
Fibonacci Heap
FM-Index
Hash Table
Trie
Top Tree
Tree
Heap
queue
Soft Heap
BST
Stack
Data
Soft Heap
Binary Hea
Weak Heap
Stack
Trie
Beap
Doubly
Rope
BIT
array
Queue
Array
Trie
Structur
Binomial Heap
Trie
Queue
Rope
Beap
Heap
Treap
Suffix Array
BST
Tree
Splay Tree
Soft Heap

4.1 Introduction

Arrays

- Structures of related data items
- Static entity (same size throughout program)

A few types

- Pointer-based arrays (C-like)
- Arrays as objects (C++)

4.2 Arrays

Array

- Consecutive group of memory locations
- Same name and type (**`int`**, **`char`**, etc.)

To refer to an element

- Specify array name and position number (index)
- Format: arrayname[position number]
- First element at position 0

N-element array c

`c[ 0 ]`, **`c[ 1 ]`** … **`c[ n - 1 ]`**

- Nth element as position N-1

4.2 Arrays

Array elements like other variables

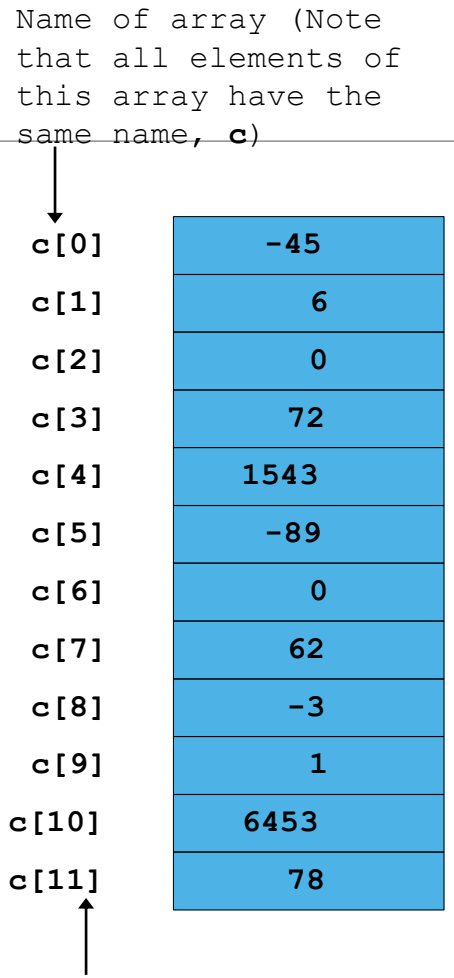
- Assignment, printing for an integer array **c**

```
c[ 0 ] =  3;
cout << c[ 0 ];
```

Can perform operations inside subscript

c[5 - 2] same as **c[3]**

4.2 Arrays


Name of array (Note that all elements of this array have the same name, c)
c[0] -45
c[1] 6
c[2] 0
c[3] 72
c[4] 1543
c[5] -89
c[6] 0
c[7] 62
c[8] -3
c[9] 1
c[10] 6453
c[11] 78
Position number of the element within array c

4.3 Declaring Arrays

When declaring arrays, specify

- Name
- Type of array
  - Any data type
- Number of elements
- *type arrayName* **[** *arraySize* **];**

```
int c[ 10 ];  // array of 10 integers
float d[ 3284 ]; // array of 3284 floats
```

Declaring multiple arrays of same type

- Use comma separated list, like regular variables

```
int b[ 100 ], x[ 27 ];
```

4.4 Examples Using Arrays

Initializing arrays

- For loop
  - Set each element
- Initializer list
  - Specify each element when array declared

```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

  - If not enough initializers, rightmost elements 0
  - If too many syntax error
- To set every element to same value

```
int n[ 5 ] = { 0 };
```

- If array size omitted, initializers determine size

```
int n[] = { 1, 2, 3, 4, 5 };
```

  - 5 initializers, therefore 5 element array

```
1   // FIG. 4.3: FIG04_03.CPP
2   // INITIALIZING AN ARRAY.
3   #INCLUDE <IOSTREAM>
5   USING STD::COUT;
6   USING STD::ENDL;
8   #INCLUDE <IOMANIP>
10  USING STD::SETW;
12  INT MAIN()
13  {
14     INT N[ 10 ]; // N IS AN ARRAY OF 10 INTE
16     // INITIALIZE ELEMENTS OF ARRAY N TO 0
17     FOR ( INT I = 0; I < 10; I++ )
18        N[ I ] = 0;  // SET ELEMENT AT LOCATION I TO
20     COUT << "ELEMENT" << SETW( 13 ) << "VALUE" << ENDL;
22     // OUTPUT CONTENTS OF ARRAY N IN TABULAR FORMAT
23     FOR ( INT J = 0; J < 10; J++ )
24        COUT << SETW( 7 ) << J << SETW( 13 ) << N[ J ] << ENDL;
25
```

Declare a 10-element array of integers.

Initialize array to **0** using a for loop. Note that the array has elements **`n[0]`** to **`n[9]`**.

```
26     RETURN 0; // INDICATES SUCCESSFUL TERMINATION
27
28  } // END MAIN
```

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

```
1   // FIG. 4.4: FIG04_04.CPP
2   // INITIALIZING AN ARRAY WITH A DECLARATION.
3   #INCLUDE <IOSTREAM>
5   USING STD::COUT;
6   USING STD::ENDL;
8   #INCLUDE <IOMANIP>
10  USING STD::SETW;
12  INT MAIN()
13  {
14    // USE INITIALIZER LIST TO INITIALIZE ARRAY N
15    INT N[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
17    COUT << "ELEMENT" << SETW( 13 ) << "VALUE" << ENDL;
19    // OUTPUT CONTENTS OF ARRAY N IN TABULAR FORMAT
20    FOR ( INT I = 0; I < 10; I++ )
21       COUT << SETW( 7 ) << I << SETW( 13 ) << N[ I ] << ENDL;
23    RETURN 0; // INDICATES SUCCESSFUL TERMINATION
25  } // END MAIN
```

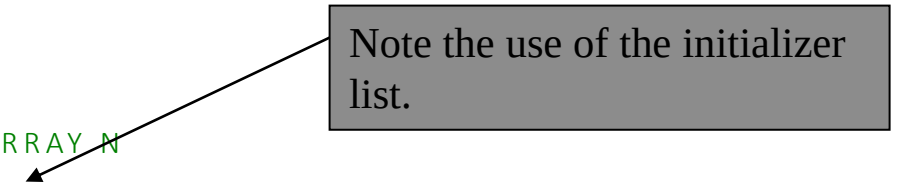


fig04_04.cpp output (1 of 1)

```
ELEMENT      VALUE
      0         32
      1         27
      2         64
      3         18
      4         95
      5         14
      6         90
      7         70
      8         60
      9         37
```

4.4 Examples Using Arrays

Array size

- Can be specified with constant variable (**`const`**)
  - `const int size = 20;`
- Constants cannot be changed
- Constants must be initialized when declared
- Also called named constants or read-only variables

```
1   // FIG. 4.5: FIG04_05.CPP
2   // INITIALIZE ARRAY S TO THE EVEN INTEGERS FROM 2 TO 20.
3   #INCLUDE <IOSTREAM>
5   USING STD::COUT;
6   USING STD::ENDL;
8   #INCLUDE <IOMANIP>
10  USING STD::SETW;
12  INT MAIN()
13  {
14     // CONSTANT VARIABLE CAN BE US
15     CONST INT ARRAYSIZE = 10;
17     INT S[ ARRAYSIZE ];  // ARRAY S HAS 10 ELEMENTS
19     FOR ( INT I = 0; I < ARRAYSIZE; I++ )  // SET THE V
20        S[ I ] = 2 + 2 * I;
22     COUT << "ELEMENT" << SETW( 13 ) << "VALUE" <<
```

Note use of **`const`** keyword. Only **`const`** variables can specify array sizes.

The program becomes more scalable when we set the array size using a **`const`** variable. We can change **`arraySize`**, and all the loops will still work (otherwise, we'd have to update every loop in the program).

```
24      // OUTPUT CONTENTS OF ARRAY S IN TABULAR FORMAT
25      FOR ( INT J = 0; J < ARRAYSIZE; J++ )
26         COUT << SETW( 7 ) << J << SETW( 13 ) << S[ J ] <<
ENDL;
27
28      RETURN 0; // INDICATES SUCCESSFUL TERMINATION
29
30   } // END MAIN
```

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

```
1   // FIG. 4.6: FIG04_06.CPP
2   // USING A PROPERLY INITIALIZED CONSTANT VARIABLE.
3   #INCLUDE <IOSTREAM>
4
5   USING STD::COUT;
6   USING STD::ENDL;
7
8   INT MAIN()
9   {
10    CONST INT X = 7;  // INITIALIZED CONSTANT VARIABLE
11
12    COUT << "THE VALUE OF CONSTANT VARIABLE X IS: "
13         << X << ENDL;
14
15    RETURN 0;  // INDICATES SUCCESSFUL TERMINATION
16
17  } // END MAIN
```

Proper initialization of **`const`** variable.

```
The value of constant variable x is: 7
```

```
1    // FIG. 4.7: FIG04_07.CPP
2    // A CONST OBJECT MUST BE INITIALIZED.
3
4    INT MAIN()
5    {
6       CONST INT X; // ERROR: X MUST BE INITIALIZED
7
8       X = 7;       // ERROR: CANNOT MODIFY A CONST
VARIABLE
9
10      RETURN 0;    // INDICATES SUCCESSFUL
TERMINATION
11
12   } // END MAIN
```

Uninitialized **`const`** results in a syntax error. Attempting to modify the **`const`** is another error.

```
d:\cpphtp4_examples\ch04\Fig04_07.cpp(6) : error C2734: 'x' :
   const object must be initialized if not extern
d:\cpphtp4_examples\ch04\Fig04_07.cpp(8) : error C2166:
   l-value specifies const object
```

```
1   // FIG. 4.8: FIG04_08.CPP
2   // COMPUTE THE SUM OF THE ELEMENTS OF THE ARRAY.
3   #INCLUDE <IOSTREAM>
5   USING STD::COUT;
6   USING STD::ENDL;
8   INT MAIN()
9   {
10    CONST INT ARRAYSIZE = 10;
12    INT A[ ARRAYSIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14    INT TOTAL = 0;
16    // SUM CONTENTS OF ARRAY A
17    FOR ( INT I = 0; I < ARRAYSIZE; I++ )
18      TOTAL += A[ I ];
20    COUT << "TOTAL OF ARRAY ELEMENT VALUES IS " << TOTAL << ENDL;
22    RETURN 0;  // INDICATES SUCCESSFUL TERMINATION
24  } // END MAIN
```

```
Total of array element values is 55
```

```
1   // FIG. 4.9: FIG04_09.CPP
2   // HISTOGRAM PRINTING PROGRAM.
3   #INCLUDE <IOSTREAM>
5   USING STD::COUT;
6   USING STD::ENDL;
8   #INCLUDE <IOMANIP>
10  USING STD::SETW;
12  INT MAIN()
13  {
14    CONST INT ARRAYSIZE = 10;
15    INT N[ ARRAYSIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
17    COUT << "ELEMENT" << SETW( 13 ) << "VALUE"
18       << SETW( 17 ) << "HISTOGRAM" << ENDL;
20    // FOR EACH ELEMENT OF ARRAY N, OUTPUT A BAR IN HISTOGRAM
21    FOR ( INT I = 0; I < ARRAYSIZE; I++ ) {
22      COUT << SETW( 7 ) << I << SETW( 13 )
23         << N[ I ] << SETW( 9 );
25      FOR ( INT J = 0; J < N[ I ]; J++ )  // PRINT ONE BAR
26        COUT << '*';
```

Prints asterisks corresponding to size of array element, **`n[i]`**.

```
27
28          COUT << ENDL;  // START NEXT LINE OF OUTPUT
29
30       } // END OUTER FOR STRUCTURE
31
32       RETURN 0;  // INDICATES SUCCESSFUL TERMINATION
33
34    } // END MAIN
```

```
Element         Value         Histogram
      0            19         ********************
      1             3         ***
      2            15         ***************
      3             7         *******
      4            11         ***********
      5             9         *********
      6            13         *************
      7             5         *****
      8            17         *****************
      9             1         *
```

```
1   // FIG. 4.10: FIG04_10.CPP
2   // ROLL A SIX-SIDED DIE 6000 TIMES.
3   #INCLUDE <IOSTREAM>
5   USING STD::COUT;
6   USING STD::ENDL;
8   #INCLUDE <IOMANIP>
10  USING STD::SETW;
12  #INCLUDE <CSTDLIB>
13  #INCLUDE <CTIME>
15  INT MAIN()
16  {
17    CONST INT ARRAYSIZE = 7;
18    INT FREQUENCY[ ARRAYSIZE ] = { 0 };
20    SRAND( TIME( 0 ) );  // SEED RANDOM-NUMBE
22    // ROLL DIE 6000 TIMES
23    FOR ( INT ROLL = 1; ROLL <= 6000; ROLL++ )
24      ++FREQUENCY[ 1 + RAND() % 6 ]; // REPLAC
25                                // OF FIG. 3.8
```

Remake of old program to roll dice. An array is used instead of 6 regular variables, and the proper element can be updated easily (without needing a **`switch`**).

This creates a number between 1 and 6, which determines the index of **`frequency[]`** that should be incremented.

```
26
27      COUT << "FACE" << SETW( 13 ) << "FREQUENCY" << ENDL;
28
29      // OUTPUT FREQUENCY ELEMENTS 1-6 IN TABULAR FORMAT
30      FOR ( INT FACE = 1; FACE < ARRAYSIZE; FACE++ )
31        COUT << SETW( 4 ) << FACE
32            << SETW( 13 ) << FREQUENCY[ FACE ] << ENDL;
33
34      RETURN 0;  // INDICATES SUCCESSFUL TERMINATION
35
36   } // END MAIN
```

```
Face     Frequency
   1          1003
   2          1004
   3           999
   4           980
   5          1013
   6          1001
```

4.4 Examples Using Arrays

Strings (more in ch. 5)

- Arrays of characters
- All strings end with **`null`** (**`'\0'`**)
- Examples
  - **`char string1[] = "hello";`**
    - **`Null`** character implicitly added
    - **`string1`** has 6 elements
  - **`char string1[] = { 'h', 'e', 'l', 'l',  'o', '\0' };`**
- Subscripting is the same

  `String1[ 0 ]` is **`'h'`**

  `string1[ 2 ]` is **`'l'`**

4.5 Passing Arrays to Functions

Specify name without brackets

- To pass array **`myArray`** to **`myFunction`**

```
int myArray[ 24 ];
myFunction( myArray, 24 );
```

- Array size usually passed, but not required
  - Useful to iterate over all elements

4.5 Passing Arrays to Functions

Arrays passed-by-reference

- Functions can modify original array data
- Value of name of array is address of first element
  - Function knows where the array is stored
  - Can change original memory locations

Individual array elements passed-by-value

- Like regular variables
- `square( myArray[3] );`

4.5 Passing Arrays to Functions

Functions taking arrays

- Function prototype
  - `void modifyArray( int b[], int arraySize );`
  - `void modifyArray( int [], int );`
    - Names optional in prototype
  - Both take an integer array and a single integer
- No need for array size between brackets
  - Ignored by compiler
- If declare array parameter as **`const`**
  - Cannot be modified (compiler error)
  - `void doNotModify( const int [] );`

fig04_14.cpp (1 of 3)

Syntax for accepting an array in parameter list.

```
1   // FIG. 4.14: FIG04_14.CPP
2   // PASSING ARRAYS AND INDIVIDUAL ARRAY ELEMENTS TO FUNCTIONS.
3   #INCLUDE <IOSTREAM>
4
5   USING STD::COUT;
6   USING STD::ENDL;
7
8   #INCLUDE <IOMANIP>
9
10  USING STD::SETW;
```

fig04_14.cpp (2 of 3)

```
26
27    COUT << ENDL;
28
29    // PASS ARRAY A TO MODIFYARRAY BY REFERENCE
30    MODIFYARRAY( A, ARRAYSIZE );
31
32    COUT << "THE VALUES OF THE MODIFIED ARRAY ARE:\N";
33
34    // OUTPUT MODIFIED ARRAY
35    FOR ( INT J = 0; J < ARRAYSIZE; J++ )
36       COUT << SETW( 3 ) << A[ J ];
37
38    // OUTPUT VALUE OF A[ 3 ]
39    COUT << "\N\N\N"
40         << "EFFECTS OF PASSING ARRAY ELEMENT BY VALUE:"
41         << "\N\NTHE VALUE OF A[3] IS " << A[ 3 ] << '\N';
42
43    // PASS ARRAY ELEMENT A[ 3 ] BY VALUE
44    MODIFYELEMENT( A[ 3 ] );
45
46    // OUTPUT VALUE OF A[ 3 ]
47    COUT << "THE VALUE OF A[3] IS " << A[ 3 ] << ENDL;
48
49    RETURN 0; // INDICATES SUCCESSFUL TERMINATION
50
51 } // END MAIN
```

Pass array name (**a**) and size to function. Arrays are passed-by-reference.

Pass a single array element by value; the original cannot be modified.

fig04_14.cpp (3 of 3)

```
52
53  // IN FUNCTION MODIFYARRAY, "B" POINTS TO
54  // THE ORIGINAL ARRAY "A" IN MEMORY
55  VOID MODIFYARRAY( INT B[], INT SIZEOFARRAY )
56  {
57     // MULTIPLY EACH ARRAY ELEMENT BY 2
58     FOR ( INT K = 0; K < SIZEOFARRAY; K++ )
59        B[ K ] *= 2;
60
61  } // END FUNCTION MODIFYARRAY
62
63  // IN FUNCTION MODIFYELEMENT, "E" IS A LOCAL COPY OF
64  // ARRAY ELEMENT A[ 3 ] PASSED FROM MAIN
65  VOID MODIFYELEMENT( INT E )
66  {
67     // MULTIPLY PARAMETER BY 2
68     COUT << "VALUE IN MODIFYELEMENT IS "
69          << ( E *= 2 ) << ENDL;
70
71  } // END FUNCTION MODIFYELEMENT
```

Although named **b**, the array points to the original array **a**. It can modify **a**'s data.

Individual array elements are passed by value, and the originals cannot be changed.

```
EFFECTS OF PASSING ENTIRE ARRAY BY REFERENCE:


THE VALUES OF THE ORIGINAL ARRAY ARE:

  0  1  2  3  4

THE VALUES OF THE MODIFIED ARRAY ARE:

  0  2  4  6  8



EFFECTS OF PASSING ARRAY ELEMENT BY VALUE:


THE VALUE OF A[3] IS 6

VALUE IN MODIFYELEMENT IS 12

THE VALUE OF A[3] IS 6
```

fig04_15.cpp (1 of 2)

```
1  // FIG. 4.15: FIG04_15.CPP
2  // DEMONSTRATING THE CONST TYPE QUALIFIER.
3  #INCLUDE <IOSTREAM>
4
5  USING STD::COUT;
6  USING STD::ENDL;
7
8  VOID TRYTOMODIFYARRAY( CONST INT[] ); // FUNCTION PROTOTYPE
9
10 INT MAIN()
11 {
12    INT A[] = { 10, 20, 30 };
13
14    TRYTOMODIFYARRAY( A );
15
16    COUT << A[ 0 ] << ' ' << A[ 1 ] << ' ' << A[ 2 ] << '\N';
17
18    RETURN 0; // INDICATES SUCCESSFUL TERMINATION
19
20 } // END MAIN
21
```

Array parameter declared as **`const`**. Array cannot be modified, even though it is passed by reference.

```
22  // IN FUNCTION TRYTOMODIFYARRAY, "B" CANNOT BE USED
23  // TO MODIFY THE ORIGINAL ARRAY "A" IN MAIN.
24  VOID TRYTOMODIFYARRAY( CONST INT B[] )
25  {
26     B[ 0 ] /= 2;   // ERROR
27     B[ 1 ] /= 2;   // ERROR
28     B[ 2 ] /= 2;   // ERROR
29
30  } // END FUNCTION TRYTOMODIFYARRAY
```

fig04_15.cpp output (1 of 1)

```
d:\cpphtp4_examples\ch04\Fig04_15.cpp(26) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(27) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch04\Fig04_15.cpp(28) : error C2166:
   l-value specifies const object
```

4.6 Sorting Arrays

Sorting data

- Important computing application
- Virtually every organization must sort some data
  - Massive amounts must be sorted

Bubble sort (sinking sort)

- Several passes through the array
- Successive pairs of elements are compared
  - If increasing order (or identical), no change
  - If decreasing order, elements exchanged
- Repeat these steps for every element

4.6 Sorting Arrays

Example:

- Go left to right, and exchange elements as necessary
  - One pass for each element
- Original: 3 4 2 7 6
- Pass 1: 3 2 4 6 7 (elements exchanged)
- Pass 2: 2 3 4 6 7
- Pass 3: 2 3 4 6 7 (no changes needed)
- Pass 4: 2 3 4 6 7
- Pass 5: 2 3 4 6 7
- Small elements "bubble" to the top (like 2 in this example)

4.6 Sorting Arrays

Swapping variables

```
int x = 3, y = 4;
y = x;
x = y;
```

What happened?

- Both x and y are 3!
- Need a temporary variable

Solution

```
int x = 3, y = 4, temp = 0;
temp = x;  // temp gets 3
x = y;     // x gets 4
y = temp;  // y gets 3
```

fig04_16.cpp (1 of 3)

```
1  // FIG. 4.16: FIG04_16.CPP
2  // THIS PROGRAM SORTS AN ARRAY'S VALUES INTO ASCENDING ORDER.
3  #INCLUDE <IOSTREAM>
4
5  USING STD::COUT;
6  USING STD::ENDL;
7
8  #INCLUDE <IOMANIP>
9
10 USING STD::SETW;
11
12 INT MAIN()
13 {
14    CONST INT ARRAYSIZE = 10; // SIZE OF ARRAY A
15    INT A[ ARRAYSIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16    INT HOLD; // TEMPORARY LOCATION USED TO SWAP ARRAY ELEMENTS
17
18    COUT << "DATA ITEMS IN ORIGINAL ORDER\N";
19
20    // OUTPUT ORIGINAL ARRAY
21    FOR ( INT I = 0; I < ARRAYSIZE; I++ )
22       COUT << SETW( 4 ) << A[ I ];
23
```

fig04_16.cpp (2 of 3)

```
24    // BUBBLE SORT
25    // LOOP TO CONTROL NUMBER OF PASSES
26    FOR ( INT PASS = 0; PASS < ARRAYSIZE - 1; PASS++ )
27
28       // LOOP TO CONTROL NUMBER OF COMPARISONS PER PASS
29       FOR ( INT J = 0; J < ARRAYSIZE - 1; J++ )
30
31          // COMPARE SIDE-BY-SIDE ELEMENTS AND SWAP THEM IF
32          // FIRST ELEMENT IS GREATER THAN SECOND ELEMENT
33          IF ( A[ J ] > A[ J + 1 ] ) {
34             HOLD = A[ J ];
35             A[ J ] = A[ J + 1 ];
36             A[ J + 1 ] = HOLD;
37
38          } // END IF
39
```

Do a pass for each element in the array.

If the element on the left (index `j`) is larger than the element on the right (index `j + 1`), then we swap them. Remember the need of a temp variable.

```
40      COUT << "\NDATA ITEMS IN ASCENDING ORDER\N";
41
42      // OUTPUT SORTED ARRAY
43      FOR ( INT K = 0; K < ARRAYSIZE; K++ )
44         COUT << SETW( 4 ) << A[ K ];
45
46      COUT << ENDL;
47
48      RETURN 0; // INDICATES SUCCESSFUL TERMINATION
49
50   } // END MAIN
```

fig04_16.cpp output (1 of 1)

```
Data items in original order
   2    6    4    8   10   12   89   68   45   37
Data items in ascending order
   2    4    6    8   10   12   37   45   68   89
```

4.7 Case Study: Computing Mean, Median and Mode Using Arrays

Mean

- Average (sum/number of elements)

Median

- Number in middle of sorted list
- 1, 2, 3, 4, 5 (3 is median)
- If even number of elements, take average of middle two

Mode

- Number that occurs most often
- 1, 1, 1, 2, 3, 3, 4, 5 (1 is mode)

fig04_17.cpp (1 of 8)

```
1   // FIG. 4.17: FIG04_17.CPP
2   // THIS PROGRAM INTRODUCES THE TOPIC OF SURVEY DATA ANALYSIS.
3   // IT COMPUTES THE MEAN, MEDIAN, AND MODE OF THE DATA.
4   #INCLUDE <IOSTREAM>
5
6   USING STD::COUT;
7   USING STD::ENDL;
8   USING STD::FIXED;
9   USING STD::SHOWPOINT;
10
```

fig04_17.cpp (2 of 8)

```
26    INT FREQUENCY[ 10 ] = { 0 }; // INITIALIZE ARRAY FREQUENCY
27
28    // INITIALIZE ARRAY RESPONSES
29    INT RESPONSE[ RESPONSESIZE ] =
30          { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
31            7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
32            6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
33            7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
34            6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
35            7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
```

fig04_17.cpp (3 of 8)

We cast to a double to get decimal points for the average (instead of an integer).

```
50  // CALCULATE AVERAGE OF ALL RESPONSE VALUES
51  VOID MEAN( CONST INT ANSWER[], INT ARRAYSIZE )
52  {
53    INT TOTAL = 0;
54
55    COUT << "*********\N  MEAN\N*********\N";
56
57    // TOTAL RESPONSE VALUES
58    FOR ( INT I = 0; I < ARRAYSIZE; I++ )
59      TOTAL += ANSWER[ I ];
```

fig04_17.cpp (4 of 8)

```
75  // SORT ARRAY AND DETERMINE MEDIAN ELEMENT'S VALUE
76  VOID MEDIAN( INT ANSWER[], INT SIZE )
77  {
78     COUT << "\N*********\N MEDIAN\N*********\N"
79          << "THE UNSORTED ARRAY OF RESPONSES IS";
80
81     PRINTARRAY( ANSWER, SIZE );  // OUTPUT UNSORTED ARR
82
83     BUBBLESORT( ANSWER, SIZE );  // SORT ARRAY
84
85     COUT << "\N\NTHE SORTED ARRAY IS";
86     PRINTARRAY( ANSWER, SIZE );  // OUTPUT SORTED ARRAY
87
88     // DISPLAY MEDIAN ELEMENT
89     COUT << "\N\NTHE MEDIAN IS ELEMENT " << SIZE / 2
90          << " OF\NTHE SORTED " << SIZE
91          << " ELEMENT ARRAY.\NFOR THIS RUN THE MEDIAN IS "
92          << ANSWER[ SIZE / 2 ] << "\N\N";
93
94  } // END FUNCTION MEDIAN
95
```

Sort array by passing it to a function. This keeps the program modular.

fig04_17.cpp (5 of 8)

```
96  // DETERMINE MOST FREQUENT RESPONSE
97  VOID MODE( INT FREQ[], INT ANSWER[], INT SIZE )
98  {
99     INT LARGEST = 0;  // REPRESENTS LARGEST FREQUENCY
100    INT MODEVALUE = 0; // REPRESENTS MOST FREQUENT RESPONSE
101
102    COUT << "\N********\N  MODE\N********\N";
103
104    // INITIALIZE FREQUENCIES TO 0
105    FOR ( INT I = 1; I <= 9; I++ )
106       FREQ[ I ] = 0;
107
108    // SUMMARIZE FREQUENCIES
109    FOR ( INT J = 0; J < SIZE; J++ )
110       ++FREQ[ ANSWER[ J ] ];
111
112    // OUTPUT HEADERS FOR RESULT COLUMNS
113    COUT << "RESPONSE" << SETW( 11 ) << "FREQUENCY"
114         << SETW( 19 ) << "HISTOGRAM\N\N" << SETW( 55 )
115         << "1    1    2    2\N" << SETW( 56 )
116         << "5    0    5    0    5\N\N";
117
```

```
118    // OUTPUT RESULTS
119    FOR ( INT RATING = 1; RATING <= 9; RATING++ ) {
120       COUT << SETW( 8 ) << RATING << SETW( 11 )
121          << FREQ[ RATING ] << "      "
122
123       // KEEP TRACK OF MODE VALUE AND LARGEST FEQUENCY VALU
124       IF ( FREQ[ RATING ] > LARGEST ) {
125          LARGEST = FREQ[ RATING ];
126          MODEVALUE = RATING;
127
128       } // END IF
129
130       // OUTPUT HISTOGRAM BAR REPRESENTING FREQUENCY VALUE
131       FOR ( INT K = 1; K <= FREQ[ RATING ]; K++ )
132          COUT << '*';
133
134       COUT << '\N'; // BEGIN NEW LINE OF OUTPUT
135
136    } // END OUTER FOR
137
138    // DISPLAY THE MODE VALUE
139    COUT << "THE MODE IS THE MOST FREQUENT VALUE.\N"
140         << "FOR THIS RUN THE MODE IS " << MODEVALUE
141         << " WHICH OCCURRED " << LARGEST << " TIMES." << ENDL;
142
143 } // END FUNCTION MODE
```

fig04_17 (6 of 8)

The mode is the value that occurs most often (has the highest value in **`freq`**).

fig04_17.cpp (7 of 8)

```
144
145 // FUNCTION THAT SORTS AN ARRAY WITH BUBBLE SORT ALGORITHM
146 VOID BUBBLESORT( INT A[], INT SIZE )
147 {
148    INT HOLD; // TEMPORARY LOCATION USED TO SWAP ELEMENTS
149
150    // LOOP TO CONTROL NUMBER OF PASSES
151    FOR ( INT PASS = 1; PASS < SIZE; PASS++ )
152
153       // LOOP TO CONTROL NUMBER OF COMPARISONS PER PASS
154       FOR ( INT J = 0; J < SIZE - 1; J++ )
155
156          // SWAP ELEMENTS IF OUT OF ORDER
157          IF ( A[ J ] > A[ J + 1 ] ) {
158             HOLD = A[ J ];
159             A[ J ] = A[ J + 1 ];
160             A[ J + 1 ] = HOLD;
161
162          } // END IF
163
164 } // END FUNCTION BUBBLESORT
165
```

```
166 // OUTPUT ARRAY CONTENTS (20 VALUES PER ROW)
167 VOID PRINTARRAY( CONST INT A[], INT SIZE )
168 {
169    FOR ( INT I = 0; I < SIZE; I++ ){
170
171       IF ( I % 20 == 0 ) // BEGIN NEW LINE EVERY 20 VALUES
172          COUT << ENDL;
173
174       COUT << SETW( 2 ) << A[ I ];
175
176    } // END FOR
177
178 } // END FUNCTION PRINTARRAY
```

fig04_17.cpp (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

****

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

****

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.

4.8 Searching Arrays: Linear Search and Binary Search

Search array for a key value

Linear search

- Compare each element of array with key value
  - Start at one end, go to other
- Useful for small and unsorted arrays
  - Inefficient
  - If search key not present, examines every element

4.8 Searching Arrays: Linear Search and Binary Search

Binary search

- Only used with sorted arrays
- Compare middle element with key
  - If equal, match found
  - If key < middle
    - Repeat search on first half of array
  - If key > middle
    - Repeat search on last half
- Very fast
  - At most N steps, where 2 > # of elements
  - 30 element array takes at most 5 steps

$2^N > 30$

$N = 5$

fig04_19.cpp (1 of 2)

Takes array, search key, and array size.

```
1   // FIG. 4.19: FIG04_19.CPP
2   // LINEAR SEARCH OF AN ARRAY.
3   #INCLUDE <IOSTREAM>
4
5   USING STD::COUT;
6   USING STD::CIN;
7   USING STD::ENDL;
8
9   INT LINEARSEARCH( CONST INT [], INT, INT ); // PROTOTYPE
10
```

fig04_19.cpp (2 of 2)

```
26    // DISPLAY RESULTS
27    IF ( ELEMENT != -1 )
28       COUT << "FOUND VALUE IN ELEMENT " << ELEMENT << ENDL;
29    ELSE
30       COUT << "VALUE NOT FOUND" << ENDL;
31
32    RETURN 0; // INDICATES SUCCESSFUL TERMINATION
33
34 } // END MAIN
35
36 // COMPARE KEY TO EVERY ELEMENT OF ARRAY UNTIL LOCATION IS
37 // FOUND OR UNTIL END OF ARRAY IS REACHED; RETURN SUBSCRIPT OF
38 // ELEMENT IF KEY OR -1 IF KEY NOT FOUND
39 INT LINEARSEARCH( CONST INT ARRAY[], INT KEY, INT SIZEOFARRAY )
40 {
41    FOR ( INT J = 0; J < SIZEOFARRAY; J++ )
42
43       IF ( ARRAY[ J ] == KEY ) // IF FOUND,
44          RETURN J;            // RETURN LOCATION OF KEY
45
46    RETURN -1; // KEY NOT FOUND
47
48 } // END FUNCTION LINEARSEARCH
```

```
ENTER INTEGER SEARCH KEY: 36

FOUND VALUE IN ELEMENT 18


ENTER INTEGER SEARCH KEY: 37

VALUE NOT FOUND
```

fig04_19.cpp output (1 of 1)

fig04_20.cpp (1 of 6)

```
1  // FIG. 4.20: FIG04_20.CPP
2  // BINARY SEARCH OF AN ARRAY.
3  #INCLUDE <IOSTREAM>
4
5  USING STD::COUT;
6  USING STD::CIN;
7  USING STD::ENDL;
8
9  #INCLUDE <IOMANIP>
10
11 USING STD::SETW;
12
13 // FUNCTION PROTOTYPES
14 INT BINARYSEARCH( CONST INT [], INT, INT, INT, INT );
15 VOID PRINTHEADER( INT );
16 VOID PRINTROW( CONST INT [], INT, INT, INT, INT );
17
18 INT MAIN()
19 {
20    CONST INT ARRAYSIZE = 15;  // SIZE OF ARRAY A
21    INT A[ ARRAYSIZE ];        // CREATE ARRAY A
22    INT KEY;                   // VALUE TO LOCATE IN A
23
24    FOR ( INT I = 0; I < ARRAYSIZE; I++ )  // CREATE SOME DATA
25       A[ I ] = 2 * I;
26
```

fig04_20.cpp (2 of 6)

```
27    COUT << "ENTER A NUMBER BETWEEN 0 AND 28: ";
28    CIN >> KEY;
29
30    PRINTHEADER( ARRAYSIZE );
31
32    // SEARCH FOR KEY IN ARRAY A
33    INT RESULT =
34       BINARYSEARCH( A, KEY, 0, ARRAYSIZE - 1, ARRAYSIZE );
35
36    // DISPLAY RESULTS
37    IF ( RESULT != -1 )
38       COUT << '\N' << KEY << " FOUND IN ARRAY ELEMENT "
39           << RESULT << ENDL;
40    ELSE
41       COUT << '\N' << KEY << " NOT FOUND" << ENDL;
42
43    RETURN 0; // INDICATES SUCCESSFUL TERMINATION
44
45 } // END MAIN
46
```

fig04_20.cpp (3 of 6)

```
47  // FUNCTION TO PERFORM BINARY SEARCH OF AN ARRAY
48  INT BINARYSEARCH( CONST INT B[], INT SEARCHKEY, INT LOW,
49     INT HIGH, INT SIZE )
50  {
51     INT MIDDLE;
52
53     // LOOP UNTIL LOW SUBSCRIPT IS GREATER THAN HIGH SUBSCRIPT
54     WHILE ( LOW <= HIGH ) {
55
56        // DETERMINE MIDDLE ELEMENT OF SUBARRAY BEING SEARCHED
57        MIDDLE = ( LOW + HIGH ) / 2;
58
59        // DISPLAY SUBARRAY USED IN THIS LOOP ITERATION
60        PRINTROW( B, LOW, MIDDLE, HIGH, SIZE );
61
```

Determine middle element

```
62       // IF SEARCHKEY MATCHES MIDDLE ELEMENT, RETURN MIDDLE
63       IF ( SEARCHKEY == B[ MIDDLE ] ) // MATCH
64          RETURN MIDDLE;
65
66       ELSE
67
68          // IF SEARCHKEY LESS THAN MIDDLE ELEMENT,
69          // SET NEW HIGH ELEMENT
70          IF ( SEARCHKEY < B[ MIDDLE ] )
71             HIGH = MIDDLE - 1; // SEARCH LOW END OF ARRAY
72
73          // IF SEARCHKEY GREATER THAN MIDDLE ELEMENT,
74          // SET NEW LOW ELEMENT
75          ELSE
76             LOW = MIDDLE + 1;  // SEARCH HIGH END OF ARRAY
77       }
78
79       RETURN -1; // SEARCHKEY NOT FOUND
80
81    } // END FUNCTION BINARYSEARCH
```

fig04_20.(4 of 6)

Use the rule of binary search:
If key equals middle, match

If less, search low end

If greater, search high end

Loop sets low, middle and high dynamically. If searching the high end, the new low is the element above the middle.

fig04_20.cpp (5 of 6)

```
82
83  // PRINT HEADER FOR OUTPUT
84  VOID PRINTHEADER( INT SIZE )
85  {
86     COUT << "\NSUBSCRIPTS:\N";
87
88     // OUTPUT COLUMN HEADS
89     FOR ( INT J = 0; J < SIZE; J++ )
90        COUT << SETW( 3 ) << J << ' ';
91
92     COUT << '\N'; // START NEW LINE OF OUTPUT
93
94     // OUTPUT LINE OF - CHARACTERS
95     FOR ( INT K = 1; K <= 4 * SIZE; K++ )
96        COUT << '-';
97
98     COUT << ENDL; // START NEW LINE OF OUTPUT
99
100 } // END FUNCTION PRINTHEADER
101
```

fig04_20.cpp (6 of 6)

```
102 // PRINT ONE ROW OF OUTPUT SHOWING THE CURRENT
103 // PART OF THE ARRAY BEING PROCESSED
104 VOID PRINTROW( CONST INT B[], INT LOW, INT MID,
105    INT HIGH, INT SIZE )
106 {
107    // LOOP THROUGH ENTIRE ARRAY
108    FOR ( INT M = 0; M < SIZE; M++ )
109
110       // DISPLAY SPACES IF OUTSIDE CURRENT SUBARRAY RANGE
111       IF ( M < LOW || M > HIGH )
```

fig04_20.cpp output (1 of 2)

```
ENTER A NUMBER BETWEEN 0 AND 28: 6

SUBSCRIPTS:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 FOUND IN ARRAY ELEMENT 3
```

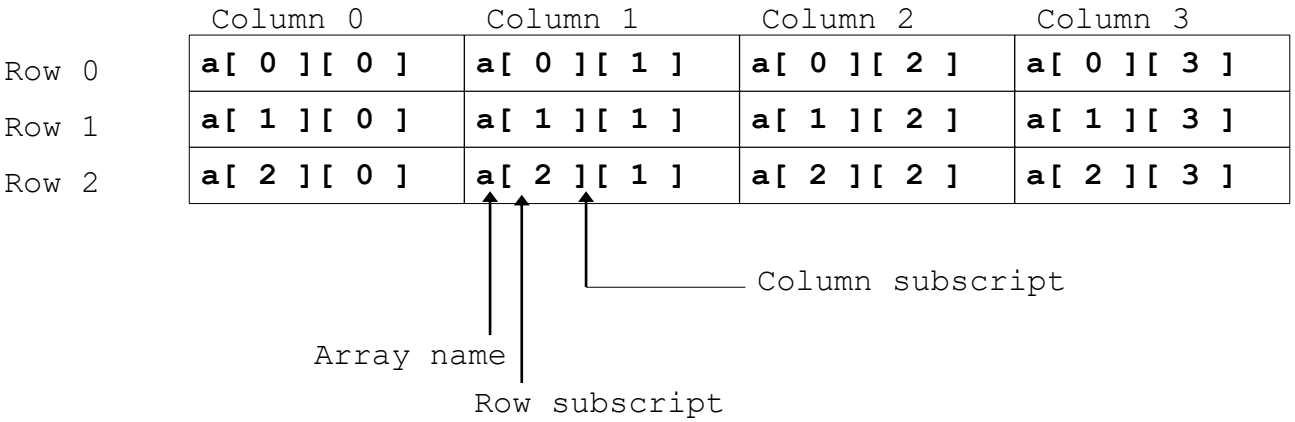
```
ENTER A NUMBER BETWEEN 0 AND 28: 8


SUBSCRIPTS:

  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14

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

  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28

  0   2   4   6*  8  10  12

              8  10* 12

              8*


8 FOUND IN ARRAY ELEMENT 4
```

fig04_20.cpp output (2 of 2)

4.9 Multiple-Subscripted Arrays

Multiple subscripts

- `a[ i ][ j ]`
- Tables with rows and columns
- Specify row, then column
- "Array of arrays"
  - `a[0]` is an array of 4 elements
  - `a[0][0]` is the first element of that array



4.9 Multiple-Subscripted Arrays

To initialize

- Default of **0**
- Initializers grouped by row in braces

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Row 0

Row 1

1	2
3	4

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

1	0
3	4

4.9 Multiple-Subscripted Arrays

1	0
3	4

Referenced like normal

```
cout << b[ 0 ][ 1 ];
```

- Outputs **0**
- Cannot reference using commas

```
cout << b[ 0, 1 ];
```

  - Syntax error

Function prototypes

- Must specify sizes of subscripts
  - First subscript not necessary, as with single-scripted arrays
- **`void printArray( int [][ 3 ] );`**

fig04_22.cpp (1 of 2)

```
1  // FIG. 4.22: FIG04_22.CPP
2  // INITIALIZING MULTIDIMENSIONAL ARRAYS.
3  #INCLUDE <IOSTREAM>
4
5  USING STD::COUT;
6  USING STD::ENDL;
7
8  VOID PRINTARRAY( INT [][ 3 ] );
9
10 INT MAIN()
11 {
12    INT ARRAY1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13    INT ARRAY2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
14    INT ARRAY3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
15
16    COUT << "VALUES IN ARRAY1 BY ROW ARE:" << ENDL;
17    PRINTARRAY( ARRAY1 );
18
19    COUT << "VALUES IN ARRAY2 BY ROW ARE:" << ENDL;
20    PRINTARRAY( ARRAY2 );
21
22    COUT << "VALUES IN ARRAY3 BY ROW ARE:" << ENDL;
23    PRINTARRAY( ARRAY3 );
24
25    RETURN 0; // INDICATES SUCCESSFUL TERMINATION
26
27 } // END MAIN
```

Note the format of the prototype.

Note the various initialization styles. The elements in **`array2`** are assigned to the first row and then the second.

fig04_22.cpp output (1 of 1)

```
28
29  // FUNCTION TO OUTPUT ARRAY WITH TWO ROWS AND THREE COLUMNS
30  VOID PRINTARRAY( INT A[][ 3 ] )
31  {
32     FOR ( INT I = 0; I < 2; I++ ) {  // FOR EACH ROW
33
34        FOR ( INT J = 0; J < 3; J++ )   // OUTPUT COLUMN VALUES
35           COUT << A[ I ][ J ] << ' ';
36
37        COUT << ENDL;  // START NEW LINE OF OUTPUT
38
39     } // END OUTER FOR STRUCTURE
40
41  } // END FUNCTION PRINTARRAY
```

For loops are often used to iterate through arrays. Nested loops are helpful with multiple-subscripted arrays.

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

4.9 Multiple-Subscripted Arrays

Next: program showing initialization

- After, program to keep track of students grades
- Multiple-subscripted array (table)
- Rows are students
- Columns are grades

	Quiz1	Quiz2
Student0	95	85
Student1	89	80

fig04_23.cpp (1 of 6)

```
1  // FIG. 4.23: FIG04_23.CPP
2  // DOUBLE-SUBSCRIPTED ARRAY EXAMPLE.
3  #INCLUDE <IOSTREAM>
4
5  USING STD::COUT;
6  USING STD::ENDL;
7  USING STD::FIXED;
8  USING STD::LEFT;
9
10 #INCLUDE <IOMANIP>
11
12 USING STD::SETW;
13 USING STD::SETPRECISION;
14
15 CONST INT STUDENTS = 3;   // NUMBER OF STUDENTS
16 CONST INT EXAMS = 4;      // NUMBER OF EXAMS
17
18 // FUNCTION PROTOTYPES
19 INT MINIMUM( INT [][ EXAMS ], INT, INT );
20 INT MAXIMUM( INT [][ EXAMS ], INT, INT );
21 DOUBLE AVERAGE( INT [], INT );
22 VOID PRINTARRAY( INT [][ EXAMS ], INT, INT );
23
```

fig04_23.cpp (2 of 6)

```
24 int main()
25 {
26    // initialize student grades for three students (rows)
27    int studentGrades[ students ][ exams ] =
28       { { 77, 68, 86, 73 },
29         { 96, 87, 89, 78 },
30         { 70, 90, 86, 81 } };
31
32    // output array studentGrades
33    cout << "The array is:\n";
34    printArray( studentGrades, students, exams );
35
36    // determine smallest and largest grade values
37    cout << "\n\nLowest grade: "
38         << minimum( studentGrades, students, exams )
39         << "\nHighest grade: "
40         << maximum( studentGrades, students, exams ) << '\n';
41
42    cout << fixed << setPrecision( 2 );
43
```

```
44    // CALCULATE AVERAGE GRADE FOR EACH STUDENT
45    FOR ( INT PERSON = 0; PERSON < STUDENTS; PERSON++ )
46       COUT << "THE AVERAGE GRADE FOR STUDENT " << PERSON
47            << " IS "
48            << AVERAGE( STUDENTGRADES[ PERSON ], EXAMS )
49            << ENDL;
50
51    RETURN 0; // INDICATES SUCCESSFUL TERMINATION
52
53 } // END MAIN
54
55 // FIND MINIMUM GRADE
56 INT MINIMUM( INT GRADES[][ EXAMS ], INT PUPILS, INT TESTS )
57 {
58    INT LOWGRADE = 100; // INITIALIZE TO HIGHEST POSSIBLE GRADE
59
60    FOR ( INT I = 0; I < PUPILS; I++ )
61
62       FOR ( INT J = 0; J < TESTS; J++ )
63
64          IF ( GRADES[ I ][ J ] < LOWGRADE )
65             LOWGRADE = GRADES[ I ][ J ];
66
67    RETURN LOWGRADE;
68
69 } // END FUNCTION MINIMUM
```

fig04_23.cpp (3 of 6)

Determines the average for one student. We pass the array/row containing the student's grades. Note that **`studentGrades[0]`** is itself an array.

fig04_23.cpp (4 of 6)

```
70
71  // FIND MAXIMUM GRADE
72  INT MAXIMUM( INT GRADES[][ EXAMS ], INT PUPILS, INT TESTS )
73  {
74     INT HIGHGRADE = 0; // INITIALIZE TO LOWEST POSSIBLE GRADE
75
76     FOR ( INT I = 0; I < PUPILS; I++ )
77
78        FOR ( INT J = 0; J < TESTS; J++ )
79
80           IF ( GRADES[ I ][ J ] > HIGHGRADE )
81              HIGHGRADE = GRADES[ I ][ J ];
82
83     RETURN HIGHGRADE;
84
85  } // END FUNCTION MAXIMUM
86
```

fig04_23.cpp (5 of 6)

```
87 // DETERMINE AVERAGE GRADE FOR PARTICULAR STUDENT
88 DOUBLE AVERAGE( INT SETOFGRADES[], INT TESTS )
89 {
90    INT TOTAL = 0;
91
92    // TOTAL ALL GRADES FOR ONE STUDENT
93    FOR ( INT I = 0; I < TESTS; I++ )
94       TOTAL += SETOFGRADES[ I ];
95
96    RETURN STATIC_CAST< DOUBLE >( TOTAL ) / TESTS; // AVERAGE
97
98 } // END FUNCTION MAXIMUM
```

fig04_23.cpp (6 of 6)

```
99
100 // PRINT THE ARRAY
101 VOID PRINTARRAY( INT GRADES[][ EXAMS ], INT PUPILS, INT TESTS )
102 {
103    // SET LEFT JUSTIFICATION AND OUTPUT COLUMN HEADS
104    COUT << LEFT <<                [0] [1] [2] [3]";
105
106    // OUTPUT GRADES IN TABULAR FORMAT
107    FOR ( INT I = 0; I < PUPILS; I++ ) {
108
109       // OUTPUT LABEL FOR ROW
110       COUT << "\NSTUDENTGRADES[" << I << "] ";
111
112       // OUTPUT ONE GRADES FOR ONE STUDENT
113       FOR ( INT J = 0; J < TESTS; J++ )
114          COUT << SETW( 5 ) << GRADES[ I ][ J ];
115
116    } // END OUTER FOR
117
118 } // END FUNCTION PRINTARRAY
```

```
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
```