Raman Kahlon

CS 240 Lab 1

Problem 1:

After compiling and running the program in v1 multiple times, I observed that the program prints two lines of output. The first line is always the same, since it prints out the integer 5, which is the value of the variable s declared in main.c under v1. The second line prints out the address of the variable s. The address is used to refer to the location of the integer in RAM. Based on the address outputs, I observed that the addresses have a base in common, 0x7ff.

Problem 2:

After calling the method changeval1() in v2, the value of s printed on stdout continues to be 5 because it is not passed as a pointer. In C, arguments are passed to functions by value so changeval1() cannot affect the variable s in main(). The method changeval1() simply makes a copy of the variable s and places it in int a. In changeval1(), a is set to the value of 3, however, this doesn't affect the value of s in main(). Thus, the output of the program is still 5.

Problem 3:

After calling changeval() in v3, the value of x printed on stdout is 3 because the program passes a pointer to the value of x as a parameter when changeval() is called. At the beginning of the program, changeval() has been defined to be a pointer to an integer. In main(), changeval is called with the parameter &x, which represents the address of the variable x. After the address has been printed, the line of code *a = 3 means that the content of a has been changed to the value 3. In other words, the value that corresponds to the address of becomes 3.

If we change "*a = 3" to "a = 3", gcc gives a warning during compilation. The warning states that "assignment makes pointer from integer without a cast". This means that *a is a pointer and can't be assigned the data type of an int. However, *a can be a pointer pointing to the data type int. In the original statement *a = 3, a was used as a pointer to an int, which is the value 3 in this case. During runtime, a.out prints the value of x, then the two same addresses of the variable, and finally the value of x again. After changing *a = 3 to a = 3, however, the value of x doesn't change, instead it remains at 5 before and after calling changeval().

Problem 4:

If the filename a.out is undesirable for the executable file, there are two methods for changing it to a different name. The first method is to use the mv command to rename the file. For example, if I would like to change the executable file name a.out to final.out, I would simply type in the command line: mv a.out final.out, resulting in all of the content of a.out being shifted to m.out. The second method is to use the -o option to rename a.out. For example, in gcc I would type: -g -o final.out a.out.

Problem 5:

The output produced by main.c in v4 is 4 lines long. The number 5 is printed out on the first line, which is the value of the variable x in the program. Next, the program prints out the address of x. On the following line, the number 5 is printed out again because the variable y is defined to be the address of x. Thus, this third line of code prints out *y, which is a reference to the content of x, which is 5. On the final line, the value of y is printed out, which is the aforementioned address of x.

When the assignment "z = &y" is added to the end of main, z must be declared as "int **z". In other words, z is a reference of a reference. In order to print the value of x using z, we would use **z as the parameter of printf().

Problem 6:

A segmentation fault is a failure condition raised by hardware with memory protection, notifying the operating system that the software has attempted to access a restricted area of memory. In other words, this is a memory access violation when a user is trying to access RAM that doesn't belong to them. The fault arises in main() because the method changeval() is called with an incorrect parameter. At the beginning of main(), changeval() is defined to take a parameter of a pointer, however, when changeval() is called after the program prints out '3' under "ok 2", the parameter is a variable of type int, not a pointer. The segmentation fault occurs after the program attempts to execute changeval(x). In this case, x is defined as an integer, yet the changeval() method only accepts a parameter of a pointer to an int.

Problem 7:

Arrays are always laid out from low memory locations to high memory locations. In v7, the 1-D array s[3] is of type int. When placed in RAM, the array is treated as a set of integers in consecutive memory locations. The data type int takes up a space of 4 bytes, therefore if the first index s[0] starts at the address of 1000 and ends at 1003, then s[1] will begin at 1004 and take up space until 1007, and so on.

Problem 8:

The program main.c in v8 results in a segmentation fault because h is simply declared as a pointer to a singular integer. In main(), the program attempts to assign the values 100, 200, and 300 to the first three indices, respectively, but these statements can't be executed because h is defined as an integer. As a result, a segmentation fault occurs when attempting to run the program. To fix this issue, the declaration of h needs to be changed from "int *h" to "int h[3]". The former declaration is a pointer to a singular integer, while the latter declaration is an array of size 3 with integer data types for each index. After this line of code is changed, then the program properly executes the code in main() and prints out the content of the array in successive lines, 100 200 300.

Problem 9:

The silent run-time error in v9 occurs when the last two for-loops are executed. At the beginning of main(), the array a is defined as having 5 indices, each of type int. The first two for-loops are executed properly because the condition i<5 only allows each loop to run for 5 times, matching the number of indices in the array. The condition for the last two for-loops are i<6, which means that each loop would run 6 times. In this case, the array is defined as having a size of 5, so the last two loops are not supposed to run 6 times. However, during run-time, the array a is printed from 0 to 4 then 0 to 5. These run-time bugs could be prevented by using the #ifdef and #endif debug blocks in between the for-loops of the program to check if everything is being executed properly. For example, in v6, there are several print function statements ("ok 1", "ok 2", and "ok 3") that would produce a certain type of output based on whether a bug is found or not. For instance, if no bug is found, the program would print "ok" followed by the number of the debug statement.

If you change the limit of the third for-loop from 6 to 7, the program will print out array a from 0 to 4, then 0 to 5, and will give a statement: stack smashing detected: a.out terminated. The program then proceeds to backtrace, then prints out a memory map, and finally aborts.

If you run gcc with the -fno-stack-protector, the stack smashing error doesn't show up, even if the program is attempting to place a 6th array index into array a, which has been defined as having only 5 indices. Essentially, the -fno-stack-protector disables the stack smashing error by allocating more space on the stack. This allows the program to check whether the stack has been overwritten while in the function.

Problem 10:

In v10, the 1-D array a[6] is of type char. This particular array is treated as a set of characters in consecutive memory locations in RAM. The data type char takes up a total space of 1 byte, therefore if the first index a[0] starts at the address of 100 and ends at 101, then a[1] will begin at 1002 and take up space until 1003, and so on. The '\n' is a null character, indicating the end of the current string of characters. Therefore, when the statement printf("%s\n", a) is executed, the %s indicates the string of characters in a. This includes ab, but due to the null character at the third index, a[2], the string terminates and the program simply prints "ab".

When calling the method strcpy(), a[6] can be used to store the string "abcdef" because it initially only contains the 5 characters "abcde" and can change the character at the 6th index from '\0' to 'f'. Since the string "abcde" was initially placed into the array a, the last index was already null and was now changed to another character, 'f'.

Bonus Problem:

It is certainly possible to use the method strcpy() incorrectly, resulting in a bug. This bug states that anything might happen if the destination string parameter *dest* of strcpy() is not large enough to contain the original string parameter *src.* This is related to the silent run-time error in Problem 9 because the array a in v9 would add a 6th integer to the array a, which was only defined to contain 5 indices (0-4), without gcc giving an error during compilation time. Similarly, the string "abcdef" is written into the array a in v10 because an extra character 'f' was added to the string as the 6th character in the 5th index. This is clearly a bug and could be prevented if the programmer checks for the size of the *dest* string to determine whether it can fit the characters of the original *src*.