hw

1.

a)

Here `refCount` should be of type `int *` since we need this data to be shared across all instances, thus we should only hold an address that can access to update shared information.

b)

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

c)

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

d)

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

e)

```
my_shared_ptr &operator=(const my_shared_ptr &obj) {
    // don't do anything if we reassign to ourself
    if (&obj == this) {
        return *this;
    }
    // destruct the self
    this->~my_shared_ptr();

    // reinitialize self with the passed in obj, increment reference count
    this->ptr = obj.ptr;
    this->refCount = obj.refCount;
    (*this->refCount)++;
    return *this;
}
```

2.

a)

We don't want to use a garbage collected language because:

a) If we use GC via reference counting, we must forfeit any (potentially necessary) performance benefits from multithreading as multithreading can be very problematic when using reference counting (data races in the reference count). b) If we use mark and compact/sweep, we could be subject to long, expensive calls to the GC that will cause our runtime to not perform as quick as necessary and be potentially unpredictable.

b)

I think Ava is mostly right because reference counting is quicker and more predictable, though like I said earlier we would not be able to take advantage of multithreading, and additionally we could suffer memory leaks if our asteroid detecting program has cyclical references that would not be freed by our GC.

c)

I would say that he should use C#, since mark-and-compact GC is less prone to fragmentation since we regroup each block of memory into a contiguous segment, rather than mark-and-sweep which just clears individual chunks, leaving gaps in memory.

d)

Since Go destructors may not run long after an object is unreachable (and may not run at all!) I would advise Yvonne to implement her own manual socket cleanup method that doesn't rely on a finalizer/destructor to run.

3.

a)

Here the parameters are possibly passed by most likely passed by reference, since there is no discernable boxing (not pass by object reference) or dereferencing (not pass by pointer) and yet the variables are modified after being passed into the function (not pass by value).

b)

Here we may now be using pass by value or pass by object reference, or pass by pointer (assigning x or y could just perform pointer arithmetic on a copy of the address which would be why our variables aren't mutated). This could not be pass by reference because if this was the case our variables would be mutated.

c)

If it was pass by value, it would print `2`, otherwise it would print `5`. This is because pass by value entails creating a copy of the entire instance of the `X` class before passing it as a parameter to the function. Pass by object reference means passing a copy of a pointer to the object, which still points to the same memory and thus is able to change the boxed field `x`.

4.

Based on this assembly code it looks like conversions are being applied on lines 3, 4, and 5. This is because the

other assembly segments highlighted match the original base case (besides the call to `...basic_ostream...` being a bit different in the unsigned int case to account for the type specifications) while in the other sections:

- the `boolean` section implements additional code that converts the value to a `1` if the expression evalutes to `true` or `0` otherwise
- the `short` section implements a `movswl` instruction which (narrowingly) converts the data to a `short`
- the `float` section implements a `cvtsi2ssl` instruction to convert the bits to floating point representation before calling the `...basic_ostream...` instruction, specifying the input as a float with the `f` suffix