Trevor Smith
CS 354-1
March 27, 2020

TA2

3.2) The differences of how variables are allocated depends on the design of the programming language. Languages like Java or C# have garbage collection so they can allocate to the heap without seeing many issues with memory leaks. Languages like C, however only allocate to the heap when needed as it requires manual allocation and deallocation. Languages like Scheme are also built for recursion, so it makes more sense for the allocation to be static.

An example of a Pascal program that does not work with a statically allocated variable would be the overload.p program in the pub/ch3 folder, since fact is called within itself.

An example of a Scheme program that wouldn't work with local variable allocation to the stack would be the tsar.scm program that we wrote for LA1. Since this program uses recursion, it would not work well being allocated to the stack.

3.4) One example would be in a Java program where you have some integer variable named i, and then in a for loop you create another "I" variable to use as an iterator. The first i would still be live, but would not be within the scope of the for loop.

Another example is where in a Java program you could have a variable that is passed through to a method. Then within that method you can have another variable with the same name, and the method would use that local variable. Therefore, the original variable is still live but not used within scope of the method.

In a C program you can have a subroutine that is called which creates a struct. This struct would then be alive in memory but no within scope of the main program

3.5) C: 113112
Modula-3: 333312

3.7) When reversing the list, the new list (reversed) points to L and there is now nothing pointing to the old list. This memory leak causes the heap to fill after some iterations of main.

The output is corrupted because when “list_node* T = reverse(L);” is called, it is making a reference to the same spot that delete_list will be freeing. This causes a dangling reference where the reference previously made by the pointer no longer exists, therefore causing the references to be corrupted and corrupting the output

3.14) Static scoping: the output would be 1122. This is because each time x is set, it is being set in the set( ) method which assigns the parameter value to the global x, making the x variable in second( ) obsolete. Each print that is called is printing the value of the global x variable.

Dynamic scoping: the output would be 1121. Here, 11 is printed in the same way as before, but the difference is when second( ) is called. When set(2) is called within second( ), it ends up setting the local variable within second( ) since that was the most recent x variable as it was just recently created. Print(x), within second( ), then prints the local variable. Then when print( ) is called in the main function, the global variable x is printed, which is still 1 from setting it after calling first( ), so 1 is printed.

3.18) Shallow binding: 10 20 30 40
This is the output because each time set(0) is called, it’s setting the global x, which is then printed after the foo( ) function prints.

Deep binding: 10 52 33 44
The output for deep binding is this way because the variable x is bound by the call to set( ) within foo( ), since deep binding is done at the time of the function call versus time of function definition. So each time foo is called, x is reset by whichever set( ) call it enters, depending on what the n parameter is.