```
int a = 15;
```

1: Make a pointer named ap and point it to a.

```
int* ap = &a;
```

Reason: first we need to declare the variable ap. Since C++ is strongly typed, it's not enough to declare it as a pointer – it must be a pointer to some type. *int* ap* declares ap as a variable capable of holding the address of an integer.

Then, the address of the variable a is retrieved (using the ampersand) and assigned to ap.

```
a = 20;
*ap = 30;
```

2: What is the value of a?

30

Reason: *ap dereferences ap; in other words, it can be considered another name for a. Then the second line of code can be read "a = 30".

```
double x = 3.5;
double* xp = &x;
x *= 2;
cout << *xp << endl;
```

3: What is the output of the above code?

7

Reason: *xp dereferences xp, which holds the address of x. Again, this means that *xp can be thought of as x. Then the code *cout << *xp* should have the same effect as *cout << x*.

```
double y = 3.5;
```

4: Write two lines of code that will A) create a pointer to y named yp, and B) multiply the value of y by 2 *without* directly referencing y (i.e., using only the pointer).

```
double* yp = &y;
*yp *= 2; //alternatively, *yp = *yp * 2;
```

Reason: the first line of code is like the solution to Q1, except here we pointer to a double. The second line of code doubles the value of y using only the dereferenced pointer yp. Notice that we actually dereference

yp twice: once on the right hand side of the assignment operator (for access), once on the left (for modification).

```
double b = 3.14;
double c = 4.2;
double* bp = &b;
double* cp = &c;
b = 10.1;
cp = bp;
*bp = 10.2;
```

5: What is the value of *cp?

10.2

Reason: the line *cp = bp* is a regular assignment which copies over the address that bp holds into cp. Since bp is originally set to hold the address of b, then cp now holds that address too. If we keep track of what occurs at that address: first it is set to 3.14, then it is set to 10.1, and then bp is dereferenced to assign 10.2 to that location.

```
double dArray[6]{1,2,3,4,5,6};
cout << *(dArray + 0) << endl;
```

6: What would be the output of the above code?

1

Reason: there are two ways to think about this. The first is that any expression of the form **(arrayName + x)* is equivalent to *arrayName[x]*, so all we're doing here is printing out dArray[0]. The other way is by thinking about pointer arithmetic: dArray, like any array name, can be seen as a pointer to the start of the array in memory. By dereferencing it without advancing it (we say + 0), we get another name for the first element, dArray[0]. When we print it, we get 1.

```
cout << *(dArray) << endl;
```

7: What would be the output of the above code?

1

Reason: this would be exactly like in question 6; the +0 isn't necessary when trying to get the first element.

*cout << *(dArray + 3) << endl;*

8: What would be the output of the above code?

4

Reason: *(dArray + 3) is equivalent to dArray[3]. As before, dArray points to the start of the array. Writing dArray + 3 will move forward 3 elements, leaving us with the *address* of the element at index 3. To get the name of the element itself so that we can print it, we have to dereference this value.

int intArray[2][3]{9,8,7,6,5,4};

9: Draw what the 2D array intArray looks like in table form (rows and columns).

9	**8**	**7**
6	**5**	**4**

Reason: a good approach is to first draw the grid and make sure you have the dimensions right. With uniform initialization, numbers are filled in row by row, left to right.

int intArrayTwo[2][3]{{9,8,7},{6,5,4}}

10: Draw the 2D array intArrayTwo in table form (rows and columns).

Same as the solution to 9.

Reason: this is simply another way to initialize a 2D array in C++ and the result will be the same as the previous method. This might be a more intuitive notation, since 2D arrays are simply 1D arrays of 1D arrays, and this notation highlights this fact (each "row" is really an internal array, and each "column" is really the index of the element in the internal array).

int intArrayThree[3][2]{{4,2},{1,6},{5,3}}

11: Draw the 2D array intArrayThree in table form (rows and columns).

4	**2**
1	**6**
5	**3**

Reason: same as 10. This time we have 3 rows and 2 columns, which is the same as saying we have 3 internal arrays of size 2 each.

12: What is the meaning of *intArray? How is it different from intArray? (in terms of the same intArray initialized before question 9)

***intArray is another name for the 1D array that makes up the first row of intArray, {9, 8, 7}. intArray, meanwhile, is the name of the entire 2D array.**

> **More info: remember that in C++, array names "decay" to pointers under certain conditions, for example when passing them into functions. Interestingly, both *intArray and intArray would be pointing to the exact same memory location.**
>
> **The difference is that intArray points to the first element of the 2D array, intArray, while *intArray points to the first element *of the first row* of intArray. Though both sit at the same memory address, the difference can be seen when you try to do pointer arithmetic.**
>
> **If you were to say **intArray + 1*, you would move to the location where the 8 is held, 1 column to the right; but if you said *intArray + 1* (without dereferencing), you would move to the location where the 6 is held, 1 column down. The reason is that when you do pointer arithmetic, "+ 1" changes meaning depending on what your array holds. Remember that all 2D arrays are just 1D arrays of 1D arrays – each "row" is really an internal array. So intArray holds 2 arrays, and when you say intArray + 1, you jump to the next array (i.e. the next row). But *intArray, is the internal array, which holds integers – *not* arrays! So when you say *intArray + 1, you jump to the next integer of intArray[0].**

13: What is the meaning of *(intArray + 1)? How is it different from intArray + 1? And how is it different from *intArray + 1?

***(intArray + 1) is another name for the 1D array that makes up the second row of intArray, {6,5,4}.**
intArray + 1 is the memory address where the 1D array on the second row of intArray lives.
And *intArray + 1 is the memory address of the 8 in intArray.

> **More info: the difference between *(intArray + 1) and intArray + 1 is subtle. It's like the difference between intArray and *intArray in question 12. Both point to the same memory address, but technically, intArray + 1 points to the second array (i.e. second row) as a whole, while *(intArray + 1) points to the first element of the second array (i.e. second row).**
>
> ***intArray + 1 is different. Because of operator precedence rules, the dereferencing is done first, so *intArray evaluates to the first array of intArray. Then, +1 moves to the second element of this array.**

14: Print the value 4 from intArrayThree using bracket notation (e.g. array[i][j]).

cout << intArrayThree[0][0] << endl;

> **Reason: this is just standard 2D array access.**

15: Print the value 4 from intArrayThree using pointer arithmetic (no brackets).

cout << **intArrayThree << endl;

> **Reason: this is just like saying *(*(intArrayThree + 0) + 0). We dereference intArrayThree once to get the address of the first element of the first array of intArrayThree, then we dereference that to get another name for this value. With this name, we can print.**

16: Print the value 3 from intArrayThree using pointer arithmetic (no brackets).

cout << *(*(intArrayThree + 2) + 1) << endl;

> **Reason: first we move forward two internal arrays (rows). Then we dereference this address so that we have another name for this array and can do pointer arithmetic on it as well. To this internal array we add 1 to progress to the second element's address, then we dereference this value as well to get another name for the element, which we then print.**

```
int arr[3][2];
for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
                arr[i][j] = (i+1)*(j+2);
        }
}
```

17: Using pointer arithmetic on arr, write a statement that prints 9.

Hint:

> First draw out what the array looks like and run through the code by hand to fill in the values at each position. arr[3][2] would have 3 rows and 2 columns.

cout << *(*(arr + 2) + 1) << endl;

> **Reason: after filling out the array, it looks like this:**

2	**3**
4	**6**
6	**9**

To arrive at 9, first get to the third 1D array (i.e. row 3). The code arr+2 will point to the address where the third array is, but to actually work within this array we need to dereference arr+2.

***(arr + 2) will give us access to work on this array. Then, to get to the 9, we can move forward by 1 element, i.e. *(arr + 2) + 1. This again gives us the *address* of the element, but not the value of the element. To get to what's actually held in this position, we dereference again: *(*(arr+2)+1).**