# Week 6 Tutorial Exercise 1

Case Study: Computing Mean, Median and Mode Using Arrays

Text book Examples

# 6.9 Case Study: Computing Mean, Median and Mode Using Arrays

- Computers are commonly used for survey data analysis to compile and analyze the results of surveys and opinion polls.
- Figure 6.16 uses array `response` initialized with 99 responses to a survey.
- Each response is a number from 1 to 9.
- The program computes the mean, median and mode of the 99 values.
- Figure 6.17 contains a sample run of this program.
- This example includes most of the common manipulations usually required in array problems, including passing arrays to functions.

```
1  // Fig. 6.16: fig06_16.c
2  // Survey data analysis with arrays:
3  // computing the mean, median and mode of the data.
4  #include <stdio.h>
5  #define SIZE 99
6
7  // function prototypes
8  void mean(const unsigned int answer[]);
9  void median(unsigned int answer[]);
10 void mode(unsigned int freq[], unsigned const int answer[]) ;
11 void bubbleSort(int a[]);
12 void printArray(unsigned const int a[]);
13
14 // function main begins program execution
15 int main(void)
16 {
17    unsigned int frequency[10] = {0}; // initialize array frequency
18
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 1 of 8.)

```
19      // initialize array response
20      unsigned int response[SIZE] =
21         {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
22          7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
23          6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
24          7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
25          6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
26          7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
27          5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
28          7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
29          7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
30          4, 5, 6, 1, 6, 5, 7, 8, 7};
31
32      // process responses
33      mean(response);
34      median(response);
35      mode(frequency, response);
36   }
37
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 2 of 8.)

```
38  // calculate average of all response values
39  void mean(const unsigned int answer[])
40  {
41     printf("%s\n%s\n%s\n", "********", "  Mean", "********");
42
43     unsigned int total = 0; // variable to hold sum of array elements
44
45     // total response values
46     for (size_t j = 0; j < SIZE; ++j) {
47        total += answer[j];
48     }
49
50     printf("The mean is the average value of the data\n"
51            "items. The mean is equal to the total of\n"
52            "all the data items divided by the number\n"
53            "of data items (%u). The mean value for\n"
54            "this run is: %u / %u = %.4f\n\n",
55            SIZE, total, SIZE, (double) total / SIZE);
56  }
57
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 3 of 8.)

```
58  // sort array and determine median element's value
59  void median(unsigned int answer[])
60  {
61     printf("\n%s\n%s\n%s\n%s",
62             "********", " Median", "********",
63             "The unsorted array of responses is");
64
65     printArray(answer); // output unsorted array
66
67     bubbleSort(answer); // sort array
68
69     printf("%s", "\n\nThe sorted array is");
70     printArray(answer); // output sorted array
71
72     // display median element
73     printf("\n\nThe median is element %u of\n"
74            "the sorted %u element array.\n"
75            "For this run the median is %u\n\n",
76            SIZE / 2, SIZE, answer[SIZE / 2]);
77  }
78
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 4 of 8.)

```
79  // determine most frequent response
80  void mode(unsigned int freq[], const unsigned int answer[])
81  {
82     printf("\n%s\n%s\n%s\n","********", "  Mode", "********");
83
84     // initialize frequencies to 0
85     for (size_t rating = 1; rating <= 9; ++rating) {
86        freq[rating] = 0;
87     }
88
89     // summarize frequencies
90     for (size_t j = 0; j < SIZE; ++j) {
91        ++freq[answer[j]];
92     }
93
94     // output headers for result columns
95     printf("%s%11s%19s\n\n%54s\n%54s\n\n",
96            "Response", "Frequency", "Histogram",
97            "1    1    2    2", "5    0    5    0    5");
98
99     // output results
100    unsigned int largest = 0; // represents largest frequency
101    unsigned int modeValue = 0; // represents most frequent response
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 5 of 8.)

```
102
103     for (rating = 1; rating <= 9; ++rating) {
104        printf("%8u%11u           ", rating, freq[rating]);
105
106        // keep track of mode value and largest frequency value
107        if (freq[rating] > largest) {
108           largest = freq[rating];
109           modeValue = rating;
110        }
111
112        // output histogram bar representing frequency value
113        for (unsigned int h = 1; h <= freq[rating]; ++h) {
114           printf("%s", "*");
115        }
116
117        puts(""); // being new line of output
118     }
119
120     // display the mode value
121     printf("\nThe mode is the most frequent value.\n"
122            "For this run the mode is %u which occurred"
123            " %u times.\n", modeValue, largest);
124  }
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 6 of 8.)

```
125
126 // function that sorts an array with bubble sort algorithm
127 void bubbleSort(unsigned int a[])
128 {
129    // loop to control number of passes
130    for (unsigned int pass = 1; pass < SIZE; ++pass) {
131
132       // loop to control number of comparisons per pass
133       for (size_t j = 0; j < SIZE - 1; ++j) {
134
135          // swap elements if out of order
136          if (a[j] > a[j + 1]) {
137             unsigned int hold = a[j];
138             a[j] = a[j + 1];
139             a[j + 1] = hold;
140          }
141       }
142    }
143 }
144
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 7 of 8.)

```
145 // output array contents (20 values per row)
146 void printArray(const unsigned int a[])
147 {
148    // output array contents
149    for (size_t j = 0; j < SIZE; ++j) {
150
151       if (j % 20 == 0) { // begin new line every 20 values
152          puts("");
153       }
154
155       printf("%2u", a[j]);
156    }
157 }
```

**Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 8 of 8.)

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788
```

**Fig. 6.17** | Sample run for the survey data analysis program. (Part 1 of 3.)

```
********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7
```

**Fig. 6.17** | Sample run for the survey data analysis program. (Part 2 of 3.)

```
********
  Mode
********
Response   Frequency          Histogram

                                    1    1    2    2
                               5    0    5    0    5

       1           1           *
       2           3           ***
       3           4           ****
       4           5           *****
       5           8           ********
       6           9           *********
       7          23           ***********************
       8          27           ***************************
       9          19           *******************

The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

**Fig. 6.17** | Sample run for the survey data analysis program. (Part 3 of 3.)

# 6.9 Case Study: Computing Mean, Median and Mode Using Arrays

***Mean***

- The *mean* is the arithmetic average of the 99 values.
- Function `mean` (Fig. 6.16) computes the mean by totaling the 99 elements and dividing the result by 99.

***Median***

- The median is the "*middle* value."
- Function `median` determines the median by calling function `bubbleSort` to sort the array of responses into ascending order, then picking `answer[SIZE / 2]` (the middle element) of the sorted array.

# 6.9 Case Study: Computing Mean, Median and Mode Using Arrays (Cont.)

- When the number of elements is even, the median should be calculated as the mean of the two middle elements.
- Function `median` does not currently provide this capability.
- Function `printArray` is called to output the `response` array.

# 6.9 Case Study: Computing Mean, Median and Mode Using Arrays (Cont.)

***Mode***

- The *mode* is the *value that occurs most frequently* among the 99 responses.
- Function `mode` determines the mode by counting the number of responses of each type, then selecting the value with the greatest count.
- This version of function `mode` does not handle a tie (see Exercise 7.14).
- Function `mode` also produces a histogram to aid in determining the mode graphically.

# Week 6 Tutorial Exercise 2

Multidimensional Arrays

Text book Examples

# 6.11 Multidimensional Arrays (Cont.)

***Two-Dimensonal Array Manipulations***

- Figure 6.22 performs several other common array manipulations on 3-by-4 array `studentGrades` using `for` statements.
- Each row of the array represents a student and each column represents a grade on one of the four exams the students took during the semester.
- The array manipulations are performed by four functions.
- Function `minimum` determines the lowest grade of any student for the semester.

# 6.11 Multidimensional Arrays

- Function `maximum` determines the highest grade of any student for the semester.
- Function `average` determines a particular student's semester average.
- Function `printArray` outputs the two-dimensional array in a neat, tabular format.

```c
// Fig. 6.22: fig06_22.c
// Two-dimensional array manipulations.
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

// function prototypes
int minimum(const int grades[][EXAMS], size_t pupils, size_t tests);
int maximum(const int grades[][EXAMS], size_t pupils, size_t tests);
double average(const int setOfGrades[], size_t tests);
void printArray(const int grades[][EXAMS], size_t pupils, size_t tests);

// function main begins program execution
int main(void)
{
   // initialize student grades for three students (rows)
   int studentGrades[STUDENTS][EXAMS] =
      { { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 } };

   // output array studentGrades
   puts("The array is:");
   printArray(studentGrades, STUDENTS, EXAMS);
```

**Fig. 6.22** | Two-dimensional array manipulations. (Part 1 of 7.)

```
25
26      // determine smallest and largest grade values
27      printf("\n\nLowest grade: %d\nHighest grade: %d\n",
28         minimum(studentGrades, STUDENTS, EXAMS),
29         maximum(studentGrades, STUDENTS, EXAMS));
30
31      // calculate average grade for each student
32      for (size_t student = 0; student < STUDENTS; ++student) {
33         printf("The average grade for student %u is %.2f\n",
34            student, average(studentGrades[student], EXAMS));
35      }
36   }
37
```

**Fig. 6.22** | Two-dimensional array manipulations. (Part 2 of 7.)

```
38  // Find the minimum grade
39  int minimum(const int grades[][EXAMS], size_t pupils, size_t tests)
40  {
41     int lowGrade = 100; // initialize to highest possible grade
42
43     // loop through rows of grades
44     for (size_t i = 0; i < pupils; ++i) {
45
46        // loop through columns of grades
47        for (size_t j = 0; j < tests; ++j) {
48
49           if (grades[i][j] < lowGrade) {
50              lowGrade = grades[i][j];
51           }
52        }
53     }
54
55     return lowGrade; // return minimum grade
56  }
57
```

**Fig. 6.22** | Two-dimensional array manipulations. (Part 3 of 7.)

```
58  // Find the maximum grade
59  int maximum(const int grades[][EXAMS], size_t pupils, size_t tests)
60  {
61     int highGrade = 0; // initialize to lowest possible grade
62
63     // loop through rows of grades
64     for (size_t i = 0; i < pupils; ++i) {
65
66        // loop through columns of grades
67        for (size_t j = 0; j < tests; ++j) {
68
69           if (grades[i][j] > highGrade) {
70              highGrade = grades[i][j];
71           }
72        }
73     }
74
75     return highGrade; // return maximum grade
76  }
77
```

**Fig. 6.22** | Two-dimensional array manipulations. (Part 4 of 7.)

```
78  // Determine the average grade for a particular student
79  double average(const int setOfGrades[], size_t tests)
80  {
81     int total = 0; // sum of test grades
82
83     // total all grades for one student
84     for (size_t i = 0; i < tests; ++i) {
85        total += setOfGrades[i];
86     }
87
88     return (double) total / tests; // average
89  }
90
```

**Fig. 6.22** | Two-dimensional array manipulations. (Part 5 of 7.)

```
91  // Print the array
92  void printArray(const int grades[][EXAMS], size_t pupils, size_t tests)
93  {
94     // output column heads
95     printf("%s", "                  [0]  [1]  [2]  [3]");
96
97     // output grades in tabular format
98     for (size_t i = 0; i < pupils; ++i) {
99
100       // output label for row
101       printf("\nstudentGrades[%u] ", i);
102
103       // output grades for one student
104       for (size_t j = 0; j < tests; ++j) {
105          printf("%-5d", grades[i][j]);
106       }
107    }
108 }
```

**Fig. 6.22** | Two-dimensional array manipulations. (Part 6 of 7.)

```
The array is:
                 [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

**Fig. 6.22** | Two-dimensional array manipulations. (Part 7 of 7.)

# 6.11 Multidimensional Arrays (Cont.)

- Functions `minimum`, `maximum` and `printArray` each receive three arguments—the `studentGrades` array (called `grades` in each function), the number of students (rows of the array) and the number of exams (columns of the array).
- Each function loops through array `grades` using nested `for` statements.

# 6.11 Multidimensional Arrays (Cont.)

- The following nested `for` statement is from the function `minimum` definition:
  - ```
    // loop through rows of grades
    for (size_t i = 0; i < pupils; ++i) {
       // loop through columns of grades
       for (size_t j = 0; j < tests; ++j) {
          if (grades[i][j] < lowGrade) {
             lowGrade = grades[i][j];
          }
       }
    }
    ```
    ```

# 6.11 Multidimensional Arrays (Cont.)

- The outer `for` statement begins by setting `i` (i.e., the row index) to `0` so that the elements of that row (i.e., the grades of the first student) can be compared to variable `lowGrade` in the body of the inner `for` statement.
- The inner `for` statement loops through the four grades of a particular row and compares each grade to `lowGrade`.
- If a grade is less than `lowGrade`, `lowGrade` is set to that grade.
- The outer `for` statement then increments the row index to `1`.
- The elements of that row are compared to variable `lowGrade`.

# 6.11 Multidimensional Arrays (Cont.)

- The outer `for` statement then increments the row index to 2.
- The elements of that row are compared to variable `lowGrade`.
- When execution of the *nested* statement is complete, `lowGrade` contains the smallest grade in the two-dimensional array.
- Function `maximum` works similarly to function `minimum`.
- Function `average` takes two arguments—a one-dimensional array of test results for a particular student called `setOfGrades` and the number of test results in the array.

# 6.11 Multidimensional Arrays (Cont.)

- When `average` is called, the first argument `studentGrades[student]` is passed.
- This causes the address of one row of the two-dimensional array to be passed to `average`.
- The argument `studentGrades[1]` is the starting address of row 1 of the array.
- Remember that a two-dimensional array is basically an array of one-dimensional arrays and that the name of a one-dimensional array is the address of the array in memory.
- Function `average` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result.