Chapter 4 - Arrays

Outline

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

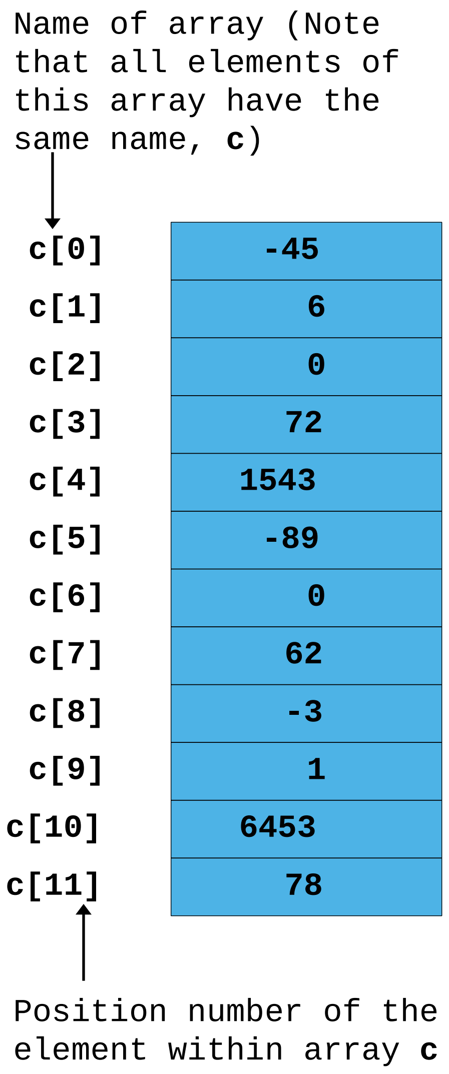
- Array elements like other variables
  - Assignment, printing for an integer array **`c`**

```
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

fig04_03.cpp
(1 of 2)

```
// Fig. 4.3: fig04_03.cpp
// Initializing an array.
#include <iostream>

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

#include <iomanip>

using std::setw;

int main()
{
   int n[ 10 ];  // n is an array o

   // initialize elements of array
   for ( int i = 0; i < 10; i++ )
      n[ i ] = 0;   // set element at location i to 0

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int j = 0; j < 10; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;

```

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]`.

```
    return 0;  // indicates successful termination

} // end main
```

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

Outline

fig04_03.cpp (2 of 2)

fig04_03.cpp output (1 of 1)

```
// Fig. 4.4: fig04_04.cpp
// Initializing an array with a declaration.
#include <iostream>

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

#include <iomanip>

using std::setw;

int main()
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int i = 0; i < 10; i++ )
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0;  // indicates successful termination

} // end main
```

Note the use of the initializer list.

Outline

fig04_04.cpp
(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
```

fig04_04.cpp output (1 of 1)

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

Outline

fig04_05.cpp
(1 of 2)

```
// Fig. 4.5: fig04_05.cpp
// Initialize array s to the even integers from 2 to 20.
#include <iostream>

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

#include <iomanip>

using std::setw;

int main()
{
   // constant variable can be used to
   const int arraySize = 10;

   int s[ arraySize ];  // array s has 10 e

   for ( int i = 0; i < arraySize; i++ )  //
      s[ i ] = 2 + 2 * i;

   cout << "Element" << setw( 13 ) << "Valu

```

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).

Outline

fig04_05.cpp (2 of 2)

fig04_05.cpp output (1 of 1)

```cpp
   // output contents of array s in tabular format
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;

   return 0;  // indicates successful termination

} // end main
```

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

Outline

fig04_06.cpp
(1 of 1)

fig04_06.cpp
output (1 of 1)

```
// Fig. 4.6: fig04_06.cpp
// Using a properly initialized constant variable.
#include <iostream>

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

int main()
{
   const int x = 7;  // initialized constant variable

   cout << "The value of constant variable x is: "
        << x << endl;

   return 0;  // indicates successful termination

} // end main
```

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

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

Outline

fig04_07.cpp (1 of 1)

fig04_07.cpp output (1 of 1)

```
// Fig. 4.7: fig04_07.cpp
// A const object must be initi

int main()
{
   const int x;   // Error: x m

   x = 7;         // Error: cannot modify a const variable

   return 0;      // indicates successful termination

} // 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
```

Outline

fig04_08.cpp (1 of 1)

fig04_08.cpp output (1 of 1)

```cpp
// Fig. 4.8: fig04_08.cpp
// Compute the sum of the elements of the array.
#include <iostream>

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

int main()
{
   const int arraySize = 10;

   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   int total = 0;

   // sum contents of array a
   for ( int i = 0; i < arraySize; i++ )
      total += a[ i ];

   cout << "Total of array element values is " << total << endl;

   return 0;  // indicates successful termination

} // end main
```

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

Outline

fig04_09.cpp
(1 of 2)

```
// Fig. 4.9: fig04_09.cpp
// Histogram printing program.
#include <iostream>

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

#include <iomanip>

using std::setw;

int main()
{
   const int arraySize = 10;
   int n[ arraySize ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };

   cout << "Element" << setw( 13 ) << "Value"
        << setw( 17 ) << "Histogram" << endl;

   // for each element of array n, output a bar in histogram
   for ( int i = 0; i < arraySize; i++ ) {
      cout << setw( 7 ) << i << setw( 13 )
           << n[ i ] << setw( 9 );

      for ( int j = 0; j < n[ i ]; j++ )   // print one bar
         cout << '*';
```

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

```

         cout << endl;  // start next line of output

      } // end outer for structure

      return 0;  // indicates successful termination

   } // 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          *
```

Outline

fig04_09.cpp (2 of 2)

fig04_09.cpp output (1 of 1)

Outline

fig04_10.cpp
(1 of 2)

```
// Fig. 4.10: fig04_10.cpp
// Roll a six-sided die 6000 times.
#include <iostream>

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

#include <iomanip>

using std::setw;

#include <cstdlib>
#include <ctime>

int main()
{
   const int arraySize = 7;
   int frequency[ arraySize ] = { 0 };

   srand( time( 0 ) );  // seed random-number

   // roll die 6000 times
   for ( int roll = 1; roll <= 6000; roll++ )
      ++frequency[ 1 + rand() % 6 ]; // replac
                                     // of Fig
```

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.

Outline

fig04_10.cpp
(2 of 2)

fig04_10.cpp
output (1 of 1)

```

   cout << "Face" << setw( 13 ) << "Frequency" << endl;

   // output frequency elements 1-6 in tabular format
   for ( int face = 1; face < arraySize; face++ )
      cout << setw( 4 ) << face
           << setw( 13 ) << frequency[ face ] << endl;

   return 0;  // indicates successful termination

} // end main
```

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

Outline

fig04_11.cpp
(1 of 2)

```cpp
// Fig. 4.11: fig04_11.cpp
// Student poll program.
#include <iostream>

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

#include <iomanip>

using std::setw;

int main()
{
   // define array sizes
   const int responseSize = 40;   // size of array responses
   const int frequencySize = 11;  // size of array frequency

   // place survey responses in array responses
   int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };

   // initialize frequency counters to 0
   int frequency[ frequencySize ] = { 0 };

```

Outline

fig04_11.cpp
(2 of 2)

```
   // for each answer, select value of an element of array
   // responses and use that value as subscript in array
   // frequency to determine element to increment
   for ( int answer = 0; answer < responseSize; answer++ )
      ++frequency[ responses[answer] ];

   // display results
   cout << "Rating" << setw( 17 ) <<

   // output frequencies in tabular f
   for ( int rating = 1; rating < frequencySize; rating++ )
      cout << setw( 6 ) << rating
           << setw( 17 ) << frequency[ rating ] << endl;

   return 0;  // indicates successful termination

} // end main
```

`responses[answer]` is the rating (from 1 to 10). This determines the index in **`frequency[]`** to increment.

Outline

fig04_11.cpp output (1 of 1)

```
Rating          Frequency
     1                  2
     2                  2
     3                  2
     4                  2
     5                  5
     6                 11
     7                  5
     8                  7
     9                  1
    10                  3
```

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'`**

4.4 Examples Using Arrays

- Input from keyboard

```
char string2[ 10 ];
cin >> string2;
```

  - 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

Outline

fig04_12.cpp
(1 of 2)

```
// Fig. 4_12: fig04_12.cpp
// Treating character arrays as strings.
#include <iostream>

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

int main()
{
   char string1[ 20 ],                 /
   char string2[] = "string literal"; /

   // read string from user into a
   cout << "Enter the string \"hel
   cin >> string1;  // reads "hello" [space terminates input]

   // output strings
   cout << "string1 is: " << string1
        << "\nstring2 is: " << string2;

   cout << "\nstring1 with spaces between characters is:\n";

```

Two different ways to declare strings. **`string2`** is initialized, and its size determined automatically .

Examples of reading strings from the keyboard and printing them out.

Outline

fig04_12.cpp
(2 of 2)

fig04_12.cpp
output (1 of 1)

```
    // output characters until null character is reached
    for ( int i = 0; string1[ i ] != '\0'; i++ )
       cout << string1[ i ] << ' ';

    cin >> string1;  // reads "there"
    cout << "\nstring1 is: " << string1 << en

    return 0;  // indicates successful termin

 } // end main
```

Can access the characters in a string using array notation. The loop ends when the **`null`** character is found.

```
Enter the string "hello there": hello there
string1 is: hello
string2 is: string literal
string1 with spaces between characters is:
h e l l o
string1 is: there
```

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

Outline

fig04_13.cpp
(1 of 3)

```
// Fig. 4.13: fig04_13.cpp
// Static arrays are initialized to zero.
#include <iostream>

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

void staticArrayInit( void );      // function prototype
void automaticArrayInit( void );   // function prototype

int main()
{
   cout << "First call to each function:\n";
   staticArrayInit();
   automaticArrayInit();

   cout << "\n\nSecond call to each function:\n";
   staticArrayInit();
   automaticArrayInit();
   cout << endl;

   return 0;  // indicates successful termination

} // end main

```

fig04_13.cpp
(2 of 3)

```
// function to demonstrate a stati
void staticArrayInit( void )
{
   // initializes elements to 0 first time function is called
   static int array1[ 3 ];

   cout << "\nValues on entering staticArrayInit:\n";

   // output contents of array1
   for ( int i = 0; i < 3; i++ )
      cout << "array1[" << i << "] = " << array1[ i ] << "  ";

   cout << "\nValues on exiting stati

   // modify and output contents of array1
   for ( int j = 0; j < 3; j++ )
      cout << "array1[" << j << "] = "
           << ( array1[ j ] += 5 ) << "  ";

} // end function staticArrayInit

```

Static array, initialized to zero on first function call.

Array data is changed; the modified values stay.

```
// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
   // initializes elements each time function is called
   int array2[ 3 ] = { 1, 2, 3 };

   cout << "\n\nValues on entering automaticArrayInit:\n";

   // output contents of array2
   for ( int i = 0; i < 3; i++ )
      cout << "array2[" << i << "] = " << array2[ i ] << "  ";

   cout << "\nValues on exiting automaticAr

   // modify and output contents of array2
   for ( int j = 0; j < 3; j++ )
      cout << "array2[" << j << "] = "
           << ( array2[ j ] += 5 ) << "  ";

} // end function automaticArrayInit
```

Automatic array, recreated with every function call.

Although the array is changed, it will be destroyed when the function exits and the changes will be lost.

Outline

fig04_13.cpp
output (1 of 1)

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

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 [] );`**

Outline

fig04_14.cpp
(1 of 3)

```
// Fig. 4.14: fig04_14.cpp
// Passing arrays and individual array elements to functions.
#include <iostream>

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

#include <iomanip>

using std::setw;

void modifyArray( int [], int );  // appears strange
void modifyElement( int );

int main()
{
   const int arraySize = 5;                // size of array a
   int a[ arraySize ] = { 0, 1, 2, 3, 4 };  // initialize a

   cout << "Effects of passing entire array by reference:"
        << "\n\nThe values of the original array are:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 3 ) << a[ i ];
```

Syntax for accepting an array in parameter list.

fig04_14.cpp
(2 of 3)

```

   cout << endl;

   // pass array a to modifyArray (
   modifyArray( a, arraySize );

   cout << "The values of the modified array are:\n";

   // output modified array
   for ( int j = 0; j < arraySize; j++ )
      cout << setw( 3 ) << a[ j ];

   // output value of a[ 3 ]
   cout << "\n\n\n"
        << "Effects of passing array e
        << "\n\nThe value of a[3] is "

   // pass array element a[ 3 ] by val
   modifyElement( a[ 3 ] );

   // output value of a[ 3 ]
   cout << "The value of a[3] is " << a[ 3 ] << endl;

   return 0;  // indicates successful termination

} // 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.

Outline

ig04_14.cpp (3 of 3)

```

// in function modifyArray, "b" points to
// the original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
   // multiply each array element by 2
   for ( int k = 0; k < sizeOfArray; k++ )
      b[ k ] *= 2;

} // end function modifyArray

// in function modifyElement, "e" is a lo
// array element a[ 3 ] passed from main
void modifyElement( int e )
{
   // multiply parameter by 2
   cout << "Value in modifyElement is "
        << ( e *= 2 ) << endl;

} // 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_14.cpp
output (1 of 1)

Outline

fig04_15.cpp
(1 of 2)

```
// Fig. 4.15: fig04_15.cpp
// Demonstrating the const type qualifier.
#include <iostream>

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

void tryToModifyArray( const int [] );  //

int main()
{
   int a[] = { 10, 20, 30 };

   tryToModifyArray( a );

   cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';

   return 0;  // indicates successful termination

} // end main

```

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

```
// In function tryToModifyArray, "b" cannot be used
// to modify the original array "a" in main.
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2;    // error
   b[ 1 ] /= 2;    // error
   b[ 2 ] /= 2;    // error

} // end function tryToModifyArray
```

```
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
```

fig04_15.cpp
(2 of 2)

fig04_15.cpp
output (1 of 1)

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 $\underline{2\ \ 4}$ $\underline{6\ \ 7}$ (elements exchanged)
  - Pass 2: $\underline{2\ \ 3}$ 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
```

Outline

fig04_16.cpp
(1 of 3)

```
// Fig. 4.16: fig04_16.cpp
// This program sorts an array's values into ascending order.
#include <iostream>

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

#include <iomanip>

using std::setw;

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\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << a[ i ];

```

fig04_16.cpp
(2 of 3)

```
   // bubble sort
   // loop to control number of passes
   for ( int pass = 0; pass < arraySize - 1; pass++ )

      // loop to control number of comparisons per pass
      for ( int j = 0; j < arraySize - 1; j++ )

         // compare side-by-side elements and swap t
         // first element is greater than second ele
         if ( a[ j ] > a[ j + 1 ] ) {
            hold = a[ j ];
            a[ j ] = a[ j + 1 ];
            a[ j + 1 ] = hold;

         } // end if

```

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.

Outline

fig04_16.cpp (3 of 3)

fig04_16.cpp output (1 of 1)

```
   cout << "\nData items in ascending order\n";

   // output sorted array
   for ( int k = 0; k < arraySize; k++ )
      cout << setw( 4 ) << a[ k ];

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

```
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)

Outline

```
// Fig. 4.17: fig04_17.cpp
// This program introduces the topic of survey data analysis.
// It computes the mean, median, and mode of the data.
#include <iostream>

using std::cout;
using std::endl;
using std::fixed;
using std::showpoint;

#include <iomanip>

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

void mean( const int [], int );
void median( int [], int );
void mode( int [], int [], int );
void bubbleSort( int[], int );
void printArray( const int[], int );

int main()
{
   const int responseSize = 99;  // size of array responses

```

fig04_17.cpp
(1 of 8)

Outline

fig04_17.cpp (2 of 8)

```
   int frequency[ 10 ] = { 0 };  // initialize array frequency

   // initialize array responses
   int response[ responseSize ] =
          { 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 };

   // process responses
   mean( response, responseSize );
   median( response, responseSize );
   mode( frequency, response, responseSize );

   return 0;  // indicates successful termination

} // end main

```

fig04_17.cpp
(3 of 8)

```cpp
// calculate average of all response values
void mean( const int answer[], int arraySize )
{
   int total = 0;

   cout << "********\n  Mean\n********\n";

   // total response values
   for ( int i = 0; i < arraySize; i++ )
      total += answer[ i ];

   // format and output results
   cout << fixed << setprecision( 4 );

   cout << "The mean is the average value of the data\n"
        << "items. The mean is equal to the total of\n"
        << "all the data items divided by the numb
        << "of data items (" << arraySize
        << "). The mean value for\nthis run is: "
        << total << " / " << arraySize << " = "
        << static_cast< double >( total ) / arraySize
        << "\n\n";

} // end function mean

```

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

Outline

fig04_17.cpp
(4 of 8)

```
// sort array and determine median element's value
void median( int answer[], int size )
{
   cout << "\n********\n Median\n********\n"
        << "The unsorted array of

   printArray( answer, size );  //

   bubbleSort( answer, size );  // sort array

   cout << "\n\nThe sorted array is";
   printArray( answer, size );  // output sorted array

   // display median element
   cout << "\n\nThe median is element " << size / 2
        << " of\nthe sorted " << size
        << " element array.\nFor this run the median is "
        << answer[ size / 2 ] << "\n\n";

} // end function median

```

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

```
// determine most frequent response
void mode( int freq[], int answer[], int size )
{
   int largest = 0;    // represents largest frequency
   int modeValue = 0;  // represents most frequent response

   cout << "\n********\n  Mode\n********\n";

   // initialize frequencies to 0
   for ( int i = 1; i <= 9; i++ )
      freq[ i ] = 0;

   // summarize frequencies
   for ( int j = 0; j < size; j++ )
      ++freq[ answer[ j ] ];

   // output headers for result columns
   cout << "Response" << setw( 11 ) << "Frequency"
        << setw( 19 ) << "Histogram\n\n" << setw( 55 )
        << "1    1    2    2\n" << setw( 56 )
        << "5    0    5    0    5\n\n";

```

fig04_17.cpp
(5 of 8)

Outline

fig04_17.cpp
(6 of 8)

```
   // output results
   for ( int rating = 1; rating <= 9; rating++ ) {
      cout << setw( 8 ) << rating << s
           << freq[ rating ] << "

      // keep track of mode value and
      if ( freq[ rating ] > largest ) {
         largest = freq[ rating ];
         modeValue = rating;

      } // end if

      // output histogram bar representing frequency value
      for ( int k = 1; k <= freq[ rating ]; k++ )
         cout << '*';

      cout << '\n';  // begin new line of output

   } // end outer for

   // display the mode value
   cout << "The mode is the most frequent value.\n"
        << "For this run the mode is " << modeValue
        << " which occurred " << largest << " times." << endl;

} // end function mode
```

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

Outline

fig04_17.cpp
(7 of 8)

```

// 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;

         } // end if

} // end function bubbleSort

```

Outline

fig04_17.cpp
(8 of 8)

```
// output array contents (20 values per row)
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 ];

   } // end for

} // end function printArray
```

Outline

fig04_17.cpp output (1 of 2)

```
********
  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
```

Outline

fig04_17.cpp output (2 of 2)

```
********
  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 $2^N > \#$ of elements
    - 30 element array takes at most 5 steps

      $2^5 > 30$

Outline

fig04_19.cpp
(1 of 2)

```
// 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 );

```

Takes array, search key, and array size.

Outline

fig04_19.cpp (2 of 2)

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

   return 0;  // indicates successful termination

} // end main

// compare key to every element of array until location is
// found or until end of array is reached; return subscript of
// element if key or -1 if key not found
int linearSearch( const int array[], int key, int sizeOfArray )
{
   for ( int j = 0; j < sizeOfArray; j++ )

      if ( array[ j ] == key )  // if found,
         return j;              // return location of key

   return -1;  // key not found

} // 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)

Outline

fig04_20.cpp
(1 of 6)

```
// 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;

```

Outline

fig04_20.cpp
(2 of 6)

```
    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 << '\n' << key << " found in array element "
               << result << endl;
    else
       cout << '\n' << key << " not found" << endl;

    return 0;  // indicates successful termination

 } // end main

```

```
// function to perform binary search of an array
int binarySearch( const int b[], int searchKey, int low,
   int high, int size )
{
   int middle;

   // loop until low subscript is gre
   while ( low <= high ) {

      // determine middle element of subarray being searched
      middle = ( low + high ) / 2;

      // display subarray used in this loop iteration
      printRow( b, low, middle, high, size );

```

Determine middle element

fig04_20.cpp
(3 of 6)

Outline

fig04_20.cpp
(4 of 6)

```
      // if searchKey matches middle element, return middle
      if ( searchKey == b[ middle ] )  // match
         return middle;

      else

         // if searchKey less than middle e
         // set new high element
         if ( searchKey < b[ middle ] )
            high = middle - 1;  // search low end of array

         // if searchKey greater than middl
         // set new low element
         else
            low = middle + 1;   // search h
   }

   return -1;  // searchKey not found

} // end function binarySearch
```

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.

```

// print header for output
void printHeader( int size )
{
   cout << "\nSubscripts:\n";

   // output column heads
   for ( int j = 0; j < size; j++ )
      cout << setw( 3 ) << j << ' ';

   cout << '\n';  // start new line of output

   // output line of - characters
   for ( int k = 1; k <= 4 * size; k++ )
      cout << '-';

   cout << endl;  // start new line of output

} // end function printHeader

```

fig04_20.cpp
(5 of 6)

```
// print one row of output showing the current
// part of the array being processed
void printRow( const int b[], int low, int mid,
   int high, int size )
{
   // loop through entire array
   for ( int m = 0; m < size; m++ )

      // display spaces if outside current subarray range
      if ( m < low || m > high )
         cout << "    ";

      // display middle element marked with a *
      else

         if ( m == mid )           // mark middle value
            cout << setw( 3 ) << b[ m ] << '*';

         // display other elements in subarray
         else
            cout << setw( 3 ) << b[ m ] << ' ';

   cout << endl;  // start new line of output

} // end function printRow
```

fig04_20.cpp
(6 of 6)

Outline

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: 25

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
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 not found
```

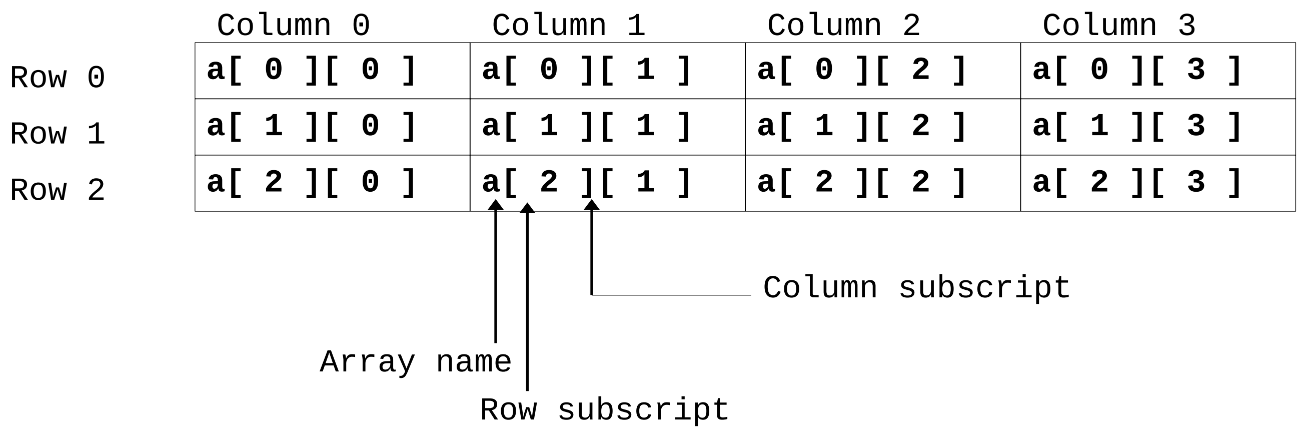
```
Enter a number between 0 and 28: 8

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
```

Outline

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



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

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

- Referenced like normal

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

1	0
3	4

  - 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 ] );`**

Outline

fig04_22.cpp
(1 of 2)

```cpp
// Fig. 4.22: fig04_22.cpp
// Initializing multidimensional arrays.
#include <iostream>

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

void printArray( int [][ 3 ] );

int main()
{
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   cout << "Values in array1 by row are:" << endl;
   printArray( array1 );

   cout << "Values in array2 by row are:" << endl;
   printArray( array2 );

   cout << "Values in array3 by row are:" << endl;
   printArray( array3 );

   return 0;  // indicates successful termination

} // 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.

```

// function to output array with two rows
void printArray( int a[][ 3 ] )
{
   for ( int i = 0; i < 2; i++ ) {    // f

      for ( int j = 0; j < 3; j++ )   // output column values
         cout << a[ i ][ j ] << ' ';

      cout << endl;  // start new line of output

   } // end outer for structure

} // 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
```

fig04_22.cpp
(2 of 2)

fig04_22.cpp
output (1 of 1)

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)

```cpp
// 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 [][ exams ], int, int );
int maximum( int [][ exams ], int, int );
double average( int [], int );
void printArray( int [][ exams ], int, int );

```

Outline

fig04_23.cpp
(2 of 6)

```
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:\n";
   printArray( studentGrades, students, exams );

   // determine smallest and largest grade values
   cout << "\n\nLowest grade: "
        << minimum( studentGrades, students, exams )
        << "\nHighest grade: "
        << maximum( studentGrades, students, exams ) << '\n';

   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 termi

} // end main

// find minimum grade
int minimum( int grades[][ exams ], int pup
{
   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;

} // end function minimum
```

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

Outline

fig04_23.cpp
(4 of 6)

```

// find maximum grade
int maximum( int grades[][ 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;

} // end function maximum

```

Outline

fig04_23.cpp
(5 of 6)

```
// determine average grade for particular student
double average( int setOfGrades[], int tests )
{
   int total = 0;

   // total all grades for one student
   for ( int i = 0; i < tests; i++ )
      total += setOfGrades[ i ];

   return static_cast< double >( total ) / tests;  // average

} // end function maximum
```

Outline

fig04_23.cpp
(6 of 6)

```

// Print the array
void printArray( int grades[][ exams ], int pupils, int tests )
{
   // set left justification and output column heads
   cout << left << "                  [0]  [1]  [2]  [3]";

   // output grades in tabular format
   for ( int i = 0; i < pupils; i++ ) {

      // output label for row
      cout << "\nstudentGrades[" << i << "] ";

      // output one grades for one student
      for ( int j = 0; j < tests; j++ )
         cout << setw( 5 ) << grades[ i ][ j ];

   } // end outer for

} // 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
```

Outline

fig04_23.cpp
output (1 of 1)