Multidimensional arrays

- so far we have just considered one-dimensional arrays: a sequence of N elements of the same type
- we can also create multi-dimensional arrays
- two dimensional arrays are the most common, and are often used to represent tables, grids, or matrices
- arrays with three or more dimensions are less common, but can be useful in the right circumstances
- we need to consider declaration and access syntax, and address some compilications with respect to parameter passing

Two dimensional arrays

- the simplest way to think of 2D arrays is as a table, e.g. M rows of data, with N columns in each row
- we declare the array by specifying the number of rows and columns, e.g.

```
const int Rows = 3;
const int Cols = 5;a
float data[Rows][Cols];
```

- data is an array of 3 rows by 5 columns, each entry containing one float (15 floats in all)

Accessing elements

- we access elements by specifying the position in each dimension, row first, then column
- positions are number starting from 0

```
data[0][0] = 5.1; // first row, first column
data[0][1] = 4.6; // second row, second column
...
data[2][4] = 0.123; // last row, last column
```

Nested loops

- it's common to go through each row and column, one element at a time, e.g.

```
for (int r = 0; r < Rows; r++) {
   for (int c = 0; c < Cols; c++) {
      cin >> data[r][c]; // read data into current elem
   }
}
```

Initializing at declaration

- We can initialize a 2D array at the point of declaration, e.g.

```
int arr[3][4] = {
   { 10, 20, 30, 40 },
   { 6, 3, 1, 9 },
   { 1074, -19, 200, 42 }
};
```

- this can only be done at the point of declaration, and we must have the correct number of rows and columns throughout

Initializing 2d arrays of char

- we can use the “” notation for 2d char arrays, e.g.

```
char text[4][6] = {
     “abcde”,
     “12345”,
     “argh!”,
     “ZYXWV”
};
```

- remember the null terminator in these counts as a char

Common uses

- 2d arrays are often used to store information for things like
  - entries in a spreadsheet
  - text on a page
  - values in a matrix
  - data points on a 2d map

Memory considerations

- If the number of rows and columns gets large, we should be aware of the total memory being used
- size in bytes can be calculated as

  ```
  Rows * Cols * sizeof(float)
  ```

- when we get into arrays with more dimensions the same idea holds:
  - take the product of all the dimensions and multiply by the number of bytes needed for a single element

Passing as parameters

- when declaring a function that will accept a 2d array as a parameter, the syntax is a little different:
  - this time we actually specify the number of columns in the array as part of the parameter, but leave the number of rows empty

```
// for arrays of 10 columns, any number of rows
void print(float arr[][10], int rows);
```

- the number of columns is usually passed as an additional parameter, we still call the function in the same way, e.g.

```
print(data, 5); // assuming data is 5 rows x 10 columns
```

Declaring in structs

- parameter syntax can be simplified by the use of structs:

```
const int Rows = 3;
const int Cols = 5;

struct Table {
   float data[Rows][Cols];
};

void fill(Table &tbl);

int main() {
   Table t;
   fill(t);
}
```

```
void fill(Table &tbl)
{
   for (int r = 0; r < Rows; r++) {
        for (int c = 0; c < Cols; c++) {
             cin >> tbl.data[r][c];
        }
   }
}
```

This example requires the size be fixed across all tables, we'll look at more flexible approaches soon.