Homework 5

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)***

*** Binding semantics and parameter passing will be covered on Monday. You should have enough knowledge to attempt the last two problems that cover this topic with the help of slides and online resources like ChatGPT.**

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and hand-written solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work.

For homework #5, you must complete the following **required** problems:

- DATA4: Scope, Lifetime, Shadowing (19 min)
- DATA5 (a-d): Smart Pointers (10 min)
- DATA6: Garbage Collection (20 min)
- DATA7: Binding Semantics (5 min)
- DATA8: Binding Semantics/Parameter Passing (14 min)

For homework #5, you may optionally complete the following additional problems:

- DATA5 (e)

DATA4: Scope, Lifetime, Shadowing (19 min)

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

```
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?

Your answer:

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

```
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?

Your answer:

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!

Your answer:

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

```
01: fn main() {
02:  let x = 0;
03:  {
04:    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, 9 and 9.

Your answer:

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++.

Your answer:

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_ptrs` that point to the same object. What should the type of `refCount` be in the declaration and why?

Your answer:

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

```
my_shared_ptr(int * ptr)
{


}
```

Your answer:

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

```
my_shared_ptr(const my_shared_ptr & other)
{




}
```

Your answer:

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()
{




}
```

```
}
```

Your answer:

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

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









}
```

Your answer:

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?

Your answer:

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!

Your answer:

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?

Your answer:

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.

Your answer:

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)..

Your answer:

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?

Your answer:

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?

Your answer:

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.

Your answer: