a) What is the size of ptr?
    a. 4 bytes
b) What is the size of twod?
    a. 48 bytes
c) What is the size of twod[0] and why?
    a. 12 bytes, because it is a 1d array of 3 ints (3 * 4)
d) What is the size of twod[0][0]?
    a. 4 bytes
e) What can you say about twod and twod[0] as it relates to the name of the array?
    a. Twod is the whole 2d array, twod[0] is one of the columns of our 2d array
f) Draw a memory map that shows the memory locations of each element of the array and of ptr.

    ptr [0][1][2][10][11][12][20][21][22][30][31][32]

    Memory location: 790 all the way to 820

```
printf("twod + 3 is: %p\n", twod + 3);
```

This line will print the memory location for the 4th column of the array

Correct

```
printf("*(*(twod + 1)) is: %d\n", *(*(twod + 1)));
```

This line will print out the first value of a 2nd column of the array

Correct

```
printf("*twod + 1 is: %p\n", *twod+1);
```

This line will print out the memory address of the 2nd column of the array

Correct

```
printf("*twod[2] is: %d\n", *twod[2]);
```

This line will print out the value of the 2nd column of the first row

Correct

```
printf("*(twod + 2) + 2 is: %d\n", *(twod + 2) + 2);
```

This line will print out the memory address of the 3rd item of the 3rd row

Incorrect: d where a p should be. Otherwise guess is correct. Misreading on my part

```
printf("twod[1] is: %p\n", twod[1]);
```

This will print out the memory address of the first item of the 2nd row

Incorrect, got my memory mixed up

```
printf("twod[1][2] is: %d\n", twod[1][2]);
```

This will print out the value of the 2st row, 3nd column

Correct

```
printf("ptr %p\n", ptr);
```

This will print out the memory address of the start of our array

Correct

```
printf("twod [1] %p\n", twod [1]);
```

This will print out the memory address of the first element of the 2nd row

Correct

```
printf("ptr[1] %p\n", ptr[1]);
```

This will print out the memory address of the first element of the 2nd row

Incorrect: p where there should be a d. Incorrect because I got my dereferencing mixed up

```
printf("ptr + 1 %p\n", ptr + 1);
```

This will print out the memory address of the first element of the 2nd row

Correct

printf("*(ptr + 1) %p\n", *(ptr + 1));

This will print out the memory address of the 2nd item of the 1st row

Incorrect: p where a d should be. Also had issues with dereferencing.

printf("twod + 1 %p\n", twod+1);

This will print out the memory address of the first element of the 2nd row

Correct

printf("*twod + 1 %p\n", *twod + 1);

This will print out the memory address of the 2nd item of the 1st row

Correct

printf("ptr[8] %p\n", ptr[8]);

This will print an error message because it is out of bounds of our array dimensions

Incorrect: p where a d should be. Don't understand how this works unless we are talking about how in class our 2d array is actually all of them lined up.

4. No. Because the line you have stated in the PDF doesn't have a closing ')' or a semi colon. If this answer wasn't the case at it was a typo, then Yes. It would work but it would give us a garbage value that was stored in that location of memory. Since we can walk off lists and give stuff from those memory locations.

11. The difference is in how it is dereferenced. The first will tell the compiler that we are passing in an array that would look like this: twod[][3]. However the second, will tell the compiler we are passing in an array that would look like this: twod[3][]. Which would give the compiler errors. This error would be because we didn't pass in how many other arrays we have. We are only telling the compiler that they are size 3.

12. No. Because we are passing in a 2d array without specification as to how many columns there are. This would cause a crash to occure.

13. They are similar in that they add the same. Let me explain first! When we are using a passed in 1D array, we have to add up to our psuedo last index (which in this case would be 11). But if we use a 2D array, we have to count up to a given column and row, which when multiplied together (1 * 3) we get 3, and just like when

multiplied together (3*4) we get 12. They add together in such a way that makes sense as to why they are basically lined up in memory. The only real difference is that we have to worry about counting up in the 1D array vs the 2D array.