

# CSE 333 Section 1 Solutions - C, Pointers, and Gitlab

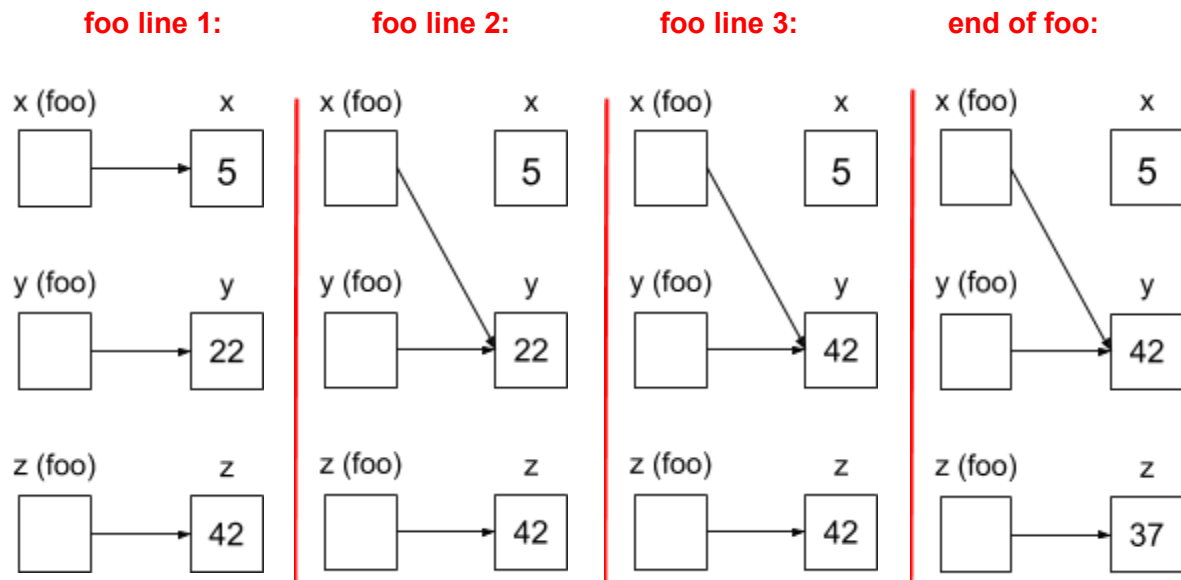
## Pointers

### Exercises:

Draw a memory diagram like the one above for the following code and determine what the output will be.

```
void foo(int32_t *x, int32_t *y, int32_t *z) {  
    x = y;  
    *x = *z;  
    *z = 37;  
}
```

```
int main(int argc, char *argv[]) {  
    int32_t x = 5, y = 22, z = 42;  
    foo(&x, &y, &z);  
    printf("%d, %d, %d\n", x, y, z);  
    return EXIT_SUCCESS;  
}
```



So, the code will output 5, 42, 37.

The following code has a bug. What's the problem, and how would you fix it?

```
void bar(char *str) {
    str = "ok bye!";
}

int main(int argc, char *argv[]) {
    char *str = "hello world!";
    bar(str);
    printf("%s\n", str); // should print "ok bye!"
    return EXIT_SUCCESS;
}
```

The problem is that modifying the argument `str` in `bar` will not effect `str` in `main` because arguments in C are always passed by value. In order to modify `str` in `main`, we need to pass a pointer to a pointer (`char **`) into `bar` and then dereference it:

```
void bar_fixed(char **str_ptr) {
    *str_ptr = "ok bye!";
}
```

## Output Parameters

### Exercise:

`strcpy` is a function from the standard library that copies a string `src` into an output parameter called `dest` and returns a pointer to `dest`. Write the function below. You may assume that `dest` has sufficient space to store `src`.

```
char *strcpy(char *dest, char *src) {
    char *ret_value = dest;
    while (*src != '\0') {
        *dest = *src;
        src++;
        dest++;
    }
    *dest = '\0'; // don't forget the null terminator!
    return ret_value;
}
```

Why do we need an output parameter? Why can't we just return an array?

If we allocate an array inside `strcpy`, it will be allocated on the stack. Thus, we have no control over this memory after `strcpy` returns, which means we can't safely use the array whose address we've returned.

We could've also returned a pointer to a `malloc`'d block of memory. How might this be better or worse than using an output parameter?

### Pros of Malloc

- We don't have to worry about an output parameter not having enough storage space for the copy.
- For very large strings, it's more efficient to allocate memory on the heap for a copy rather than allocating an array on the stack to hold it.

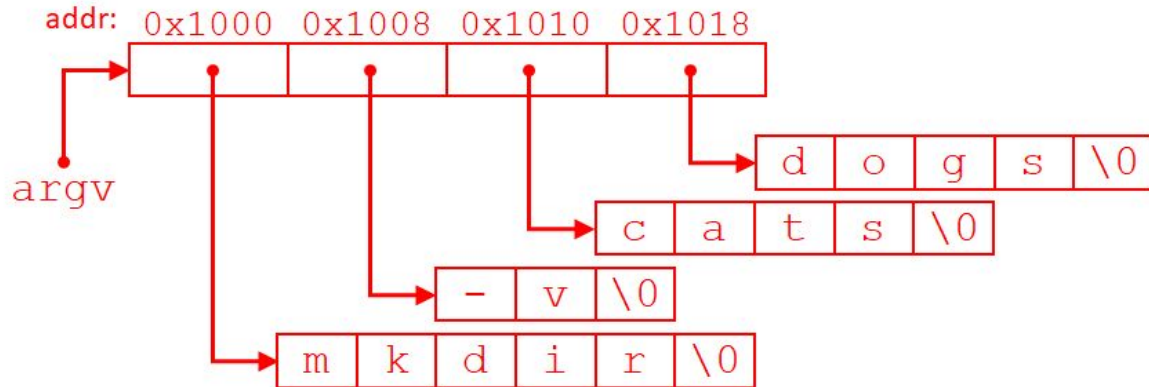
### Pros of Output Params

- We don't have to handle the case where `malloc` fails.
- More flexibility for the user - they can choose to use a stack-allocated array for storage or they can use a `malloc`'d chunk of memory.

## Pointer Arithmetic and Arrays

### Exercise:

Given the following command: "mkdir -v cats dogs" and `argv = 0x1000`, draw a box-and-arrow memory diagram of `argv` and its contents for when `mkdir` executes.



`argv` is the second parameter, so its value is stored in `%rsi` and does not take up space in memory. The character arrays have unknown/unspecified addresses that are stored in the entries of `argv`. Each character of the command-line arguments takes up 1 byte of memory and the elements of each character array have consecutive addresses, though the arrays are likely not contiguous to each other.

Using the same information from above, what can you say about the values returned by the following expressions? You may not be able to tell the exact value returned, but you should be able to describe what that value is/represents.

- 1) `argv[0]` -> address of the first character in "mkdir"
- 2) `argv + 1` -> `0x1008`
- 3) `*(argv[1] + 1)` -> 'v'
- 4) `argv[0] + 1` -> address of the second character in "mkdir"
- 5) `argv[0][3]` -> 'i'

**Exercise:**

A prefix sum over an array is the running total of all numbers in the array up to and including the current number. For example, given the array {1, 2, 3, 4}, the prefix sum would be {1, 3, 6, 10}.

Write a function to compute the prefix sum of an array given a pointer to its first element, the pointer to the first element of the output array, and the length both arrays (assumed to be the same).

```
void prefix_sum(int *input, int *output, int length) {
    if (length == 0) {
        return;
    }
    output[0] = input[0];

    for (int i = 1; i < length; i++) {
        output[i] = output[i - 1] + input[i];
    }
}
```