COMP6991 Supplementary Exam

Getting Started

Create a new directory for this lab called `exam_supp`, change to this directory, and fetch the provided code for the exam by running these commands:

```
$ mkdir exam_supp
$ cd exam_supp
$ 6991 fetch exam
```

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

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.

EXERCISE:

Q1: Theory (10 marks)

Q1.1 (3 marks)

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

```rust
fn get_argument(index: usize) -> (bool, String) {
    let args: Vec<String> = std::env::args().collect();

    if index >= args.len() {
        return (false, "Could not find argument".to_string());
    }
    return (true, args[index].clone());
}

fn main() {
    let (found_arg, text) = get_argument(1);
    if !found_arg {
        println!("No argument found");
        return;
    }
    println!("Argument: {text}");
}
```

Question:

1. Identify a standard library Rust enum that could be used here instead of returning a tuple. *(1 mark)*
2. Why could using this enum be better, compared to returning a `(Output, Error)` tuple? Identify one aspect in which using this enum would be an improvement. *(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 (3 marks)

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

```rust
fn get_random_number() -> i32 {
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}

fn main() {
    let answer = {
        let number = get_random_number();
        if number == 4 {
            "random number."
        } else {
            "non-random number."
        }
    };

    println!("The answer is: {answer}");
}
```

Question:

1. What is the type of `answer`? *(1 mark)*
2. The variable `answer` is equal to a block of code. Identify what property of the Rust language allows variables to be assigned to blocks of code. *(1 mark)*
3. Briefly, how does this feature work? *(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)

This question will ask you to comment on the following Rust snippet, which does not compile.

```rust
fn main() {
    let x;

    {
        let y = String::from("hello");
        x = &y;
    }

    println!("{x}");
}
```

Question

1. Explain why Rust fails to compile this code. *(2 marks)*
2. The equivalent C program produces no compilation error. Is this an advantage of C over Rust? Discuss. *(2 marks)*

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
```

EXERCISE:

Q2: Practical (10 marks)

In this question, we have written three functions (and one struct) which are implemented correctly. These functions and struct are missing lifetime annotations. Your task is to add the correct lifetime annotations to these functions and struct.

```rust
#![allow(unused)]

use require_lifetimes::require_lifetimes;

/// This function prints the given input.
/// You will need to annotate its lifetimes.
/// (2 marks)
#[require_lifetimes]
pub fn print(a: &i32) {
    println!("{a}");
}

/// This function returns the first parameter it is given.
/// You will need to annotate its lifetimes.
/// (3 marks)
#[require_lifetimes]
pub fn first(a: &i32, b: &i32) -> &i32 {
    a
}

/// A struct to hold the data of a string being split.
/// You will need to annotate its lifetimes.
/// (2 marks)
pub struct StringSplitter {
    pub text: &str,
    pub pattern: &str,
}

/// This function creates a string splitter with given data.
/// You will need to annotate its lifetimes.
/// (3 marks)
#[require_lifetimes]
pub fn split(text: &str, pattern: &str) -> StringSplitter {
    StringSplitter {
        text,
        pattern,
    }
}
```

The code will run some assertions to check correctness:

```
$ 6991 cargo run --bin exam_q2
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q2`
5
$ 6991 cargo run --bin exam_q2_alt
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q2_alt`
10
```

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
```

EXERCISE:

Q3: Theory (10 marks)

Q3.1 (4 marks)

Question:

1. Identify two enum types that the `?` operator can be used on in Rust. *(2 marks)*
2. Describe what the `?` operator does operating on a successful enum variant value. *(1 mark)*
3. Describe what the `?` operator does operating on an erroneous enum variant value. *(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)

Liam has created a `Point` struct that reads as follows:

```rust
struct Point {
    x: i32,
    y: i32,
}
```

He would like to include functionality to add two points together, so he writes a function as follows:

```rust
impl Point {
    fn add(self, rhs: Point) -> Point {
        Self {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}
```

However, his friend suggests rewriting their code as follows:

```rust
use std::ops::Add;

impl Add for Point {
    type Output = Point;

    fn add(self, rhs: Point) -> Point {
        Self {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}
```

Question:

1. Discuss the two approaches to writing the `add` function, providing a judgement. *(2 marks)*
2. Describe a context in which one approach of the `add` function would work, where the other would not. *(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)

A function has been written that prints a provided item surrounded by stars, only requiring the item to implement `Display`:

```rust
use std::fmt::Display;

fn print_in_stars<D>(item: D)
where
    D: Display,
{
    println!("*** {item} ***");
}
```

Nat then tries to use this function, printing either an int or a string with stars by writing the following code:

```rust
fn print_something(cond: bool) {
    let item = if cond {
        42
    } else {
        "foo"
    };

    print_in_stars(item);
}
```

However, even though both `i32` and `&str` do implement `Display`, her code does not compile.

Question:

1. Explain (in plain terms) why her code does not compile. *(2 marks)*
2. Suggest an appropriate fix for this issue. You can choose to either adequately explain the fix, or simply write the fixed code instead. *(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
```

EXERCISE:

Q4: Theory (10 marks)

Q4.1 (3 marks)

While browsing the topical chatter of the `/r/rust` subreddit, you find a commenter bragging of Rust's safety:

> Posted by rust-fanatic-9001:
>
> "Did you know that it's totally impossible to cause a data race in Rust?"

Question:

1. How does Rust's borrowing model help to prevent programming concurrency issues? *(1 mark)*
2. Is it impossible to cause a data race in Rust? If so, explain how. If not, provide and explain a counter-example. *(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)

This question focuses on the `Rc<T>` and `Arc<T>` types from the Rust standard library.

Question:

1. The `Rc<T>` type unconditionally does not implement `Send`. Explain what issue would arise if `Rc<T>` implemented `Send`. *(1 mark)*
2. The `Arc<T>` type only implements `Send` if `T` implements `Sync`. Explain what issue would arise if `Arc<T>` implemented `Send` unconditionally regardless of `T`. *(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 (4 marks)

Andrew was originally a C programmer, but now loves writing Rust programs. However, he never quite got over Rust's syntax for declaring variables.

In Rust, variables are declared as `let foo: i32 = 42`, where the equivalent C style code would read `i32 foo = 42`.

When Andrew learned of Rust's macro system, he realised that this tool could solve his problem. Rust's macro system has some restrictions however, so the C style syntax must be slightly modified. Our previous example instead becomes `(i32) foo = 42`, where the type is wrapped in parentheses.

Your task is to implement a C style variable declaration macro for Andrew. In particular, you must implement a `cvar!` macro which accepts zero or more C style variable declarations. It should expand that syntax as follows:

```
fn main() {
    cvar! {
        (i32) foo = 42;
        (&str) bar = "hello";
        (char) baz = 'z';
    }

    // This macro expands into:

    // let foo: i32 = 42;
    // let bar: &str = "hello";
    // let baz: char = 'z';

    println!("{foo} {bar} {baz}");
    // Prints "42 hello z"
}
```

To earn the final two marks of the question, you must also support declaring mutable variables, using the following syntax:

```
fn main() {
    cvar! {
        (i32) mut foo = 42;
        (&str) bar = "hello";
        (char) baz = 'z';
    }

    // This macro expands into:

    // let mut foo: i32 = 42;
    // let bar: &str = "hello";
    // let baz: char = 'z';

    // foo should be mutable:
    foo = 123;

    println!("{foo} {bar} {baz}");
    // Prints "123 hello z"
}
```

Question:

Implement the `cvar!` macro. Provide the full `macro_rules!` declaration.

You will receive *(2 marks)* for correctly implementing the macro without `mut` support, or *(4 marks)* for correctly implementing the macro with `mut` support.

Partial marks may be awarded for progress made.

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
```

EXERCISE:

Q5: Practical (10 marks)

In this question, you will implement a generic "parallel_reduce" function.

Parallel reduce is a fictional operation that takes a sequence of elements, an identity (a.k.a. starting) element, and a reducer function that can turn two elements into just one. It then reduces that sequence of possibly many items into just a single one, splitting up the workload across two threads (by alternating which thread receives the next element).

For example, the sequence `[1, 2, 3, 4, 5]` with the identity `0` and the reducing function `|x, y| x + y` should produce a final value of `15`.

The overall flow of this algorithm can be seen as follows:

```
# Thread a
identity: 0
data: [1, 3, 5]
calculation: (((0 + 1) + 3) + 5)
result: 9

# Thread b
identity: 0
data: [2, 4]
calculation: ((0 + 2) + 4)
result: 6

# Main thread
Results from threads: [9, 6]
calculation: 9 + 6
result: 15
```

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

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

pub fn parallel_reduce(items: Vec<i32>, identity: i32, reducer: fn(i32, i32) -> i32) -> i32 {
    let (a_send, a_recv) = channel();
    let (b_send, b_recv) = channel();
    let (res_send, res_recv) = channel();

    thread::scope(|scope| {
        for chan in [a_recv, b_recv] {
            let reducer = &reducer;
            let identity = identity.clone();
            let res_send = res_send.clone();

            scope.spawn(move || {
                let mut acc = identity;

                while let Ok(elem) = chan.recv() {
                    acc = reducer(acc, elem);
                }

                res_send.send(acc).unwrap();
            });
        }

        let mut is_a = true;
        for item in items {
            let sender = if is_a { &a_send } else { &b_send };
            is_a = !is_a;
            sender.send(item).unwrap();
        }

        drop(a_send);
        drop(b_send);

        let a = res_recv.recv().unwrap();
        let b = res_recv.recv().unwrap();
        reducer(a, b)
    })
}
```

This code correctly implements `parallel_reduce` for `Vec` collections of `i32`s, an `i32` identity, and a reducer using a function pointer from two `i32`s to a single `i32`.

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.

The identity element should similarly be modified to suit.

You must modify the type of the `reducer` from a function pointer of `fn(i32, i32) -> i32` into a function closure type. You must decide which type of closure is the most generic choice out of `FnOnce`, `FnMut` and `Fn`.

The overall function return type 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 `parallel_reduce` 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.
>
> You do not need to make this generic over reducing multiple types (i.e. `fold`). That is, your reducer function should require the same type for both input parameters.

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`
15
```

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_alt
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q5_odd_even`
true
$ 6991 cargo run --bin exam_q5_alt2
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q5_duration`
120
$ 6991 cargo run --bin exam_q5_alt3
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/exam_q5_naughty_nice`
45
```

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
```

EXERCISE:
Q6: Theory (10 marks)

Q6.1 (4 marks)

Bjorn finds little value in Rust's safety claims:

> "`unsafe` code has to be manually checked over, just like equivalent C or C++ code. Having to write the `unsafe` keyword everywhere just adds unnecessary noise"

Question:

1. Does having an `unsafe` keyword provide any advantages over C or C++ code if it needs to be manually checked regardless? Provide a clear explanation why or why not. *(2 marks)*
2. Discuss any downsides and/or pitfalls of Rust's `unsafe` code system. *(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 an `Rc<T>` smart pointer, making use of `unsafe` code.

```rust
use std::ptr::{self, addr_of, addr_of_mut};
use std::ops::{Deref, DerefMut};

pub struct Rc<T> {
    ptr: *mut RcBox<T>,
}

struct RcBox<T> {
    count: usize,
    value: T,
}

impl<T> Rc<T> {
    pub fn new(value: T) -> Self {
        let inner = Box::new(RcBox {
            count: 1,
            value
        });

        Self {
            ptr: Box::into_raw(inner),
        }
    }
}

impl<T> Clone for Rc<T> {
    fn clone(&self) -> Self {
        let count_addr = unsafe { addr_of_mut!((*self.ptr).count) };
        unsafe { *count_addr += 1 };

        Self {
            ptr: self.ptr,
        }
    }
}

impl<T> Deref for Rc<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        let addr = unsafe { addr_of!((*self.ptr).value) };
        unsafe { &*addr }
   }
}

impl<T> DerefMut for Rc<T> {
    fn deref_mut(&mut self) -> &mut Self::Target {
        let addr = unsafe { addr_of_mut!((*self.ptr).value) };
        unsafe { &mut *addr }
   }
}

impl<T> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            let count_addr = addr_of_mut!((*self.ptr).count);
            let count = ptr::read(count_addr);
            let new_count = count - 1;

            if new_count > 0 {
                ptr::write(count_addr, new_count);
            } else {
                drop(Box::from_raw(self.ptr))
            }
        }
    }
}
```

There exists a subtle unsoundness in this code.

Question:

1. Perform a code review on the `Rc<T>` implementation, with respect to `unsafe` Rust. *(3 marks)*
2. Identify the soundness issue in the code. *(1 mark)*
3. Write a short example program that exercises the soundness issue (e.g. causes a segmentation fault in safe Rust). *(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
```

EXERCISE:

Q7: Theory (10 marks)

The final question of the exam is a more open-ended question which asks you to perform some analysis or make an argument. Your argument will be judged like an essay (i.e. are your claims substantiated by compelling arguments).

One article about using Rust concluded with the following:

```
Should I learn Rust?

Yes, if the following is true for you:
* I desire to become a really great software developer in a modern language capable of productively developing any type of software system
* I routinely need to develop software with compact memory use without being restricted to using a subset of the total language (e.g. like with C++)
* I need to develop performant software, particularly on a single CPU core (but not limited to a single core)
* I need to develop correct and memory-safe software without worry and prefer or require a runtime without a garbage collector
* I've mastered another language and I'm looking for a new challenge

No (or maybe), if the following is true for you:
* To date, I've never (or hardly) developed any software
* I need to rapidly prototype a new concept to find product-market-fit on my own or at a startup company
* I'm not willing or curious enough to persist when certain Rust concepts become challenging
* I have no interest or need to learn how memory management works for the type of software I want to develop
```

Read through the excerpt above, and **justify** or **argue against** the writer's conclusion. In other words...

Explain why these conclusions are or are not correct.

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.

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. Only your last submission will be marked.

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

Do not leave it to the deadline to submit your answers. Submit each question when you finish working on it. Running autotests does not automatically submit your code.

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

Remember you have until **2024-05-21 17:00:00**, Sydney time, to complete this exam (not including any extra time provided by ELPs).

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

```
$ 6991 classrun -check exam_q1_1
$ 6991 classrun -check exam_q1_2
$ 6991 classrun -check exam_q1_3
$ 6991 classrun -check exam_q2
...
$ 6991 classrun -check exam_q7
```

- END OF EXAMINATION. -
