Soyeon Kim, 106292803

Homework 5

CS 131, Fall 2024
Carey Nachenberg

DATA4: Scope, Lifetime, Shadowing

Part A

- `name`
  - scope: within the `boop` function
  - lifetime: during the `boop` function call
- `res`
  - scope: within the `boop` function
  - lifetime: during the `boop` function call
- the object bound to `res`
  - scope: the caller of the `boop` function
  - lifetime: until the `print` statement is executed
- `name` and `res` are temporary variables scoped to the `boop` function and their lifetime ends when the function is returned. However, the object bound to `res` lives beyond the function call until the `print` function is called.

Part B

Once the variable `x` is out of its scope `{}`, its memory is released. So when we try to access it out of the scope, the memory may have been reallocated or invalidated.

Part C

In C++, memory for local variables like `x` is freed when the variable goes out of scope, but the actual value in that memory isn't immediately overwritten. Therefore, the value of `x` may remain in memory right after its scope ends, and dereferencing `n` might appear to work.

Part D

The language uses shadow scoping. When a new variable is declared with the same name, it hides the previous variable. It finds the closest variable when it's called. Other languages that do not have shadow scoping will create an error in line 10 since the variable `x` is already declared within the same scope.

Part E

The language uses dynamic scoping within the function level. It is different from other languages I've used. In C++, it will create an error in line `x = 1;` since the variable `x` is not declared within the `boop` function, and the line `console.log(x);` will also create an error because `var x` is not declared within the scope.

Soyeon Kim, 106292803

DATA5: Smart Pointers

Part A

It should be an int pointer (`int*`) because if it is int type, then it will create its own `refCount` for every instance. We want to share `refCount` for all instances that share the same object.

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) {
    delete ptr;
    delete refCount;
  }
}
```

Part E

```
my_shared_ptr& operator=(const my_shared_ptr & obj) {
  if (this != &obj) {
    (*refCount)--;
    if (*refCount == 0) {
      delete ptr;
      delete refCount;
    }
    ptr = obj.ptr;
    refCount = obj.refCount;
    (*refCount)++;
  }
  return *this;
}
```

Soyeon Kim, 106292803

DATA6: Garbage Collection

Part A

You would not want to use a language with garbage collection in a space probe because garbage collection can introduce unpredictable pauses in execution. The garbage collector typically runs automatically when there is memory pressure, which may occur at any time and without control over its timing. These pauses can delay critical operations in a time-sensitive environment.

Part B

Using reference counting language would be better than garbage collecting language. However, in time-sensitive environments, reference counting may still cause minor overhead because the reference count has to be constantly updated every time an object is assigned and deallocated.

Part C

I believe C# would be better than Go because frequent allocation and deallocation would cause a lot of memory fragmentation, and mark-and-compact is good for reducing it.

Part D

Go uses garbage collection. It does not immediately free up resources, so sockets could remain open. Furthermore, with a high-performance CPU, the garbage collector may not run frequently. In this case, Yvonne can create a finalizer and call it explicitly whenever the socket has to be released.

DATA7: Binding Semantics

The language seems `==` operator compares the value, and the same values of basic types are cached to optimize memory and share the same memory location. This is similar to Python.

DATA8: Binding Semantics/Parameter Passing

Part A

It uses pass-by object reference because the values of `x` and `y` in the `main` function are modified within function `f`.

Part B

It uses pass-by-value because it means the values of `x` and `y` in the `main` function are not modified within function `f`. It creates a copy of variables when it's passed to a function.

Part C

- Pass by value: 5
  - A copy of `x` will be created, however, `X` is a mutable object, so its value can be modified within the `func` function.

- Pass by object reference: 5
  - `x.x` value gets modified within the `func` function.