**Homework 5 – Spring 2023** (Due: May 10, 2023)

In this homework, you'll explore more concepts from data palooza: garbage collection, parameter passing semantics, and type conversions. Some questions have multiple, distinct answers which would be acceptable, so there might not be a "right" answer: what's important is your ability to justify your answer with clear and concise reasoning that utilizes the appropriate terminology discussed in class. Each question has a time estimate; you'll know you're ready for the exam when you can solve them roughly within their time constraints.

We understand, however, that as you learn these questions may take more time. For that reason, only **starred red** questions need to be completed when you submit this homework (the rest should be used as exam prep materials). Note that for some multi-part questions, not all parts are marked as red so you may skip unstarred subparts in this homework.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and handwritten 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.

1. ** 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)
  {
  }
};
```

**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?

**`refCount` should be a <u>pointer</u> 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_ptr`s 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_ptr`s to be updated concurrently. By having an `int` pointer, all of the `shared_ptr`s that point to the same object also share the same reference counter.**

**b)** ** (2 min.) Fill in the code inside the constructor:

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

**c)** ** (2 min.) Fill in the code inside the copy constructor:

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

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

**e)** ** (5 min.) Fill in the code inside the copy assignment operator:

```
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;
}
```

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

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?

**This is a more fleshed-out argument of what's on Slide 128 of 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.**

**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!

**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.**

**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?

**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!**

**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.

**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!**
    a. **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**

3. ** 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);
}
```

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

**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.**

**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?

**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.**

**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.

**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.**

**4.** ** (5 min.) This question is about distinguishing casts from conversions. As we learned in class, sometimes it's not so easy to figure out when a language is using a cast vs a conversion. That is, unless you actually look at its assembly output.

Consider the following program:

```
int main() {
  int a = 5;
  cout << a;
  cout << (const int) a;          // Line 1
  cout << (unsigned int) a;        // Line 2
  cout << (short) a;               // Line 3
  cout << (bool) a;                // Line 4
  cout << (float) a;               // Line 5
```

```
}
```

When this program is compiled with g++'s `-S` option (`g++ -S foo.cpp`), g++ produces the following assembly language output:

```
_main:                                  ## main()
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
# int a = 5;
  movl  $5, -4(%rbp)     # a is stored on the stack at [rbp-4]
######################
# cout << a;
  movl  -4(%rbp), %esi
  movq  __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
  callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEElsEi
######################
# cout << (const int)a;
  movl  -4(%rbp), %esi
  movq  __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
  callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEElsEi
# cout << (unsigned int)a;
  movl  -4(%rbp), %esi
  movq  __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
  callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEElsEj
# cout << (short)a;
  movl  -4(%rbp), %eax
  movq  __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
  movswl  %ax, %esi
  callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEElsEs
# cout << (bool)a;
  cmpl  $0, -4(%rbp)
  setne %al
  movzbl  %al, %esi
  andl  $1, %esi
  movq  __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
  callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEElsEb
# cout << (float)a;
  cvtsi2ssl -4(%rbp), %xmm0
  movq  __ZNSt3__14coutE@GOTPCREL(%rip), %rdi
  callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEElsEf
```

```
# end of main()
  xorl  %eax, %eax
  addq  $16, %rsp
  popq  %rbp
  retq
  .cfi_endproc
```

Based on this assembly language output, which of the lines (1-5) above are using casts and which are using conversions? If conversions are used, how is the C++ compiler performing the conversion?

**Hint:** You don't need to be an expert at assembly language to solve this problem. Simply compare the code generated for the `cout << a;` statement (which we have delineated using a plethora of octothorpes) to the other versions to look for differences. Also, try googling instructions (like `cvtsi2ssl`).

**Notice that for `(const int)`, `(unsigned int)` and `(short)`, the compiler only generates `movl` instructions. This means it's essentially just copying the bits from one location to another. This means it is just performing a cast.**

**For `(bool)` and `(float)`, the compiler generates doesn't generate simple `movl` instructions, but instead generates code to convert the value (that's what the `cvtsi2ssl` (convert signed integer to scalar single-precision) instruction does).**