1. Consider the following:

```c
void mystery (int *a, int *b) {
    int whoKnows = *a;
    *a = *b;
    *b = whoKnows;
}
```

```c
int main (int argc, char** argv) {
    int x = 18;
    int y = 3;
    mystery(&x, &y);

    printf("x = %d ", x);
    printf("y = %d", y);

    return 0;
}
```

What does the above program output? (Circle One)

(a) x = 18 y = 3  
(b) x = 3 y = 18  
(c) x = 'memory address of y' y = 'memory address of x'  
(d) Gets a seg fault  
(e) Doesn’t compile

Solution: 
B

This is a basic swap function. If we passed x and y without passing a reference to them then any changes we made in mystery would only be local and would never be reflected in x and y in the calling function. This is because C passes everything by value. So instead of passing the value stored in x and y we pass the memory location of x and y by value instead. Since they are pointers in mystery, then I can dereference them to access the location where x and y are stored. When I dereference them I can change their values directly and thus any changes I make in mystery will be reflected in x and y back in main. So the correct answer, is x = 3 and y = 18.
2. Consider the following:

```c
int main() {
    void* a = malloc(800);
    void* b = &((int32_t*)a)[3];
    printf("%ld\n", b - a); // this line does compile
}
```

What does the above program output? (Circle One)

(a) 0  
(b) 8  
(c) 12  
(d) Nothing, doesn't compile  
(e) Nothing, runtime error

Solution:

C

We malloc 800 bytes of memory in `a` and then we assign `b` to point to the third offset from the start of `a`. The assignment `&((int32_t*)a)[3]` takes the pointer value stored in `a` and then casts it to a 32 bit integer. Then when we dereference using `[3]` it is doing so in the context that each element is a 32 bit integer. We then assign `b` the memory address of this location with the `&` operator. So if `a` had memory address `x` then `b` would be assigned `x + 3(4) = x + 12` since we are using 32 bit integers and each integer has 4 bytes and we are looking at the third offset in the array. So now when we subtract `b` from `a` we are subtracting their memory addresses. This gives us `b - a = x + 12 - x = 12` where `x` is whatever memory address is initially assigned to `a`. 
Consider the following code for the next two questions:

```c
// Accepts an array of "strings" and converts all the strings in the array
// to all upper case letters. The new strings are placed in an output parameter
// __B?__.
int toUpper(char** data, __B?__ /* declares parameter 'result' */) {
    if (!data || !result ) return 0;
    <some code>

    return 1;
}

int main(int argc, char ** argv) {
    char** upperData;
    char* data[3] = { "test str",
                     "another str",
                     NULL };

    if ( toUpper(data, __A?__ ) == 0 ) exit(1);

    char** p = upperData;
    while ( *p != NULL ) {
        printf("%s\n", *p);
        p = p + 1;
    }
}
```

The above code prints:
TEST STRING
ANOTHER STR

What is the best choice to replace __A?__ (Circle One)

(a) upperData
(b) &upperData
(c) *upperData
(d) &data
(e) malloc(sizeof(data))

Solution:

**B**

The main thing to notice in this problem is that upperData is an output parameter. Last week in section we mentioned how output parameters have to be passed and why that is the case. Here we are actually passing an array of c strings. Which is just an array of char * arrays. We want to modify the passed array and place the results into an output parameter. As I mentioned last week, in order to change the location of where a pointer points, you need
to pass it by reference. This is for the same reason we needed to pass x and y by reference in question 1. So in this case we needed to pass the address of where we will be placing the result. So the answer is &upperData.

What is the best choice to replace __B__ (Circle One)

(a) char * result[ ]
(b) char ** result
(c) char *** result
(d) char *** result[ ]
(e) void * result

Solution:

C

The answer here comes naturally if you saw how to do part a. Since we need to pass the output parameter by reference and the original type of upperData was a char ** then we have created a pointer to a char ** which is a char ***. Thus the answer is char *** result.
Consider the following code for the next two questions:

```c
// Compare two arrays and print whether or not they're equal.

int main() {
    int a[] = {10, 20, 30, 40, 50, -1};
    int* b = malloc(sizeof(a));
    // copy a to b
    memcpy(b, a, sizeof(a));

    if ( a == b ) printf("Equal\n");
    else printf("Not equal\n");

    free(a);
    free(b);

    return 0;
}
```

This code doesn’t run, and even if we fix it so it does, it doesn’t do what we want.

Write a brief phrase explaining two problems that shows us you’ve identified the problems:

1. _______________________________________________________________________

2. _______________________________________________________________________

Solution:
There were two main answers that we were looking for here:

(a) `a == b` does not properly compare the ‘arrays’. Both a and b are just pointers to the beginning of the memory that stores the values. Comparing a and b only compares the pointers and doesn’t look at any of the values in either of those blocks of memory.

(b) You cannot free(a) because it was never malloc’d. This will cause a runtime error.

Many also mentioned that `sizeof(a)` will not work because it doesn’t know any information about the size of int a[]. This is not actually correct. Using `sizeof(a)` will actually return the correct value of 24. However, even if `sizeof(a)` didn’t return the correct value it still wouldn’t crash the program. Instead you would just end up comparing much less of the array than you intended.
Consider the following code for the next two questions:

```c
// This function applies a user supplied function, f, to
// find the most preferred Car in an array of cars. f compares
// two Cars and returns the one it prefers.

Car* pickCar(Car* carVec, int size, PreferFunc f) {
    if ( size <= 0 ) return NULL;
    Car* result = &carVec[0];
    for (int index=1; index<size; index++) {
        result = f(result, &carVec[index]);
    }
    return result;
}
```

1. Write a C typedef statement to define PreferFunc, the function pointer type that is the type of pickCar’s second parameter.

Solution:
```c
typedef Car *(* PreferFunc)(Car*, Car*)
```

We talked about function pointers in section and how they would be important to the rest of your time with C and C++. Very few people got full credit on this question. While you haven’t had to actually write a function pointer in your homeworks up to this point, they are still a very important part of C/C++ programming. As a reminder the syntax is as follows:

Syntax for typedef’s are as follows: `typedef type name`, for example: `typedef int my_int` would assign my_int the type of int.

Syntax for function pointers are as follows: `return_type(* name)(parameters..)`, for this question it would be: `Car * (*PreferFunc)(Car*, Car*)`

Combining these two we get or solution. The only subtle issue here is that you don’t need the extra name of the function at the end of the typedef. You only need the PreferFunc name in the actual function pointer. If you added an extra name we only deducted one point.