Colby Frison
OUID: 113568816

CS-3323 || Homework 3

2/10/2025

3.17

As part of the development team at MumbleTech.com, Janet has written a list manipulation library for C that contains, among other things, the code in Figure 3.16.

(a) Accustomed to Java, new team member Brad includes the following code in the main loop of his 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.

(b) 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 insufficient 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?

a) Why is memory running out?

There are a few possible reasons:

- The insert function probably creates a new node to insert without deleting the old node
- The reverse function likely creates a new list in reverse without deallocating the old list leading to memory leaks
- Lastly as more widgets are added it takes more and more memory until it runs out.

b) Why is the output corrupted?

Most likely deleting the original list after reversing will alter the new list. To solve this problem he will have to change how the delete_list is handled by either modifying the function or creating a deep copy of the list before deleting. He could also take an entirely different approach to memory management such as creating a temp list of nodes before reversing.

3.14

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()

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 scoping? What does it print with dynamic scoping? Why?

i) What does this print if the language uses static scoping

```
1
1
2
2
```

set(0) → first() → set(1) → print() → print()
↳ second() → set(2) → print() → print()

ii) What about dynamic scoping

```
1
1   ← sets global x to 1
2   ← creates local x and sets it to 2
1   ← prints out GLOBAL x which is still 1
```

3.39

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

Coercion has its use-cases. On one hand it is very useful in increasing the flexibility & readability of the code. On the other hand it can make the code less predictable, and more error prone which is obviously not ideal.

Whether coercion is good is dependant on the language and context, but generally explicit type conversion is preffered due to its clarity & readability.