CSci 365: Organizations of Programming Languages

Name: Kenneth Jahnke

**Assignment 7 (chap. 9): 55 points**

Q1. [30] Consider the following program written in C syntax:

**void** swap(**int** a, **int** b) {

**int** temp;

temp = a;

a = b;

b = temp;

}

**void** main() {

**int** value = 2, list[5] = {1, 3, 5, 7, 9};

swap(value, list[0]); -- (A)

swap(list[0], list[1]); -- (B)

swap(value, list[value]); -- (C)

}

For each of the following parameter-passing methods, what are all of the values of the variables and list after each of the three calls to swap?

Assume the calls are not accumulative; that is, they are always called with the initialized values of the variables, so their effects are not accumulative.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | (A) | | (B) | | (C) | |
|  | value | list[0 ..4] | value | list[0..4] | value | list[0..4] |
| 1) Pass by Value | 2 | {1, 3, 5, 7, 9} | 2 | {1, 3, 5, 7, 9} | 2 | {1, 3, 5, 7, 9} |
| 2) Pass by Reference | 1 | {2, 3, 5, 7, 9} | 2 | {3, 1, 5, 7, 9} | 5 | {1, 3, 2, 7, 9} |
| 3) Pass by Value-Result | 1 | {2, 3, 5, 7, 9} | 2 | {3, 1, 5, 7, 9} | 5 | {1, 3, 2, 7, 9} |

Q2. [15] Consider the following program written in C syntax:

**void** fun (**int** first, **int** second) {

first += first;

second += second;

}

**void** main() {

**int** list[2] = {1, 3};

fun(list[0], list[1]);

}

For each of the following parameter-passing methods, what are the values of the list array after execution?

|  |  |
| --- | --- |
|  | list[0 ..1] |
| 1) Pass by Value | {1, 3} |
| 2) Pass by Reference | {2, 6} |
| 3) Pass by Value-Result | {2, 6} |

Q3. [10] Speculate on the issue of allowing nested subprograms in programming languages—why are they not allowed in many contemporary languages?

====BEGIN ANSWER Q3====

To start, I don’t think it’s fair to say that nested subprograms aren’t *allowed* in contemporary languages. They are *allowed*, but their use is mostly discouraged in favor of simpler program structure, and thus don’t see frequent implementation.

A few issues:

1. Complexity. Nested subprograms add more layers to the overall program. In other words, less flat means more complex.

2. Debugging. This ties into the complexity argument. With more layers to sift through should something go wrong, it becomes increasingly difficult to pinpoint the cause of the error.

3. Resource depletion. Memory will allocated to new variables should the subprograms initialize them. This is an unnecessary expense if the functionality of the subprogram can somehow be rolled into a calling program’s reference environment. Furthermore, the calling program will be paused until the subprogram completes, potentially eating up time unnecessarily.

4. Convolution. While there may be niche cases where a nested subprogram provides a “cleaner” implementation, on the whole, this is a flawed method of development. It’s likely more costly in terms of overhead, readability, and headache to create subprograms than it is to not.

5. Reference environment. An intimate knowledge of the language at hand is likely needed to avoid referencing errors for nested subprograms.

====END ANSWER Q3====