Homework 5 Solutions

This homework covers the following topics from class:

- Data palooza, part 3
  - Scoping Strategies (Lexical vs. Dynamic), Garbage Collection, Ownership Model, Object Destruction/Finalization, Memory Safety, Mutability
- Data-Function Palooza, Function Palooza part 1
  - Variable Binding Semantics and Parameter Passing (Value, Reference, Object Reference, Name/Need, Pass by Value Result, Pass by Macro Expansion)

DATA4: Scope, Lifetime, Shadowing (19 min)

Part A: (5 min.) Consider this Python code:

```python
num_boops = 0
def boop(name):
  global num_boops
  num_boops = num_boops + 1
  res = {"name": name, "booped": True, "order": num_boops }
  return res

print(boop("Arjun"))
print(boop("Sharvani"))
```

For `name`, `res`, and the object bound to `res`, explain:

- What is their scope?
- What is their lifetime?
- Are they the same/different, and why?

Solution:

For `name`: the scope and the lifetime of the variable itself are the same (starting from the parameter name, and ending at the return statement). This is not the same as the string that is bound to `name`, which has a lifetime that exists before, during, and after the function call.

For `res`: the scope and the lifetime are the same (starting from the declaration, and ending at the return statement). Note that, even though the object's lifetime persists after the return statement, the identifier/variable `res` doesn't!

For the object bound to `res`: the lifetime starts within the boop function, but extends until the end of the program – since we then print the result! Note that this is *different* from the lifetime of `res`. For values, we don't really have a concept of a scope.

Part B: (3 min.) Consider this C++ code:

```cpp
int* n;
{
  int x = 42;
  n = &x;
}
std::cout << *n;
```

This is undefined behavior. In the language of scoping and lifetimes, why would that be the case?

Solution:

C++ is lexically scoped, so the scope of the variable `x` ends with the closing brace. In addition, for local variables in C++, the lifetime and scope of variables is the same! So, after the program reaches the closing brace, the lifetime of `x` – and, the memory it refers to – is over.

Since n points to the same memory address as x, after the closing brace we're accessing a value whose lifetime already ended. Essentially, this is a dangling pointer! This demonstrates one way in which C/C++ are weakly typed.

Aside: Rust's Reference and Borrow system (the "borrow-checker") prevents errors of this type! This lets Rust provide "compile-time memory safety".

Part C: (4 min.) It turns out, this code tends to work and print out the expected value of 42 – even though it is undefined behavior. Why might that be the case?

Hint: This has to do with *something else* covered in data palooza!

Solution:

This has to do with C++'s memory management model and optimizations. Broadly (and in most programming languages), when a variable's scope and lifetime ends, the memory *holding* its value isn't zeroed out. Instead, it just stays there! Usually, it'll only get changed once that piece of memory is "given" to another variable. On top of that, there's no garbage collector, so memory isn't constantly being reallocated. And, we haven't made another addition to the stack. So, it's likely that the memory that holds 42 still does hold the same bits when we print it.

This is still undefined behavior, and you shouldn't write code like this!

Part D: (2 min.) This block of code from a mystery language compiles and outputs properly.

```
fn main() {
 let x = 0;
 {
   let x = 1;
05:    println!(x);  // prints 1
06:   }
07:
08:   println!(x);    // prints 0
09:
10:   let x = "Mystery Language";
11:   println!(x);    // prints 'Mystery Language'
12: }
```

We'll note that even though it doesn't look like it, this language is statically typed. With that in mind, what can you say about its variable scoping strategies? How is it similar or different to other languages you've used?

Hint: The scope of the integer variable x, defined by let x = 0; on line 2 is limited to lines 2, 3, 7, 8 and 9.

Solution:

This is Rust; the example is almost directly copied from the Wikipedia page on shadowing.

This language is lexically scoped: if it wasn't, the second print statement would also print "1". It seems to lexically scope based on blocks, similar to C++ (see part c).

However, the really interesting part happens on line 10: we somehow are allowed to redefine a variable with the same identifier, *and it has a different type!* In a statically typed language, this is only really possible if we treat them as different variables. The logical conclusion of this is that Rust must allow variable shadowing *in the same scope* (a relatively unique feature); we can confirm this by reading the docs on shadowing.

So the definition of x on line 10 is actually creating a new variable x of type `String`, which hides the old variable x of type `Int` defined on line 2. Both variables still exist

on lines 10/11, but only the most recently defined x is accessible (the original x is shadowed and no longer accessible!). Note: This is super unusual - most of the time when we see the ability to "change a type" of a variable, we're dealing with a dynamically-typed language.

Part E: (5 min.) Here's an (adapted) example from a lesson that the TA taught a couple of years ago on a "quirky" language:

```
function boop() {
  if (true) {
    var x = 2;
    beep();
  }
  console.log(x);
}
function beep() {
  x = 1;
}
boop();
console.log(x);
// this prints:
// 2
// Error: x is not defined
```

Briefly explain the scoping strategy this language seems to use. Is it similar to other languages you've used, or different?

Hint: Try writing the same function in C++.

Solution:

This answer is as instructive as we can make it; a more concise answer would get full marks on a test!

This is JavaScript. First, we can immediately note that this language seems to use lexical scoping. If it was dynamically scoped, the first `console.log(x)` would print 1, since that's the last place (in the program) that `x` was set. In a dynamically scoped language like bash, this would have logged 1. Thus, we can infer that this language scopes variables to some lexical construct (a function, a block, or something else).

What's curious is that the `console.log` in boop prints 2, and not an undefined variable error. Languages like C++ (and most mainstream languages) lexically scope to blocks (e.g., if-else, while, for-loop blocks). Since the declaration is not in the same block as the print statement, this would be an error in C++. In contrast, JavaScript's `var` keyword scopes to functions. This behaves similarly to Python (try it!). For full marks on a test, we'd expect you to note this.

An aside: JavaScript is one of the only mainstream languages that has *different scoping rules* depending on what keyword you use. If you're curious, check out the MDN docs on `let` (the other general declaration).

DATA5: Smart Pointers (15 min)

We learned in class that C++ doesn't have garbage collection. But it does have a concept called smart pointers which provides key memory management functionality. A smart pointer is an object (via a C++ class) that holds a regular C++ pointer (e.g., to a dynamically allocated value), as well as a reference count. The reference count tracks how many different copies of the smart pointer have been made:

- Each time a smart pointer is constructed, it starts with a reference count of 1.

- Each time a smart pointer is copied (e.g., passed to a function by value), it increases its reference count, which is shared by all of its copies.
- Each time a smart pointer is destructed, it decrements the shared reference count.

When the reference count reaches zero, it means that no part of your program is using the smart pointer, and the value it points to may be "deleted." You can read more about smart pointers in the Smart Pointer section of our Data Palooza slides. In this problem, you will be creating your own smart-pointer class in C++!

Concretely, we want our code to look something like this

```
auto ptr1 = new int[100];
auto ptr2 = new int[200];
my_shared_ptr m(ptr1); // should create a new shared_ptr for ptr1
my_shared_ptr n(ptr2); // should create a new shared_ptr for ptr2
n = m; // ptr2 should be deleted, and there should be 2
shared_ptr pointing to ptr1
```

We want our shared pointer to automatically delete the memory pointed to by its pointer once the last copy of the smart pointer is destructed. For this, we need our shared pointer class to contain two members, one that stores the pointer to the object, and another that stores a reference count. The reference count stores how many pointers currently point to the object.

You are given the following boilerplate code:

```
class my_shared_ptr
```

```
{
private:
  int * ptr = nullptr;
  ______ refCount = nullptr; // a)

public:
  // b) constructor
  my_shared_ptr(int * ptr)
  {
  }

  // c) copy constructor
  my_shared_ptr(const my_shared_ptr & other)
  {
  }

  // d) destructor
  ~my_shared_ptr()
  {
  }

  // e) copy assignment
  my_shared_ptr& operator=(const my_shared_ptr & obj)
  {
  }
};
```

Part A: (4 min.) The type of `refCount` cannot be `int` since we want the counter to be shared across all `shared_ptr`s that point to the same object. What should the type of `refCount` be in the declaration and why?

Solution:

`refCount` should be a pointer to an `int` (or equivalent type like `unsigned int`). This is because the `refCount` must be shared across multiple objects.

To see why, consider the case where there are multiple `shared_ptrs` pointing to a single object. If the `refCount` were just a simple `int`, then there would be no way for the `refCount` of other `shared_ptrs` to be updated concurrently. By having an `int` pointer, all of the `shared_ptrs` that point to the same object also share the same reference counter.

Part B: (2 min.) Fill in the code inside the constructor:

```
my_shared_ptr(int * ptr)
{


}
```

Solution:

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

Part C: (2 min.) Fill in the code inside the copy constructor:

```
my_shared_ptr(const my_shared_ptr & other)
{
}
```

Solution:

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

Part D: (2 min.) Fill in the code inside the destructor:

Hint: You only need to delete the object when the reference count hits 0.

```
~my_shared_ptr()
{









}
```

Solution:

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

Part E: (5 min.) Fill in the code inside the copy assignment operator:

```
my_shared_ptr& operator=(const my_shared_ptr & obj)
{













}
```

Solution:

```
my_shared_ptr& operator=(const my_shared_ptr & obj)
{
  if (this == &other)
    return *this;

  (*refCount)--;
  if (*refCount == 0)
  {
    if (nullptr != ptr)
      delete ptr;
    delete refCount;
  }

  // Assign incoming object's data to this object
  this->ptr = obj.ptr; // share the underlying pointer
  this->refCount = obj.refCount;
  // if the pointer is not null, increment the refCount
  (*this->refCount)++;

  return *this;
}
```

DATA6: Garbage Collection (20 min)

These questions test you on concepts involving memory models and garbage collection. These are similar to interview questions you may get about programming languages!

Part A: (5 min.) Rucha and Ava work for SNASA on a space probe that needs to avoid collisions from incoming asteroids and meteors in a very short time frame (let's say, < 100 ms).

They're trying to figure out what programming language to use. Rucha thinks that using C, C++, or Rust is a better idea because they don't have garbage collection.

Finish Rucha's argument: why would you not want to use a language with garbage collection in a space probe?

Solution:

This is a more fleshed-out argument of what's covered in Data Palooza. This answer is more instructive than what we'd expect on an exam.

This SNASA probe is a mission-critical, real-time software system. In the problem, we've imposed that the collision-avoidance routine needs to always run in a specific time frame. In these situations, we want totally predictable or deterministic behavior: no matter how many times we run the same code, the same thing should always happen, in the same amount of time.

Garbage collection is non-deterministic/unpredictable behavior: at any point in the program, memory pressure or a clock-cycle GC algorithm could pause execution and perform garbage collection. This could make the collision avoidance routine take longer than 100ms, and would lead to a very, very expensive mistake. So, we would want to avoid this behavior, and use a language without garbage collection. With manual memory management, we know that manual allocation/deallocation happens in a relatively constant time frame.

Part B: (5 min.) Ava disagrees and says that Rucha's concerns can be fixed with a language that uses reference counting instead of a mark-and-* collector, like Swift. Do you agree? Why or why not?

Fun fact: NASA has very aggressive rules on how you're allowed to use memory management. `malloc` is basically banned!

Solution:

No, we disagree. Even though reference counting is a different flavor of garbage collection from mark-and-sweep and mark-and-compact, it still is garbage collection. The reclamation of memory can trigger side effects, which still pause execution.

To cite one example, if an object's reference count goes to zero, and the object refers to thousands or millions of other objects (e.g., in a dictionary it holds), that will cause those objects' reference counts to also go to zero. Thus garbage collection of a single object could cascade into millions of collected objects, taking milliseconds or more.

Part C: (5 min.) Kevin is writing some systems software for a GPS. He has to frequently allocate and deallocate arrays of lat and long coordinates. Each pair of coordinates is a fixed-size tuple, but the number of coordinates is variable (you can think of them as random).

Here's some C++-like pseudocode:

```
struct Coord {
  float lat;
  float lng;
};

function frequentlyCalledFunc(count) {
  Array[Coord] coords = new Array[Coord](count);
}
```

He’s trying to decide between using C# (has a mark-and-compact GC) and Go (has a mark-and-sweep GC). What advice would you give him?

Solution:

There are two key insights for this question:

1. **Arrays in most languages are contiguous memory blocks**
2. **Since we’re rapidly allocating and deallocating random-length arrays, we’ll get “holes” of random sizes in memory, of different sizes: this leads to memory fragmentation.**

As soon as you see memory fragmentation, you should immediately jump to the core goal of mark-and-compact: avoiding memory fragmentation. With mark-and-compact, the compaction shifts all the allocated blocks to be one contiguous space, removing the holes (temporarily). So, Kevin may want to use C#!

Aside: if allocation/deallocation is super frequent (could cause thrashing), a non-GC language may be more appropriate!

Part D: (5 min.) Yvonne works on a messaging app, where users can join and leave many rooms at once.

The original version of the app was written in C++. The C++ code for a `Room` looks like this:

```
class Socket { /* ... */ };
class RoomView {
  RoomView() {
    this.socket = new Socket();
  }
  ~RoomView() {
    this.socket.cleanupFd();
  }
  // ...
};
```

Recently, the company has moved its backend to Go, and Yvonne is tasked with implementing the code to leave a room.

When Yvonne tests her Go version on her brand-new M3 Macbook, she finds that the app quickly runs out of sockets (and socket file descriptors)! She's confused: this was never a problem with the old codebase, and there are no compile or runtime errors. Give one possible explanation of the problem she's running into, and what she could do to solve it.

Hint: You may wish to Google a bit about Go and destructors, and when they run to help you solve this problem.

Solution:

There are three key insights for this question:

1. **C++ uses destructors; destructors always run at the end of object lifetimes**
2. **Go uses finalizers; finalizers are only run at garbage collection, which is not deterministic! Aside: if this was a test, we'd tell you this information.**

3. **The RoomView class only frees up its resource (the socket file descriptor) in its destructor. Implicitly, Yvonne's Go version would only call it in the finalizer.**

Since finalizers don't always run, Yvonne's cleanup code could never run (and in this example, that's what's happening)! This is particularly likely when her machine has a ton of memory, and doesn't need to run GC frequently (or at all).

There are many ways she could solve this. The most common would be for Yvonne to explicitly call the cleanup code (this.socket.cleanupFd) manually, before the object is removed from memory. This solution is language-agnostic, and what we'd expect on a midterm.

If you're curious, there are Go-specific ways to resolve this problem. We do not expect you to know this!

1. **Go allows you to run the GC manually; see the runtime package**
2. **A soft convention is to start a goroutine to run cleanup**

DATA7: Binding Semantics (5 min)

(5 min.) With reference to variable binding semantics, examine the behavior of this language.

In particular, comment on the differences between n1 == n2 and s1 == s2. Is this similar to other languages you've used?

```
n1 = 3
n2 = 3
n1.object_id
```

```
# 7
n2.object_id
# 7
puts n1 == n2
# true
# ...
s1 = "hello"
s2 = "hello"
puts s1.object_id
# 25700
puts s2.object_id
# 32880
puts s1 == s2
# true
s2 = s1
puts s2.object_id
# 25700
```

Note: In this language, object_id is an attribute of all objects that is used to uniquely identify that object from other objects. You may assume that every distinct object has a unique object_id, and can think of this as being analogous to the address of the object in memory (thus two distinct objects at different locations in memory will have different IDs)..

Solution:
This is Ruby.

The observations we're looking for you to make are:

- **for the "number/int" type, it seems like the same literal value always has the same object ID – a shared reference**
    - **this is not enough information to infer that *all* numbers behave like this. When you see this, it's usually safe to assume that this works for "small enough" numbers, i.e. a 32-bit or 64-bit int.**

    - **in Ruby, this isn't true for arbitrarily-large numbers; try: 999999999999999999999999999999999999999999999**
- **but, for the "string" type, it seems like the same literal value can have different object IDs – not a shared reference**
- **yet, the equality operator seems to account for this:**
    - **for the "number/int" type, it could be either performing a value comparison or an ID comparison - you can't tell from this example**
    - **but, for the "string" type, it's definitely performing a value comparison - the strings are equal, even though their object IDs aren't!**
- **you could then infer that Ruby probably uses object reference semantics (which it does); however, technically this example isn't enough to confirm or deny that**

DATA8: Binding Semantics/Parameter Passing (14 min)

You are given the following piece of code from a mystery language:

```
void main()
{
  int x = 2;
  int y = 2;

  f(x,y);

  print(x); // outputs 16
  print(y); // outputs 0

}

void f(x, y)
{
  if (y <= 0)
    return;

  x = x * x;
  y = y - 1;
  f(x,y);
}
```

Part A: (4 min.) What is/are the possible parameter passing convention(s) used by this language? Why?

Solution:

The language has to follow a pass by reference convention. Notice how the values of the variables x and y change when you pass them to the function f. In general, if you find that the value of a variable changes once you pass it to a function, the language uses pass by reference semantics.

Part B: (5 min.) Now let’s assume that print(x) and print(y) both output 2. What types of parameter passing conventions might the language be using? Why?

Solution:

The language can be using either pass by value or pass by object reference semantics. It’s straightforward to see why it could be pass by value: by definition, the values of x and y in main cannot be modified through the invocation of f. The argument for why it could be pass by object reference is a little more tricky. The key insight is to notice that when we perform an assignment (like y = 3) in a language with object reference parameter passing semantics, we are creating a new object with value 3 and making y point to that. Thus, the statements assigning to x and y in f would not change what the x and y variables in main point to.

Part C: (5 min.) Consider the following snippet of code:

```
class X:
  def __init__(self):
    self.x = 2
x = X()
def func(x):
  x.x = 5

f(x)
print(x.x)
```

If this language used pass by value semantics, what would the code output? What about if it used pass by object reference semantics? Justify your answers.

Solution:

If the language used pass by value semantics, essentially a new copy of the class would be passed to the function, so any changes made by the function would not be visible once the scope of the function ends. So the print statement just outputs 2.

If the language used pass by object reference semantics, then a reference to the object would be passed to the function. In line 3, when the function assigns to `x.x`, we are modifying the original object. Therefore, the print statement outputs 5.