The arrays we have been looking at until now have been linear, or sequential, lists. One-dimensional. A *vector* is a one-dimensional array. The word array usually (but not always) refers to a two- or more dimensional array. It is a vector whose elements are vectors. Multi-dimensional arrays are useful for organizing data which is dependent on two or more variables.

If X is a matrix of numbers, it is said to be 2-dimensional. A 2-dimensional array can be thought of as a table of data items, with rows and columns. The 3rd element in the 2nd row is referred to as X(2,3).

Example:

```
//multidimarray.cpp
//modified from Hubbard ex. 5.17, p. 141

#include <iostream>

int main(){
   int x[3][5];
                              //read the 2-dimensional array
   cout << "Enter 15 integers, 5 per row:\n";
   for (int i=0; i<3; i++){
      cout << "Row " << i << ": ";
      for (int j=0; j<5; j++)
         cin >> x[i][j];
   }
                              //print the 2-dimensional array
   cout << "\nYour data matrix contains the following:";
   for (int i=0; i<3; i++){
      cout << endl;
      for (int j=0; j<5; j++)
         cout << "  " << x[i][j];
   }
   cout << "\n\nPress any key to close console window.";
   char c; cin >> c;
   return 0;
}
```

Output:

```
Enter 15 integers, 5 per row:
Row 0: 44 33 87 88 100
Row 1: 22 33 45 75 66
Row 2: 4 6 66 44 110

Your data matrix contains the following:
  44  33  87  88  100
  22  33  45  75  66
  4  6  66  44  110

Press any key to close console window.
```

A 3-dimensional array would be declared with three dimensions, e.g.:

```
int x [25][10][5];
```

and three for loops with three index variables (e.g. i, j, k) would be used for processing it. A 3-dimensional array can be thought of as a collection of tables, like pages in a book.

NOTE: When a multidimentional array is an argument to a function, only the size of the first dimension can be missing. The others must be specified in the parameter list, e.g.,

```
int sumMatrix (int x[][100], int rows, int cols);
```

Examples:

Two-level array – Rating Table – Insurance rates by age and job classification:

		Job Classification			
Age		Class 1 [0]	Class 2 [1]	Class 3 [2]	Class 4 [3]
18-34	[0]	23.50	25.25	27.05	52.90
35-39	[1]	24.00	35.75	27.55	53.40
40-44	[2]	24.60	36.35	28.15	54.00
45-49	[3]	25.30	37.05	28.85	54.70
50-54	[4]	26.30	38.05	29.85	55.70
55-59	[5]	28.00	39.75	31.55	57.40

Three-level array – Rating Table – Insurance rates by age, sex, and job classification:

		Men [0]		Women [1]	
		Job Classification		Job Classification	
Age		Class 1 [0]	Class 2 [1]	Class 1 [0]	Class 2 [1]
18-34	[0]	23.50	25.25	27.05	52.90
35-39	[1]	24.00	35.75	27.55	53.40
40-44	[2]	24.60	36.35	28.15	54.00
45-49	[3]	25.30	37.05	28.85	54.70
50-54	[4]	26.30	38.05	29.85	55.70
55-59	[5]	28.00	39.75	31.55	57.40

Sparse arrays

e.g.:

0	0	0	0	1	0	0	2	0	0
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	4	0	0	0	0	0	0
0	0	0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0

The problem here is that there is a lot of wasted space. Consider, a 1000 by 1000 array, 1 million elements and, say, 1500 are nonzero. What is a more efficient way to represent this data?

How about an array of triplets, or 3 vectors:

Row[i]	Column[i]	Value[i]
1	5	1
1	8	2
2	2	1
3	1	1
5	4	4
6	8	2
8	1	2
8	2	1

Or – even better – a record (struct) or object.

Practice Assignment: Read in triplets (as records). Print out original matrix. Print our transpose. Do not at any time store the entire 8 x 10 array in main memory.