On the return of f(), it skips the first printf() because in pointer in f() *A adds ten to the return address of f() and skips the instructions for `printf("I called f\n");` and does the instructions of the next printf(). All of the instructions live in the code segment in memory and we can go to any spot in the code if we wanted to. For example, we can subtract 10 to continuedly loop the call of f().

When we add variables, we are increasing the activation record of f() and A[6] is no longer points at the return address and we get the runtime error, we need to add two more spots, or A[8] to get it to point at the return address. These new variables are 32-bit integers that get stuffed into 64-bit slot, for a 64-bit architecture, so the complier can stuff another integer in the same spot at the high or low end, whatever is not filled. So, that when `i` and `j` get allocated, the are stuffed in the low and high ends of the 64-bit spot. So when we add in `k`, we get a new 64-bit spot for k and the activation record gets shifted down and A[6] no longer points at the return address and produces the runtime error.

Also, when we get the runtime error, it looks like to is messing with the Stack Pointer that put backs to main for the activation record of main, when we add 10 to it we are moving that stack pointer to 10 above the beginning of main's activation record and screwing up the data of main. If we print L[] after the function of f(), we get garbage instead of the 100,200,300, and 400 we set it to earlier. And I believe when it deallocates the activation record of main, it goes above the cap of the memory and Seg faults. Or maybe when it resets the activation record to the top of memory, to does not get the right spot to find main's stored stack pointer and get garbage to point the top of the stack somewhere else.

```
/* Program to demonstrate how to over write the
* return address inside of function
* we will use a global variable to store
* the address we want to go to on return
* and we will use an array in the function to
* seek the location and replace with the new value



Shaun Cooper

2020 September




Jeffrey Lansford
Program 3
9/18/2020
Program to show how the runtime stack works and how language uses memory.
*/
#include <stdio.h>

// dummy function which makes one important change
```

```c
void f()
{

    unsigned int *A;
    int i;

    // varibles to use to increase activation record
    // we can leave all the varibles declared, as the optimizer will not allocate
 varibles not assigned
    int j;
    int k;
    int l;
    int m;
    int n;

    j = 12;
    k = 13;
    A = (unsigned int *)&A;

    for (i = 0; i <= 10; i++)
        printf("%d %u\n", i, A[i]);

    // add two to index for two varibles added
    A[8] = A[8] + 10;
    printf("A is %u \n", A);
    for (i = -4; i <= 10; i++)
        printf("%d %u\n", i, A[i]);
}

int main()
{

    int A[100];
    unsigned int L[400];
    L[0] = 100;
    L[1] = 200;
    L[2] = 300;
    L[3] = 400;

    printf("A[] is at %u \n", A);
    printf("L[] is at %u \n", L);

    for (int i = 0; i < 100; i++)
        A[i] = i;
    for (int i = 0; i <= 3; i++)
        printf("L:%d %u\n", i, L[i]);

    printf("main is at %lu \n", main);

    printf("f is at %lu \n", f);
    printf("I am about to call f\n");
    f();
    printf("I called f\n");

    // tests to see how the SP changes
    // L[0] = 100;
    printf("here \n");
    for (int i = 0; i <= 3; i++)
        printf("L:%d %u\n", i, L[i]);

out:
    printf(" I am here\n");
}
```