First I did some experiments messing around with the statement **A[6]=A[6]+10;** I added one more print statement to the end of the main function just to double check. So, what I did is test values to see How much I needed to add to skip the original **out: printf…** line. I found out that +20 would make me skip the line. I tested it one more time with another print statement and changed it to +30, and I skipped the new line I added. One last time I changed it to -10 expecting that the program would be stuck in an infinite loop calling f(), and that is what happened.

Therefore, it is reasonable to claim that A[6] is a return address, and by adding 10n to it you can move up n lines of code. I then made the printing loops print all the way up to 15 instead of 10 to confirm my suspicion. When I did that, I immediately saw the variables 0-4 of main's A integer array in positions 12-15. Essentially therefore, it seems that the activation record of a function has the first declared variables first, and then at the end it has the return address and stack pointer. Therefore A[6] in f() controls where the function f() will return to at its termination.

Next, I assigned a value to I and nothing happened. So, I made 2 new variables in f and it finally segfaulted. I tested with 3 variables and it still gave a segfault. But curiously when I gave it four variables, there was no segmentation fault. The other thing that happened is that whenever two variables were added, the position of L[0] moved up 2, it started at 8, then moved to 10, and finally moved to 12 when there were 4 variables added. It is worth noting that the variables that I used were int. So I tested it out with smaller char variables (1 byte versus 4 bytes for the int), and I found that the moment I used 5 chars (5 bytes) it caused it to segmentation fault and L[0] was moved to position 10.

Fixing it so that it skipped the print statement was easy enough, I just had to change the line **A[6]=A[6]+10** to **A[8]=A[8]+10.** After I did that it skipped the print statement like normal and only printed I am here at the end.

I do not quite understand why the position of the return address would move by two positions (8 bytes) when 5 bytes where added, and then move 4 positions (16 bytes) when 13 bytes where added. It seems to me that the variable I (4 bytes) and then the added 5 chars (5 bytes), that it created a brand-new space of 8 bytes for the last char to exist in alone. This may be to help support between 32-bit and 64-bit operating systems. In this way, in a 32-bit operating system 2 ints (4 bytes each) inhabit an 8 byte space together, that by itself in a 64-bit operating system would be simply inhabited by 1 int (at 8 bytes).

Example output of default code	Example output of Seg fault with 2 int variables added to f(), [they are at 3 and 4].
main is at 4195835 f is at 4195655 I am about to call f 0 4262346032 1 32765 2 4195440 3 3 4 4262346496 5 32765 6 4195980 7 0 8 100 9 200 10 300 A is 4262346032 -4 4262346480 -3 32765 -2 4195822 -1 0 0 4262346032 1 32765 2 4195440 3 3	main is at 4195849 f is at 4195655 I am about to call f 0 1572439944 1 32766 2 1572440416 3 15 4 14 5 5 6 1572440416 7 32766 8 4195994 9 0 10 100 A is 1572439944 -4 4195788 -3 0 -2 1572440400 -1 32766 0 1572439944 1 32766 2 1572440416 3 15

4 4262346496 5 32765 6 4195990 7 0 8 100 9 200 10 300 I am here	4 14 5 5 6 1572440426 7 32766 8 4195994 9 0 10 100 I called f I am here Segmentation fault (core dumped)