit

By sitting or submitting an assessment on the scheduled assessment date, a student is declaring that they are fit to do so and cannot later apply for Special Consideration.

If, during an exam a student feels unwell to the point that they cannot continue with the exam, they should take the following steps:

1. Stop working on the exam and take note of the time
2. Contact us immediately, using cs6991.exam@cse.unsw.edu.au, and advise us that you are unwell
3. Immediately submit a Special Consideration application saying that you felt ill during the exam and were unable to continue
4. If you were able to advise us of the illness during the assessment (as above), attach screenshots of this conversation to the Special Consideration application

Technical Issues

If you experience a technical issue, you should take the following steps:

1. If your issue is with the connection to CSE, please follow the following steps:
   - **If you are using VLab:** Try exiting VLAB and reconnecting again - this may put you on a different server, which may improve your connection. If you are still experiencing problems, you can try changing how you connect to the CSE servers. Consider:
     - By using VSCode (with SSH-FS extension): https://www.cse.unsw.edu.au/~learn/homecomputing/vscode/
     - By using SSH: https://taggi.cse.unsw.edu.au/FAQ/Logging_In_With_SSH/
   - **If you are using VSCode remote-ssh:** Try disconnecting VSCode, and then changing the URL from `vscode.unsw.edu.au` to `vscode2.unsw.edu.au`.
   - **If you are using SSH:** Try disconnecting SSH and reconnecting again.
2. If things are still NOT working, take screenshots of as many of the following as possible:
   - error messages
   - screen not loading
   - timestamped speed tests
   - power outage maps
3. Contact should be made immediately to advise us of the issue at cs6991.exam@cse.unsw.edu.au
4. A Special Consideration application should be submitted immediately after the conclusion of the assessment, along with the appropriate screenshots.

(OPTIONAL) EXERCISE:
Q1

Question 1

Q1.1 (2 marks)

A C programmer who is starting to learn Rust has asked: "Aren't match statements just complicated if statements?". Give a specific example of a situation where you believe a match statement would significantly improve code quality, instead of a series of if/else statements.

Q1.2 (2 marks)

The following Rust code fails to compile, but equivalent code in other popular programming languages (e.g. C, Java, Python) compiles and/or works correctly. Explain what issue(s) prevent the Rust compiler from building this code, and the philosophy behind this language decision.

```rust
struct Coordinate {
    x: i32,
    y: i32,
};

let coord1 = Coordinate {x: 1, y: 2};
let coord2 = coord1;
let coord_sum = Coordinate { x: coord1.x + coord2.x, y: coord1.y + coord2.y };
```

Q1.3 (3 marks)

In other languages, the operation: `"first_string" + "second_string"` produces a new string, `"first_stringsecond_string"`. This particular operation **does not** work in Rust.

1. Why does Rust not implement this operation on the `&str` type?
2. Would it be possible for the Rust language developers to implement this? What rust feature would they use to implement it?
3. Do you think the Rust language developers should implement this operation? Give one reason to justify your answer.

Q1.4 (3 marks)

Rust beginners have posted some questions on a programming forum:

1. How can I turn an owned value into a shared borrow?
2. How can I turn a shared borrow into an exclusive borrow!
3. Why am I allowed to turn an exclusive borrow into a shared borrow?

Provide a short answer to each question. Importantly, note that some questions might ask for something that is not possible (in which case, you should say so and explain why).

(OPTIONAL) EXERCISE:

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.

You have been given starter code which does not yet compile. Your task is to fill in both `todo!()` statements, as well as to add lifetimes where required in order to build your code.

You are not permitted to change the return type of functions, the names of structs, or the types of structs. You may also not change the main function, and you should expect that the main function could be changed during testing. You will, however, have to add lifetimes to existing types in order to successfully compile your code.

This is an example of the expected behaviour:

```
$  6991 cargo run test_data/test_data.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
    Running `target/debug/prac_q2`
there
very
prove
the universe
  Ctrl-D
Found 1 results for 'there'.
Found 9 results for 'very'.
Found 1 results for 'prove'.
Found 11 results for 'the universe'.
'8 billion years ago, space expanded very quickly (thus the name "Big Bang")' occured 1 times.
'According to the theory the universe began as a very hot, small, and dense superforce (the mix of the four fundamental
forces), with no stars, atoms, form, or structure (called a "singularity")' occured 2 times.
'Amounts of very light elements, such as hydrogen, helium, and lithium seem to agree with the theory of the Big Bang'
occured 1 times.
'As a whole, the universe is growing and the temperature is falling as time passes' occured 1 times.
'Because most things become colder as they expand, scientists assume that the universe was very small and very hot when it
started' occured 2 times.
'By measuring the redshift, scientists proved that the universe is expanding, and they can work out how fast the object is
moving away from the Earth' occured 2 times.
'Cosmology is the study of how the universe began and its development' occured 1 times.
'Other observations that support the Big Bang theory are the amounts of chemical elements in the universe' occured 1
times.
'The Big Bang is a scientific theory about how the universe started, and then made the stars and galaxies we see today'
occured 1 times.
'The Big Bang is the name that scientists use for the most common theory of the universe, from the very early stages to
the present day' occured 2 times.
'The more redshift there is, the faster the object is moving away' occured 1 times.
'The most commonly considered alternatives are called the Steady State theory and Plasma cosmology, according to both of
which the universe has no beginning or end' occured 1 times.
'The most important is the redshift of very far away galaxies' occured 1 times.
'These electromagnetic waves are everywhere in the universe' occured 2 times.
'This radiation is now very weak and cold, but is thought to have been very strong and very hot a long time ago' occured 1
times.
'With very exact observation and measurements, scientists believe that the universe was a singularity approximately 13'
occured 2 times.
```

(OPTIONAL) EXERCISE:

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.

(OPTIONAL) EXERCISE:

Q4

Q4.1 (2 marks)

Steve is writing some Rust code for a generic data structure, and creates a (simplified) overall design alike the following:

```rust
struct S {
    // some fields...
}

impl S {
    fn my_func<T>(value: T) {
        todo!()
    }
}
```

He soon finds that this design is not sufficient to model his data structure, and revises the design as such:

```rust
struct S<T> {
    // some fields...
}

impl<T> S<T> {
    fn my_func(value: T) {
        todo!()
    }
}
```

Give an example of a data-structure that Steve could be trying to implement, such that his first design would not be sufficient, and instead his second design would be required for a correct implementation. Furthermore, explain why this is the case.

Q4.2 (3 marks)

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:

1. An enum
2. A trait object
3. 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.

Q4.3 (5 marks)

Rust's macro system offers an extremely flexible method for code generation and transfiguring syntax, but this language feature comes with certain costs. Identify 3 downsides to the inclusion, design, or implementation of Rust's macro system.

(Note that your downsides may span any amount and combination of the categories above. e.g. you could write all 3 on just one category, or one on each, or anything in-between.)

(OPTIONAL) EXERCISE:

Q5

Q5.1 (3 marks)

In many other popular programming languages, mutexes provide `lock()` and `unlock()` methods which generally do not return any value (i.e. `void`).

What issues could this cause?

How does Rust differently implement the interface of a `Mutex`, and what potential problems does that help solve?

Q5.2 (2 marks)

In Rust, locking a `Mutex` returns a `Result`, instead of simply a `MutexGuard`. Explain what utility this provides, and why a programmer might find this important.

Q5.3 (3 marks)

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`.

Q5.4 (2 marks)

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.

(OPTIONAL) EXERCISE:

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.

You have been given some starter code for the type `RcuType<T>`, including some suggested fields, and the required interface. Ensure you first understand the requirements of this task, and then implement the methods described in the starter code.

(OPTIONAL) EXERCISE:

Q7

Q7.1 (5 marks)

Gavin writes a blog post critical of Rust, especially with respect to `unsafe`. In his blog post, he claims that it's not possible to have any confidence in the overall safety of a Rust program since "even if you only write safe Rust, most standard functions you call will have unsafe code inside them".

1. State to what extent you agree with Gavin's claim.
2. Give at least three arguments that support your conclusion.

Q7.2 (5 marks)

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.

1. Identify the issue that is causing the program to misbehave.
2. Describe a practical solution Hannah could use to fix the bug.
3. Explain why Rust wasn't able to catch this issue at compile-time.

(OPTIONAL) EXERCISE:

Q8

The final question of the exam will be a more open-ended question which will ask you to perform some analysis or make an argument. Your argument will be judged alike an essay (are your claims substantiated by compelling arguments). Remember that you **will not** get any marks for blindly supporting Rust.

A friend of yours has just read this article, and thinks that it means they shouldn't learn Rust.

Read through the article, and **discuss** the following prompt:

Rust is not worth learning, as explained by this article.

The overall structure of your answer is **not** marked. For example, your answer may include small paragraphs of prose accompanied by dot-points, or could instead be posed as a verbal discussion with your friend. Regardless of the structure / formatting you choose, the **substance** of what you write is the most important factor, and is what will determine your overall mark for this question.

NOTE:

From UNSW's Glossary of Task Words:

Discuss	Investigate or examine by argument. Examine key points and possible interpretations, sift and debate, giving reasons for and against. Draw a conclusion.

(OPTIONAL) EXERCISE:

22T3 Q1: Theory (10 marks)

Q1.1 (3 marks)

Question:

1. Explain the difference between the `Option` and `Result` types. *(1 mark)*
2. Give an example of where each might be used. *(2 marks)*

Write your answer in `exam_q1/q1_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_1 q1_1.txt
```

Q1.2 (2 marks)

The following Rust code fails to compile, but equivalent code in other popular programming languages (e.g. C, Java, Python) compiles and/or works correctly.

```rust
let i: u32 = 32;
let j: i32 = -1;
println!("{}", i + j);
```

Question:

1. Explain what issue(s) prevent the Rust compiler from building this code *(1 mark)*.
2. Explain the philosophy behind this language decision *(1 mark)*.

Write your answer in `exam_q1/q1_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_2 q1_2.txt
```

Q1.3 (2 marks)

Your friend has asked you to teach them Rust. They think a great place to begin would be your teaching them to write their own implementation of a doubly-linked list (i.e. a list in which each node is stored on the heap, with references to the previous node and the next node).

Question:

Give two reasons to explain why this is not a problem well-suited for Rust. *(1 mark per reason)*

Write your answer in `exam_q1/q1_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_3 q1_3.txt
```

Q1.4 (3 marks)

A COMP6991 student has decided to build their own operating system in Rust. When implementing the ability to open and read from files, they design these functions based on how Linux deals with files.

```rust
/// This function takes a path to a file; and a mutable reference to a `usize`.
/// If the file at `file_path` can be opened, the function will write a unique
/// `file_id` to the given mutable reference, and return 0. If the file cannot
/// be opened, the function will return an error code.
fn open_file(file_path: &str, file_id: &mut usize) -> usize;

/// This`function takes a `file_id` the user has already obtained, as well as a
/// `buffer` to write bytes to, and a `max_read_size`. The function tries to
/// copy `max_read_size` bytes into `buffer`. It returns the actual number of
/// bytes read. If there is an error, it returns a negative error code.
fn read_file(file_id: i32, buffer: &[u8], max_read_size: usize) -> i32;
```

Question:

Identify three issues in this students plan (either with their design, or their ability to implement their plan in Rust), and three accompanying Rust features that could be used to fix them. *(1 mark per issue+fix)*

Write your answer in `exam_q1/q1_4.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_4 q1_4.txt
```

(OPTIONAL) EXERCISE:

22T3 Q2: Practical (10 marks)

In this activity, you will be finding the differences between two rolls of people. A COMP6991 tutor has collected two rolls of people, and wants to know who's unique to the first one; who's unique to the second one; and who is on both lists.

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`.

You have been given starter code which does not yet compile. Your task is to fill in the `todo!()` statement, as well as to modify the lifetimes where required in order to build your code.

You are **not permitted** to change the return type of functions, the names of structs, or the types of structs. You may also not change the main function, and you should expect that the main function will be changed during testing. Specifically, the main function could be changed to extract `DiffResult::RightOnly` or `DiffResult::Both`. You will, however, have to add or modify lifetimes to existing types in

order to successfully compile your code.

This is an example of the expected behaviour:

```
$ 6991 cargo run test_data/test_data.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
     Running `target/debug/prac_q2`
# Roll 1
Tom
Shrey
Zac
Hailey
Toby
Barry
Netanya

# Roll 2
Tom
Shrey
Hailey
Ctrl-D
Left Only: Barry
Left Only: Netanya
Left Only: Toby
Left Only: Zac
```

Write your answer in `exam_q2/src/lib.rs`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q2 lib.rs
```

(OPTIONAL) EXERCISE:

22T3 Q3: Practical (10 marks)

In this task, you will build a data-structure, called a "DBMap". This 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.

Your implementation should compile with both the main functions provided, however you should assume that more main functions may be tested during marking.

You may assume that any **one individual `DBMap`** will be comprised of totally unique keys.

An example of a merge is shown below:

Stock Prices

Key	Value
Apples	3.5
Pears	4.5

Stock Quantity

Key	Value
Pears	50
Apples	100

Merge of Stock Prices and Quantity

Key	Value
Apples	(3.5, Some(100))
Pears	(4.5, Some(50))

Caviar	200	Peaches	200	Caviar	(200, None)

For example,

```
$ 6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q3`
#1: Max Verstappen (Red Bull Racing)
#3: Daniel Riccardo (None)
#4: Lando Norris (McLaren)
#5: Sebastian Vettel (None)
#6: Nicholas Latifi (None)
#7: Kimi Räikkönen (None)
#9: Nikita Mazepin (None)
#11: Sergio Pérez (Red Bull Racing)
```

Write your answer in `exam_q3/src/lib.rs`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q3 lib.rs
```

(OPTIONAL) EXERCISE:

22T3 Q4: Theory (10 marks)

Q4.1 (2 marks)

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;
```

Question:

1. Explain the difference between the two approaches. *(1 mark)*
2. Give one reason why Therese might choose each approach. *(1 mark)*

Write your answer in `exam_q4/q4_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_1 q4_1.txt
```

Q4.2 (2 marks)

Daniel is trying to design a function that counts the number of whitespace characters in some text. He starts with the first design...

```
fn number_of_whitespace_chars(string: String) -> usize {
    string.chars()
        .filter(char::is_whitespace)
        .count()
}
```

... and submits this for code review. Another software engineer on his team suggests the following change...

```
fn number_of_whitespace_chars(string: &str) -> usize {
    string.chars()
        .filter(char::is_whitespace)
        .count()
}
```

... which Daniel accepts and resubmits. Finally, a senior engineer suggests a further change:

```
fn number_of_whitespace_chars<T: AsRef<str>>(t: T) -> usize {
    t.as_ref()
        .chars()
        .filter(char::is_whitespace)
        .count()
}
```

Question:

1. Why is the first suggested change an improvement on Daniel's original design? *(1 mark)*
2. Why is the second suggested change an improvement on the second design? *(1 mark)*

Write your answer in `exam_q4/q4_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_2 q4_2.txt
```

Q4.3 (3 marks)

Olivia is working on a roll-call system to help take attendance at a class she teaches. She writes the following code:

```
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}")
    }
}

fn call_roll(students: Vec<Box<dyn IntoRollCall>>) {
    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 }),
    ]);
}
```

Although there is currently only one implementor of `IntoRollCall`, she plans to soon add more implementors for other schools.

However, when she attempts to compile the code, she gets a confusing error:

```
$ rustc my_program.rs
error[E0161]: cannot move a value of type `dyn IntoRollCall`
  --> my_program.rs:19:24
   |
19 |         println!("{}", student.into_roll_call());
   |                        ^^^^^^^^^^^^^^^^^^^^^^^^ the size of `dyn IntoRollCall` cannot be statically determined

error: aborting due to previous error

For more information about this error, try `rustc --explain E0161`.
```

Question:

1. Explain the reasoning behind this error message, with specific reference to the requirements of trait objects. *(2 marks)*
2. Identify a simple solution to fix Olivia's program. *(1 mark)*

Write your answer in `exam_q4/q4_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_3 q4_3.txt
```

Q4.4 (3 marks)

Patel does not think Rust's generics system is that special. He claims he can write his own macro which does everything that Rust's generics system can do.

His macro, and an example of him using it, is shown below.

```rust
macro_rules! generic {
    ($($name:ident = $type:ty),+; fn $fn_name:ident($($arg_name:ident: $arg_type:ty),* ) -> $return_type:ty $blk:block) =>
{
        $(type $name = $type);+;

        fn $fn_name($($arg_name: $arg_type),*) -> $return_type {
            $blk
        }
    }
}

generic!(I = i32, O = String; fn add(a: I, b: I) -> O {
    format!("{}", a + b)
});

fn main() {
    let a: i32 = 1;
    let b: i32 = 1;
    println!("{}", add(a, b));
}
```

Question:

Explain to Patel three features of Rust's generics system which his macro does not implement, with explicit reference to an example of how his system would have to implement it. *(1 mark per feature)*

Write your answer in `exam_q4/q4_4.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_4 q4_4.txt
```

(OPTIONAL) EXERCISE:

22T3 Q5: Theory (10 marks)

Q5.1 (3 marks)

The following code attempts to creates multiple threads which each increment an `i32` protected by a `Mutex` by one.

```rust
use std::thread;
use std::sync::Mutex;

fn main() {
    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());
}
```

However, there is a mistake in this code. Fortunately, it is caught by the Rust compiler instead of causing a crash or undefined behaviour at runtime.

Question:

1. Explain how the Rust compiler knows (statically, at compile time) that `i` cannot be sent across threads. *(1 mark)*
2. Explain how you would change the code to compile, while maintaining the behaviour that each thread increments the `i32` by exactly one. *(1 mark)*
3. Explain why your change fixes the issue. *(1 mark)*

Write your answer in `exam_q5/q5_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q5_1 q5_1.txt
```

Q5.2 (2 marks)

Question:

1. Identify a type that should never be marked as `Sync`. *(1 mark)*
2. Explain why that type should never be marked as `Sync`. *(1 mark)*

Write your answer in `exam_q5/q5_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q5_2 q5_2.txt
```

Q5.3 (3 marks)

Zain claims on the course forum that Rust makes concurrency impossible to get wrong.

However, the next post shows a student struggling with their concurrent code:

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

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

    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 {}

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

Question:

1. Explain two concurrency issues with the student's code. *(1 mark per issue)*
2. Discuss to what extent you agree with Zain's claim. *(1 mark)*

Write your answer in `exam_q5/q5_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q5_3 q5_3.txt
```

Q5.4 (2 marks)

Question:

1. Analyse two Rust features which the [Rayon crate]() is able to take advantage of in order to provide static (i.e. compile-time) guarantees as to the safety of a parallel computation. Ensure to explain both the Rust feature, and *how* it allows Rayon to provide such guarantees. *(1 mark per feature)*

Write your answer in `exam_q5/q5_4.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q5_4 q5_4.txt
```

(OPTIONAL) EXERCISE:

22T3 Q6: Practical (10 marks)

In this exercise, we will be making some mathematical operations occur in parallel. You have been given the code to calculate the factors of a number, and to find common factors between two numbers. This code is quite slow, as it only executes on a single thread.

Your task is to parallelise the code in your main function. You can use any concurrency tools within the standard library, but you **must not** use Rayon, or any other non-standard crates. You must not change any code outside the `main.rs` file.

You will receive full marks if you can ensure that (theoretically), all calls to `get_factors` could run simultaneously; and seperately that all calls to `get_common_factors` could run simultaneously.

You should **not** put a constant limit on the number of threads you spawn (i.e., do not use thread-pooling). We suggest that you spawn a new thread for each `get_factors` call, and a new thread for each `get_common_factors` call.

Write your answer in `exam_q6/src/main.rs`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q6 main.rs
```

(OPTIONAL) EXERCISE:

22T3 Q7: Theory (10 marks)

Q7.1 (5 marks)

Larry is trying to implement a difficult data structure in Rust, and is running into some issues with the borrow checker.

Steven, noticing Larry's struggle, offers some unsolicited advice: "Just use `unsafe`, it disables the borrow checker and the rest of the other safety checks."

Stephanie overhears the advice and retorts: "It's more complicated than that. Maybe you should rethink your ownership model before resorting to `unsafe`."

Question:

1. Which safety checks does `unsafe` code opt-out of? *(1 mark)*
2. How and why might `unsafe` be a useful tool for resolving borrow-checking issues when writing, for example, a complex data structure in Rust? *(2 marks)*
3. Explain why using `unsafe` is not always the optimal solution in a case like this, and what procedures might be important to attempt to validate its soundness? *(2 marks)*

Write your answer in `exam_q7/q7_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q7_1 q7_1.txt
```

Q7.2 (5 marks)

The following code attempts to write an (inefficient, but simple) implementation of a `Mutex` using `unsafe`.

```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;
    }
}
```

It can be used like this, which produces the correct, expected output:

```rust
mod my_mutex;

fn main() {
    use std::thread;
    use my_mutex::MyMutex;

    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})");
}
```

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/mutex_broken`
Final value: 20000 (expected 20000)
```

Furthermore, running with Miri (eventually) produces nothing out of the ordinary:

```
$ cargo +nightly miri run
Preparing a sysroot for Miri (target: x86_64-unknown-linux-gnu)... done
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `/home/zac/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/bin/cargo-miri runner target/miri/x86_64-unknown-linux-gnu/debug/mutex_broken`
Final value: 20000 (expected 20000)
```

Note that if you plan to run the code with Miri yourself, you might want to consider decreasing the constants' values - Miri is particularly slow to run.

However, there exists a subtle unsoundness in `MyMutex`. In certain conditions, it is possible for an end-user of `MyMutex` to cause **undefined behaviour** without writing any `unsafe`. In particular, it is possible to cause a data race using only safe code.

> NOTE:
>
> Note that `MyMutex` uses a `Mutex` internally to store the `lock_held` state, however a more appropriate type here would have been an `AtomicBool` instead. A `Mutex` was specifically chosen to limit the scope of unsoundness, and its choice in place of an `AtomicBool` is not relevant to the unsoundness you are tasked to find.

Question:

1. What is the soundness issue in `MyMutex`, and how could it be fixed? *(3 marks)*
2. Provide a short example `main` function using only safe Rust that can reproduce a data race by exploiting the located unsoundness when run with Miri. *(2 marks)*

Write your answer in `exam_q7/q7_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q7_2 q7_2.txt
```

(OPTIONAL) EXERCISE:

22T3 Q8: Theory (10 marks)

The final question of the exam will be a more open-ended question which will ask you to perform some analysis or make an argument. Your argument will be judged alike an essay (i.e. are your claims substantiated by compelling arguments). Remember that you **will not** get any marks for blindly supporting / opposing Rust.

A friend of yours has just read an article critical of Rust, written approximately 7 years ago. Following is a relevant excerpt of the article for you to read. The original article is a very long and challenging read, so please only read / consider this excerpt.

Criticizing the Rust Language, and Why C/C++ Will Never Die

Eax Melanhovich, May 12 2015.

I believe Rust is overhyped, and that the death of C and C++ are over-exaggerated. It is crystal clear for every sane programmer that C/C++ is not going to die in the near future. No one is going to rewrite almost all of the existing desktop applications, operating system kernels, browser engines, tons of other C-libraries, and so on and so forth, into other languages. This is a huge mass of fast, debugged, and time-proven code. Rewriting it is way, way too expensive, risky, and, honestly, doesn't seem to make sense except in the heads of the most frantic Rust fans. The demand for C/C++ programmers has always been high and will remain so for a long time to come.

C/C++ is criticized for a variety of reasons. Briefly, the issue with C++ is that it is very fast but it is not safe in the sense that it allows array overruns, addressing freed memory, and so on. Back in the past, this problem urged programmers to develop a variety of safe languages such as Java, C#, Python, and others. But they have proved to be too resource-demanding compared to C++. That's why programmers are struggling to create a language as fast as C++ but also safe. Rust is one of the candidates.

And what actually makes Rust safe, by the way? To put it simple, this is a language with a built-in code analyzer, and it's a pretty tough one: it can catch all the bugs typical of C++, dealing not only with memory management, but multithreading as well. Pass a reference to an assignable object through a pipe to another thread and then try to use this reference yourself - the program will just refuse to compile. And that's really cool.

But other languages haven't stood still during the last 30 years, and there are many compile-time and runtime analysers which will check your code for issues. In any serious project, you use a continuous integration system and run tons of tests when compiling builds. If you don't, then your troubles are much worse than the language's lack of safety because static typing doesn't guarantee correct execution logic! So, since you run tests anyway, why not use sanitizers as well? True, they don't find all the bugs. But it's still pretty good, and you don't need to deal with performance issues caused by checking the bounds of an array at runtime, or the pain that the Rust compiler forces you into. Even without sanitizers, you'll find lots of stuff just building the project with various compilers on different platforms with assert's checking your code's invariants in the "assert(obj->isValid)" fashion and with proper fuzzing.

Even apart from that, I'm also skeptical about the language's design as such. In particular with regards to the many types of pointers used in it. On the one hand, it's not bad to make programmers ponder if their variables are stored in the stack or heap and if they can or cannot be handled by several threads at a time. But on the other hand, imagine you are writing a program and discover at one moment that some variable should be stored in the heap instead of the stack. So you rewrite the code to use Box. Then you figure out that you actually need Rc or Arc. Again, you rewrite all that code. And then, once again, you rewrite it all to have an ordinary variable in the stack. Regular expressions won't help. Or you might just end up with a nightmare like "Vec<Rc<RefCell<Box<Trait>>>>" - say hello to Java! It would be much more convenient to let the programmer simply declare a variable and explicitly specify Box or Rc where necessary. From this viewpoint, Rust's developers have screwed up the whole thing.

Like many of new languages, Rust is walking the path of simplification. I can generally understand why it doesn't have inheritance and exceptions, but the fact itself that someone is making decisions for me regarding things like that makes me feel somewhat displeased. C++ doesn't restrict programmers regarding what they can or cannot use.

I can't help but remind you one more time that the source of troubles is usually in humans, not technology. If your C++ code is not good enough or Java code is painfully slow, it's not because the technology is bad - it's because you haven't learned how to use it right. In that sense, you won't be satisfied with Rust either - but just for some other reasons. Isn't it easier to learn how to use more popular tools and start liking them?

So, to sum it up, personally I will be investing my time into studying C/C++ rather than Rust in the next 5 or so years. C++ is an industrial standard.

```
Programmers have been using it to solve a huge variety of tasks for over 30
years now. As for Rust and stuff like that - they are just odd toys with a vague
future. People have been predicting C++'s soon death since the 2000-s, but C/C++
haven't become less used and demanded for since then.
```

From https://pvs-studio.com/en/blog/posts/0324/.

Read through the excerpt, and **discuss** the following prompt:

Rust is too strict and too painful to be useful, as explained by this article.

The overall *structure* of your answer is **not** marked. For example, your answer may include small paragraphs of prose accompanied by dot-points, or could instead be posed as a verbal discussion with your friend. Regardless of the structure / formatting you choose, the **substance** of what you write is the most important factor, and is what will determine your overall mark for this question.

Only responses less than 1000 words will be marked for this question. There will be many good answers that are significantly shorter (see above), this limit is a cap to save our marking time, and your sanity.

> NOTE:
>
> From UNSW's Glossary of Task Words:
>
> | | |
> |---|---|
> | **Discuss** | Investigate or examine by argument. Examine key points and possible interpretations, sift and debate, giving reasons for and against. Draw a conclusion. |

Write your answer in `exam_q8/q8.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q8 q8.txt
```

(OPTIONAL) EXERCISE:

23T1 Q1: Theory (10 marks)

Q1.1 (4 marks)

Question:

1. Explain the difference between the `&str` and `String` types, with respect to *ownership*. *(1 mark)*
2. Explain how the behaviour regarding *lifetimes* differ between `&str` and `String`. *(1 mark)*
3. Outline an example of when a `&str` might be used instead of a `String`. *(1 mark)*
4. Outline an example of when a `String` might be used instead of a `&str`. *(1 mark)*

Write your answer in `exam_q1/q1_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_1 q1_1.txt
```

Q1.2 (2 marks)

The following Rust code fails to compile, but equivalent code in other popular programming languages (e.g. C, Java, Python) compiles and/or works correctly.

```rust
fn main() {
    let my_string = String::from("Hello, World!");

    // print the string twice
    print_string(my_string);
    print_string(my_string);
}

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

Question:

1. Explain what issue(s) prevent the Rust compiler from building this code. *(1 mark)*
2. Explain the philosophy behind this language decision. *(1 mark)*

Write your answer in `exam_q1/q1_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_2 q1_2.txt
```

Q1.3 (4 marks)

Question:

Idiomatic Rust programs often extensively use enums when considering errors or other failure cases. Select another programming language of your choice (e.g. C, Java, Python, Go, etc.), and contrast its equivalent error-handling features with Rust.

Provide at least three individual points of contrast *(3 marks)*, along with an overall judgement as to their merits and disadvantages. *(1 mark)*

> NOTE:
>
> Any reasonable programming language is an acceptable choice for this question.
>
> Esoteric languages (e.g., Brainf***, INTERCAL, etc.) are not acceptable, and will not receive marks.
>
> You must select a **programming** language. Languages that do not involve programming (e.g. markup languages (HTML, Markdown), query languages (SQL, jq), etc.) are not acceptable, and will not receive marks.
>
> Programming languages that are slightly more niche (e.g. Haskell, Zig, Swift, Lisp) are perfectly acceptable.
>
> If you are unsure whether your programming language is "reasonable", simply default to C.

Write your answer in `exam_q1/q1_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_3 q1_3.txt
```

(OPTIONAL) EXERCISE:

23T1 Q2: Practical (10 marks)

Boris is attempting to write a function `split_once`, which takes two string slices: `string` and `split_on`. If `split_on` occurs within `string`, it will return `Some` tuple of the slice of `string` that exists before the first occurence of `split_on`, and the remainder of `string` after the first occurence.

Boris is certain that he has written the implementation of the function correctly, but is not so confident in Rust when it comes to lifetimes. Sadly, his code does not compile currently.

```rust
pub fn split_once<'a, 'b>(string: &'a str, split_on: &'b str) -> Option<(&'b str, &'a str)> {
    let index = string.find(split_on)?;
    let tuple = (&string[..index], &string[index + split_on.len()..]);

    Some(tuple)
}
```

Your task is to fix the lifetimes issue with Boris' broken code. You only have to modify lifetimes in order to solve the issue, and thus you are not permitted to change any code except that relevant to lifetimes in `src/lib.rs`. You are permitted to add, remove, and modify lifetimes in this exercise.

This is an example of the expected behaviour:

```
$ 6991 cargo run --bin exam_q2
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q2`
Some(("hello", "world"))

$ 6991 cargo run --bin exam_q2_alt
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q2_alt`
Some(("hello", "world"))
```

Write your answer in `exam_q2/src/lib.rs`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q2 lib.rs
```

(OPTIONAL) EXERCISE:

23T1 Q3: Theory (10 marks)

Luke is trying to define a library of sports cars for use in his Rust programs. He decides that he would like to make use of the trait system to do so, and so defines the following trait:

```rust
pub trait Car {
    fn brand(&self) -> CarBrand;
    fn horsepower(&self) -> u32;
}

pub enum CarBrand {
    Toyota,
    Subaru,
    Nissan,
}
```

Because defining a type and implementing a trait for each individual car requires a substantial amount of boilerplate, he also defines a macro to make the process simpler:

```rust
#[macro_export]
macro_rules! car {
    ($name:ident, $brand:expr, $horsepower:literal) => {
        pub struct $name;
        impl Car for $name {
            fn brand(&self) -> CarBrand {
                use CarBrand::*;
                $brand
            }
            fn horsepower(&self) -> u32 {
                $horsepower
            }
        }
    }
}
```

And so defines the individual cars like such:

```rust
car!(Corolla,  Toyota, 100);
car!(Cressida, Toyota, 160);
car!(Chaser,   Toyota, 220);

car!(Liberty,  Subaru, 100);
car!(Impreza,  Subaru, 130);
car!(Wrx,      Subaru, 200);

car!(Pulsar,   Nissan, 90);
car!(Silvia,   Nissan, 200);
car!(Skyline,  Nissan, 220);
```

This all seems to have worked correctly, so he creates his first function, `favourite_car`, which given a particular `CarBrand` will return back his favourite car model from that brand:

```rust
fn favourite_car(brand: CarBrand) -> impl Car {
    use CarBrand::*;

    match brand {
        Toyota => Cressida,
        Subaru => Liberty,
        Nissan => Skyline,
    }
}
```

However, attempting to build this leads Luke into a problem:

```
$ 6991 cargo build
   Compiling cars v0.1.0
error[E0308]: `match` arms have incompatible types
  --> src/favourite.rs:8:13
   |
6  | /     match brand {
7  | |         Toyota => Cressida,
   | |                   -------- this is found to be of type `cars::Cressida`
8  | |         Subaru => Liberty,
   | |                   ^^^^^^^ expected struct `cars::Cressida`, found struct `cars::Liberty`
9  | |         Nissan => Skyline,
10 | |     }
   | |_____- `match` arms have incompatible types
```

Q3.1 (4 marks)

Question:

1. Explain why Luke is receiving this error message, even with an `impl Car` return type. *(1 mark)*
2. Explain two issues Rust (the language) would face if this behaviour were to be allowed. *(2 marks)*
3. Provide an example of a possible fix for Luke's issue. *(1 mark)*

Write your answer in `exam_q3/q3_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q3_1 q3_1.txt
```

Q3.2 (3 marks)

Luke decides that his `car!` macro still has a bit too much boilerplate. Instead of defining cars individually, he would like to only perform one macro invocation for a whole brand of cars at a time.

That is, he would like to modify his macro such that the following code works equivalently:

```
car! {
    brand  = Toyota;
    models = [
        Corolla  = 100,
        Cressida = 160,
        Chaser   = 220,
    ];
}

car! {
    brand  = Subaru;
    models = [
        Liberty = 100,
        Impreza = 130,
        Wrx     = 200,
    ];
}

car! {
    brand  = Nissan;
    models = [
        Pulsar  = 90,
        Silvia  = 200,
        Skyline = 220,
    ];
}
```

Question

Rewrite Luke's macro as described. Provide the full `macro_rules!` definition in your response. *(3 marks)*

Write your answer in `exam_q3/q3_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q3_2 q3_2.txt
```

Q3.3 (3 marks)

Luke would finally like to parameterise the `Car` trait to include type-level information about the car's transmission layout. There are three possible transmission layouts in Luke's library: front-wheel drive, rear-wheel drive, and all-wheel drive.

Luke knows that most cars are only sold with one choice of transmission layout. For example, all `Wrx`s are all-wheel drive, and all `Silvia`s are rear-wheel drive.

However, some cars may support different transmission layouts. For example, while `Corolla`s were originally rear-wheel drive, newer `Corolla`s can come in both front-wheel drive *and* all-wheel drive!

With this in mind, Luke settles on two options for his trait design:

```rust
// Layouts
struct RearWheelDrive;
struct FrontWheelDrive;
struct AllWheelDrive;

// Trait definition:

// Option 1
pub trait Car<Layout> {
    fn brand(&self) -> CarBrand;
    fn horsepower(&self) -> u32;
}

// Option 2
pub trait Car {
    type Layout;

    fn brand(&self) -> CarBrand;
    fn horsepower(&self) -> u32;
}
```

Question

1. Explain the differences between Luke's two options. *(2 marks)*
2. With the information provided, which option is more suitable for Luke's case? *(1 mark)*

Write your answer in `exam_q3/q3_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q3_3 q3_3.txt
```

(OPTIONAL) EXERCISE:

23T1 Q4: Theory (10 marks)

Q4.1 (4 marks)

The following program attempts to launch two new threads, which each print a shared string.

```rust
use std::thread;

fn main() {
    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);
    });

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

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

However, the code does not compile.

Note: the signature of `std::thread::spawn` is as follows:

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

Question:

1. Explain why the code does not compile. *(1 mark)*
2. Identify the **specific** part of `std::thread::spawn`'s signature that is responsible for this compiler error. *(1 mark)*
3. Explain two possible solutions to fix this issue. *(2 marks)*

Write your answer in `exam_q4/q4_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_1 q4_1.txt
```

Q4.2 (3 marks)

When `lock()` is called on a `std::sync::Mutex`, a `Result` containing a `MutexGuard` is returned.

Question:

1. How does the `Mutex` automatically unlock itself? *(1 mark)*
2. How does Rust statically prevent programmers from extracting the guarded value out of a `MutexGuard`? *(2 marks)*

Write your answer in `exam_q4/q4_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_2 q4_2.txt
```

Q4.3 (3 marks)

The signature of `std::iter::Iterator::map` looks approximately like the following:

```rust
pub trait Iterator {
    type Item;

    ...

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

However, the `rayon` crate's equivalent function, `rayon::iter::ParallelIterator::map` contains some subtle differences:

```rust
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,
    {
        ...
    }
}
```

Question:

1. Justify the change in type bounds to the `type Item` associated type parameter. *(1 mark)*

2. Justify the change in type bounds involving `Send` and `Sync` within `fn map`. *(1 mark)*
3. Justify the change in function trait (`FnMut` to `Fn`) for the provided closure to `map`. *(1 mark)*

Write your answer in `exam_q4/q4_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_3 q4_3.txt
```

(OPTIONAL) EXERCISE:

23T1 Q5: Practical (10 marks)

In this question, you will implement a basic in-memory cache.

A cache is a data-type which when given some input, will perform some calculation and return the result. However, if the calculation for that specific input had already been performed previously, the cache will simply return the result of the previous calculation, instead of doing the hard work again.

You have been given `src/lib.rs`, which describes a very simple cache primitive:

```rust
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(),
        }
    }

    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
        }
    }
}
```

This code correctly implements an in-memory cache that works only for key-value pairs of type `String`. That is, the cache will work for a `calculator` of type `fn(&String) -> String`, and allow you to query/fill the cache with the `get` function turning a `String` key into an `Rc<String>` value.

Your task is to make this cache more generic. You must modify the type of the calculator from a function pointer of `&String -> String` into a closure of `&Key -> Value` for any possible types `Key` and `Value`. As part of this, you must also decide which type of closure is the most generic choice out of `FnOnce`, `FnMut` and `Fn`.

This will also require you to change the generic parameters within the `HashMap` from `<String, Rc<String>>` into `<Key, Rc<Value>>`, and similarly within the `get` function, as previously described in the calculator closure.

You will be required to constrain some generic type parameters as you solve the exercise. You must ensure that you do not overly constrain the types, only requiring what is minimally needed.

Write your answer in `exam_q5/src/lib.rs`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q5 lib.rs
```

(OPTIONAL) EXERCISE:

23T1 Q6: Theory (10 marks)

Q6.1 (4 marks)

Question:

1. Is it possible to invoke undefined behaviour in Rust? If so, how? If not, why? *(1 mark)*
2. Explain the difference between an unsafe block and an unsafe function in Rust. *(1 mark)*
3. Describe the technique of unsafe implementations behind safe abstractions, and explain how it reduces Rust's scope for unsoundness. *(2 marks)*

Write your answer in `exam_q6/q6_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q6_1 q6_1.txt
```

Q6.2 (6 marks)

The following code attempts to write an (inefficient, but simple) implementation of a `channel` using `unsafe`. In particular, it implements a bounded-buffer single-producer single-consumer channel where the bound is always 1.

```rust
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; }
    }
```

```
}
```

It can be used like this:

```rust
use exam_q6_lib::channel;

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);
    });
}
```

```
$ 6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q6`
Sending 1...
Sending 2...
Sending 3...
Thread got 1!
Sending 4...
Thread got 2!
Thread got 3!
Thread got 4!
Sending 5...
Thread got 5!
Sending finished!
Thread finished
```

However, sometimes when it is run, the thread never receives the final message!

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q6`
Sending 1...
Sending 2...
Sending 3...
Thread got 1!
Thread got 2!
Thread got 3!
Sending 4...
Sending 5...
Sending finished!
Thread got 4!
Thread finished
```

Question:

1. Perform a code review on the channel implementation *(not the main function)*, carefully examining the usage of `unsafe` and make a judgement on the overall soundness. *(4 marks)*
2. Demonstrate or otherwise explain how to fix the issue shown above where the thread sometimes does not receive the final message. Pasting a `diff` of the necessary changes to `lib.rs` is sufficient. *(2 marks)*

Write your answer in `exam_q6/q6_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q6_2 q6_2.txt
```

(OPTIONAL) EXERCISE:

23T1 Q7: Theory (10 marks)

The final question of the exam will be a more open-ended question which will ask you to perform some analysis or make an argument. Your argument will be judged alike an essay (i.e. are your claims substantiated by compelling arguments). Remember that you **will not** get any marks for blindly supporting / opposing Rust.

On November 10th 2022, the USA's National Security Agency (i.e. NSA) published a "Cybersecurity Information Sheet" investigating software memory safety. In this 7 page document (which you **are not** expected to read, but is accessible here and as a PDF in the `exam_q7` directory as `nsa.pdf`), the NSA explains that memory safety vulnerabilities represent a potent security threat, to the extent that memory-unsafe languages should be avoided if at all possible. Their overall conclusion is included within the following excerpt (which you **are** expected to read, and is also available in the `exam_q7` directory as `nsa_excerpt.txt`):

```
Examples of memory safe language include C#, Go, Java®, Ruby™, Rust®, and Swift®.

[...]

Memory issues in software comprise a large portion of the exploitable vulnerabilities
in existence. NSA advises organizations to consider making a strategic shift from
programming languages that provide little or no inherent memory protection, such as
C/C++, to a memory safe language when possible.

[...]

Memory safe languages provide differing degrees of memory usage protections, so available
code hardening defenses, such as compiler options, tool analysis, and operating system
configurations, should be used for their protections as well. By using memory safe
languages and available code hardening defenses, many memory vulnerabilities can be
prevented, mitigated, or made very difficult for cyber actors to exploit.
```

In response to this advisory, Bjarne Stroustrup (the designer and implementor of the C++ programming language) published a C++ Standards committee paper titled **A call to action: Think seriously about 'safety'; then do something sensible about it** on December 6th 2022. This paper (in a slightly redacted form) is provided below (and as a text file as `bjarne.txt` in the `exam_q7` directory).

```
[ in response to the NSA information sheet ]

That specifically and explicitly excludes C and C++ as unsafe. As is far too
common, it lumps C and C++ into the single category C/C++, ignoring 30+ years of
progress. Unfortunately, much C++ use is also stuck in the distant past, ignoring
improvements, including ways of dramatically improving safety.

Now, if I considered any of those "safe" languages superior to C++ for the range of
uses I care about, I wouldn't consider the fading out of C/C++ as a bad thing, but
that's not the case. Also, as described, "safe" is limited to memory safety, leaving
out on the order of a dozen other ways that a language could (and will) be used to
violate some form of safety and security.

[...]

There is not just one definition of "safety", and we can achieve a variety of kinds of
safety through a combination of programming styles, support libraries, and enforcement
through static analysis. [C++ language proposal P2410r0] gives a brief summary of
the approach. I envision compiler options and code annotations for requesting rules
to be enforced. The most obvious would be to request guaranteed full type-and-resource
safety. [C++ language proposal P2687R0] is a start on how the standard can support this,
R1 will be more specific. Naturally, comments and suggestions are most welcome.

Not everyone prioritizes "safety" above all else. For example, in application domains
where performance is the main concern, the [C++ language proposal P2687R0] approach
lets you apply the safety guarantees only where required and use your favorite tuning
techniques where needed. Partial adoption of some of the rules (e.g., rules for range
checking and initialization) is likely to be important. Gradual adoption of safety
rules and adoption of differing safety rules will be important. If for no other reason
than the billions of lines of C++ code will not magically disappear, and even "safe"
code (in any language) will have to call traditional C or C++ code or be called by
traditional code that does not offer specific safety guarantees.

Ignoring the safety issues would hurt large sections of the C++ community and undermine
much of the other work we are doing to improve C++. So would focusing exclusively
on safety.

What might "something sensible to do" be? I suggest making a list of issues that could
be considered safety issues (including UB) and finding ways of preventing them within
the framework of [C++ language proposal P2687R0]. That's what I plan to do.
```

From https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2739r0.pdf.

Read through the excerpt above, and **discuss** the following prompt:

Memory safety does not pose a significant enough threat to warrant substituting C or C++ code for Rust, as explained by Bjarne Stroustrup.

The overall *structure* of your answer is **not** marked. For example, your answer may include small paragraphs of prose accompanied by dot-points, or could instead be posed as a verbal discussion with your friend. Regardless of the structure / formatting you choose, the **substance** of what you write is the most important factor, and is what will determine your overall mark for this question.

Only responses less than 1000 words will be marked for this question. There will be many good answers that are significantly shorter (see above), this limit is a cap to save our marking time, and your sanity.

> NOTE:
>
> From UNSW's Glossary of Task Words:
>
> | | |
> |---|---|
> | **Discuss** | Investigate or examine by argument. Examine key points and possible interpretations, sift and debate, giving reasons for and against. Draw a conclusion. |

Write your answer in `exam_q7/q7.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q7 q7.txt
```

(OPTIONAL) EXERCISE:

23T3 Q1: Theory (10 marks)

Q1.1 (3 marks)

This question will ask you to comment on the following Rust snippet, which compiles correctly.

Question:

```
fn main() {
    let name = Some("Tom".to_string());

    print_name(name);
}

fn print_name(name: Option<String>) {
    match name {
        Some(n) => println!("Name is {}", n),
        None => println!("No name"),
    }
}
```

1. On the first line of the main function, we call the `to_string` method. Explain what this function does. *(1 mark)*
2. On the first line of the main function, we use the expression `Some(...)`. Explain the purpose of this code. *(1 mark)*
3. On the second line of the print_name function, we write `Some(n)`. Explain what this does in the context of a match block. *(1 mark)*

Write your answer in `exam_q1/q1_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_1 q1_1.txt
```

Q1.2 (3 marks)

Julian has just learned how to use command line arguments in Rust. The `std::env::args().nth(1)` expression returns an `Option` containing the first command line argument (or `None` if there isn't an argument).

This code is his first attempt at using this function.

Question:

```
fn main() {
    let Some(name) = std::env::args().nth(1);

    println!("First argument was: {}", name);
}
```

Question:

1. Explain in principle (i.e. from a high level) why this code does not work. *(2 marks)*
2. Rewrite Julian's code to use a `let ... else` or `match` expression to assign the variable `name` to the command line argument if present or an empty string otherwise. Do not modify the `println!` code. *(1 mark)*

Write your answer in `exam_q1/q1_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_2 q1_2.txt
```

Q1.3 (4 marks)

Your friend Imran sends you the following message:

> Honestly Rust is so annoying. C and C++ let me just pass pointers or references around without having to worry about this `&mut` and `&` stuff, and none of those confusing lifetime annotations. My code would compile and work just fine in any other language!

Question:

Discuss Rust's ownership and borrowing model with reference to Imran's comment.

In your response, highlight two advantages *(2 marks)* and two disadvantages *(2 marks)* of the ownership and borrowing model.

Write your answer in `exam_q1/q1_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q1_3 q1_3.txt
```

(OPTIONAL) EXERCISE:

23T3 Q2: Practical (10 marks)

In this question, we have written 3 functions which are implemented correctly. These functions are missing lifetime annotations. Your task is to add the correct lifetime annoations to these functions.

```rust
use require_lifetimes::require_lifetimes;
/// This function returns the input it is given.
/// You will need to annotate its lifetimes
/// (2 marks)
#[require_lifetimes]
pub fn identity(a: &i32) -> &i32 {
    a
}

/// This function swaps the two references it is given.
/// You will need to annotate its lifetimes
/// (4 marks)
#[require_lifetimes]
pub fn swap(a: &i32, b: &i32) -> (&i32, &i32) {
    (b, a)
}

//// This function returns the two references it is given in sorted order,
//// with the smallest one first.
//// (4 marks)
#[require_lifetimes]
pub fn sort_references(a: &i32, b: &i32) -> (&i32, &i32) {
    if *a > *b {
        (b, a)
    } else {
        (a, b)
    }
}
```

The code will run some assertions to check correctness, so the expected behaviour is empty output:

```
$ 6991 cargo run --bin exam_q2
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q2`
$ 6991 cargo run --bin exam_q2_alt
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q2_alt`
```

Write your answer in `exam_q2/src/lib.rs`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q2 lib.rs
```

(OPTIONAL) EXERCISE:

23T3 Q3: Theory (10 marks)

Q3.1 (5 marks)

The following code reads all digit characters from a given file and parses it into a single `i32`. For example, given a file with the contents `1, 2, 3-4-5, once I caught a fish alive!` it would output the single number `12345`.

```rust
use std::{error::Error, fs::File, path::Path, io::Read};

fn read_num_from_file(file_path: &Path) -> Result<i32, Box<dyn Error>> {
    let mut file = File::open(file_path)?;

    let mut contents = String::new();
    file.read_to_string(&mut contents)?;

    contents.retain(|ch| ('0'..='9').contains(&ch));
    let num = contents.parse()?;

    Ok(num)
}
```

This code in particular makes use of `Box<dyn Error>` and the `?` (`Try`) operator.

Question:

1. Explain what `Box<dyn Error>` represents. *(1 mark)*
2. Explain how the `?` (`Try`) operator works, with reference to the example code. *(1 mark)*
3. Discuss how this function's approach differs from using a concrete error `enum`, for both the function's author and its users. Provide a judgement. *(3 marks)*

Write your answer in `exam_q3/q3_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q3_1 q3_1.txt
```

Q3.2 (2 marks)

Hannah is trying to design a function that sums up a sequence of `i32`s. She starts with the first design...

```rust
fn sum_sequence(sequence: &[i32]) -> i32 {
    let mut sum = 0;
    for num in sequence {
        sum += num;
    }
    sum
}
```

... and submits this for code review. Another software engineer on her team suggests the following change...

```rust
fn sum_sequence(sequence: impl Iterator<Item = i32>) -> i32 {
    let mut sum = 0;
    for num in sequence {
        sum += num;
    }
    sum
}
```

... which Hannah accepts and resubmits. Finally, a senior engineer suggests a further change:

```rust
fn sum_sequence(sequence: impl IntoIterator<Item = i32>) -> i32 {
    let mut sum = 0;
    for num in sequence {
        sum += num;
    }
    sum
}
```

Question:

1. Why is the first suggested change an improvement on Hannah's original design? *(1 mark)*
2. Why is the second suggested change an improvement on the second design? *(1 mark)*

Write your answer in `exam_q3/q3_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q3_2 q3_2.txt
```

Q3.3 (3 marks)

Olivia is trying to create a `collect_sorted_vec` helper function that permits the shorthand of collecting an `Iterator` into a `Vec`, and then sorting that resulting `Vec`.

In short, she is trying to create a function working in the following way:

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

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

She elects to achieve this through an extension trait that should be implemented on all candidate iterators. She defines the trait as follows:

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

Question:

Write the code to implement this trait for all candidate `Iterators`. A candidate `Iterator` is any `Iterator` whose `Item` implements `Ord`. *(3 marks)*

Write your answer in `exam_q3/q3_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q3_3 q3_3.txt
```

(OPTIONAL) EXERCISE:

23T3 Q4: Theory (10 marks)

Q4.1 (3 marks)

While wandering the halls of UNSW's K17 building, you overhear some young and excited undergraduates discussing their favourite programming languages.

> "Rust has solved concurrency! I haven't run into a single concurrent issue since I started with it!"

Question:

1. Explain how Rust's aliased XOR mutable borrowing model helps to prevent programming concurrency issues. *(2 marks)*
2. Does Rust prevent all concurrency issues? If so, explain how. If not, provide and explain a counter-example. *(1 mark)*

Write your answer in `exam_q4/q4_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_1 q4_1.txt
```

Q4.2 (3 marks)

Lachlan is attempting to use `channels` in a concurrent context. He writes some short example code that spawns a thread using `thread::scope`, and communicates between the main thread and spawned thread through a `channel`.

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

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

    thread::scope(|scope| {
        scope.spawn(move || {
            let item = recv.recv().unwrap();

            println!("Received {item} on thread!");
        });
    });

    send.send("hello").unwrap();
}
```

He expects that when run, the thread should print `"Received hello on thread!"`, but instead the program simply hangs endlessly, and has to be terminated manually with `^C`:

```
$ 6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/channel_test`
[...]
^C
$
```

Question:

1. Explain in detail why the program hangs forever. *(2 marks)*
2. Suggest a fix for this issue. *(1 mark)*

Write your answer in `exam_q4/q4_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_2 q4_2.txt
```

Q4.3 (4 marks)

Annie often writes concurrent code making use of `Arc<Mutex<...>>`s. She values this particular abstraction, but dislikes the syntactic overhead of writing code this way.

In particular, she dislikes writing `Arc::new(Mutex::new(...))`, and having to use a new scope along with cloning her required `Arc`s each time she needs to spawn a thread. Her current sample code looks like the following:

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

fn main() {
    let a = Arc::new(Mutex::new(42));
    let b = Arc::new(Mutex::new("hello"));
    let c = Arc::new(Mutex::new(3.14));

    {
        let a = a.clone();
        let b = b.clone();

        thread::spawn(move || {
            println!("{} {}", a.lock().unwrap(), b.lock().unwrap());
        }).join().unwrap();
    }

    {
        let b = b.clone();
        let c = c.clone();

        thread::spawn(move || {
            println!("{} {}", b.lock().unwrap(), c.lock().unwrap());
        }).join().unwrap();
    }
}
```

She would like to make use of two macros, `arc_mutex!` and `use_arc_mutex!` in order to cut out some of the boilerplate. Her resulting equivalent code should be written as the following:

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

fn main() {
    arc_mutex!(
        a = 42;
        b = "hello";
        c = 3.14;
    );

    use_arc_mutex!(
        a;
        b;
        {
            thread::spawn(move || {
                println!("{} {}", a.lock().unwrap(), b.lock().unwrap());
            }).join().unwrap();
        }
    );

    use_arc_mutex!(
        b;
        c;
        {
            thread::spawn(move || {
                println!("{} {}", b.lock().unwrap(), c.lock().unwrap());
            }).join().unwrap();
        }
    );
}
```

When the macros are expanded, they should produce the original sample code.

Note that `arc_mutex!` should work for any number of declared variables, and `use_arc_mutex!` should work for any number of cloned `Arc`s.

Your task is to implement these two macros for Annie.

Question:

1. Implement the `arc_mutex!` macro. Provide the full `macro_rules!` declaration. *(2 marks)*
2. Implement the `use_arc_mutex!` macro. Provide the full `macro_rules!` declaration. *(2 marks)*

Write your answer in `exam_q4/q4_3.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q4_3 q4_3.txt
```

(OPTIONAL) EXERCISE:

23T3 Q5: Practical (10 marks)

In this question, you will implement a 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,
```

You have been given the following starter code in `src/lib.rs`:

```rust
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
}
```

This code correctly implements `group_add_items` for `Vec` collections of `i32`s, and a grouper using a function pointer from `i32` borrows to `i32`s.

Your task is to make this function far more generic.

Instead of the item list being concretely a `Vec` of `i32`s, it should be a generic sequence of any applicable type.

You must modify the type of the `grouper` from a function pointer of `&i32 -> i32` into a closure of `&Item -> Group` for any possible types `Item` and `Group`. As part of this, you must also decide which type of closure is the most generic choice out of `FnOnce`, `FnMut` and `Fn`.

The returned types within the `HashMap` should similarly be modified to suit.

You will be required to constrain some generic type parameters as you solve the exercise. You must ensure that you do not overly constrain the types, only requiring what is minimally needed.

> NOTE:
>
> You must not modify the body of the `group_add_items` function at all.
>
> That is, the actual running code is already entirely correct. You must **only** modify the signature of the function (optionally adding a `where` section) to make this function more generic.
>
> You are also permitted to import standard library types.

The code already runs correctly on the basic test case outlined above:

```
$ 6991 cargo run --bin exam_q5
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q5`
```

However does not yet typecheck on the other test cases provided. Once you have modified the types to make the function as generic as possible, you should find these other test cases now compile and run:

```
$ 6991 cargo run --bin exam_q5_odd_even
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q5_odd_even`
$ 6991 cargo run --bin exam_q5_duration
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q5_duration`
$ 6991 cargo run --bin exam_q5_naughty_nice
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q5_naughty_nice`
```

Partial marks will be awarded for solutions which pass a subset of the test cases.

Write your answer in `exam_q5/src/lib.rs`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q5 lib.rs
```

(OPTIONAL) EXERCISE:

23T3 Q6: Theory (10 marks)

Q6.1 (4 marks)

The following code demonstrates two possible appearances of so-called "safety comments" in Rust, with the details redacted.

```rust
/// # Safety
/// ...
///
pub unsafe fn my_unsafe_fn(x: *const char) -> *const i32 {
    // Safety: ...
    unsafe {
        // ...
    }
}
```

Question:

1. Explain the purpose of the first safety comment. *(1 mark)*
2. Explain the purpose of the second safety comment. *(1 mark)*
3. Explain how safety comments are validated in Rust. Make reference to at least 1 specific technique. *(2 marks)*

Write your answer in `exam_q6/q6_1.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q6_1 q6_1.txt
```

Q6.2 (6 marks)

The following code attempts to write an implementation of a linked list, making use of `unsafe` code.

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

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

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) {
        let node = &mut Node {
            data,
            next: null_mut(),
        } as *mut Node<T>;

        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 {
                let data = ptr::read(self.head).data;

                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;
                }

                let data = ptr::read(self.tail).data;
                ptr::read(curr).next = null_mut();

                Some(data)
            }
        }
    }
}
```

Question:

1. Perform a code review on the linked list implementation carefully examining the usage of `unsafe`. *(5 marks)*
2. Make a judgement on the overall soundness of the code. *(1 mark)*

Write your answer in `exam_q6/q6_2.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q6_2 q6_2.txt
```

(OPTIONAL) EXERCISE:

23T3 Q7: Theory (10 marks)

The final question of the exam will be a more open-ended question which will ask you to perform some analysis or make an argument. Your argument will be judged alike an essay (i.e. are your claims substantiated by compelling arguments). Remember that you **will not** get any marks for blindly supporting / opposing Rust.

In early October 2023, Bjarne Stroustrup (creator of the C++ language) gave a keynote speech at the 2023 C++ conference. This talk outlined important concerns he had about C++, and about how they would be addressed. The New Stack, a tech-focused news site, published an article that summarised his talk, which we have excerpted below. You are **not** expected to read the full article, though you can find it here The excerpt is copied below (which you **are** expected to read, and is also available in the `exam_q7` directory as `bjarne.txt`):

Bjarne Stroustrup's plan for bringing safety to C++

Published by The New Stack, by David Cassel

The 72-year-old creator of C++, [Bjarne Stroustrup,] gave a forward-looking keynote address last month as the programming language's annual convention.

...

Early in the presentation, Stroustrup shared a slide titled "Safety is not just type safety," highlighting everything from resource leaks and overflows to memory corruption and timing errors. There's concurrency errors, termination errors — and of course, type errors. "Often the safety mentioned is just memory safety — that's not enough… And the need to interoperate with other languages, including C++ and C, tend not to be mentioned. And the cost of conversion can be ferocious. That's rarely mentioned…"

"And anyway — which other language? The way I have seen it argued, we are going to have C++ replaced by about seven different languages, as of suggestions of about now. By the time it happens — 40 years from now — we'll probably have 20 different ones, and they have to interoperate. This is going to be difficult."

Elsewhere in the talk Stroustrup also points out that "A lot of the so-called 'safe' languages outsource all the low-level stuff to C or C++," temporarily escaping the original language to access hardware resources or even the operating system (which is often written in C) — or even "trusted code" which may actually be very old, tucked away in an external library… or written in an entirely different programming language.

As Stroustrup sees it, "This idea of just building a new system over on the side, without any of the problems of the old one, is a fantasy. But it's a very popular fantasy."

...

So about 54 minutes into the talk, Stroustrup told his audience, "Now I want to get to future stuff: where do we go from here…? "

...

Stroustrup has arrived at his solution: profiles. (That is, a set of rules which, when followed, achieve specific safety guarantees.)
...
Stroustrup lays out the general strategy: using static analysis to eliminate potential errors. But "Global static analysis is just unaffordable," Stroustrup adds. "So basically we need rules to simplify what we are writing to something that can be analyzed efficiently and cheaply — local static analysis… And then provide libraries to make relying on these rules feasible."

One slide also noted another advantage: that "Gradual conversion from older code to modern code offering guarantees is supported. The slide emphasized that there'll be a standard "fundamental" set of guarantees, with a larger, open universe of available guarantees. Stroustrup says "I'm imagining type- and resource-safety, memory safety, range safety. Arithmetic safety, things like that, could be standardized." And there will also be rules for applying different guarantees to different fragments of code.

Code could even gets explicit expressions of which guarantees were applied (thus reassuring future readers). Stroustrup again put up his slide listing the "many notions of safety" — a slide titled "Safety is not just type safety," highlighting resource leaks, overflows, memory corruption, timing errors, concurrency errors, termination errors — and of course, type errors.

One slide succinctly makes the case: "Being careful" doesn't scale. So while the core guidelines may suggest safe coding practices, "We need enforced rules." As Stroustrup puts it, "We have to formulate the rules for safe use. We have to provide ways of verifying that people actually do what they're trying to do." Stroustrup points out that much of what he's describing has already been tried, even at scale. "But nowhere has it all been integrated into a consistent, coherent whole. And that's what I'm arguing we should do."

```
One slide lays out the formula in six words: hygiene rules + static analysis +
run-time checks. Stroustrup put up a slide saying C++ can eliminate many common
errors — including uninitialized variables, range errors, null pointer
dereferencing, resource leaks, and dangling references.

[End of excerpt]
```

Read through the excerpt above, and **discuss** the following prompt:

Gradual conversion of existing code to a new "profiles" system is a compelling alternative to widespread adoption of Rust.

The overall *structure* of your answer is **not** marked. For example, your answer may include small paragraphs of prose accompanied by dot-points, or could instead be posed as a verbal discussion with your friend. Regardless of the structure / formatting you choose, the **substance** of what you write is the most important factor, and is what will determine your overall mark for this question.

Only responses less than 1000 words will be marked for this question. There will be many good answers that are significantly shorter (see above), this limit is a cap to save our marking time, and your sanity.

NOTE:

From UNSW's Glossary of Task Words:

Discuss	Investigate or examine by argument. Examine key points and possible interpretations, sift and debate, giving reasons for and against. Draw a conclusion.

Write your answer in `exam_q7/q7.txt`.

When you are finished working on your answer, submit your work with **give**:

```
$ give cs6991 exam_q7 q7.txt
```

Submission

When you are finished each exercise make sure you submit your work by running `give`.

You can run `give` multiple times.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

The due date for this week's exercises is **2024-05-13 21:00:00**.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

Automarking will be run continuously throughout the term, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run you can view your results here or by running this command on a CSE machine:

```
$ 6991 classrun -sturec
```