These are unfinished sections that I deemed to advanced to belong in the beginning portion (part 2) of my book. They will be re-instated at later chapters. All of this content refers to the elusive concept of pointing to pointers and how it is used.

Returning Strings from Functions

Arrays returned from functions suffer the same fate as strings. You cannot use assignment to place the value of one string into another and likewise you cannot return an array. There are two ways around this: return a pointer (the address of the first element in the array, or character in the case of strings) or use an output parameter.

Multiple Dimensions

Arrays are not limited to flat, linear lists of values. They can be multi-dimensional. Think of a grid: it has cells that can be accessed by both row and column. A grid is two-dimensional because points within it are accessed by two indices (rows and columns). Arrays can have two, three, four, or any amount of dimensions you wish. For each dimension you will need one more index.

Author's Preference: Arrays beyond the third dimension are difficult to visualize and best avoided.

To declare a multidimensional array you must specify a size in square brackets for the size of each dimension:

```
int two_d[10][10];
```

The above declaration creates a two-dimensional array called 'two_d'. To access a subscript in the above array you will need to specify both dimensional indices:

```
int i = 0, j = 0;
two_d[i][j] = 0;
```

Two indices are required to access a single subscript. The number of elements, or subscripts, in a multidimensional array is calculated by multiplying the size of each dimension. The array 'two_d' as declared earlier has one hundred (100) total subscripts because it has two (2) dimensions each with ten (10) elements.

In this example, the variable 'two_d' might be thought of as an array of arrays; it is perfectly example to think of multidimensional arrays in this manner. The first dimension *contains* the second dimension. That is, specifying only the first index in 'two_d' is like referring to a row that then itself contains ten (10) subscripts of its own. Thus, to position yourself in an exact spot you must specify the row and then the column of that row:

```
two_d[row][col];
```

It is important to remember that the left-most dimensions specified contain those to the right. That is, the row contains ten (10) columns, but a column does *not* contain ten (10) rows. The code 'two_d[row]' specifies a single row, thus becoming the same as a flat, one-dimensional array of ten (10) subscripts. You could then use this subscript array as you would any normal array:

To be done: example of passing each row of a md array to a function accepting a flat array (remember to include a flat array in the example to show the association between the two).

Blarg.

Multiplication Table and Analogies

The multiplication table is a perfect example of a multidimensional array. If you never learned it, the multiplication table is a two-dimensional grid of numbers that contains all the answers for multiplying the row number to the column number:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

The table above only shows the results of multiplying any two (2) numbers in the range of one (1) to five (5). For example, to see the result of multiplying three (3) and two (2) you would simply place your finger where the left-most column is three (3) and the right-most column is two (2). The answer is six (6). This is much like accessing a multidimensional array where you are accessing the subscript at index '3,2' or '[3][2]'. The following program creates a 5x5 multiplication table in a multidimensional array and calculates multiplication answers from it:

To be done: insert multiplication table program here.

Another analogy of a multidimensional array is a chess or checker board. The location of each piece is not determined by a single value, it is determined by its row and column. Paper is considered flat: two dimensional. That is, when you have your pencil on a piece of paper, its location is determined by its horizontal and vertical offsets from the sides of the page. Lastly, remember the hype surrounding 3D gaming. The world we live in requires at least three dimensions to represent: vertical, horizontal, and depth. When you think of a square, you think of width and height. When you think of a cube you think of width, height, and depth: three dimensions.

It is my hope that I have instilled the essence of what multidimensional arrays are. They are blocks of data whose elements are located by multiple indices, one for each dimension.

Initializing Multiple Dimensions

A multidimensional array can be initialized in two ways. The first is by specifying the values of each dimension separately and the second is to initialize all the subscripts with a linear list of values.

Author's Preference: Initializing a multidimensional array with a linear list of values can confuse the fact that the array has more than one dimension. I usually avoid that method, but I won't pretend it doesn't exist either. ☺

Remember how you initialize a flat array:

```
int arr[5] = { 0, 1, 2, 3, 4 };
```

A multidimensional array can be initialized in the same way, however you must initialize each dimension correctly. A dimension that simply contains another array must be initialized as an "array of arrays" whereas the right-most dimension actually has the values. Basically, for a two-dimensional array you will have the normal flat array initialization repeated a number of times equivalent to the size of the first dimension.

```
int mdarr[5][4];
```

The array above, 'mdarr', will have five (5) normal initializations. Each of those initializations will contain four (4) values. To initialize the entire array to zeroes:

```
int mdarr[5][4] =
{
    { 0, 0, 0, 0 },
    { 0, 0, 0, 0 },
    { 0, 0, 0, 0 },
    { 0, 0, 0, 0 },
    { 0, 0, 0, 0 }
};
```

Notice we have five (5) series of '{ 0, 0, 0, 0 }'. Each one of those initializes a single row of 'mdarr' to all zeroes. Every flat initialization is followed by a comma, and the whole thing is contained in a set of curly braces. Notice that each individual list of values is not followed by a semi-colon. A semi-colon terminates a statement, thus it comes at extreme end of the entire block. Where semi-colons would be there are now commas. But there are only commas *between* each list of values, there is not one after the last list.

Author's Opinion: Initializing multidimensional arrays can be truly confusing. At two dimensions it can be grasped easily with practice. At three dimensions, visualizing what you are doing becomes hideously difficult and at four and beyond can get lost in an alien world.

Pointer Pointer Arrays

Pointers have a unique relationship with arrays: they can represent them completely. This is true with multidimensional arrays as well as flat ones. Using a single index with a two-dimensional array specifies a single row of values. That is, the value of that subscript is a flat array. This flat array can be stored in a pointer:

```
int stuff[5][5];
int *p = stuff[0];
```

The above code causes 'p' to point to the first value of the first row of 'stuff'. The pointer 'p' can now be used as if it was a flat array:

```
p[0] = 7;
cout << p[0] << endl;
```

And pointer arithmetic can also be applied to it to move through the row:

```
p++;
*p = 8;
p++;
*p = 9;
```

This has been seen before. The concept of a pointer to a list of values is interesting, but what about a pointer to a pointer of a list of values? A pointer pointer is just that. It is basically the pointer version of a two dimensional array and can be declared by preceding the pointer name with an additional asterisk:

```
int **pp;
```

A pointer pointer is practically equivalent to a two-dimensional array and the two can associated with one another:

```
pp = stuff;
pp[0][0] = 10;
```

The pointer pointer can be used as if it was a two-dimensional array. It can also be used as a flat array if it is dereferenced:

```
(*pp)[0] = 10;
```

A pointer pointer can take the place of a multidimensional array in function parameters as well and it is easier to pass than an array. A pointer pointer is identical to a normal pointer: its value is a memory address. However, the value of a pointer pointer is the address of another pointer variable. The address contained by *that* pointer variable is the location of an actual value.

To be continued …

Functions and Multidimensional Arrays

When I was first writing this chapter I got an email from someone who was trying to pass a multidimensional array to a function that accepted one. His compiler was giving him strange errors; as it did me when I tried it for myself. The reason was that multidimensional arrays must be treated uniquely when used as parameters to functions. The following is *not* valid:

```
void takemulti(int arr[][]);
```

It looks like it validly takes a two-dimensional array parameter called ‘arr’. I’m sure there’s a reason for it, but this is not technically legal. There are a few ways to remedy the problem.

```
void takemulti(int arr[][10]);
void takemulti(int *arr[]);
void takemulti(int **arr);
```