Sarah Ford

Computer Systems HW2 Part A

1. **Program 1:**

The output of this program is the value of the variable y, which is 20. The main function initializes the variable y by declaring it an integer and setting it equal to 20. Next, the main function calls the function foo (y). In a separate stack frame, the void foo (int x) function passes y by its value of 20. After declaring new local variables, a and b, set to values of 100 and 40 respectively, the function locally assigns the input variable (y) equal to the sum of a squared and four multiplied by b. This function, having run to completion, then pops off of the stack frame without having altered the actual value of y which is stored in a separate memory address linked to the variable y. Now, the main function stack frame prints out a double with the value stored in the variable y. Since the foo (int x) function did not alter the memory-stored value of y, the main function prints out the value 20 as shown below.

2. **Program 2:**

The output of this program is the value of the variable y, which is 121. The main function initializes the variable y by declaring it an integer and setting it equal to 20. Next, the main function calls the function foo (&y). In a separate stack frame, the void foo (int \*ptr) function passes y by reference to its memory address. After declaring new local variables, a and b, set to values of 10 and 4 respectively, the function dereferences the pointer to y’s memory address and thus alters the value of y through its pointer to equal the sum of a squared and the product of four, b, and five. The foo function, having run to completion, then pops off of the stack frame, having altered the value of the variable y at its memory location. Now, the main function stack frame prints out a double with the value stored in the variable y. By virtue of the calculation in the foo function, the value of y is now 121 and this value becomes the output of the program.

3. **Program 3:**

The output of this program can be shown as:

x = 10.25

\*ptr = 10.254575

x = 20.25475

\*ptr = 20.25475

x = 21.2545

\*ptr = 21.2545

First, the main function declares two variables of type double, x and and \*ptr. The function then initializes the pointer variable by setting it equal to the memory address of variable x. Next, it dereferences the pointer variable and sets the value stored at the memory location of x equal to 10.254575. The first print statement outputs a string sequence (“ x = “) with this value of x, accurate to two decimal places (10.25). The second print statement outputs a string sequence (“ \*ptr = “) with the default representation of the value indicated by the pointer to x (10.254575). Again, by dereferencing the pointer to x, the code then adds ten to the previous value stored at the memory location for variable x (so that the sum is now 20.254575). The third print statement in the main function outputs a string sequence (“ x = “) with the updated value stored in variable x in its default representation (20.254575). The fourth print statement similarly outputs a string sequence (“ \*ptr = “) with the default representation of the value indicated by the pointer to x (20.254575). Lastly, by dereferencing the pointer to x, the code adds one to the previous value stored at the memory location for variable x. The fifth print statement outputs a string sequence (“ x = “) with the updated value stored in variable x, accurate to four decimal places (21.2545). The sixth print statement outputs a final string sequence (“ \*ptr = “) with the value indicated by the pointer to x, accurate to four decimal places (21.2545).

4. **Program 4:**

The output of this program is the pair of values stored in variables i and j, which is 0 and 125. First, the program initializes integer variables i and j and sets them equal to 0 and 1 respectively. Next, inside the main function, the code calls function void foo (int \*p, int \*q), passing by reference the memory addresses for variables i and j. This function call generates a new stack frame for foo wherein two new local variables (a and b) are declared and assigned the values 10 and 5 respectively. Next, the assignment of q to p says that the memory address of pointer p (which was previously equal to the memory address of variable i) should now equal the memory address of pointer q (which points to the memory address of variable j). In other words, the pointer p, which previously pointed to the variable i, should now point to the same address as pointer q which, in turn, points to the variable j. Finally, the last line in the function foo dereferences the pointer p so as to manipulate the value stored in the memory location for variable j, setting it equal to the sum of b squared and the product of ten and a. Now the foo function, having run to completion, pops off the stack frame. The main function continues with a print statement that outputs the values of i and j. Since the pointer p, which initially pointed at the memory location of i was almost immediately assigned the memory address of pointer q (and variable j) in the function foo, the value for variable i never changed from 0 when the code dereferenced p and manipulated the value at its address. However, because pointer p shared the same address as pointer q (and, subsequently, variable j), the dereferencing and manipulation of the value at the address indicated by pointer p *did* alter the value in variable j, changing it to 125. Therefore, the print output of the program results in the string “0 125” followed by a line break. The second-to-last line of code getchar(); allows the user to input some string sequence at the command line before the main function returns and the program halts.

5. **Program 5:**

The output of this program can be represented as follows:

(Computer Systems, CSCI 2271)

(CSCI 2271, Computer Systems)

First, the program initializes two character-type pointer variables, x and y, setting these equal to the constant string arrays “Computer Systems” and “CSCI 2271” respectively rather than to memory addresses. The program also declares character-type pointer t but does not immediately initialize it. Next, the code calls the function swap (x, y), generating a new stack frame and passing the pointers x and y which indicate their respective strings (not modifiable memory addresses). The first line in the swap function initializes a local pointer t and sets it equal to pointer x (which simply points to the string “Computer Systems”, not a variable’s memory address). Next, the code attempts to set pointer x equal to pointer y. However, since pointer x refers to a string and not a memory address of a modifiable variable, the value indicated by x changes only locally—not globally. The same is true in the next line of code where t is assigned to y. Therefore, when the swap function runs to completion and pops off the stack frame, the subsequent print statement in the main function outputs the string (Computer Systems, CSCI 2271) with the original strings indicated by pointers x and y. Next, inside the main function, the code sets the pointer t equal to the pointer x, so that t points to the same string value indicated by the pointer x (that is, “Computer Systems”). The pointer x is then set equal to pointer y so that the pointer x now indicates the string to which y also points (that is, “CSCI 2271”). Lastly, the pointer y is set equal to the pointer t so that y now points to the value indicated by t (that is, “Computer Systems”). When the program implements a print statement in the next line, the changes to the x and y pointer values, stored locally within the main function stack frame, result in a swapped output of (CSCI 2271, Computer Systems).