Homework 5 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Question 1

Question 1a

`refCount` should instead be a **pointer** to an `int`. This way, multiple `shared_ptr`s can share the same `int` counter by each having their own pointer member that points to this same `int`.

Question 1b

```
my_shared_ptr(int *ptr)
{
    this->ptr = ptr;
    this->refCount = new int(1);
}
```

Question 1c

```
my_shared_ptr(const my_shared_ptr &other)
{
    this->ptr = other.ptr;
    this->refCount = other.refCount;
    *this->refCount++;
}
```

Question 1d

```
~my_shared_ptr()
{
    *this->refCount--;
    if (*this->refCount == 0)
        delete this->ptr;
}
```

Question 1e

```
my_shared_ptr &operator=(const my_shared_ptr &obj)
{
    // Prevent aliasing.
    if (this == &obj)
        return *this;
    // Modify current shared_ptr, cleaning up if necessary.
    *this->refCount--;
    if (*this->refCount == 0)
        delete this->ptr;
    // Copy over data from obj.
    this->ptr = obj.ptr;
    this->refCount = obj.refCount;
    *this->refCount++;
    return *this;
}
```

Question 2

Question 2a

The collision avoidance system of a space probe is an example of **real-time** software. Such systems must be able to readily perform operations at any given time and not stall or slow down for background processes such as garbage collection (GC). Because C, C++, and Rust do not have GC, they have faster runtime performance.

Question 2b

While a language with reference counting GC does not suffer from freezes, there is still a constant overhead (both in performance and memory) associated with updating reference counts on virtually every memory-related operation. Something like collision avoidance should be optimized for speed. Furthermore, critical real-time software should have well-defined, predictable behavior at all times. Such a constraint should eliminate the need for having GC in the first place.

Question 2c

A **mark-and-compact** GC language such as C# will be a better choice. Mark-and-compact fixes a problem that mark-and-sweep has, which is memory **fragmentation**. While `Coord` is fixed-size, the software needs random-length *arrays* of these structures, and arrays must be contiguous. Thus, Kevin needs to ensure that there is always a memory segment available that can accommodate any size of array.

Question 2d

Go is a garbage-collected language unlike C++, so it does not support object destructors. While C++ guarantees that `this.socket.cleanupFD` is called when a `RoomView`'s lifetime ends, there is no such promise in Go. Thus, many socket file descriptors go unreleased, resulting in the problem Yvonne faces.

According to ChatGPT, Go instead offers a `defer` keyword and mechanism, which **schedules** a function to run at the end of the current function. To fix Yvonne's problem, she should place `defer` statements in every function that a `RoomView` is created. For example:

```
func someFunction()
{
  roomView := RoomView{};
  defer roomView.socket.cleanUpFd();
  // Some operations using roomView...
  // When someFunction terminates, cleanUpFd is automatically called.
}
```

> The prompt I used for ChatGPT was "does the go programming language have destructors or finalizers". It then responded by saying they have neither and proceeded to explain the use of and an example of `defer`.

Question 3

Question 3a

The language likely uses **pass by reference** semantics. `x` and `y` are modified in `main`'s scope after the call to `f(x,y)`, meaning it cannot be pass by value. `f` reassigns `x` and `y` in its body with new values, and since this change is visible in the caller (`main`), it cannot be pass by object reference either. There is also no evidence that passing by name is happening.

Question 3b

The language could be using either **pass by value** or **pass by object reference** semantics. The former is possible because copies could be made for the `x` and `y` arguments of `f`, meaning any mutation on them will not affect the `x` and `y` in `main`. Pass by object reference is also possible. The `x` and `y` in `f` may start out as the same reference to `main`'s version, but then `f` reassigns the names `x` and `y`, overwriting its local reference and thus no longer referencing `main`'s version.

Question 3c

If the language used pass by value semantics, the code would output `2`. This is because a copy of the `x` instance would be made when calling `func`. The line `x.x = 5` is then modifying its local copy, so the global `x` is unchanged.

If the language used pass by object reference semantics, the code would output `5`. This is because the `x` in `func` will reference the same `x` instance in the global scope, so the line `x.x = 5` would modify that instance.

Question 4

Lines 1 and 2 are **casts** while lines 3 to 5 are **conversions**. We can determine this by comparing the memory usage compared to the `cout << a;` assembly. The first two are not using any new memory, only the existing data at the stack position representing the variable `a` (at `-4(%rbp)`). The next three use extra memory to change the type of the data. The `short` conversion requires an operation on the `%eax` register before `mov`ing it to `%esi`. Similarly, the `bool` one uses `%al`, and the `float` one uses `%xmm0`.