Homework 5

Tejas Kamtam

305749402

CS 131 - Fall 2023

Problem 1

Part a

The language is **dynamically typed** because we are able to assign both strings and integers to the same variable.

Part b

The language uses **lexical scoping** because the variable `x` remains unchanged regardless of the called function `beep`. However, the context is not drawn around `if` statements as the console can access the variable `x` even if it is defined in the `if` statement as long as it's in the same function context.

Part c

It is lexically scoped. Similar to C it is scoped by blocks, however these blocks can be arbitrary and don't need a definition

Problem 2

Part a

`name`: the scope and the lifetime of the variable itself are the same (
from the param name to the return statement).
`res`: the scope and the lifetime are the same (from the definition to the return statement).
different from the lifetime of res. For values, we don't really have a
concept of a scope.
`object bound to res`: the lifetime starts within the boop function, but
extends until the end of the program

Part b

C++ is lexically scoped, so the scope and lifetime of the variable x ends with the closing
brace.

Part c

This is because although the scope and lifetime ends, it has not been garbage collected yet, the value is still in memory.

Problem 3

Part a

`refCount` should be an `int pointer` to be shared across objects.

Part b

```
my_shared_ptr(int * ptr)
{
  this→ptr = ptr;
```

```
  refCount = new int(1);
}
```

Part c

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

Part d

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

Part e

```
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;
  }
  this->ptr = obj.ptr;
  this->refCount = obj.refCount;
  (*this->refCount)++;
  return *this;
}
```

Problem 4

Part a

Garbage collection is non-deterministic, so it is not possible to know when the garbage collector will run. This means that it is not possible to know that the collision routine will run in a timely manner.

Part b

No, reference counting is still a method ofgarbage collection. So, the same problem applies.

Part c

Mark-and-compact garbage collection avoids memory fragmentation by moving all the live objects to one side of the heap, and then freeing the other side. This means that the memory is always contiguous (temporarily).

Part d

Finalizers don't always run in a lanugage like Go. But, Yvonne could fix this by explicitly calling the finalizer.

Problem 5

For `const int, unsigned int, short`, the compiler is performing casts because it is simply copying the memory. For `bool,float`, the compiler is calling a function to convert the value.