# CSI 1430
# Week 7 Resource

Colin Burdine

2/28/21

---

**Major Topics:**

1. Advanced Arrays/Vectors
2. Functions

**Keywords**: *arrays, vectors, STL, functions, header file, source file*

---

*Reminder*: Group tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

## 1 Notes on Major Topics:

### 1.1 Advanced Arrays/Vectors

Last week we introduced arrays and vectors in C++. Both of these data structures are useful for storing sequential values of the same type; however, arrays are more memory-efficient because they are of a fixed size, while vectors are less memory-efficient but can grow in size to accommodate more elements. This week we will be talking about some additional functionality of arrays and vectors. In particular, we will focus on 2D arrays/vectors.

2D arrays may sound complicated, but how they function is not conceptually difficult. In fact, a 2D array in C++ is only an *array of arrays*. (Likewise for vectors, a 2D vector is simply a *vector of vectors*.) 2D arrays are useful because they allow us to retrieve data of a certain type from a table. The syntax for creating 2D arrays and vectors is shown below:

```
// this creates a 12x12 array of doubles:
double my2DArray[12][12];


// this creates a 12x12 array of vectors:
vector<vector<double>> my2DArray =
    vector<vector<double>>(12,vector<double>(12));
```

We can modify or access the element in row $r$ and column $c$ using the same square bracket notation as we do with 1D arrays, (i.e. `my2DArray[r][c]`), treating the 2D array just as if it was an array of arrays. Typically, we will want to iterate over each element in these arrays to perform some kind of data entry or data processing. We can do that with nested `for` loops that iterate over each (row, column) pair. As an example, consider the program below, which prints out a difference table, where `diffTable[r][c]` stores the value of `r - c`:

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

using namespace std;

int main(){

    // initialize difference table:
    const int TABLE_SIZE = 10;
    int diffTable[TABLE_SIZE][TABLE_SIZE];

    // populate the entries of the difference table:
    for(int r = 0; r < TABLE_SIZE; ++r){
        for(int c = 0; c < TABLE_SIZE; ++c){
            diffTable[r][c] = (r - c);
        }
    }

    // print the difference table:
    for(int r = 0; r < TABLE_SIZE; ++r){
        for(int c = 0; c < TABLE_SIZE; ++c){
            // use setw to print numbers with equal spacing:
            cout << setw(5) << diffTable[r][c];
        }
        cout << endl;
    }


    return 0;
}
```

## 1.2 Functions

In C++, *functions* are a way of grouping together code segments or operations that may be executed with similar data types. For example, you may have used functions like `sqrt(x)` from the `cmath` library to calculate the square root of a double value. We can actually define our own functions to facilitate writing cleaner and more robust code. However, when we define our function, we must define it in such a way that the compiler can recognize invocations of your defined function. This includes declaring two things in your C++ code, a *function prototype* and a *function body*. The function prototype simply declares to the compiler that your function exists and may be called somewhere throughout your source code. The function body specifies the actual code that is executed upon an invocation of your function.

Furthermore, a function may also have a *return type*, which specifies the type that a call to a function results in. If a function returns no type (i.e. it only performs operations on its parameters and does not evaluate to anything, then we state the function's return type as *void*.

For example, you have probably used the declaration of the `main` function in your C++ code several times:

```
int main(){ ...  }
```

Note that the main function is a special function that is always called first when a program executes. In C++, it returns the type `int`. This integer is the exit code. In most cases, you should only return 0 as the exit code.

Now, let's create an example function called `linearFunction`, which takes as parameters a `double m`, an `int b` and a `double x` and returns the value of $y = mx + b$. Normally, we would want to make $b$ a double as well, but we will assume that in this particular case, the user can only specify an integer as the y-intercept $b$. First, we would need to create a special file for our prototype function, which is referred to as a *header file*. Header files are different from regular source files because they use the `.h` file extension. Our header file will only include the function prototype and will be called `linearFunction.h`:

```
1 /* linearFunction.h */
2
3 double linearFunction(double m, int b, double x);
```

Next, we will create a separate source file the function body of `linearFunction`, called `linearFunction.cpp`. Inside this file, we will include the header file we just created so that the compiler knows that the function exists:

```cpp
/* linearFunction.cpp */

// include the function prototype:
#include "linearFunction.h"

double linearFunction(double m, int b, double x){
    double y = (m*x + static_cast<double>(b));
    return y;
}
```

Finally, we can create a main source file (we'll call it `main.cpp`) in which we use our function. However, in order to use it, we will need to include the header file, so the compiler knows that the function `linearFunction` exists. Our main source file is shown below:

```cpp
/* main.cpp */
#include <iostream>

// include the function prototype:
#include "linearFunction.h"

int main(){
    double m, x, result;
    int b;

    // prompt the user for m, b, and x:
    cout << "Enter a value for m: ";
    cin >> m;
    cout << "Enter an integer value for b: ";
    cin >> b;
    cout << "Enter a value for x: ";
    cin >> x;

    // call our function:
    result = linearFunction(m, b, x);

    // print the result:
    cout << "The result is: " << result << endl;

    return 0;
}
```

Note that above, we can call our user-defined function just like any other function, so long as we supply the correct types (in this case the types were (`double`, `int`, `double`). One other think that is worth mentioning: **Never include a `.cpp` file; only include `.h` files!**. Although the compiler will let you include `.cpp` files, you can easily get yourself into trouble when the compiler cannot find a function declaration or starts to complain about undefined references.

## 2 Programming Example Problem

To illustrate how we can use arrays and vectors to solve problems, consider the following example programming problem:

The game of chess is played on an 8x8 checkerboard. The queen pieces are (arguably) the most versatile pieces in the game of chess; they can move and attack other pieces that are in the same row, column, or diagonal on the chessboard. Now consider a scenario where we only have two queen pieces on a chessboard. We want to write a program to (1) display the position of the queens on the chessboard visually, and (2) determine if the queens are in attacking range each other (i.e. in the same row, column, or diagonal of the chessboard). In your program you will prompt the user for the row and column of the first queen and then the second queen. The rows and columns will be indexed starting at 0, and you can only assume that the user will enter valid integers in the range 0 to 7 inclusive to the row and column position of each queen.

To help make the problem easier, you may want to create some functions in the source and header files `chessboard.cpp/.h` respectively. To help get you started, my header file for my user defined functions is given below:

```
1 /* chessboard.h */
2
3 // checks if two queens are in range of eachother:
4 bool isInRange(int a_r, int a_c, int b_r, int b_c);
5
6 // prints an 8x8 chessboard:
7 void printChessboard(char chessboard[][8]);
```

Some examples of the program output are below:

```
Enter the row and column (0-7) of the first queen: 3 4
Enter the row and column (0-7) of the second queen: 6 1
#_#_#_#_
_#_#_#_#
#_#_#_#_
_#_#A#_#
#_#_#_#_
_#_#_#_#
#B#_#_#_
_#_#_#_#
The queens are in attacking range
```

```
Enter the row and column (0-7) of the first queen: 3 6
Enter the row and column (0-7) of the second queen: 2 2
#_#_#_#_
_#_#_#_#
#_B_#_#_
_#_#_#A#
#_#_#_#_
_#_#_#_#
#_#_#_#_
_#_#_#_#
The queens are not in attacking range
```

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to use the empty space below to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week and last week.

```cpp
/*  main.cpp
    your correct header comments go here :)
*/
#include <iostream>
#include "chessboard.h"

using namespace std;

int main(){

    // prompt for the position of first queen:
    int queenA_r, queenA_c;
    int queenB_r, queenB_c;
    const int BOARD_SIZE = 8;
    char chessboard[BOARD_SIZE][BOARD_SIZE];

    // prompt for the position of the first queen:
    cout << "Enter the row and column (0-7) of the first queen: ";
    cin >> queenA_r >> queenA_c;

    // prompt for the position of the second queen:
    cout << "Enter the row and column (0-7) of the second queen: ";
    cin >> queenB_r >> queenB_c;

    // initialize chessboard to B/W squares:
    for(int r = 0; r < BOARD_SIZE; ++r){
        for(int c = 0; c < BOARD_SIZE; ++c){
            if((r+c)%2 == 0){
                chessboard[r][c] = '#';
            } else {
                chessboard[r][c] = '_';
            }
        }
    }

    // set position of queens and print chessboard:
    chessboard[queenA_r][queenA_c] = 'A';
    chessboard[queenB_r][queenB_c] = 'B';
    printChessboard(chessboard);

    // check if queens are in atacking range of eachother:
    if(isInRange(queenA_r, queenA_c, queenB_r, queenB_c)){
        cout << "The queens are in attacking range" << endl;
    } else {
        cout << "The queens are not in attacking range" << endl;
    }

    return 0;
}
```

```h
/* chessboard.h */

// checks if two queens are in range of eachother:
bool isInRange(int a_r, int a_c, int b_r, int b_c);

// prints an 8x8 chessboard:
void printChessboard(char chessboard[][8]);
```

```cpp
/* chessboard.cpp */

#include <iostream>

#include "chessboard.h"

using namespace std;

// checks if queen A is in range of queen B:
bool isInRange(int a_r, int a_c, int b_r, int b_c){

    bool inRange = false;
    // check if queens are in the same row/col:
    if(a_r == b_r || a_c == b_c){
        inRange = true;

    // check if queens are in the same diagonal:
    } else if((a_r+a_c) == (b_r+b_c) || (a_r-a_c) == (b_r-b_c)){
        inRange = true;
    }

    return inRange;
}

// prints a chessboard:
void printChessboard(char chessboard[][8]){
    for(int r = 0; r < 8; ++r){
        for(int c = 0; c < 8; ++c){
            cout << chessboard[r][c];
        }
        cout << endl;
    }
}
```