CS240 Lab2

Maninder (Kaurman) Kaur

July 2, 2025

Question 1

There were the following adresses that were printed each time I ran main.bin: 0x7ffdd6fdd924, 0x7ffe3dfeef14, 0x7ffcb44651a4, 0x7fff977aee34 meaning the memory adress changes with each run of main.bin. The different outputs tell us that the variable x's address is in the "temporary memory" of the computer, which as the name suggests, means it will be used while running and then wiped to reset and reduce memory allocation. From right to left, the address is binary is: 0111 1111 1111 1111 1001 0111 0111 1010 1110 1110 0011 0100. There are 48 bits total in that address. The x-bit computer that you have directly indicates the max amount of physical memory that a computer can make into a address.
Now that we have removed x=7, instead of printing 7, it will print out 0. Again, it it not required that the memory addresses stay the same as it is in temporary memory, which is a pre-allocated part of the stack.

Question 2

When you run a.out from v2/main.c, you get the following:

```
7
0x7ffe364fab1c
7
0x7ffe364fab1c
```

The memory addresses will be different for each run, keep in mind, but this is what I got when I ran it so we will use it as an example. Two integer values are created, one called x that stores 7, and another is a pointer y that points at x. The first two lines prints the value at the address and then the address itself where x is stored. The two last lines first call the value inside of the address that y is pointing to, and then the pointer itself with is simply an address. item[] You are unable to refrence x with **r without using *y in between, as you cannot write `r = &&x`. With each time you reference a pointer, instead of x, the address changes, as the pointer itself has a spot on the memory stack.

Question 3

The main.c prints out `100 0` out on three lines.

The reason that one() changes to `1 2`, is because, initially it was accessing the closest instance of the variables which was inside. Because, a and b were not given values when local to one(). one() printed out arbitrary values for them. When we removed the local variables, one() searched for local, found nothing, looked for global calls of a and b, found it, and returned those.

When we remove the local variables, it removes the variable addresses from the function's stack allocation and goes to main()'s overall. When it becomes global, it often goes to a higher memory address, becoming larger.

Question 4

`7 200` is printed, because the first one simply exists in the function, but is not stored anywhere. This x is printed and not changed. On the other hand, &y means that the function should apply all the changes to the value at the address itself, thus it is changed.
The r and x are not the same because they are in different memory slots. R is the blueprint, while x is a copy. On the other hand, y and r are the same because y is a direct pointer to r. Because of that, the value inside of y is not a address to x. The y pointer derefrences the pointer to y, which allows us access to the value at y. When we assign 200, we essentially change the value at r because y points to it.

Question 5

We split the main.c into 3 separate files and compile it into a executable called changeling.bin. Just as before, it does as intended.

Question 6

- explain binary search + why efficient : The binary search works like this: you surround your code segments with debug statements using printf(). You divide code into halves. If the debug runs without error, you know that the fault is in the second half, otherwise the first. You repeat the halves until you find your error. It is efficient due to how it halves the amount of statements that need to be used, instead of checking line by line.
  The steps needed in the worst case can be given by the formula: $\log_2(N)$, where rounded up to the nearest whole number would give you the N

amount of steps in worst case scenerio. Example: if $N = 1000 \rightarrow \log_2(1000) \approx 9.97 \rightarrow 10$ steps
(I removed and replaced the newline after seeing the effects) If you were to remove the newline character, the output would not be flushed properly, meaning it might stay until the program is terminated. This can lead to issues and make you confused and screw up your terminal. When debugging, it'll make it harder to find where the errors are. If you want to make sure this does not happn, you use `fflush(stdout)` that forces the flush off the program buffer and removes an confusion.

Question 7

An array like z is put in memory slots next to one another in memory unlike being placed randomly around like separate int vals. Z for example, is int val, so each value takes up 4bytes and are built atop one another. Address 0x0000 for example could hold z[0], 4 bytes above at 0x0004 is z[1].
From the explanation, you can see why (z+3) and z[3] are similar. All pointers for an array, point to the first adress, so z[0]. The (z+3) deferences the pointer and moves 3 elements towards the fourth value which is z[4] ot 400.
We outputted the addresses of z and &z. It is the same because both point to the same memory location even if z is type int* and &z is a pointer to the whole array.
why layour incorrect
As mentioned above, gcc puts arrays on top of one another on the memory stack to allow +3 and accessing them easily.?

Question 8

The silent run-time error happens is when the loop goes beyond the array size 5, because it goes to 6. The array s is through 0-4 indiex, but it attempts to access s[5]. This is simply undefined, making the program continue with invisible errors like overwriting some memory slot or messing up the stack.
Silent errors are worse because you don't immeditely tell you what's wrong, which can lead to errors later down the road. Alongside this, you'd never know what data or memory is wrong or changed. It's worse to trace as well, as the program continues.
If you chnage it to 8, the program will lead to a segmentation fault, because it is over bounds compeletly, and goes into another memory slot that we do not have access to
Compiling with `-fno-stack-protector` will make the segmentation fault errors go away. No stack protection that usually adds canaries to check

for returns. With it disabled, it will not crash immediately, but is not a good practice.

Question 9

It differs from the original in many ways. One, the placement of the array in memory is differnet as now it is placed in the global var location within the file location. Because of its different memory location, it does not have a canary, which does not check if it is going out of bounds. Instead, it goes out of the s[5] and starts overwrriting other memory locations.

Question 10

The string ”hello” assigns each character as a single value and ending it with a null terminator. The total bytes, because a char is 1 byte and a null terminator is 6 bytes. The longest possible string would be 9 bytes.

Bonus

The string would not end and become undefined. All strings end with it and it lets the computer know that it is finished.Without it, the program could crash, or could cause an infinite loop.
To not have any issues, check the last character in the array and make sure it is null terminated. You could limit how much is printed to find the error as well.