Hunter Black

CS 3323-01

**Homework 3**

**3.7, 3.14, 3.39 (pg. 200)**

**3.7:**

**a) Question:**

Accustomed to Java, new team member Brad includes the following code in the main loop of the program:

list\_node\* L = 0;

while(more\_widgets()) {

L = insert(next\_widget(), L);

}

L = reverse(L);

Sadly, after running for a while, Brad’s program always runs out of memory and crashes. Explain what’s going wrong.

**Answer:**

Looking at the reverse function and Brad’s subsequent calling of the reverse function, his program is crashing because he is assigning the reversed list to L. This causes all nodes from the old list to be inaccessible. Thus, after iterating through the program a couple of times, the main heap fills up with inaccessible nodes, causing the program to run out of memory and crash.

**b) Question:**

After Janet patiently explains the problem to him, Brad gives it another try:

list\_node\* L = 0;

while(more\_widgets()) {

L = insert(next\_widget(), L);

}

list\_node\* T = reverse(L);

delete\_list(L);

L = T;

This seems to solve the memory problem, but where the program used to produce correct results (before running out of memory), now its output is strangely corrupted, and Brad goes back to Janet for advice. What will she tell him this time?

**Answer:**

When calling the delete\_list() function, all memory associated with the list (both nodes and associated widgets) get freed, thus leaving empty nodes in their place due to their pointers referring to areas where their associated data no longer exists. This causes the output to become corrupted.

**3.14:**

**Question:**

Consider the following pseudocode:

x : integer --global

procedure set\_x(n : integer)

x := n

procedure print\_x()

write\_integer(x)

procedure first()

set\_x(1)

print\_x(1)

procedure second()

x : integer

set\_x(2)

print\_x()

set\_x(0)

first()

print\_x()

second()

print\_x()

What does this program print if the language uses static scope? What does it print with dynamic scope? Why?

**Answer:**

Static scope: *1, 1, 2, 2*

The above is printed because when using static scope, the global variables (in this case x) exist throughout the entirety of the program, and are immediately referred to whenever a call to that variable is made and no local version of it exists. This really only comes into play when second() is called. Even though a “new” variable x is created, the global variable is referenced because when set\_x(2) is called, no local version of x exists in that procedure.

Dynamic scope: *1, 1, 2, 1*

The above is printed because when using dynamic scope, global variables are only referred to when all dynamic links are expired and no other version of the variable exists that is “closer”. For example, when second() is called, because a new x variable is created, and then set\_x(2) is called, the newly created x variable is the closest to the call to modify x in set\_x(2). Thus that is the x that is referred to.

**3.39:**

**Question:**

Do you think coercion is a good idea? Why or why not?

**Answer:**

Coercion is the automatic conversion performed by a compiler for converting a value of one type into a value of another data type. In other words, it is an *implicit* type conversion.

Overall, I believe that coercion is a good idea as it removes some of the trivial and tedious conversions that would otherwise need to be done by the programmer, and in most cases these conversions do not affect the overall program. There are of course some downsides, such as the potential loss of data if a conversion from a more accurate value is converted to a less accurate value to fit a certain procedure. This can cause errors and bugs that can be hard to track down.