CS 162C++
Parallel and Multi-Dimensional Arrays

Thus far, we have talked about simple 1-dimensional arrays. In this document, we will first consider parallel arrays, then we will look at multi-dimensional arrays; particularly focusing on 2-D arrays

Parallel Arrays

Sometimes we have several groups of things that are related. For example, we might have an array that contains the names of the students in the class and another array that contains their grades. Or we might have an array of the names of the items we have in inventory, another one of how many we have of each, and a third that contains the prices for each item. These are all examples of parallel arrays.

Lets first look at the example of names and grades. If we define two arrays as follows:

```
const int SIZE = 6;
string names[SIZE] = {"Ann", "Bob", "Chris", "Dean", "Egbert", "Freda"};
int grades[SIZE] = {98, 85, 75, 90, 65, 100};
```

Then we could imagine the arrays in memory as follows:

Names	Grades
Ann	98
Bob	85
Chris	75
Dean	90
Egbert	65
Freda	100

You can see that the index value of 2 is the same for Chris and her grade of 75. This is how parallel arrays work. You can look up a name to see what grade they got as in the following code snippet:

```
string name;
cout << "What name do you want? ";
cin >> name;
bool found = false;
for(int i = 0; i < SIZE; i++)
    if ( names[i] == name )
    {
        cout << name << " has a grade of " << grades[i] << endl;
        found = true;
        break;
    }
if ( !found )
    cout << "There was no student with that name, sorry." << endl;
```

Or display a table of names and grades like this:

```
cout << setw(10) << "Names" << setw(6) << "Grades" << endl;
for(int i = 0; i < SIZE; i++)
    cout << setw(10) << names[i] << setw(6) << grades[i] << endl;
cout << endl;
```

For practice with parallel arrays, look at the in-class exercise of Chips and Salsa.

Multi-dimensional Arrays

The simplest way to consider a two-dimensional array is to think of a table with rows and columns like this:

	0	1	2	3	4	5	6
0	12	22	32	42	52	62	72
1	00	11	22	33	44	55	66
2	1	3	5	7	9	11	13
3	2	4	8	16	32	64	128
4	3	9	27	81	243	729	2187
5	91	87	73	61	59	43	37

This is an array with 7 columns and six rows. You could define it with the following statement:

```
const int ROWS = 6;
const int COLS = 7;
int values[ROWS][COLS] = { {12, 22, 32, 42, 52, 62, 72},   {00, 11, 22, 33, 44, 55, 66},
                           {1, 3, 5, 7, 9, 11, 13},        {2, 4, 8, 16, 32, 64, 128},
                           {3, 9, 27, 81, 243, 729, 2187}, {91, 87, 73, 61, 59, 43, 37}};
```

In this example, **`values[3][2]`** is **8** and values**`[5][1]`** is **87.**

Note that in writing the initialization list each row is delineated with {} and the rows are lined up to make it easier to see how they are organized. While this is not necessary, it makes your code easier to review and maintain. Just like using constants and comments helps. If the initializer list had not had six rows defined or if some of the rows had less than seven elements, the non-specified elements would be set to zero; just as in a 1D array.

Accessing 2D Arrays

When you want to loop through a 2D array to access all its values, for example to output it, you use nested loops as in the following code snippet:

```
for(int i = 0; i < ROWS; i++)
{
   for(int j = 0; j < COLS; j++)
   {
       cout << setw(5) << values[i][j];
   }
   cout << endl;
}
cout << endl;
```

Note that I used the constants from the above definitions so that my loops would go through the entire array. If I had wanted to only display part of the array, I would have used values less than `ROWS` or `COLS`.