Alexander Wise
November 13, 2023
EECE 2160
Homework 3

Problem 1:

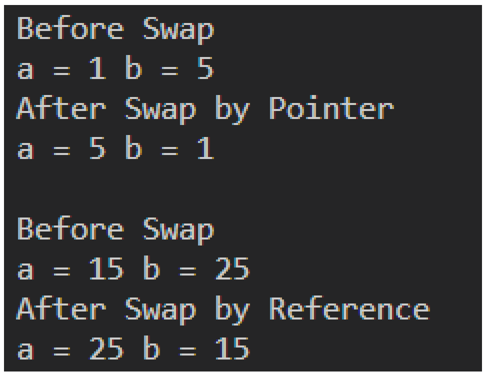
For the SwapP function, it took into two pointers for num1 and num2 and then swapped those positions directly in memory.

```
// Swaps numbers through pass-by-pointer
void SwapP(int* num1, int* num2){
    // Temporary value to store num 1
    int temp = *num1;
    // Replace num1 with num2
    *num1 = *num2;
    // Replace num2 with temp value
    *num2 = temp;
}
```

For the SwapR, the values were sent from main and the function took in two references. These references are then swapped thus, swapping their memory locations.

```
// Swaps numbers through pass-by-reference
void SwapR(int& num1, int& num2){
    // Temporary value to store num1
    int temp = num1;
    // Replaces num1 with num2
    num1 = num2;
    // Replaces num2 with temp value
    num2 = temp;
}
```

When run, the following output was produced:



```
Before Swap
a = 1 b = 5
After Swap by Pointer
a = 5 b = 1

Before Swap
a = 15 b = 25
After Swap by Reference
a = 25 b = 15
```

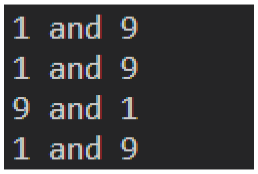
Problem 2:

From the main function, x and y are declared as well as pointers p1 and p2. These pointers are then set equal to the reference of x and y. This means that the PointerSwap function should only modify the locations that are being pointed to and not the variables x and y. With this in mind, the following PointerSwap function was created:

```cpp
// Function definition for swapping two integer pointers by reference
void PointerSwap(int* &p1, int* &p2) {
    int* temp;
    // Store the memory address of p1 in a temporary variable
    temp = p1;
    // Assign the address of p2 to p1
    p1 = p2;
    // Assign the memory address from the temporary variable to p2
    p2 = temp;
}
```

The function takes in a pointer to a reference of p1 and p2, allowing for modification of the original pointers and not just their copy. Next, a temporary pointer was created to store the memory address of p1. p1 is then replaced with the address of p2 and p2 is replaced with the temp variable.

This results in the following output when run:



```
1 and 9
1 and 9
9 and 1
1 and 9
```

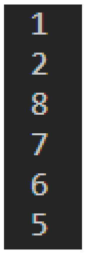
Problem 3:

From the main array, we know that we have a pre declared array of 6 elements. Similarly, the mirror function takes in the array, *v*, and the length of v.

With this in mind, the following mirror function was created.

```
void mirror(int* v, int n) {
    // Makes a copy of n to count backwards
    int reverseCounter = n - 1;
    int temp;
    // Loops through the first half of the array
    for(int i = 0; i < n/2; i++, reverseCounter--){
        // Stores the front element in a temporary variable
        temp = v[i];
        // Replaces front element with rear element
        v[i] = v[reverseCounter];
        // Replaces rear element with front element
        v[reverseCounter] = temp;
    }
}
```

Mirror takes in a pointer to *v* and the length of the array, *n*. With these, the array loops through the first half of elements and stores them in a temporary variable. At the same time, there is a counter starting at the length of the array, n, that is used to swap the rear value with the front value before decrementing. The for loop decrements the *reverseCounter* and increments *i* at the same time so both of these counters move towards one another until the loop exits. This results in the following output when ran:



```
1
2
8
7
6
5
```

Problem 4:

Taking the character ‘a’ for example. Its ASCII value is 97 and its associated binary value is 01100001. The character ‘A’ has an ASCII value of 65 (01000001). The difference between these two binary values is in the 6th bit. Lower case ‘a’ has a 1 in the 6th bit while upper case ‘A’ has a 0 in the 6th bit. I will refer to this as the case bit. Knowing that the case bit is what determines if a character is upper or lower case, we just have to AND/OR the ASCII value in binary with a value to flip the case bit.

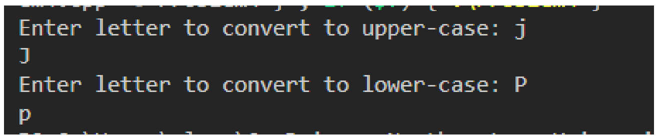
For the conversion from lower to upper case, the binary value for the inputted lower-case character needs and AND operation to convert that 6th bit to a 0. This means that every bit needs to be 1 except for the 6th bit which must be a zero. The operation in this case would be myChar & 11011111 (223 in decimal). The implementation looks like this:

```
    char myChar;
    cout << "Enter letter to convert to upper-case: ";
    cin >> myChar;
    // AND operation with the binary number of 11011111 (223)
    // This makes the 6th bit a 0 which converts it to upper case
    myChar = myChar & 223;

    cout << myChar << endl;
```

Similarly, to convert from upper to lower case, we need the OR operation to convert the 6th bit to a 1. The operation here would look like myChar | 00100000. This also ensures that no other bits are modified in the original ASCII value. The implementation looks like this:

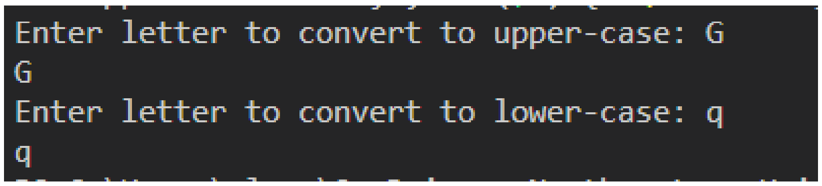
```
    cout << "Enter letter to convert to lower-case: ";
    cin >> myChar;
    // OR operation with the binary number 00100000 (32)
    // This makes the 6th bit a 1 which converts it to lower case
    myChar = myChar | 32;
    cout << myChar << endl;
```

When run, the following is produced:



```
Enter letter to convert to upper-case: j
J
Enter letter to convert to lower-case: P
p
```

If an upper case letter is input for case A, then an upper case letter would also be returned. This is because the and operator is being used and the case bit is already a 1 so it would stay a 1 and return an upper case letter. Same for case B, with a lower case letter, a lower case letter will also be returned because with the or operation, there will be two 0s in the case bit resulting in the case staying the same:



```
Enter letter to convert to upper-case: G
G
Enter letter to convert to lower-case: q
q
```

Problem 5:

First, a struct was defined for the Students which accepted a student's name and their grade:

```
struct Students{
    string Name;
    int Grade;
};
```

Within the main function, the program takes in the size of the class and creates a dynamic array of structs based on the size input.

```
    int classSize;
    int i = 0;

    // Prompts user to input class size
    cout << "Enter the size of the class: ";
    cin >> classSize;

    // Creates a new dynamic array of length ClassSize
    Students *classData = new Students[classSize];
```

Next, a for loop iterates through each item within the classData array initialized above and takes in user input to assign values to the name and grade for each student:

```
    // Loops thorugh each index to input student name and grade
    for(int i = 0; i < classSize; i++){
        int grade;

        cout << "\nEnter student name: ";
        // Assigns student name to associated value within the struct
        cin >> classData[i].Name;

        // Checks to see if the inputed student grade is valid. Loops until valid
input
        do {
            cout << "Enter student grade (0 to 100): ";
            cin >> grade;
        } while(grade > 100 || grade < 0);
        // Assigns grade to associated value within the struct
        classData[i].Grade = grade;
    }
```

The do while loop is used to ensure that the given grade is between 100 and 0 inclusive. If it isn't, it will loop through once again.

To print the class data, the function printClass was created. printClass takes in the array of Students structs as well as the size of the class.

```cpp
// Prints the class' information
void printClass(Students *classData, int classSize) {
    sortGrades(classData, classSize);
    cout << "\nStudent:\tGrade:\n";
    // Iterates through each student within the array and prints their grade and
name
    for(int i = 0; i < classSize; i++) {
        cout << classData[i].Name << "\t\t" << classData[i].Grade << endl;
    }
}
```

The function iterates through each element within the array and prints the name and grade for each student. Before this is done through, the array is sorted in descending order by grade using selection sort.

```cpp
// Sorts student grades into descending order
void sortGrades(Students *classData, int classSize) {
    for (int i = 0; i < classSize - 1; i++) {
        int max = i;
        // Find the maximum element
        for (int j = i + 1; j < classSize; j++) {
            // Compare to see if grade is greater than current max
            if (classData[j].Grade > classData[max].Grade) {
                max = j;
            }
        }
        // Swap if a greater element is found
        if (max != i) {
            swap(classData[i], classData[max]);
        }
    }
}
```

The selection sort looks for the max value within the array and swaps it with whatever index the nested loop is on. This will sort the array by grades in descending order.

Next, the average and median grade was calculated. Both of these operations were done in a function called averageGrade which takes in the array of students and size of the class. For the mean, the sum of grades was calculated and then divided by the class size to determine the average grade:

```cpp
    int sum = 0;
    // Calculates the sum of all the grades
    for(int i = 0; i < classSize; i++){
        sum += classData[i].Grade;
    }
    // Calculates the average
    float average = sum/classSize;
```

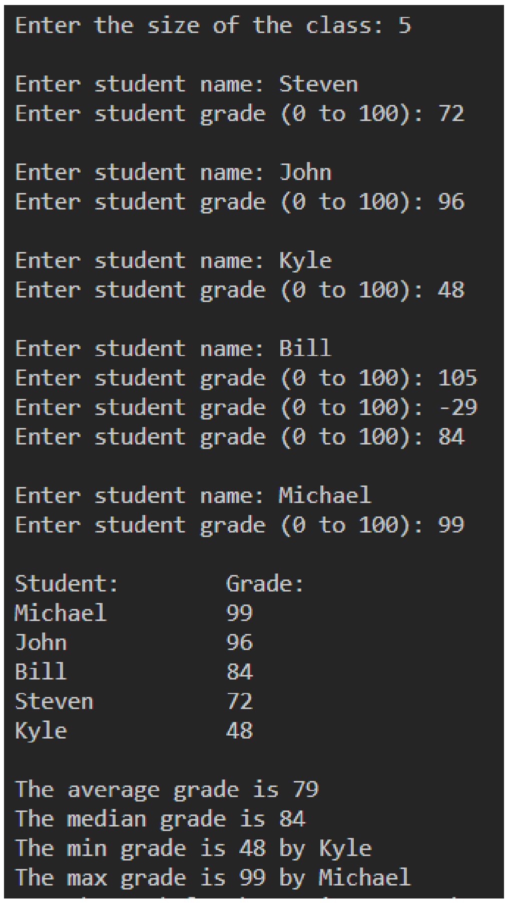
The median was calculated next. First, the program checks to see if the class size is even or odd. If it's odd, it will return the grade at the middle index. If even, it will find the average of the two middle indices:

```
    float median;

    // Checks if there are an odd or even number of students in the class
    if(classSize % 2 == 0){
        // Calculates median based on an even number of students
        median = (classData[(classSize-1)/2].Grade + classData[((classSize-1)/2)
+ 1].Grade)/2.0;
    } else {
        // Calculates median based on an odd number of students
        median = classData[classSize/2].Grade;
    }
```

Since the array of students has been sorted in descending order by grade, the max grade will be found at the first index and the min grade will be found at the max index. With this in mind, the min grade and student, and the max grade and student are found with the two following functions:

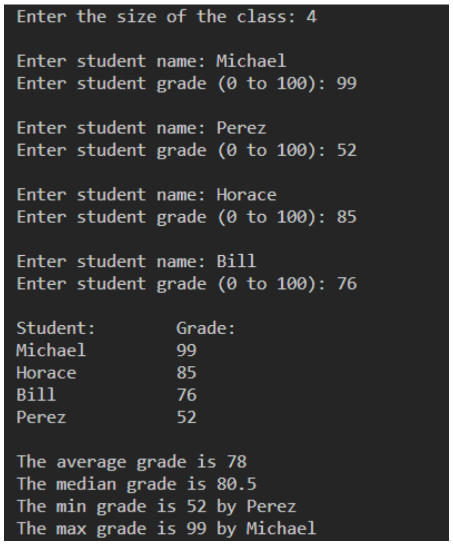
```
// Displays the max grade and student
void maxGrade(Students *classData, int classSize){
    // the max grade will be the first index after sorting
    cout << "The max grade is " << classData[0].Grade << " by " <<
classData[0].Name << endl;
}

// Displays the min grade and student
void minGrade(Students *classData, int classSize){
    // The min grade will be the last index after sorting
    cout << "The min grade is " << classData[classSize - 1].Grade << " by " <<
classData[classSize - 1].Name << endl;
}
```

When run with 5 students, the following output is produced:



```
Enter the size of the class: 5

Enter student name: Steven
Enter student grade (0 to 100): 72

Enter student name: John
Enter student grade (0 to 100): 96

Enter student name: Kyle
Enter student grade (0 to 100): 48

Enter student name: Bill
Enter student grade (0 to 100): 105
Enter student grade (0 to 100): -29
Enter student grade (0 to 100): 84

Enter student name: Michael
Enter student grade (0 to 100): 99

Student:        Grade:
Michael         99
John            96
Bill            84
Steven          72
Kyle            48

The average grade is 79
The median grade is 84
The min grade is 48 by Kyle
The max grade is 99 by Michael
```

As seen, with an inputted grade not within 0 and 100, the program will loop until a proper grade is inputted. For 4 students:



```
Enter the size of the class: 4

Enter student name: Michael
Enter student grade (0 to 100): 99

Enter student name: Perez
Enter student grade (0 to 100): 52

Enter student name: Horace
Enter student grade (0 to 100): 85

Enter student name: Bill
Enter student grade (0 to 100): 76

Student:        Grade:
Michael         99
Horace          85
Bill            76
Perez           52

The average grade is 78
The median grade is 80.5
The min grade is 52 by Perez
The max grade is 99 by Michael
```

As seen, the median value is between the grades for Horace and Bill.