Homework 5 – Fall 2023

In this homework, you'll explore more concepts from data palooza: garbage collection, typing, scoping, 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.

Classify That Language (Late Night Edition)

We're such huge fans of "Classify That Language", so we've brought you a special episode in this homework.

1. **

    a) ** (2 min.) Consider the following code snippet from a language a former TA worked with in his summer internship:

```
user_id = get_most_followed_id(group) # returns a string
user_id = user_id.to_i
# IDs are zero-indexed, so we need to add 1
user_id += 1
puts "Congrats User No. #{user_id}, you're the most followed
user in group #{group}!"
```

    Without knowing the language, is the language dynamically or statically typed? Why?

    b) ** (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++.

c) ** (2 min.) This block of code from a mystery language compiles and outputs properly.

```
fn main() {
  let x = 0;
  {
    let x = 1;
    println!(x);  // prints 1
  }

  println!(x);    // prints 0

  let x = "Mystery Language";
  println!(x);    // prints 'Mystery Language'
}
```

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 `let x = 0;` is only lines 2, 7, 8, and 9.

How to Save a Lifetime

2. **

   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?

   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?

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!

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

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

```
my_shared_ptr(int * ptr)
{



}
```

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

```
my_shared_ptr(const my_shared_ptr & other)
{




}
```

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()
{




}
```

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

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









}
```

4. ** 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?

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!

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?

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.

5. ** (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`).