CSE 333 Section 1 - C, Pointers, and Gitlab

Sample Solutions

Exercise 1:

Consider the following snippet of code

```
void division(int  numerator,
              int  denominator,
              int* quotient,
              int* remainder) {
  *quotient = numerator / denominator;
  *remainder = numerator % denominator;
}

int main(int argc, char* argv[]) {
  int quot, rem;
  division(22, 5, _____, _____);
  printf("%d rem %d\n", _____, _____);
  return EXIT_SUCCESS;
}
```

Which parameters of division() are output parameters?

`quotient` and `remainder`

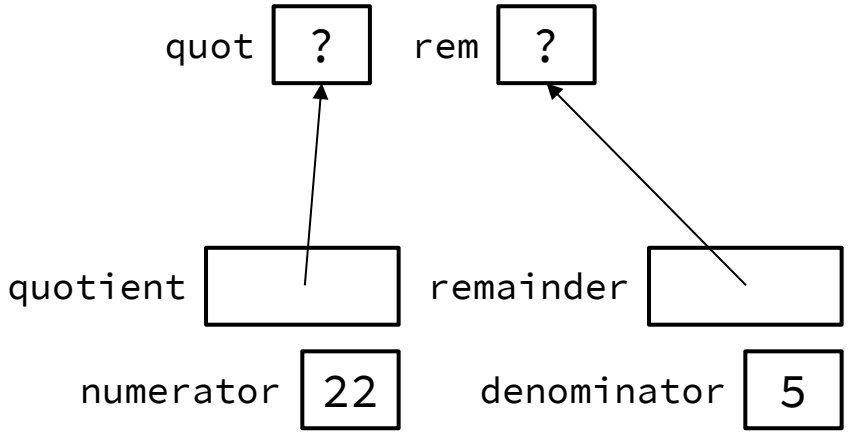
What variables should go in the blank spaces in our call to division()?

`&quot` and `&rem`

What should go in the blank spaces in our call to printf()?

`quot` and `rem`

Draw out a memory diagram of the beginning of this call to division().



Exercise 2:
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];
      }
}
```

Exercise 3 (bonus):
The following code has a bug. What's the problem, and how would you fix it?
Changes shown in red:

```
void bar(char* ch) {
  *ch = '3';
}

int main(int argc, char* argv[]) {
  char fav_class[] = "CSE331";
  bar(&fav_class[5]);
  printf("%s\n", fav_class);  // should print "CSE333"
  return EXIT_SUCCESS;
}
```

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

Exercise 4 (bonus):
`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;
}
```

How is the caller able to see the changes in `dest` if C is pass-by-value?

The caller can see the copied over string in `dest` since we are dereferencing `dest`. Note that modifications to dest that do not dereference will not be seen by the caller(such as `dest++`). Also note that if you used array syntax, then `dest[i]` is equivalent to `*(dest+i)`.

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

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.

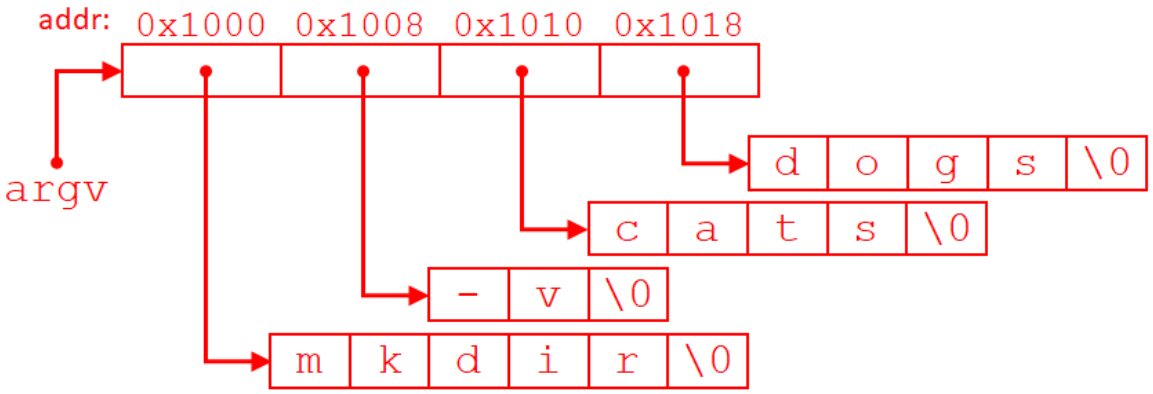
Exercise 5 (bonus):
More practice with output parameters and arrays.

Write a function to compute the sum of values and product of all values in an array. The function is given a pointer to the first element in an array, the length of the array, and two output parameters to return the product and sum.

```
void product_and_sum(int *input, int length, int *product,
                                                    int *sum) {
  int temp_sum = 0;
  int temp_product = 1;
  for (int i = 0; i < length; i++) {
    temp_sum += input[i];
     temp_product *= input[i];
  }
  *sum = temp_sum;
  *product = temp_product;

}
```

Exercise 6 (Bonus):
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'**