2-D Arrays

We define 2-D arrays similar to 1-D arrays, except that we must specify the size of the second dimension. The following is how we can declare a 5x5 int array:

int grid[5][5];

Essentially, this gives us a 2-D int structure that we index using two indexes instead of one. Thus, the following would set the location in row 0, column 0 to 0:

grid[0][0] = 0.

Of course, to set each grid location to 0, we have to use a loop structure as follows (assume i and j are already defined):

```
for (i=0; i<5; i++)
  for (j=0; j<5; j++)
    grid[i][j] = 0;
```

Typically, we think of the first index as the row in the array and the second index as the column into the array. But, it's not necessary to think of it that way. The real thing that matters is that you use an array you declare consistently. If you initialize it thinking the first index is a row and the second is a column value, then when you access the array, you must still think that!

Here's an example of what grid could store:

row/col	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	6	8
3	0	3	6	9	12
4	0	4	8	12	16

We can store these values in grid as follows:

```
for (i=0; i<5; i++)
  for (j=0; j<5; j++)
    grid[i][j] = i*j;
```

Just like with 1-D arrays, it will be typical for us to loop through all the elements in a 2-D array using a double loop structure like above. Also, we will likely want to do some sort of manipulation with each array index. Sometimes, as is the case above, that manipulation will be based upon the values of the loop indexes, so it's important to figure out how the loop indexes relate to what you might be trying to do with your array. In the example above, we are storing a multiplication table.

Difference between the first and second dimension

Although it's not necessary to think about the rows and columns as I mentioned previously, there is a difference between the first and second indexes into an array. In particular, consider an example where the two dimensions are of different sizes:

```
int grid[3][2];
```

One way to think about this is that this allocates 3 integer arrays of size 2, each. It does NOT allocate 2 integer arrays of size 3. (Thus, there are three rows, and two columns. Each row is of size 2 while each column is of size 3.)

Furthermore, these 3 integer arrays are located contiguously in memory.

Namely, in memory, here is the order that they are stored:

grid[0][0], grid[0][1], grid[1][0], grid[1][1], grid[2][0], grid[2][1]

Thus, "rows" are stored contiguously, but columns are not. What this means is that we can refer to a 1-D array that is a single row of a 2-D array as follows:

grid[1]

but that there is no way to refer to a 1-D array that stores a single column of a 2-D array, in the typical sense.

2-D Arrays as Parameters to Functions

With a one-dimensional array, it is NOT necessary to specify the size of the array in the formal parameter. Thus, the following is valid function prototype that specifies a function that takes in a one-dimensional array:

```
void printArray(int vals[], int length);
```

However, the same is not exactly true for a 2D array. For a 2D array, you MUST specify the size of the second dimension of the array as follows:

```
void printArray(int values[][2], int len);
```

The reason for this is that technically, in both examples, the array values is really just a pointer to the memory location where the zero index of the array is stored. In the function, we an element of a 1-D array is accessed, such as vals[4], the computer "knows" to look 4 memory addresses AFTER where vals points. BUT, with a 2-D array, if we attempt to access values[2][1], the computer CAN'T "know" how many memory addresses after values to look. The reason for this is that index 2,1 of the array means that there are 2 full rows before this element, along with 1 element. To determine how many elements this is, we NEED to know the length of a row. The length of a row is simply the number of columns in the 2-D array, the size of the second dimension of the array. Using our example above, since each row is of size 2, values[2][1] is exactly 2*2+1 = 5 memory addresses after the pointer values.

To display one more example, consider an array declared as follows:

```
int X[4][5];
```

Consider accessing the element X[2][4]. We just need to know that there are 5 values in each row (this is the size of the second dimension and can then calculate the appropriate offset: 2*5 + 4 = 14. We can verify this by listing out the 14 elements prior to X[2][4] in memory:

X[0][0], X[0][1], X[0][2], X[0][3], X[0][4],
X[1][0], X[1][1], X[1][2], X[1][3], X[1][4],
X[2][0], X[2][1], X[2][2], X[2][3]

Pascal Triangle Example

Let's conclude by writing a program that prints out Pascal's Triangle. You may remember this from math class. It's a triangle of values that if formed by creating each element in the subsequent row as the sum of two elements in the previous row. Here are the first few rows:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
```

Each number is simply the sum of the two numbers above it to the left and right, while the first and last number on each row is 1, and each subsequent row contains one number more than the previous one.

In an array, we would store the zeroth row in the array location:

```
pascaltri[0][0].
```

We'd store the first row in the locations:

```
pascaltri[1][0], pascaltri[1][1]
```

We'd store the second row in the locations:

```
pascaltri[2][0], pascaltri[2][1],
pascaltri[2][2]
```

etc.

Basically, about half of our array slots will go unused because the first index into the array is always greater than or equal to the second index. (E.g. nothing will be stored in pascaltri[1][3].)

In order to print out this triangle, we will first fill in all of the 1's in the appropriate array locations.

Then we will fill out the rest of the array, top to bottom, as we would fill in the triangle by hand.

To print the triangle, we'll simply go through the whole array row by row, and print each value on each row out before advancing to the next row.

The implementation is on the next page:

```
#include <stdio.h>

int main() {

  int pascaltri[11][11];
  int r, c;

  // Initialize beginning and end of each
  // row to 1.
  for (r=0; r<11; r++) {
    pascaltri[r][0] = 1;
    pascaltri[r][r] = 1;
  }

  // Fill in the table.
  for (r=2; r<11; r++)
    for (c=1; c<r; c++)
      pascaltri[r][c] = pascaltri[r-1][c-1]+
                        pascaltri[r-1][c];

  // Loop through each row of the table.
  for (r=0; r<11; r++) {

    // Loop through each value of the row.
    for (c=0; c<=r; c++)
      printf("%4d ", pascaltri[r][c]);

    // Go to the next line.
    printf("\n");
  }

  return 0;
}
```

Magic Square Example

A Magic Square of size n by n is a square that contains each of the numbers 1, 2, 3, ..., n^2 exactly once and has the sum of the numbers in each of its rows, columns and diagonals equal to the same thing. (If you do a bit of math, you find out that each row, column and diagonal must add to $n(n^2 + 1)/2$.)

We will develop a program that reads in a potential 3x3 magic square and then determines whether or not it is indeed a magic square.

Here are our major tasks:

(1) Reading in the values of the square into a 3x3 integer array.

(2) Checking to see if each number from 1 through 9 is stored in the square exactly once.

(3) Checking if the sum of each row, column and diagonal is equal to 15.

The first can be done with a double for loop, which reads each item into successive spots in our 3x3 array.

The second task itself needs an auxiliary array which stores how many of each number from 1 through 9 we've seen. This is a frequency array. In the beginning our frequency array will look like this:

index	1	2	3	4	5	6	7	8	9
value	0	0	0	0	0	0	0	0	0

In order for our values to create a real Magic Square, we must have the frequency array look like this after we go through our numbers:

index	1	2	3	4	5	6	7	8	9
value	1	1	1	1	1	1	1	1	1

In particular, we must look into each cell of our magic square, take that number, and add one to our index in our frequency array. (Essentially, the value stored in one array is used as an index into the other array.)

Finally, if it turns out that each number is stored in the square exactly once, we must turn to task #3.

We have to check the following:

3 rows
3 columns
2 diagonals

One way to do this would be eight separate if statements.

While this is feasible, we would like a solution that would be easy to extend to 4x4 magic squares, or larger magic squares.

Our key idea will be to use a loop. The basic set up will be as follows:

```
for (i=0; i<3; i++) {
  // Checks if row i adds up to 15.
}
```

Now, our problem boils down to determining if row i does add up to 15. Here's how to do it:

```
sum = 0;
for (j=0; j<3; j++)
  sum = sum + square[i][j];

if (sum != 15)
  checksquare = 0;
```

Note: when we set checksquare to 0, we are indicating that our square is NOT a Magic Square. Here are the three row checks put together:

```
for (i=0; i<3; i++) {

  // Checks if row i adds up to 15.
  sum = 0;
  for (j=0; j<3; j++)
    sum = sum + square[i][j];

  if (sum != 15)
    checksquare = 0;
}
```