1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework01, put each programming exercise into its own subdirectory of this directory, zip the entire Homework01 directory up into the file Homework01.zip, and then submit this zip file to Homework #1.

Make sure that you:

1. Follow the coding and documentation standards for the course as published in the MyClasses page for the class.
2. Check the contents of the zip file before uploading it. Make sure all the files are included.

2 Programming Exercises

In these exercises you will be using dynamically allocated array structures exclusively. You are not to create any "program memory" arrays or use any other data structures such as the STL vector, list, etc.

1. Write a program that has the following functions. Use the main program given below to test all of the functions. Make sure that all the memory is cleaned up before the end of the program so that there are no memory leaks, no multiple frees, and no invalid array accesses.

    - div: A function that takes no parameters nor returns anything, simply prints out a division line with blank spaces above and below to make output easier to read.
    - duplicateArray: This function will make a copy of an array and return a pointer to the new array. Specifically, it takes two parameters, a pointer to an integer array and the size of the array. It creates a new array, dynamically allocated, copies the contents to the new array and then returns a pointer to the new array. The parameter array should not be altered in any way.
    - display: This function takes two parameters, a pointer to an integer array and the size of the array and prints the array to the console on a single line with a single space between the entries.
    - getRandomNumbers: This function takes one parameter, the size of the array to be constructed. It creates a new integer array, dynamically allocated, populates it with random integers and returns a pointer to the new array.
    - sort: This function takes two parameters, a pointer to an integer array and the size of the array and sorts the array in ascending order using either the bubble sort, insertion sort, or selection sort.

- sorted: This function takes two parameters, a pointer to an integer array and the size of the array and determines if the array is sorted in ascending order. If it is then the function returns true and if it is not then the function returns false.
- concat: This function takes the pointers of two integer arrays and the two array sizes. It will concatenate the second array onto the first array. So the first array will be altered but the second array is not to altered in any way.
- remove: This function takes four parameters, a pointer to an array, the size of the array, a starting index and an ending index. It will remove all the elements in the array from the starting index up to but not including the ending index. For example, if the array A is

  ```
  1 2 3 4 5 6 7 8 9 10
  ```

  Then the finction call

  ```
  remove(A, 10, 3, 7);
  ```

  would remove the elements at indicies 3, 4, 5, and 6. The parameter array will now have size 6 and contain the following.

  ```
  1 2 3 8 9 10
  ```

  Make sure that you check the validity of the start and end indexes. If the start is greater than or equal to end then there is nothing to remove. If the start is negative you should start removing at the 0 index. If the end is beyond the end of the array you should remove out to the last index. Also, if the start and end values encompass the entire array then the array should be altered to nullptr.
- shuffle: This function takes two parameters, a pointer to an integer array and the size of the array and randomly shuffles the contents of the array. You may use the `random_shuffle` function from the `algorithm` library if you would like for this function.
- sub: This function takes four parameters, a pointer to an array, the size of the array, a starting index, and an ending index. It will change the array to the sub-array consisting of the entries from the starting index up to but not including the ending index. For example, if the array A is

  ```
  1 2 3 4 5 6 7 8 9 10
  ```

  Then the finction call

  ```
  sub(A, 10, 3, 7);
  ```

  would change A to the elements at indexes 3, 4, 5, and 6. The parameter array will now have size 4 and contain the following.

  ```
  4 5 6 7
  ```

Make sure that you check the validity of the start and end indexes. If the start is greater than or equal to end then there is nothing to change. If the start is negative you should start at the 0 index. If the end is beyond the end of the array you should stop at the last index.

- insert: This function takes five parameters, two pointers to integer arrays, their sizes, and an integer index. It will alter the first array to have the second array inserted into it at the position specified by the last parameter. For example, if arrays A and B are the following, with size and sizeB their respective sizes,

  ```
  1 2 3 4 5 6 7 8 9 10
  31 20 22 87 0
  ```

  Then the function call

  ```
  insert(A, size, B, sizeB, 2);
  ```

  would alter A to the following array. The B array should be unaltered.

  ```
  1 2 31 20 22 87 0 3 4 5 6 7 8 9 10
  ```

  Make sure that you check the validity of the position index. If it is negative then B should be inserted at the start of the array A and if it is beyond the end of the array the array B should be added to the end of A.

- resize: This function takes three parameters, a pointer to an integer array, the size of the array, and another integer that will be the new size of the array. The function will resize the array to the new given size. If the new array size is smaller then the entries will be truncated and if the new size is larger the extra entries will be set to 0. For example, if the array A is the following, and size is 10,

  ```
  1 2 3 4 5 6 7 8 9 10
  ```

  The function call

  ```
  resize(A, size, 20);
  ```

  will produce,

  ```
  1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0
  ```

  and the function call

  ```
  resize(A, size, 5);
  ```

  will produce,

  ```
  1 2 3 4 5
  ```

  For a lengthier example, the following main produced the output below.

```
int main() {
    int size = 0;

    cout << "Input array size: ";
    cin >> size;

    int *A = getRandomNumbers(size);

    display(A, size);
    sort(A, size);
    display(A, size);

    if (sorted(A, size))
        cout << "Array is sorted." << endl;
    else
        cout << "Array is not sorted." << endl;

    div();

    delete[] A;
    A = new int[10];
    size = 10;

    for (int i = 0; i < size; i++)
        A[i] = i + 1;
    int *copyA = duplicateArray(A, size);

    display(A, size);
    shuffle(A, size);
    display(A, size);
    cout << sorted(A, size) << endl;
    sort(A, size);
    display(A, size);
    cout << sorted(A, size) << endl;

    div();

    int *B = new int[5];
    int sizeB = 5;

    for (int i = 0; i < sizeB; i++)
        B[i] = rand() % 100;

    display(B, sizeB);
    display(A, size);
    concat(A, size, B, sizeB);
    display(B, sizeB);
    display(A, size);

    div();

    // Reset A to original data.
    delete[] A;
    size = 10;
    A = duplicateArray(copyA, size);

    display(A, size);
    remove(A, size, 3, 7);
    display(A, size);

    div();

    // Reset A to original data.
    delete[] A;
```

```
    size = 10;
    A = duplicateArray(copyA, size);

    display(A, size);
    sub(A, size, 3, 7);
    display(A, size);

    div();

    // Reset A to original data.
    delete[] A;
    size = 10;
    A = duplicateArray(copyA, size);

    display(A, size);
    display(B, sizeB);
    insert(A, size, B, sizeB, 2);
    display(A, size);

    div();

    // Reset A to original data.
    delete[] A;
    size = 10;
    A = duplicateArray(copyA, size);

    display(A, size);
    resize(A, size, 20);
    display(A, size);

    div();

    // Reset A to original data.
    delete[] A;
    size = 10;
    A = duplicateArray(copyA, size);

    display(A, size);
    resize(A, size, 5);
    display(A, size);

    delete[] A;
    A = nullptr;
    delete[] copyA;
    copyA = nullptr;
    delete[] B;
    B = nullptr;

    return 0;
}
```

Output:

```
Input array size: 5
1198147922 458026869 772450542 1895792 231677570
1895792 231677570 458026869 772450542 1198147922
Array is sorted.

--------------------------------

1 2 3 4 5 6 7 8 9 10
2 4 5 8 9 1 3 10 6 7
0
1 2 3 4 5 6 7 8 9 10
```

```
1

--------------------------------

19 64 86 56 28
1 2 3 4 5 6 7 8 9 10
19 64 86 56 28
1 2 3 4 5 6 7 8 9 10 19 64 86 56 28

--------------------------------

1 2 3 4 5 6 7 8 9 10
1 2 3 8 9 10

--------------------------------

1 2 3 4 5 6 7 8 9 10
4 5 6 7

--------------------------------

1 2 3 4 5 6 7 8 9 10
19 64 86 56 28
1 2 19 64 86 56 28 3 4 5 6 7 8 9 10

--------------------------------

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10 0 0 0 0 0 0 0 0 0 0

--------------------------------

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5
```

2. In this exercise you will write a program that will load in a file to an array of structures. The files are csv files (comma separated values), where each consecutive entry is separated by a comma. This is a standard format for spreadsheets and most will allow you to save in csv form. The two files you will be using for testing are `StateData001.csv` and `StateData002.csv`, you may want to open them in a text editor and a spreadsheet (on Linux there is one called LibreOffice Calc). This will show you both the layout of the text you will be reading in and how it would line up in spreadsheet form.

   The data in the `StateData001.csv` file is energy-related carbon dioxide emissions by year in millions of metric tons of energy-related carbon dioxide for each state in the US by year for 1970–2020. The data in the `StateData002.csv` file is the same but for a subset of states and years in the same range.

   Your program should first create a new struct `StateData` which has two fields, a string for the state name and a pointer to a double which will store the array of numeric data for each state. That is, from the data file, each line after the first consists of the name of the state, that gets put into state name field, and after the name there is a list of values for each year for that state, those will be stored into the array that is pointed to by this pointer.

   So there will be an instance of a StateData struct for each state in the file. These will be stored in a dynamically allocated array of StateData types. Since different files

may have different listed states you cannot assume what the size of this array will be without reading the file. You also do not know what years will be listed nor do you know if the years listed will be contiguous or if some will be missing. One thing you can assume is that each state that is listed will have a value of each year that is listed, so there is no missing data.

Once the data is loaded into the program, you will print out a list of states and have the user select one, by number. Then the program will print out a list of all the years in the file and have the user type in the year they want. The program will then output the data for that state and year. The program will also ask the user to input the filename of the data file they want to load. Your program may assume that all data files have the same structure, header line of years, then each line below that a state name followed by a decimal value for each year. Two runs of the program are below,

```
Input the filename: StateData001.csv
Select a State
1. Alabama
2. Alaska
3. Arizona
4. Arkansas
5. California
6. Colorado
7. Connecticut
8. Delaware
9. District of Columbia
10. Florida
11. Georgia
12. Hawaii
13. Idaho
14. Illinois
15. Indiana
16. Iowa
17. Kansas
18. Kentucky
19. Louisiana
20. Maine
21. Maryland
22. Massachusetts
23. Michigan
24. Minnesota
25. Mississippi
26. Missouri
27. Montana
28. Nebraska
29. Nevada
30. New Hampshire
31. New Jersey
32. New Mexico
33. New York
34. North Carolina
35. North Dakota
36. Ohio
37. Oklahoma
38. Oregon
39. Pennsylvania
40. Rhode Island
41. South Carolina
42. South Dakota
43. Tennessee
44. Texas
```

```
45. Utah
46. Vermont
47. Virginia
48. Washington
49. West Virginia
50. Wisconsin
51. Wyoming
Selection: 39

Select a Year
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
Selection: 2012

The energy-related carbon dioxide emissions for Pennsylvania in millions of
```

```
metric tons in the year 2012 was 239.8.
```

```
Input the filename: StateData002.csv
Select a State
1. Alabama
2. Alaska
3. Arizona
4. Arkansas
Selection: 3

Select a Year
1970
1971
1981
1982
1983
1984
1985
1986
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2003
2004
2016
2017
2018
2019
2020
Selection: 1999

The energy-related carbon dioxide emissions for Arizona in millions of
metric tons in the year 1999 was 80.62.
```

Here is a basic outline of the program construction.

(a) Get the filename from the user, if the file does not exist print out an error and exit the program.

(b) Read in the first line of the file, this is the header of the spreadsheet. Find the number of years in the spreadsheet. Remember this has to be done in general, different files may have different years listed. Create a dynamically allocated integer array of the correct size to store all of the years and load the years into the array. You will probably want to use the stoi or atoi functions to convert strings to integers.

(c) Read the rest of the file to determine the number of states that are in the file. Create a dynamically allocated StateData array of the correct size to store all of the state data. At this point you will need to go back to the beginning of the file so you can read it again and populate the StateData array with names and values. Recall that you can do this by either closing the file and repoening it or you can clear the file (resetting the EOF bit) and then seeking the beginning. For example, if the ifstream variable is inputFile, the following will reset the file pointer to the beginning.

```
inputFile.clear();
inputFile.seekg(0L, ios::beg);
```

(d) Now read the file again and for each line extract the state name and data to add to the array of StateData types. For the data, you will create a dynamically allocated double array and store the values. You may want to use the stod or atof functions to convert strings to numeric values. Once the array is loaded set the data pointer in the StateData struct to the array you created. This process is the standard parsing technique of storing the position of the previous comma, using the find function for strings to find the next comma, extract the substring between them, and finally use stod or atof to convert to a double.

(e) Close the file. It will not need to be reread anymore since all your data is in the array.

(f) Print out the list of states with a number beside the state as in the examples above. Have the user select the state by typing in the corresponding number. Error check this input and if the value the user typed in is outside the range have the program ask for input again until a legitimate value is entered.

(g) Then print out a list of years to select from and have the user type in the year they wish to see. Again, error check this and if a year is input that is not in the list have the program ask again until a legitimate year is input.

(h) Finally have the program find the correct data value for the user input and print out a message like the ones above.

(i) Make sure that all the memory is cleaned up before the end of the program so that there are no memory leaks, no multiple frees, and no invalid array accesses.

3. **Optional Exercise for Extra Credit:** This exercise is similar to previous one except that the parsing of the data file is a little more difficult and in this case the arrays that are being stored are not all the same length.

   You will again be working with csv files (comma separated values), where each consecutive entry is separated by a comma. The difference here is that some of the data entries have a comma in them (the formal names). In this case is it common to put double quotes around the data entries.

   The two files you will be using for testing are `MarData001.csv` and `MarData002.csv`. The `MarData001.csv` file is displayed below. As before you may want to open them in a text editor and a spreadsheet to see the layout of the text you will be reading in

and how it would line up in spreadsheet form. The data is fictitious data but is to represent a cross-country team's members and their marathon times. Since each member may have run a different number of marathons the rows of data will not always contain the same number of entries, unlike the data in the previous exercise.

```
"Jones, Martha","3-32-15"
"Noble, Donna","4-1-52","3-59-18"
"Oswald, Clara Oswyn","3-51-22","4-5-19","3-40-15"
"Pond, Amy","4-31-25"
"Potts, Bill","4-52-01","4-43-20","4-5-54","3-58-25","3-42-19"
"Smith, John","4-10-55"
"Smith, Mickey","3-51-8","3-44-10","4-35-1"
"Smith, Sarah Jane","3-12-19"
"Tyler, Rose","4-10-32","4-2-57","3-49-55"
"Williams, Rory","4-25-17","3-39-20","3-35-10","3-30-17"
```

In this file all the entries are in double quotes and are separated by commas, and the formal names have commas in them. The way you would read this is that Martha Jones ran one marathon and her time was 3 hours, 32 minutes, and 15 seconds. Clara Oswyn Oswald ran 3 marathons and her times were 3 hours, 51 minutes, and 22 seconds, 4 hours, 5 minutes, and 19 seconds, and 3 hours, 40 minutes, and 15 seconds respectively.

Your program should first create a new struct `PersonTimeData` which has five fields, a string for the person's first name, a string for the person's middle name, a string for the person's last name, a pointer to a double which will store the list of marathon times for the person, and an integer that will store the number of marathons they ran. So as with the previous exercise each line of the file will represent a `PersonTimeData` structure and these will be stored in a dynamically allocated array of `PersonTimeData` types. As with the last exercise, you will not know the number of people on the team and they are of course running different numbers of marathons so the arrays of data being stored will be of different sizes, hence the need to store the number they ran in the struct as well.

Once the data is loaded into the program close the file, it will not be needed. Do not do any of the calculations until the file is closed. You will now print out a team summary as in the two example runs below. The summary will display all the times for the person, note that single digit minutes or seconds are in two digit format, e.g. 5 minutes is represented as 05. It will also display their average time, personal best, and the best time for the entire team and who ran it.

As with the last exercise you will load in the data from the file into dynamically allocated arrays, one array of PersonTimeData types for the entire database and each data type has its own dynamically allocated array of times. The times are to be stored as doubles in hours. So for Martha Jones the time of 3:32:15 would be $3.5375 = 3 + 32/60 + 15/3600$. This format will make it easier to calculate averages and find minimums.

```
Input the filename: MarData001.csv
Report for Martha Jones
Times: 3:32:15
```

```
Average Time: 3:32:15
Personal Best: 3:32:15

Report for Donna Noble
Times: 4:01:52  3:59:18
Average Time: 4:00:35
Personal Best: 3:59:18

Report for Clara Oswyn Oswald
Times: 3:51:22  4:05:19  3:40:15
Average Time: 3:52:19
Personal Best: 3:40:15

Report for Amy Pond
Times: 4:31:25
Average Time: 4:31:25
Personal Best: 4:31:25

Report for Bill Potts
Times: 4:52:01  4:43:20  4:05:54  3:58:25  3:42:19
Average Time: 4:16:24
Personal Best: 3:42:19

Report for John Smith
Times: 4:10:55
Average Time: 4:10:55
Personal Best: 4:10:55

Report for Mickey Smith
Times: 3:51:08  3:44:10  4:35:01
Average Time: 4:03:26
Personal Best: 3:44:10

Report for Sarah Jane Smith
Times: 3:12:19
Average Time: 3:12:19
Personal Best: 3:12:19

Report for Rose Tyler
Times: 4:10:32  4:02:57  3:49:55
Average Time: 4:01:08
Personal Best: 3:49:55

Report for Rory Williams
Times: 4:25:17  3:39:20  3:35:10  3:30:17
Average Time: 3:47:31
Personal Best: 3:30:17

The team's best time was 3:12:19 by Sarah Jane Smith.
```

```
Input the filename: MarData002.csv
Report for Martha Jones
Times: 3:32:15
Average Time: 3:32:15
Personal Best: 3:32:15

Report for Donna Noble
Times: 4:01:52  3:59:18
Average Time: 4:00:35
Personal Best: 3:59:18
```

```
Report for Clara Oswyn Oswald
Times: 3:51:22  4:05:19  3:40:15
Average Time: 3:52:19
Personal Best: 3:40:15

Report for Amy Pond
Times: 4:31:25  4:11:05
Average Time: 4:21:15
Personal Best: 4:11:05

Report for Bill Potts
Times: 4:52:01  4:43:20  4:05:54  3:58:25  3:42:19
Average Time: 4:16:24
Personal Best: 3:42:19

Report for John Smith
Times: 4:10:55  3:31:51  3:56:21
Average Time: 3:53:02
Personal Best: 3:31:51

Report for Mickey Smith
Times: 3:51:08  3:44:10  4:35:01
Average Time: 4:03:26
Personal Best: 3:44:10

Report for Sarah Jane Smith
Times: 3:12:19
Average Time: 3:12:19
Personal Best: 3:12:19

Report for Rose Tyler
Times: 4:10:32  4:02:57  3:49:55  3:10:58
Average Time: 3:48:36
Personal Best: 3:10:58

Report for Rory Williams
Times: 4:25:17  3:39:20  3:35:10  3:30:17
Average Time: 3:47:31
Personal Best: 3:30:17

The team's best time was 3:10:58 by Rose Tyler.
```