CS 11 Data Structures and Algorithms

Lesson 10: 2-D Arrays

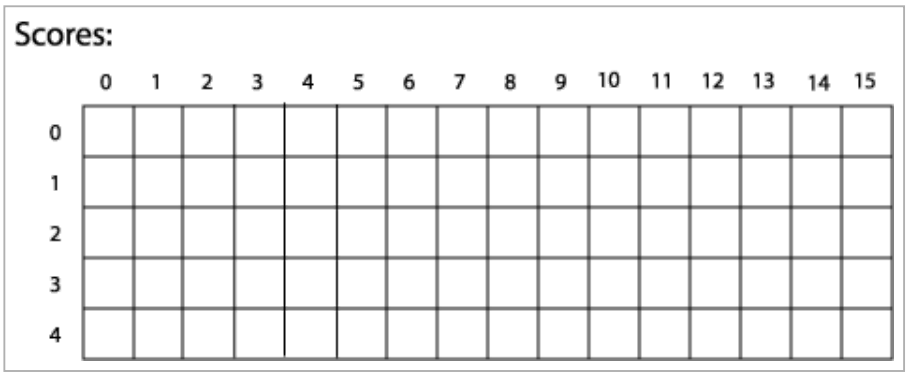
Skip to Main Content

Section 10.1: Introduction to 2-D Arrays

Often it is more convenient to store data in a 2-dimensional structure instead of a 1-dimensional structure. Data that would normally be displayed in a table format usually falls into this category. For example, scores for all of the students in a class would naturally be stored in a table, with a list of students along the left side of the table and a list of assignments and exams along the top of the table. To store data for a class with 5 students (I've picked a small class size to keep the figures small) and 16 assignments/exams, we would declare a 2-dimensional array like this:

```
int scores[5][16];
```

This declaration would allocate memory for a structure that we would then picture like this:

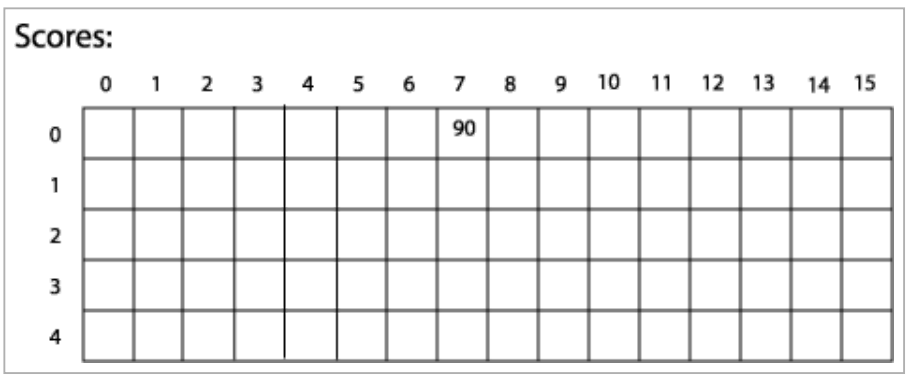


(Notice that we have not stored the names of the students or the descriptions of the assignments/exams. A common approach to this situation is to introduce a 1-dimensional array to store the names of the students and a second 1-dimensional array to store the descriptions of the assignments/exams.)

Then, for example, to indicate that student 0 scored 90 points on assignment 7 we might have a statement like this:

```
scores[0][7] = 90;
```

and we would picture the new situation like this:



Although in most cases it is best to think of 2-dimensional arrays in this way, the truth is that a 2-dimensional array in C++ is implemented as an array of arrays. In our example, we have declared a 1-dimensional array with 5 elements. The base type of this array is a 1-dimensional array with 16 elements; in other words, each element of this array is a 1-dimensional array with 16 elements. To say this yet another way, each row of the structure pictured above is actually one element in a 1-dimensional array. While it is usually best to ignore this underlying implementation detail and treat arrays such as `scores` simply as 2-dimensional arrays (because it is simpler to view them this way), there are some situations in which you will need to remember that it is actually an array of arrays. For example, suppose you have a function named `sumElements` which takes a (1-dimensional) array of integers as an argument and sums the integers in the array. You might want to call this function, sending the scores in the first row of our `scores` array as the argument. Because a 2-dimensional array is actually an array of arrays, you can refer to row 0 or row 1 of the `score` variable by saying `scores[0]` or `scores[1]` (respectively). The function call might look like this:

```
sumElements(scores[0]);
```

We will spend most of this lesson developing a (relatively) large case study example. Before we do we need to discuss the rules for using 2-dimensional arrays as arguments and parameters. For the most part the same rules that applied to 1-dimensional arrays will apply here. As with 1-dimensional arrays, when we want to pass an entire 2-dimensional array as an argument, we simply put the name of the array in the argument list. Like 1-dimensional arrays, 2-dimensional arrays are automatically passed by reference, so we will never use an ampersand (&) with a 2-dimensional array. The rule for using a 2-dimensional array as a parameter, however, is a little more complex. When you use a 2-dimensional array as a parameter, you can leave the first set of square brackets empty as you did with 1-dimensional arrays; however, you must include the correct size in the second set of square brackets. The reason for this is as follows: With 1-dimensional arrays, the compiler does not need to know the size of the array because each element is located by simply adding the index to the base address of the array. For example, if we have a 1-dimensional array named `list`, the compiler can locate `list[4]` by finding the location where `list` begins (the "base address"), and then moving over 4 elements. With 2-dimensional arrays the situation gets a bit more complex. Remember that a 2-dimensional array is actually an array of arrays, and that the computer's memory is 1-dimensional, not 2-dimensional. So when the 2-dimensional array is stored in memory it is stored one row at a time: first `scores[0]` is stored, followed by `scores[1]`, then `scores[2]`, and so on. How will the compiler know how to locate (for example) `scores[1][5]`? The compiler will have to first move over 1 entire row to get to the beginning of row 1, and then move over 5 additional elements to get to the correct element within row 1. But to skip over 1 entire row the compiler will need to know how many elements are in each row. This is why we must include the correct size in the second set of square brackets in the parameter list. This constant indicates the number of elements in each row.

The good news is that you can probably survive for awhile without understanding the details in the last paragraph. For now just make sure you can apply the basic rule: When you use a 2-dimensional array as a parameter, you can leave the first set of square brackets empty, but you must include the correct size in the second set of square brackets. You will see this rule applied in the following example.

Section 10.2: Votes Example

The rest of this lesson will be spent developing an extended example that uses a 2-dimensional array as well as a 1-dimensional array. Note to online students: The best way to learn the most from this type of lesson is to read the specifications of the program first, and then try to write the program yourself before moving on. When you have finished the program, or feel that you are completely stuck, go ahead and take a look at the first piece of code developed (which will be `main`). Use the `main` function developed here, and then go write the program again. Continue in this manner, taking one piece of code at a time from the lesson.

Our program will do an analysis of an election that involved 4 candidates receiving votes in 4 different precincts. The input will be an arbitrary number of votes stored in a file, with each vote represented as a pair of integers: a precinct number (1 through 4) and a candidate number (1 through 4). In addition, we will ask the user to enter the names of the candidates from the keyboard. For example, here is one possible input file (the pairs are displayed here in three columns to save space even though in the actual file they will all be in a single column):

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

Our program will begin by asking the user to enter the names of the candidates:

```
Enter name of candidate #1: Jones
Enter name of candidate #2: Smith
Enter name of candidate #3: Adams
Enter name of candidate #4: Smiley
```

It will then produce a report which it will write to an output file. Assuming we use the input file illustrated above, the report should look like this:

```
                Jones     Smith     Adams     Smiley
Precinct  1         2         2         1          1
```

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

Total votes for      Jones: 6
Total votes for      Smith: 7
Total votes for      Adams: 8
Total votes for     Smiley: 9

Total votes for precinct 1: 6
Total votes for precinct 2: 7
Total votes for precinct 3: 6
Total votes for precinct 4: 11
```

What data structures will we need? (For now, you can think of a "data structure" as a complex variable, such as an array, struct, or class object). First, we will want a 2-dimensional array to store the number of votes received by each candidate in each precinct. We will picture this 2-dimensional array as a table similar to the first piece of the output shown above except that, of course, the row and column labels will not appear in the 2-dimensional array. Instead, the rows and columns will simply be numbered from 0 to 3.

Second, we will need a 1-dimensional array to store the list of names. We won't need an array to store the precincts, since they are simply identified by their number.

In addition to declaring these 2 data structures, `main` will have to call functions to read the names, read the votes, and print the report. We will assume the existence of 2 global constants, NUM_PRECINCTS and NUM_CANDIDATES, both of which will be 4 in our example. Notice that in the code below we have placed the number of precincts in the first square brackets and the number of candidates in the second. This is because rows are by convention listed before columns when we deal with 2-dimensional arrays. Here is our `main` function:

```
int main()
{
    int votes[NUM_PRECINCTS][NUM_CANDIDATES];
    string names[NUM_CANDIDATES];

    readNames(names);
    readVotes(votes);
    printReport(names, votes);
}
```

The definition of the `readNames` function is straightforward. The only thing to watch out for is that we must use the getline function rather than the extraction operator to read our names so that we allow names with spaces (for example, first and last name). The code follows:

```
void readNames(string names[])
{
    for (int count = 0; count < NUM_CANDIDATES; count++){
        cout << "Enter name of candidate #" << count+1 << ": ";
        getline(cin, names[count]);
    }
}
```

In the succeeding sections we will develop the `readVotes` and `printReport` functions.

Section 10.3: Votes Example -- `readVotes`

In our `readVotes` function we must start with the preliminary work of opening the file where the votes are stored and initializing the 2-dimensional array of `ints` to reflect the fact that each candidate starts out with 0 votes from each precinct. We will handle this initialization in a separate function named `initVotes`. Then we read a vote (i.e. a precinct - candidate pair), and as long as we haven't reached the end of the file we increment the appropriate element in the votes array and read the next vote. In order to increment the appropriate element in the array we need to subtract 1 from the values that we read in, since the candidates and precincts are numbered from 1 to 4 but the array elements are numbered from 0 to 3. Here is the code:

```
void readVotes(int votes[][NUM_CANDIDATES])
{
    ifstream infile("votes.dat");
    int precinct, candidate;

    initVotes(votes);
    infile >> precinct >> candidate;
    while (infile) {
        votes[precinct-1][candidate-1]++;
```

```
            infile >> precinct >> candidate;
        }
        infile.close();
    }
```

The `initVotes` function must set each element of the array to 0. This is an important example because it illustrates an important and common technique for visiting each element of a 2-dimensional array (this is called "traversing" the array). We will start by writing a for loop which will visit each row of the array:

```
    for (int row = 0; row < NUM_PRECINCTS; row++){
        [process one row of the 2-dimensional array]
    }
```

How will we process one row of the array? We will visit each element within that row, using a for loop. Placing this second for loop in the place of the words "process one row of the 2-dimensional array" results in this code:

```
    for (int row = 0; row < NUM_PRECINCTS; row++){
        for (int col = 0; col < NUM_CANDIDATES; col++){
            [process one element of the 2-dimensional array]
        }
    }
```

In the case of the `initVotes` function, processing one element of the array involves setting it to 0.

It turns out that this task of traversing a 2-dimensional array is so common that we will normally not concern ourselves with how this code was developed and just remember that anytime we need to traverse a 2-dimensional array we need nested for loops with `row` and `col` as the loop indices.

The final version of `initVotes` follows. Notice that all we have done is replaced the words "process one element of the 2-dimensional array" with the statement

```
>votes[row][col] = 0;
```

because that is what we want to do to each element of the array in this case.

```
    void initVotes(int votes[][NUM_CANDIDATES])
    {
        for (int row = 0; row < NUM_PRECINCTS; row++){
            for (int col = 0; col < NUM_CANDIDATES; col++){
                votes[row][col] = 0;
            }
        }
    }
```

We now need to write the `printReport` function. Since it prints out three independant tables, we will decompose this function into three subroutines:

```
    void printReport(const string names[],
                     const int votes[][NUM_CANDIDATES])
    {
        ofstream outfile("votes.out");
        printTable(votes, names, outfile);
        printCandidateTotals(votes, names, outfile);
        printPrecinctTotals(votes, outfile);
    }
```

In the next section of this lesson you will find the complete code for the votes example. I strongly suggest that you attempt to write the `printTable`, `printCandidateTotals`, and `printPrecinctTotals` functions before looking at the solution given there. There are some tricky formatting issues involved (such as getting the columns to line up). Try not to get bogged down trying to get these right. The more important issues are being able to use the arrays correctly, including summing up rows and columns. You will find in each of these functions that the nested for loop pattern that we saw in the `initVotes` function is there, but in each case additions have been made to the basic pattern in order to handle the specific situation encountered.

Section 10.4: Votes Example -- Complete Code

```
#include <iostream>
#include <fstream>
#include <iomanip>
```

```
using namespace std;

const int NUM_PRECINCTS = 4;
const int NUM_CANDIDATES = 4;
const int COLUMN_WIDTH = 10;

void readVotes(int votes[][NUM_CANDIDATES]);
void initVotes(int votes[][NUM_CANDIDATES]);
void readNames(string names[]);
void printPrecinctTotals(const int votes[][NUM_CANDIDATES],
                         ofstream &outfile);
void printCandidateTotals(const int votes[][NUM_CANDIDATES],
                          const string names[],
                          ofstream &outfile);
void printTable(const int votes[][NUM_CANDIDATES],
                const string names[],
                ofstream &outfile);
void printReport(const string names[], const int votes[][NUM_CANDIDATES]);



int main()
{
    int votes[NUM_PRECINCTS][NUM_CANDIDATES];
    string names[NUM_CANDIDATES];

    readNames(names);
    readVotes(votes);
    printReport(names, votes);
}



void readNames(string names[])
{
    for (int count = 0; count < NUM_CANDIDATES; count++){
        cout << "Enter name of candidate #" << count+1 << ": ";
        getline(cin, names[count]);
    }
}



void readVotes(int votes[][NUM_CANDIDATES])
{
    ifstream infile("votes.dat");
    int precinct, candidate;

    initVotes(votes);
    infile >> precinct >> candidate;
    while (infile) {
        votes[precinct-1][candidate-1]++;
        infile >> precinct >> candidate;
    }
    infile.close();
}



void initVotes(int votes[][NUM_CANDIDATES])
{

    for (int row = 0; row < NUM_PRECINCTS; row++){
        for (int col = 0; col < NUM_CANDIDATES; col++){
            votes[row][col] = 0;
        }
    }
}



void printReport(const string names[], const int votes[][NUM_CANDIDATES])
{
    ofstream outfile("votes.out");
    printTable(votes, names, outfile);
    printCandidateTotals(votes, names, outfile);
    printPrecinctTotals(votes, outfile);
}



void printTable(const int votes[][NUM_CANDIDATES],
                const string names[],
                ofstream &outfile)


{
```

```
    outfile << setw(11) << "";
    for (int count = 0; count < NUM_CANDIDATES; count++){
        outfile << setw(COLUMN_WIDTH) << names[count].substr(0,COLUMN_WIDTH - 1);
    }
    outfile << endl;



    for (int row = 0; row < NUM_PRECINCTS; row++){
        outfile << "Precinct " << setw(2) << row+1;
        for (int col = 0; col < NUM_CANDIDATES; col++){
            outfile << setw(COLUMN_WIDTH) << votes[row][col];
        }
        outfile << endl;
    }
    outfile << endl;
}



void printCandidateTotals(const int votes[][NUM_CANDIDATES],
                          const string names[],
                          ofstream &outfile)

{
    int sum;

    for (int col = 0; col < NUM_CANDIDATES; col++){
        sum = 0;
        for (int row = 0; row < NUM_PRECINCTS; row++){
            sum += votes[row][col];
            //sum = sum + votes[row][col];
        }

        outfile << "Total votes for " << setw(10)
                << names[col].substr(0,9) << ": " << sum << endl;

    }
    outfile << endl;
}



void printPrecinctTotals(const int votes[][NUM_CANDIDATES],
                         ofstream &outfile)

{
    int sum;

    for (int row = 0; row < NUM_PRECINCTS; row++){
        sum = 0;
        for (int col = 0; col < NUM_CANDIDATES; col++){
            sum += votes[row][col];
            //sum = sum + votes[row][col];
        }

        outfile << "Total votes for precinct " << row+1
                << ": " << sum << endl;

    }
    outfile << endl;
}
```