Jaime Guevara

CS-354 Sec. 1: TA2

October 23, 2018

* 1. The differences in how these variables are allocated depends on how the language is meant to work, and how it’s interpreted. For example, in Pascal the variable allocation is on the stack (as it’s an Algol descendant) and therefore. On the other hand, Lisp-derived PLs such as Scheme are meant to aid more with recursion and therefore it would make little sense to have its allocation be static (it’d be better to use a Stack or Heap and be unbound).

An example of a Pascal program not working (if statically allocated) would be if you were to have a program that constantly calls on the same variable and makes changes to it (like the example in /pub/ch3/overload.p where it’s constantly calling on fact), because if that variable where to be statically allocated it would not be allowed to call on itself as it does, even more so when it depends on variables that may not be known until runtime (again as is the case with the aforementioned example).

On the other hand, and example of a Scheme program that would not function correctly (if allocation occurred on the stack) would be our own super-duper.scm. Since the heap is best set for lists, sets, and general character strings, only using a stack would hinder the use of our program, as it very much depends on at least one of the three previous data structures.

* 1. I) In C++ you could have a variable that is initially set to a specific value, and further down the line (in the same scope) you could create a block closed off by brackets with an identical variable set to a different value. Though the original variable is still live, while within the sub-block of code its scope will be cut off until the sub-block returns to the original scope.

II) In PHP you could have a session variable that is set either at the beginning of the page, or whenever a certain action is called, but until another function decides to use this session variable it will remain live though not necessarily in scope.

III) For Java you could have a variable pass a local scope variable to a function in another class. In which case, much like the previous two examples, the original local value will remain live but out of scope of the called upon function.

* 1. The first print (located in inner()) will print out the b from middle() and the a from main(), the second print will print out the its own b (same as the previous b) and main’s a (not inner()’s because middle() can only look up in scope to main() and not down into inner()), the last print will print out both of its own a and b values(i.e. print #1: 1,1; print #2: 1,1; print #3: 1,2). This because as previously mentioned, in a language like C, children can inherit parent variables but parents can’t access child variables.

However, if we were to apply this to C#, while I do not have any personal experience with it, just looking at the scope rules for it I’d assume that it’d throw a syntactical error before the first print since the lifetime and scope of C# variables extend until it hits the end. Therefore declaring another variable of the same name (i.e. “b”) will conflict with the first declaration of b in main().

Looking at Modula-3, and information on it, a program written in it should have the same structure as that of a C-program. That is to say it’d work roughly the same as the C, or at least print the same things as C in the same order. Even with its specific declaration rule Modula-3 would still need at least a value or a type to be declared, so even if we didn’t tell the middle() b and inner() a what type to be, just by passing the values we want to it should allow Module-3 to interpret what type these variables ought to be.

* 1. a) Unfortunately the way that C allocates and frees memory does not align with the way Java goes around doing it (manual vs auto-garbage-collected). Therefore, whenever the size of the list\_node is reached Brad’s code will breach the memory limit if that memory block isn’t freed and will crash the program.

b) She’ll tell him that when the program reaches the last line of his new code, it’ll pass the value of T. But this value is not what Brad expects, because T is a pointer its value is not the reversed list structure but the address to that list.

3.14) With a statically scoped program, such as C, the pseudocode would print out: 1,1,2,2.

With a dynamic program given that a variable is considered active whenever the function it’s located in is in execution. My guess would be that if we we’re to do a direct assignment, like in C, it’d probably have a hard time distinguishing which ‘x’ value to use in set\_x() as it wouldn’t consider them to be the same and the initial ‘x’ wouldn’t be live during the execution of set\_x()

3.18) An instance in which overloading may not be advantageous, over just setting a variable to be a float (and coercing integers) would be whenever you’re dealing with square roots in calculations. For example doing a quadratic formula calculation could take parameters *a, b,* and *c* and these parameters could very well be integers, that we’ll coerce into being doubles/floats.