Chapter 4 - Arrays

Outline
4.1 Introduction
4.2 Arrays
4.3 Declaring Arrays
4.4 Examples Using Arrays
4.5 Passing Arrays to Functions
4.6 Sorting Arrays
4.7 Case Study: Computing Mean, Median and Mode Using Arrays
4.8 Searching Arrays: Linear Search and Binary Search
4.9 Multiple-Subscripted Arrays

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

4.3 Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
    - Any data type
  - Number of elements
  - Type arrayName[ arraySize ];
    - int a[ 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 ], c[ 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
      int n[5] = {1, 2, 3, 4, 5, 6};
  – 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

Outline

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
The value of constant variable x is: 7

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

Proper initialization of const variable.

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

LaTeX code to display graphs is included.
Remake of old program to roll die. 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.
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

- Input from keyboard
  - `char string2[10];`
    - Puts user input in string
      - Stops at first whitespace character
      - Adds `null` character
      - If too much text entered, data written beyond array
      - We want to avoid this (section 5.12 explains how)

- Printing strings
  - `cout << string2 << endl;`
    - Does not work for other array types
      - Characters printed until `null` found

4.4 Examples Using Arrays

- Recall static storage (chapter 3)
  - If `static`, local variables save values between function calls
    - Visible only in function body
    - Can declare local arrays to be `static`
      - Initialized to zero
      - `static int array[3];`
  - If not `static`
    - Created (and destroyed) in every function call
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 [] );`
Pass array name \( \textit{a} \) and size to function. Arrays are passed by reference.

```cpp
void modifyElement( int e )
// array element \( \textit{a}[3] \) passed from main
// in function modifyElement, \( \textit{e} \) is a local copy of
```
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

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)

Swapping variables

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

What happened?
  – Both x and y are 3!
  – Need a temporary variable

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

Outline

```cpp
#include <iostream>

int main()
{
    const int arraySize = 10;  // size of array a
    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

    int hold;  // temporary location used to swap array elements

    cout << "Data items in original order
";
    for ( int i = 0; i < arraySize; i++ )
        cout << setw( 4 ) << a[ i ];

    // bubble sort
    for ( int pass = 0; pass < arraySize - 1; pass++ )
        for ( int j = 0; j < arraySize - 1; j++ )
            if ( a[ j ] > a[ j + 1 ] ) {
                hold = a[ j ];
                a[ j ] = a[ j + 1 ];
                a[ j + 1 ] = hold;
            }

    cout << "Data items in ascending order
";
    for ( int k = 0; k < arraySize; k++ )
        cout << setw( 4 ) << a[ k ];

    return 0;  // indicates successful termination
}
```
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, 4, 5 (1 is mode)
The mode is the value that occurs most often (has the highest value in freq).

```cpp
output results
for ( int rating = 1; rating <= 9; rating++ ) {
    cout << setw( 8 ) << rating << setw( 11 )
        << freq[ rating ] << "          ";
    // keep track of mode value and largest frequency value
    if ( freq[ rating ] > largest ) {
        largest = freq[ rating ];
        modeValue = rating;
    }
    // output histogram bar representing frequency value
    for ( int k = 1; k <= freq[ rating ]; k++ )
        cout << '*';
    cout << '
';
}
// display the mode value
cout << "The mode is the most frequent value."
    << "For this run the mode is " << modeValue
    << " which occurred " << largest << " times."
}
// function that sorts an array with bubble sort algorithm
void bubbleSort( int a[], int size ) {
    int hold;  // temporary location used to swap elements
    // loop to control number of passes
    for ( int pass = 1; pass < size; pass++ ) {
        // loop to control number of comparisons per pass
        for ( int j = 0; j < size - 1; j++ )
            // swap elements if out of order
            if ( a[ j ] > a[ j + 1 ] ) {
                hold = a[ j ];
                a[ j ] = a[ j + 1 ];
                a[ j + 1 ] = hold;
            }
    }
}
// function that prints an array contents
void printArray( const int a[], int size ) {
    for ( int i = 0; i < size; i++ ) {
        if ( i % 20 == 0 )  // begin new line every 20 values
            cout << endl;
        cout << setw( 2 ) << a[ i ];
    }
}
```

• 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^N > 8 of elements
  - 30 element array takes at most 5 steps
    \[ 2^5 > 30 \]

Outline

```cpp
// Fig. 4.19: fig04_19.cpp
// Linear search of an array.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int linearSearch( const int[], int, int ); // prototype

int main()
{
    const int arraySize = 100;  // size of array a
    int a[ arraySize ];        // create array a
    int searchKey;             // value to locate in a

    for ( int i = 0; i < arraySize; i++ )  // create some data
        a[ i ] = 2 * i;

    cout << "Enter integer search key: ";
    cin >> searchKey;

    // attempt to locate searchKey in array a
    int element = linearSearch( a, searchKey, arraySize );

    // display results
    if ( element != -1 )
        cout << "Found value in element " << element << endl;
    else
        cout << "Value not found" << endl;

    return 0; // indicates successful termination
}
```

Takes array, search key, and array size.

Outline

```cpp
// Fig. 4.20: fig04_20.cpp
// Binary search of an array.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

// function prototypes
int binarySearch( const int[], int, int, int, int );
void printHeader( int );
void printRow( const int[], int, int, int, int );

int main()
{
    const int arraySize = 15;  // size of array a
    int a[ arraySize ];       // create array a
    int key;                  // value to locate in a

    for ( int i = 0; i < arraySize; i++ )  // create some data
        a[ i ] = 2 * i;

    cout << "Enter a number between 0 and 28: ";
    cin >> key;

    printHeader( arraySize );

    // search for key in array a
    int result =
        binarySearch( a, key, 0, arraySize - 1, arraySize );

    // display results
    if ( result != -1 )
        cout << '
' << key << " found in array element 
" << result << endl;
    else
        cout << '
' << key << " not found" << endl;

    return 0;  // indicates successful termination
}
```

Enter integer search key: 36
Found value in element 18
Enter integer search key: 37
Value not found
47 // function to perform binary search of an array
48 int binarySearch( const int b[], int searchKey, int low, int high, int size )
49 {
50     int middle;
51     
52     // loop until low subscript is greater than high subscript
53     while ( low <= high ) {
54         
55         // determine middle element of subarray being searched
56         middle = ( low + high ) / 2;
57         
58         // display subarray used in this loop iteration
59         printRow( b, low, middle, high, size );
60         
61         // if searchKey matches middle element, return middle
62         if ( searchKey == b[ middle ] )  // match
63             return middle;
64         
65         // if searchKey less than middle element,
66         // set new high element
67         else if ( searchKey < b[ middle ] )
68             high = middle - 1;  // search low end of array
69         
70         // if searchKey greater than middle element,
71         // set new low element
72         else
73             low = middle + 1;   // search high end of array
74         
75         } // end function binarySearch
76     
77     return -1;  // searchKey not found
78 } // end function binarySearch

82 // print header for output
83 void printHeader( int size )
84 {
85     cout << "
Subscripts:
";
86     
87     // output column heads
88     for ( int j = 0; j < size; j++ )
89         cout << setw( 3 ) << j << ' ';  
90     
91     cout << '
';  // start new line of output
92     
93     // output line of - characters
94     for ( int k = 1; k <= 4 * size; k++ )
95         cout << '-';
96     
97     cout << endl;  // start new line of output
98 } // end function printHeader

98 // print one row of output showing the current
99 // part of the array being processed
100 void printRow( const int b[], int low, int middle, int high, int size )
101 {
102     // loop through entire array
103     for ( int m = 0; m < size; m++ )
104         
105         // display spaces if outside current subarray range
106         if ( m < low || m > high )
107             cout << "    ";
108         
109         // display middle element marked with a *
110         else if ( m == middle )           // mark middle value
111             cout << setw( 3 ) << b[ m ] << '*';
112         
113         // display other elements in subarray
114         else
115             cout << setw( 3 ) << b[ m ] << ' ';
116         
117         } // end function printRow

Give a number between 0 and 28: 6
Subscripts:
0  2  4  6  8  10  12  14  16  18  20  22  24  26  28
0  2  4  6  8  10  12  14  16  18  20  22  24  26  28
8 found in array element 3

Give a number between 0 and 28: 25
Subscripts:
0  2  4  6  8  10  12  14  16  18  20  22  24  26  28
0  2  4  6  8  10  12  14  16  18  20  22  24  26  28
16  18  20  22  24  26  28
24  26  28
24*
25 not found

Give a number between 0 and 28: 8
Subscripts:
0  2  4  6  8  10  12  14  16  18  20  22  24  26  28
0  2  4  6  8  10  12  14  16  18  20  22  24  26  28
8  10  12
8*
8 found in array element 4
4.9 Multiple-Subscripted Arrays

• Multiple subscripts
  – \( a[i][j] \)
  – Tables with rows and columns
  – Specify row, then column
  – "Array of arrays"
• \( a[0][0] \) is an array of 4 elements
• \( a[0][0][0] \) is the first element of that array

To initialize
– Default of 0
– Initializers grouped by row in braces

```cpp
int b[2][2] = { { 1, 2 }, { 3, 4 } };  // Row 0 Row 1
```

• Referenced like normal
  – Outputs 0
  – Cannot reference using commas

```cpp
cout << b[0][1];  // Outputs 0
```

• Function prototypes
  – Must specify sizes of subscripts
  – First subscript not necessary, as with single-scripted arrays

```cpp
void printArray( int a[][3] );
```

For loops are often used to iterate through arrays. Nested loops are helpful with 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
// Fig. 4.23: fig04_23.cpp
// Double-subscripted array example.

#include <iostream>

using std::cout;
using std::endl;
using std::fixed;
using std::left;

#include <iomanip>

using std::setw;
using std::setprecision;

const int students = 3;   // number of students
const int exams = 4;      // number of exams

// function prototypes
int minimum( int [students][exams], int, int );
int maximum( int [students][exams], int, int );
double average( int [], int );
void printArray( int [students][exams], int, int );

int main()
{
    // 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
    cout << "The array is:
";
    printArray( studentGrades, students, exams );

    // determine smallest and largest grade values
    cout << "

Lowest grade: " << minimum( studentGrades, students, exams ) << "
Highest grade: " << maximum( studentGrades, students, exams ) << "
";

cout << fixed << setprecision( 2 );

// calculate average grade for each student
for ( int person = 0; person < students; person++ )
    cout << "The average grade for student " << person << " is " << average( studentGrades[person], exams ) << endl;

return 0;  // indicates successful termination
}

// find minimum grade
int minimum( int grades[students][exams], int pupils, int tests )
{
    int lowGrade = 100; // initialize to highest possible grade
    for ( int i = 0; i < pupils; i++ )
        for ( int j = 0; j < tests; j++ )
            if ( grades[i][j] < lowGrade )
                lowGrade = grades[i][j];
    return lowGrade;
}

// find maximum grade
int maximum( int grades[students][exams], int pupils, int tests )
{
    int highGrade = 0;  // initialize to lowest possible grade
    for ( int i = 0; i < pupils; i++ )
        for ( int j = 0; j < tests; j++ )
            if ( grades[i][j] > highGrade )
                highGrade = grades[i][j];
    return highGrade;
}

double average( int setOfGrades[], int tests )
{
    int total = 0;
    for ( int i = 0; i < tests; i++ )
        total += setOfGrades[i];
    return static_cast< double >( total ) / tests;  // average
}

void printArray( int grades[students][exams], int pupils, int tests )
{
    // set left justification and output column heads
    cout << left << "                 [0]  [1]  [2]  [3]";
    for ( int i = 0; i < pupils; i++ )
    {
        // output label for row
        cout << "
studentGrades[" << i << "] ";
        // output one grades for one student
        for ( int j = 0; j < tests; j++ )
            cout << setw( 5 ) << grades[i][j];
    }
} // end function printArray
The array is:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>studentGrades[0]</td>
<td>77</td>
<td>68</td>
<td>86</td>
</tr>
<tr>
<td>studentGrades[1]</td>
<td>96</td>
<td>87</td>
<td>89</td>
</tr>
<tr>
<td>studentGrades[2]</td>
<td>70</td>
<td>90</td>
<td>86</td>
</tr>
</tbody>
</table>

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