Week 09 Weekly Exercises

Objectives

- Investigate FFI in rust
- Understand what unsoundness means
- Implement your own RC type with unsafe
- Short week of exercises, so more time for assignment

Activities To Be Completed

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

- **Rust2C**
- **Unsound Unsafe**
- **MyRc**

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

> **None - all exercises this week are marked.**

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 `lab09`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab09
$ cd lab09
$ 6991 fetch lab 09
```

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

EXERCISE:
Rust2C

In this exercise, you are tasked with calling C code from within Rust!

You have been provided `secretc.h`, a header file which contains the definition of a secret C function:
`int secret_c_calculation(int x, double y, char z);`

This is joined by `libsecretc.a`, the compiled C code ready to be linked through a Foreign Function Interface (FFI) with the Rust code!

You do not need to understand the C source code, and in-fact, we do not *provide* you the C source code. You are only required to *call* the C code from within Rust, and make use of the result it provides.

You will need to write both the C function definition (using the header file as a guideline) and call that C function from within main yourself.

The linking should already be taken care of for you (with a `build.rs` script and a `#[link(...)] attribute`), so if your code is written correctly, you should be able to simply use `6991 cargo` as usual.

Note that since this exercise makes use of a pre-compiled C object file, we can only guarantee this exercise building correctly on `x86_64-unknown-linux-gnu` systems. This includes most Linux systems (including CSE) and most WSL installs, but not Windows itself nor MacOS. If you aren't running on a typical Linux system, you should work on this exercise on CSE (e.g. VLAB, SSH, lab machines, etc.). Although FFI generally works on many platforms, it is the C library in this case that restricts us to this target.

For example,

```
$ 6991 cargo run -- 42 123.456 F
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/rust2c 42 123.456 F`
The secret value is 176
```

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 this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 10 Sunday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Unsound Unsafe

In this exercise, you have been provided the `unsound_unsafe` crate.

This crate attempts to (very dangerously) implement a singly-linked list data structure making liberal use of raw pointers instead of taking advantage of Rust's safety guarantees provided through its usual types (e.g. references).

Nevertheless, we are making use of this linked list, and it seems to generally work okay! We initialise a new linked list comprised of numbers given as command-line arguments, and perform the following operations:

- Print the list;
- Delete the first element in the list;
- Print the list again;
- Free the list.

Running the program with the list `1 2 3` can be observed as follows:

```
$ 6991 cargo run -- 1 2 3
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/unsound_unsafe 1 2 3`
=== PRINTING LIST ===
1
2
3
====================
=== PRINTING LIST ===
2
3
====================
```

It seems that most of these operations work correctly (i.e. they are sound), however there appears to be a soundness issue when the `list_delete_first` function is called in the case of an already empty list! Users of our dangerous linked-list structure have reported segmentation faults in this specific scenario!

```
$ 6991 cargo run --
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/unsound_unsafe`
=== PRINTING LIST ===
====================
[1]    1402912 segmentation fault  6991 cargo run --
```

Your task is first to locate the unsoundness using Miri. Miri is already installed on CSE systems, so you can try running your code like such:

```
$ 6991 cargo +nightly miri run -- 1 2 3
Preparing a sysroot for Miri (target: x86_64-unknown-linux-gnu)... done
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `/import/glass/1/cs6991/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/bin/cargo-miri runner
target/miri/x86_64-unknown-linux-gnu/debug/foo 1 2 3`
=== PRINTING LIST ===
1
2
3
====================
=== PRINTING LIST ===
2
3
====================
```

Once you understand how to use Miri, try replicating the input demonstrating a segmentation fault from above in order to track down where this issue arises.

Finally, it's time to fix the unsoundness! You are **only** permitted to modify the `list_delete_first` function. You may not modify any other code.

Once you are satisfied that your code is most-likely sound, you should run it once again through Miri! Furthermore, the autotests for this exercise will test your code using Miri on a few different cases.

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 this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 10 Sunday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

MyRc

In this exercise, we will be implementing our own simple `Rc` type.

Your task is to build a datastructure, called `MyRc` which has the following properties:

- The type should allow multiple people to own a value.
- The value should live until the last MyRc is dropped.
- The value should be dropped when the last MyRc is dropped.

There are hints for each function you should complete, in the starter code.

No changes to main.rs should be required, only lib.rs

You will need to write unsafe code in this activity - don't forget SAFETY comments! E.G the rust API guidelines

HINT:

The documentation for the `Box::from_raw` method may be useful.

The documentation for the `Box::into_raw` method may be useful.

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 this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 10 Sunday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

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 **Week 10 Sunday 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
```
