7

Arrays and Vectors

Now go, write it before them in a table, and note it in a book.

— Isaiah 30:8

To go beyond is as wrong as to fall short.

— Confucius

Begin at the beginning... and go on till you come to the end: then stop.

— Lewis Carroll

OBJECTIVES

In this chapter you will learn:

- To use the array data structure to represent a set of related data items.
- To use arrays to store, sort and search lists and tables of values.
- To declare arrays, initialize arrays and refer to the individual elements of arrays.
- To pass arrays to functions.
- Basic searching and sorting techniques.
- To declare and manipulate multidimensional arrays.
- To use C++ Standard Library class template `vector`.

Outline

7.1 Introduction

- **Arrays**
  - **Data structures containing related data items of same type**
  - **Always remain the same size once created**
    - **Are “static” entities**
  - **Character arrays can also represent strings**
  - **C-style pointer-based arrays vs. vectors (object-based)**
    - **Vectors are safer and more versatile**

7.2 Arrays

- **Array**
  - **Consecutive group of memory locations**
    - **All of which have the same type**
  - **Index**
    - **Position number used to refer to a specific location/element**
    - **Also called subscript**
    - **Place in square brackets**
      - **Must be positive integer or integer expression**
    - **First element has index zero**
    - **Example (assume `a = 5` and `b = 6`)**
      - `c[ a + b ] += 2;`
        - **Adds 2 to array element `c[ 11 ]`**

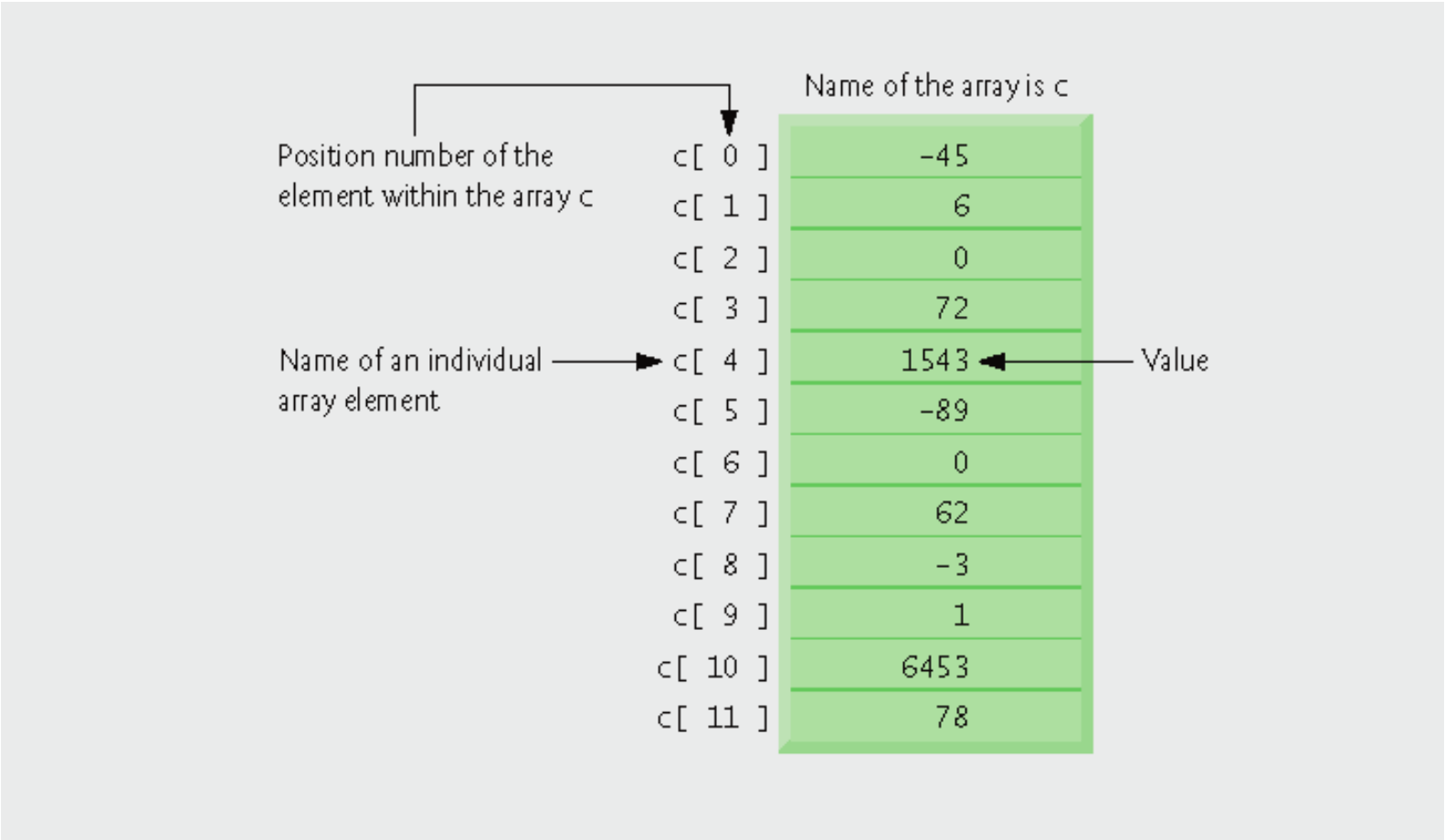

Fig.7.1 | Array of 12 elements

7.2 Arrays (Cont.)

- **Examine array `c` in Fig. 7.1**
  - **`c` is the array *name***
  - **`c` has 12 *elements* (`c[0]`, `c[1]`, … `c[11]`)**
    - **The *value* of `c[0]` is `-45`**
- **Brackets used to enclose an array subscript are actually an operator in C++**

Common Programming Error 7.1

It is important to note the difference between the "seventh element of the array" and "array element 7." Array subscripts begin at 0, so the "seventh element of the array" has a subscript of 6, while "array element 7" has a subscript of 7 and is actually the eighth element of the array. Unfortunately, this distinction frequently is a source of off-by-one errors. To avoid such errors, we refer to specific array elements explicitly by their array name and subscript number (e.g., `c[ 6 ]` or `c[ 7 ]`).

Operators	Associativity	Type
`()` `[]`	left to right	highest
`++` `--` `static_cast<` *type* `>(` *operand* `)`	left to right	unary (postfix)
`++` `--` `+` `-` `!`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig.7.2 | Operator precedence and associativity.

7.3 Declaring Arrays

- **Declaring an array**
  - **Arrays occupy space in memory**
  - **Programmer specifies type and number of elements**
    - **Example**
      - `int c[ 12 ];`
        - `c` **is an array of** `12` `ints`
  - **Array's size must be an integer constant greater than zero**
  - **Arrays can be declared to contain values of any non-reference data type**
  - **Multiple arrays of the same type can be declared in a single declaration**
    - **Use a comma-separated list of names and sizes**

Good Programming Practice 7.1

We prefer to declare one array per declaration for readability, modifiability and ease of commenting.

7.4 Examples Using Arrays

- **Using a loop to initialize the array’s elements**
  - **Declare array, specify number of elements**
  - **Use repetition statement to loop for each element**
    - **Use body of repetition statement to initialize each individual array element**

Outline

fig07_03.cpp

(1 of 2)

```
1  // Fig. 7.3: fig07_03.cpp
2  // Initializing an array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    int n[ 10 ]; // n is an array of 10 integers
13
14    // initialize elements of array n to 0
15    for ( int i = 0; i < 10; i++ )
16       n[ i ] = 0; // set element at location i to 0
17
18    cout << "Element" << setw( 13 ) << "Value" << endl;
```

Declare **n** as an array of **int**s with 10 elements

Each **int** initialized is to **0**

Outline

```
19
20    // output each array element's value
21    for ( int j = 0; j < 10; j++ )
22       cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
23
24    return 0; // indicates successful termination
25 } // end main
```

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

n[j] returns **int** associated with index **j** in array **n**

Each **int** has been initialized to **0**

fig07_03.cpp

(2 of 2)

7.4 Examples Using Arrays (Cont.)

- **Initializing an array in a declaration with an initializer list**
  - **Initializer list**
    - **Items enclosed in braces ({})**
    - **Items in list separated by commas**
    - **Example**
      - `int n[] = { 10, 20, 30, 40, 50 };`
        - **Because array size is omitted in the declaration, the compiler determines the size of the array based on the size of the initializer list**
        - **Creates a five-element array**
        - **Index values are 0, 1, 2, 3, 4**
        - **Initialized to values 10, 20, 30, 40, 50, respectively**

7.4 Examples Using Arrays (Cont.)

- **Initializing an array in a declaration with an initializer list (Cont.)**
  - **If fewer initializers than elements in the array**
    - **Remaining elements are initialized to zero**
    - **Example**
      - `int n[ 10 ] = { 0 };`
        - **Explicitly initializes first element to zero**
        - **Implicitly initializes remaining nine elements to zero**
  - **If more initializers than elements in the array**
    - **Compilation error**

Outline

fig07_04.cpp

(1 of 2)

```
1  // Fig. 7.4: fig07_04.cpp
2  // Initializing an array in a declaration.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // use initializer list to initialize array n
13    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
14
15    cout << "Element" << setw( 13 ) << "Value" << endl;
```

Declare **n** as an array of **int**s

Compiler uses initializer list to initialize array

```
16
17    // output each array element's value
18    for ( int i = 0; i < 10; i++ )
19       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
20
21    return 0; // indicates successful termination
22 } // end main
```

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

Outline

fig07_04.cpp

(2 of 2)

Common Programming Error 7.2

Providing more initializers in an array initializer list than there are elements in the array is a compilation error.

Common Programming Error 7.3

Forgetting to initialize the elements of an array whose elements should be initialized is a logic error.

7.4 Examples Using Arrays (Cont.)

- **Specifying an array's size with a constant variable and setting array elements with calculations**
  - **Initialize elements of 10-element array to even integers**
  - **Use repetition statement that calculates value for current element, initializes array element using calculated value**

Outline

fig07_05.cpp

(1 of 2)

```
1  // Fig. 7.5: fig07_05.cpp
2  // Set array s to the even integers from 2 to 20.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // constant variable can be used to specify array size
13    const int arraySize = 10;
14
15    int s[ arraySize ]; // array s has 10 elements
16
17    for ( int i = 0; i < arraySize; i++ ) // set the values
18       s[ i ] = 2 + 2 * i;
```

Declare constant variable **arraySize** using the **const** keyword

Declare array that contains 10 **int**s

Use array index to assign element's value

```
19
20    cout << "Element" << setw( 13 ) << "Value" << endl;
21
22    // output contents of array s in tabular format
23    for ( int j = 0; j < arraySize; j++ )
24       cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
25
26    return 0; // indicates successful termination
27 } // 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

fig07_05.cpp

(2 of 2)

7.4 Examples Using Arrays (Cont.)

- **Constant variables**
  - **Declared using the `const` qualifier**
  - **Also called name constants or read-only variables**
  - **Must be initialized with a constant expression when they are declared and cannot be modified thereafter**
  - **Can be placed anywhere a constant expression is expected**
  - **Using constant variables to specify array sizes makes programs more scalable and eliminates "magic numbers"**

Common Programming Error 7.4

Not assigning a value to a constant variable when it is declared is a compilation error.

Common Programming Error 7.5

Assigning a value to a constant variable in an executable statement is a compilation error.

Outline

fig07_06.cpp

(1 of 1)

```
1  // Fig. 7.6: fig07_06.cpp
2  // Using a properly initialized constant variable.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     const int x = 7; // initialized constant variable
10
11    cout << "The value of constant variable x is: " << x << endl;
12
13    return 0; // indicates successful termination
14 } // end main
```

Declaring constant value

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

Outline

```
1  // Fig. 7.7: fig07_07.cpp
2  // A const variable 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 } // end main
```

Must initialize a constant at the time of declaration

Cannot modify a constant

fig07_07.cpp

(1 of 1)

Borland C++ command-line compiler error message:

```
Error E2304 fig07_07.cpp 6: Constant variable 'x' must be initialized
   in function main()
Error E2024 fig07_07.cpp 8: Cannot modify a const object in function main()
```

Microsoft Visual C++.NET compiler error message:

```
C:\cpphtp5_examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object
   must be initialized if not extern
C:\cpphtp5_examples\ch07\fig07_07.cpp(8) : error C2166: l-value specifies
   const object
```

GNU C++ compiler error message:

```
fig07_07.cpp:6: error: uninitialized const `x'
fig07_07.cpp:8: error: assignment of read-only variable `x'
```

Error messages differ based on the compiler

Common Programming Error 7.6

Only constants can be used to declare the size of automatic and static arrays. Not using a constant for this purpose is a compilation error.

Software Engineering Observation 7.1

Defining the size of each array as a constant variable instead of a literal constant can make programs more scalable.

Good Programming Practice 7.2

Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique eliminates so-called magic numbers. For example, repeatedly mentioning the size 10 in array-processing code for a 10-element array gives the number 10 an artificial significance and can unfortunately confuse the reader when the program includes other 10s that have nothing to do with the array size.

7.4 Examples Using Arrays (Cont.)

- **Summing the elements of an array**
  - **Array elements can represent a series of values**
    - **We can sum these values**
    - **Use repetition statement to loop through each element**
      - **Add element value to a total**

Outline

fig07_08.cpp

(1 of 1)

```
1  // Fig. 7.8: fig07_08.cpp
2  // Compute the sum of the elements of the array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     const int arraySize = 10; // constant variable indicating size of array
10    int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11    int total = 0;
12
13    // sum contents of array a
14    for ( int i = 0; i < arraySize; i++ )
15       total += a[ i ];
16
17    cout << "Total of array elements: " << total << endl;
18
19    return 0; // indicates successful termination
20 } // end main
```

Declare array with initializer list

Sum all array values

```
Total of array elements: 849
```

7.4 Examples Using Arrays (Cont.)

- **Using bar charts to display array data graphically**
  - **Present data in graphical manner**
    - **E.g., bar chart**
  - **Examine the distribution of grades**
  - **Nested `for` statement used to output bars**

Outline

fig07_09.cpp

(1 of 2)

```
1  // Fig. 7.9: fig07_09.cpp
2  // Bar chart printing program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    const int arraySize = 11;
13    int n[ arraySize ] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
14
15    cout << "Grade distribution:" << endl;
16
17    // for each element of array n, output a bar of the chart
18    for ( int i = 0; i < arraySize; i++ )
19    {
20       // output bar labels ("0-9:", ..., "90-99:", "100:" )
21       if ( i == 0 )
22          cout << "  0-9: ";
23       else if ( i == 10 )
24          cout << "  100: ";
25       else
26          cout << i * 10 << "-" << ( i * 10 ) + 9 << ": ";
```

Declare **`array`** with initializer list

```
27
28       // print bar of asterisks
29       for ( int stars = 0; stars < n[ i ]; stars++ )
30          cout << '*';
31
32       cout << endl; // start a new line of ou
33    } // end outer for
34
35    return 0; // indicates successful termination
36 } // end main
```

For each array element, print the associated number of asterisks

```
Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Outline

fig07_09.cpp

(2 of 2)

Common Programming Error 7.7

Although it is possible to use the same control variable in a `for` statement and a second `for` statement nested inside, this is confusing and can lead to logic errors.

7.4 Examples Using Arrays (Cont.)

- **Using the elements of an array as counters**
  - **Use a series of counter variables to summarize data**
  - **Counter variables make up an array**
  - **Store frequency values**

Outline

fig07_10.cpp

(1 of 2)

```
1  // Fig. 7.10: fig07_10.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib>
11 using std::rand;
12 using std::srand;
13
14 #include <ctime>
15 using std::time;
16
17 int main()
18 {
19    const int arraySize = 7; // ignore element zero
20    int frequency[ arraySize ] = { 0 };
21
22    srand( time( 0 ) ); // seed random number generator
23
24    // roll die 6,000,000 times; use die value as frequency index
25    for ( int roll = 1; roll <= 6000000; roll++ )
26       frequency[ 1 + rand() % 6 ]++;
```

Declare **frequency** as array of **7 int**s

Generate **6000000** random integers in range **1** to **6**

Increment **frequency** values at the index associated with the random number

```
27
28    cout << "Face" << setw( 13 ) << "Frequency" << endl;
29
30    // output each array element's value
31    for ( int face = 1; face < arraySize; face++ )
32       cout << setw( 4 ) << face << setw( 13 ) << frequency[ face ]
33          << endl;
34
35    return 0; // indicates successful termination
36 } // end main
```

```
Face    Frequency
   1      1000167
   2      1000149
   3      1000152
   4       998748
   5       999626
   6      1001158
```

Outline

fig07_10.cpp

(2 of 2)

7.4 Examples Using Arrays (Cont.)

- **Using arrays to summarize survey results**
  - **40 students rate the quality of food**
    - **1-10 rating scale: 1 means awful, 10 means excellent**
  - **Place 40 responses in an array of integers**
  - **Summarize results**
  - **Each element of the array used as a counter for one of the survey responses**
- **C++ has no array bounds checking**
  - **Does not prevent the computer from referring to an element that does not exist**
    - **Could lead to serious execution-time errors**

Outline

fig07_11.cpp

(1 of 2)

```
1  // Fig. 7.11: fig07_11.cpp
2  // Student poll program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // define array sizes
13    const int responseSize = 40; // size of array responses
14    const int frequencySize = 11; // size of array frequenc
15
16    // place survey responses in array responses
17    const int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
18       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
19       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21    // initialize frequency counters to 0
22    int frequency[ frequencySize ] = { 0 };
23
24    // for each answer, select responses element and use that value
25    // as frequency subscript to determine element to increment
26    for ( int answer = 0; answer < responseSize; answer++ )
27       frequency[ responses[ answer ] ]++;
28
29    cout << "Rating" << setw( 17 ) << "Frequency" << er
```

Array **responses** will store **40** responses

Array **frequency** will contain **11** **int**s (ignore the first element)

Initialize **responses** with 40 responses

Initialize **frequency** to all **0**s

For each response, increment **frequency** value at the index associated with that response

```
30
31    // output each array element's value
32    for ( int rating = 1; rating < frequencySize; rating++ )
33       cout << setw( 6 ) << rating << setw( 17 ) << frequency[ rating ]
34          << endl;
35
36    return 0; // indicates successful termination
37 } // end main
```

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

Outline

fig07_11.cpp

(2 of 2)

Software Engineering Observation 7.2

The `const` qualifier should be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.

Good Programming Practice 7.3

Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.

Performance Tip 7.1

Sometimes performance considerations far outweigh clarity considerations.

Common Programming Error 7.8

Referring to an element outside the array bounds is an execution-time logic error. It is not a syntax error.

Error-Prevention Tip 7.1

When looping through an array, the array subscript should never go below 0 and should always be less than the total number of elements in the array (one less than the size of the array). Make sure that the loop-termination condition prevents accessing elements outside this range.

Portability Tip 7.1

The (normally serious) effects of referencing elements outside the array bounds are system dependent. Often this results in changes to the value of an unrelated variable or a fatal error that terminates program execution.

Error-Prevention Tip 7.2

In Chapter 11, we will see how to develop a class representing a "smart array," which checks that all subscript references are in bounds at runtime. Using such smart data types helps eliminate bugs.

7.4 Examples Using Arrays (Cont.)

- **Using character arrays to store and manipulate strings**
  - **Arrays may be of any type, including `char`s**
    - **We can store character strings in `char` arrays**
  - **Can be initialized using a string literal**
    - **Example**
      - `char string1[] = "Hi";`
    - **Equivalent to**
      - `char string1[] = { 'H', 'i', '\0' };`
  - **Array contains each character plus a special string-termination character called the null character (`'\0'`)**

7.4 Examples Using Arrays (Cont.)

- **Using character arrays to store and manipulate strings (Cont.)**
  - **Can also be initialized with individual character constants in an initializer list**

```
char string1[] =
      { 'f', 'i', 'r', 's', 't', '\0' };
```

  - **Can also input a string directly into a character array from the keyboard using `cin` and `>>`**

```
cin >> string1;
```

    - `cin >>` **may read more characters than the array can store**
  - **A character array representing a null-terminated string can be output with `cout` and `<<`**

Common Programming Error 7.9

Not providing `cin >>` with a character array large enough to store a string typed at the keyboard can result in loss of data in a program and other serious runtime errors.

Outline

fig07_12.cpp

(1 of 2)

```
1  // Fig. 7.12: fig07_12.cpp
2  // Treating character arrays as strings.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int main()
9  {
10    char string1[ 20 ]; // reserves 20 characters
11    char string2[] = "string literal"; // reserves 15 characters
12
13    // read string from user into array string1
14    cout << "Enter the string \"hello there\": ";
15    cin >> string1; // reads "hello" [space terminates input]
16
17    // output strings
18    cout << "string1 is: " << string1 << "\nstring2 is: " << string2;
19
20    cout << "\nstring1 with spaces between characters is:\n";
21
```

Store **"string literal"** as an array of characters

Initializing an array of characters using **cin**

Output array using **cin**

```
22    // output characters until null character is reached
23    for ( int i = 0; string1[ i ] != '\0'; i++ )
24       cout << string1[ i ] << ' ';
25
26    cin >> string1; // reads "there"
27    cout << "\nstring1 is: " << string1 << endl;
28
29    return 0; // indicates successful termination
30 } // end main
```

Loop until the terminating null character is reached

Accessing specific characters in the array

```
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
```


fig07_12.cpp

(2 of 2)

7.4 Examples Using Arrays (Cont.)

- `static` local arrays and automatic local arrays
  - A `static` local variable in a function
    - Exists for the duration of the program
    - But is visible only in the function body
  - A `static` local array
    - Exists for the duration of the program
    - Is initialized when its declaration is first encountered
      - All elements are initialized to zero if not explicitly initialized
        - This does not happen for automatic local arrays

Performance Tip 7.2

We can apply `static` to a local array declaration so that the array is not created and initialized each time the program calls the function and is not destroyed each time the function terminates in the program. This can improve performance, especially when using large arrays.

Outline

fig07_13.cpp

(1 of 3)

```
1  // Fig. 7.13: fig07_13.cpp
2  // Static arrays are initialized to zero.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void staticArrayInit( void ); // function prototype
8  void automaticArrayInit( void ); // function prototype
9
10 int main()
11 {
12    cout << "First call to each function:\n";
13    staticArrayInit();
14    automaticArrayInit();
15
16    cout << "\n\nSecond call to each function:\n";
17    staticArrayInit();
18    automaticArrayInit();
19    cout << endl;
20
21    return 0; // indicates successful termination
22 } // end main
```

Outline

fig07_13.cpp

```
23
24 // function to demonstrate a static local array
25 void staticArrayInit( void )
26 {
27    // initializes elements to 0 first time function is called
28    static int array1[ 3 ]; // static local array
29
30    cout << "\nValues on entering staticArrayInit:\n";
31
32    // output contents of array1
33    for ( int i = 0; i < 3; i++ )
34       cout << "array1[" << i << "] = " << array1[ i ] << "  ";
35
36    cout << "\nValues on exiting staticArrayInit:\n";
37
38    // modify and output contents of array1
39    for ( int j = 0; j < 3; j++ )
40       cout << "array1[" << j << "] = " << ( array1[ j ] += 5 ) << "  ";
41 } // end function staticArrayInit
42
43 // function to demonstrate an automatic local array
44 void automaticArrayInit( void )
45 {
46    // initializes elements each time function is called
47    int array2[ 3 ] = { 1, 2, 3 }; // automatic local array
48
49    cout << "\n\nValues on entering automaticArrayInit:\n";
```

Create a **`static`** array using keyword **`static`**

Create an automatic local array

Outline

```
50
51    // output contents of array2
52    for ( int i = 0; i < 3; i++ )
53       cout << "array2[" << i << "] = " << array2[ i ] << "   ";
54
55    cout << "\nValues on exiting automaticArrayInit:\n";
56
57    // modify and output contents of array2
58    for ( int j = 0; j < 3; j++ )
59       cout << "array2[" << j << "] = " << ( array2[ j ] += 5 ) << "   ";
60 } // end function automaticArrayInit
```

```
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
```

Values reflect changes from the previous function call – the array was not reinitialized

fig07_13.cpp

(3 of 3)

Common Programming Error 7.10

Assuming that elements of a function's local `static` array are initialized every time the function is called can lead to logic errors in a program.

7.5 Passing Arrays to Functions

- To pass an array argument to a function
  - Specify array name without brackets
    - Array `hourlyTemperatures` is declared as

      ```
      int hourlyTemperatures[ 24 ];
      ```

    - The function call

      ```
      modifyArray( hourlyTemperatures, 24 );
      ```

      passes array `hourlyTemperatures` and its size to function `modifyArray`
  - Array size is normally passed as another argument so the function can process the specific number of elements in the array

7.5 Passing Arrays to Functions (Cont.)

- **Arrays are passed by reference**
  - **Function call actually passes starting address of array**
    - **So function knows where array is located in memory**
  - **Caller gives called function direct access to caller's data**
    - **Called function can manipulate this data**

Performance Tip 7.3

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time consuming and would require considerable storage for the copies of the array elements.

Software Engineering Observation 7.3

It is possible to pass an array by value (by using a simple trick we explain in Chapter 22)—this is rarely done.

7.5 Passing Arrays to Functions (Cont.)

- **Individual array elements passed by value**
  - **Single pieces of data**
    - **Known as scalars or scalar quantities**
  - **To pass an element to a function**
    - **Use the subscripted name of the array element as an argument**
- **Functions that take arrays as arguments**
  - **Function parameter list must specify array parameter**
    - **Example**
      - `void modArray( int b[], int arraySize );`

7.5 Passing Arrays to Functions (Cont.)

- **Functions that take arrays as arguments (Cont.)**
  - **Array parameter may include the size of the array**
    - **Compiler will ignore it, though**
      - **Compiler only cares about the address of the first element**
- **Function prototypes may include parameter names**
  - **But the compiler will ignore them**
  - **Parameter names may be left out of function prototypes**

Outline

fig07_14.cpp

(1 of 3)

```
1  // Fig. 7.14: fig07_14.cpp
2  // Passing arrays and individual array elements to functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 void modifyArray( int [], int ); // appears strange
11 void modifyElement( int );
12
13 int main()
14 {
15    const int arraySize = 5; // size of array a
16    int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a
17
18    cout << "Effects of passing entire array by reference:"
19       << "\n\nThe values of the original array are:\n";
20
21    // output original array elements
22    for ( int i = 0; i < arraySize; i++ )
23       cout << setw( 3 ) << a[ i ];
24
25    cout << endl;
26
27    // pass array a to modifyArray by reference
28    modifyArray( a, arraySize );
29    cout << "The values of the modified array are:\n";
```

Function takes an array as argument

Declare **5-int** array **array** with initializer list

Pass entire array to function `modifyArray`

Outline

```
30
31    // output modified array elements
32    for ( int j = 0; j < arraySize; j++ )
33       cout << setw( 3 ) << a[ j ];
34
35    cout << "\n\n\nEffects of passing array element by value:"
36       << "\n\na[3] before modifyElement: " << a[ 3 ] << endl;
37
38    modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
39    cout << "a[3] after modifyElement: " << a[ 3 ] << endl;
40
41    return 0; // indicates successful termination
42 } // end main
43
44 // in function modifyArray, "b" points to the original arra
45 void modifyArray( int b[], int sizeOfArray )
46 {
47    // multiply each array element by 2
48    for ( int k = 0; k < sizeOfArray; k++ )
49       b[ k ] *= 2;
50 } // end function modifyArray
```

Pass array element **a[3]** to function **modifyElement**

(2 of 3)

Function **modifyArray** manipulates the array directly

```
51
52 // in function modifyElement, "e" is a local copy of
53 // array element a[ 3 ] passed from main
54 void modifyElement( int e )
55 {
56    // multiply parameter by 2
57    cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
58 } // end function modifyElement
```

Function **modifyElement** manipulates array element's copy

```
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:

a[3] before modifyElement: 6
Value of element in modifyElement: 12
a[3] after modifyElement: 6
```

Outline

fig07_14.cpp

(3 of 3)

7.5 Passing Arrays to Functions (Cont.)

- `const` array parameters
  - Qualifier `const`
  - Prevent modification of array values in the caller by code in the called function
  - Elements in the array are constant in the function body
  - Enables programmer to prevent accidental modification of data

```
1  // Fig. 7.15: fig07_15.cpp
2  // Demonstrating the const type qualifier.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void tryToModifyArray( const int [] ); // function prototype
8
9  int main()
10 {
11    int a[] = { 10, 20, 30 };
12
13    tryToModifyArray( a );
14    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
15
16    return 0; // indicates successful termination
17 } // end main
18
```

Using **const** to prevent the function from modifying the array

Array **a** will be **const** when in the body of the function

Outline

fig07_15.cpp

(1 of 2)

Outline

`fig07_15.cpp`

(2 of 2)

```
19 // In function tryToModifyArray, "b" cannot be used
20 // to modify the original array "a" in main.
21 void tryToModifyArray( const int b[] )
22 {
23    b[ 0 ] /= 2; // error
24    b[ 1 ] /= 2; // error
25    b[ 2 ] /= 2; // error
26 } // end function tryToModifyArray
```

Array cannot be modified; it is **`const`** within the body function

Borland C++ command-line compiler error message:

```
Error E2024 fig07_15.cpp 23: Cannot modify a const object
   in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 24: Cannot modify a const object
   in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 25: Cannot modify a const object
   in function tryToModifyArray(const int * const)
```

Microsoft Visual C++.NET compiler error message:

```
C:\cpphtp5_examples\ch07\fig07_15.cpp(23) : error C2166: l-value specifies
   const object
C:\cpphtp5_examples\ch07\fig07_15.cpp(24) : error C2166: l-value specifies
   const object
C:\cpphtp5_examples\ch07\fig07_15.cpp(25) : error C2166: l-value specifies
   const object
```

GNU C++ compiler error message:

```
fig07_15.cpp:23: error: assignment of read-only location
fig07_15.cpp:24: error: assignment of read-only location
fig07_15.cpp:25: error: assignment of read-only location
```

Common Programming Error 7.11

Forgetting that arrays in the caller are passed by reference, and hence can be modified in called functions, may result in logic errors.

Software Engineering Observation 7.4

Applying the `const` type qualifier to an array parameter in a function definition to prevent the original array from being modified in the function body is another example of the principle of least privilege. Functions should not be given the capability to modify an array unless it is absolutely necessary.

7.6 Case Study: Class GradeBook Using an Array to Store Grades

- Class GradeBook
  - Represent a grade book that stores and analyzes grades
  - Can now store grades in an array
- static data members
  - Also called class variables
  - Variables for which each object of a class does not have a separate copy
    - One copy is shared among all objects of the class
  - Can be accessed even when no objects of the class exist
    - Use the class name followed by the binary scope resolution operator and the name of the static data member

Outline

fig07_16.cpp

(1 of 1)

```
1  // Fig. 7.16: GradeBook.h
2  // Definition of class GradeBook that uses an array to store test grades.
3  // Member functions are defined in GradeBook.cpp
4
5  #include <string> // program uses C++ Standard Library string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // constant -- number of students who took the test
13    const static int students = 10; // note public data
14
15    // constructor initializes course name and array of grades
16    GradeBook( string, const int [] );
17
18    void setCourseName( string ); // function to set the course name
19    string getCourseName(); // function to retrieve the course name
20    void displayMessage(); // display a welcome message
21    void processGrades(); // perform various operations on the grade data
22    int getMinimum(); // find the minimum grade for the test
23    int getMaximum(); // find the maximum grade for the test
24    double getAverage(); // determine the average grade for the test
25    void outputBarChart(); // output bar chart of grade distribution
26    void outputGrades(); // output the contents of the grades array
27 private:
28    string courseName; // course name for this grade book
29    int grades[ students ]; // array of student grades
30 }; // end class GradeBook
```

students is a **static** class variable

Number of students we will be keeping track of

Declare array **grades** to store individual grades

Outline

fig07_17.cpp

(1 of 6)

```
1  // Fig. 7.17: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses an array to store test grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12 using std::setw;
13
14 #include "GradeBook.h" // GradeBook class definition
15
16 // constructor initializes courseName and grades array
17 GradeBook::GradeBook( string name, const int gradesArray[] )
18 {
19    setCourseName( name ); // initialize courseName
20
21    // copy grades from gradeArray to grades data member
22    for ( int grade = 0; grade < students; grade++ )
23       grades[ grade ] = gradesArray[ grade ];
24 } // end GradeBook constructor
25
26 // function to set the course name
27 void GradeBook::setCourseName( string name )
28 {
29    courseName = name; // store the course name
30 } // end function setCourseName
```

Copy elements from **gradesArray** to data member **grades**

Outline

fig07_17.cpp

(2 of 6)

```
31
32 // function to retrieve the course name
33 string GradeBook::getCourseName()
34 {
35    return courseName;
36 } // end function getCourseName
37
38 // display a welcome message to the GradeBook user
39 void GradeBook::displayMessage()
40 {
41    // this statement calls getCourseName to get the
42    // name of the course this GradeBook represents
43    cout << "Welcome to the grade book for\n" << getCourseName() << "!"
44       << endl;
45 } // end function displayMessage
46
47 // perform various operations on the data
48 void GradeBook::processGrades()
49 {
50    // output grades array
51    outputGrades();
52
53    // call function getAverage to calculate the average grade
54    cout << "\nClass average is " << setprecision( 2 ) << fixed <<
55       getAverage() << endl;
56
57    // call functions getMinimum and getMaximum
58    cout << "Lowest grade is " << getMinimum() << "\nHighest grade is "
59       << getMaximum() << endl;
```

Outline

fig07_17.cpp

(3 of 6)

```
60
61    // call function outputBarChart to print grade distribution chart
62    outputBarChart();
63 } // end function processGrades
64
65 // find minimum grade
66 int GradeBook::getMinimum()
67 {
68    int lowGrade = 100; // assume lowest grade is 100
69
70    // loop through grades array
71    for ( int grade = 0; grade < students; grade++ )
72    {
73       // if current grade lower than lowGrade, assign it to lowGrade
74       if ( grades[ grade ] < lowGrade )
75          lowGrade = grades[ grade ]; // new lowest grade
76    } // end for
77
78    return lowGrade; // return lowest grade
79 } // end function getMinimum
80
81 // find maximum grade

82 int GradeBook::getMaximum()
```

Loop through **grades** to find the lowest grade

```
83 {
84    int highGrade = 0; // assume highest grade is 0
85
86    // loop through grades array
87    for ( int grade = 0; grade < students; grade++ )
88    {
89       // if current grade higher than highGrade, assign it to highGrade
90       if ( grades[ grade ] > highGrade )
91          highGrade = grades[ grade ]; // new highest grade
92    } // end for
93
94    return highGrade; // return highest grade
95 } // end function getMaximum
96
97 // determine average grade for test
98 double GradeBook::getAverage()
99 {
100   int total = 0; // initialize total
101
102   // sum grades in array
103   for ( int grade = 0; grade < students; grade++ )
104      total += grades[ grade ];
105
106   // return average of grades
107   return static_cast< double >( total ) / students;
108} // end function getAverage
109
110// output bar chart displaying grade distribution
111void GradeBook::outputBarChart()
```

Loop through **grades** to find the highest grade

fig07_17.cpp

(4 of 6)

Loop through **grades** to sum grades for all students

Divide the total by the number of students to calculate the average grade

Outline

fig07_17.cpp

(5 of 6)

```
112{
113   cout << "\nGrade distribution:" << endl;
114
115   // stores frequency of grades in each range of 10 grades
116   const int frequencySize = 11;
117   int frequency[ frequencySize ] = { 0 };
118
119   // for each grade, increment the appropriate frequency
120   for ( int grade = 0; grade < students; grade++ )
121      frequency[ grades[ grade ] / 10 ]++;
122
123   // for each grade frequency, print bar in chart
124   for ( int count = 0; count < frequencySize; count++ )
125   {
126      // output bar labels ("0-9:", ..., "90-99:", "100:" )
127      if ( count == 0 )
128         cout << "  0-9: ";
129      else if ( count == 10 )
130         cout << "  100: ";
131      else
132         cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
133
134      // print bar of asterisks
135      for ( int stars = 0; stars < frequency[ count ]; stars++ )
136         cout << '*';
137
138      cout << endl; // start a new line of output
```

Loop through **grades** to calculate frequency

Display asterisks to show a bar for each grade range

```
139   } // end outer for
140} // end function outputBarChart
141
142// output the contents of the grades array
143void GradeBook::outputGrades()
144{
145   cout << "\nThe grades are:\n\n";
146
147   // output each student's grade
148   for ( int student = 0; student < students; student++ )
149      cout << "Student " << setw( 2 ) << student + 1 << ": " << setw( 3 )
150         << grades[ student ] << endl;
151} // end function outputGrades
```

Displaying each grade

Outline

fig07_17.cpp

(6 of 6)

Outline

```
1  // Fig. 7.18: fig07_18.cpp
2  // Creates GradeBook object using an array of grades.
3
4  #include "GradeBook.h" // GradeBook class definition
5
6  // function main begins program execution
7  int main()
8  {
9     // array of student grades
10    int gradesArray[ GradeBook::students ] =
11       { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
12
13    GradeBook myGradeBook(
14       "CS101 Introduction to C++ Programming", gradesArray );
15    myGradeBook.displayMessage();
16    myGradeBook.processGrades();
17    return 0;
18 } // end main
```

Use **static** data member **students** of class **GradeBook**

Declare and initialize **gradesArray** with 10 elements

Pass **gradesArray** to **GradeBook** constructor

fig07_18.cpp

(1 of 2)

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Outline

fig07_18.cpp

(2 of 2)

7.7 Searching Arrays with Linear Search

- **Arrays may store large amounts of data**
  - **May need to determine if certain key value is located in an array**
- **Linear search**
  - **Compares each element of an array with a search key**
  - **Just as likely that the value will be found in the first element as the last**
    - **On average, program must compare the search key with half the elements of the array**
  - **To determine that value is not in array, program must compare the search key to every element in the array**
  - **Works well for small or unsorted arrays**

Outline

```
1  // Fig. 7.19: fig07_19.cpp
2  // Linear search of an array.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int linearSearch( const int [], int, int ); // prototype
9
10 int main()
11 {
12    const int arraySize = 100; // size of array a
13    int a[ arraySize ]; // create array a
14    int searchKey; // value to locate in array a
15
16    for ( int i = 0; i < arraySize; i++ )
17       a[ i ] = 2 * i; // create some data
18
19    cout << "Enter integer search key: ";
20    cin >> searchKey;
21
22    // attempt to locate searchKey in array a
23    int element = linearSearch( a, searchKey, arraySize );
24
```

Function takes an array, a key value, and the size of the array as arguments

Function returns location of key value, **-1** if not found

fig07_19.cpp

(1 of 2)

Outline

fig07_19.cpp

(2 of 2)

```
25    // display results
26    if ( element != -1 )
27       cout << "Found value in element " << element << endl;
28    else
29       cout << "Value not found" << endl;
30
31    return 0; // indicates successful termination
32 } // end main
33
34 // compare key to every element of array until location is
35 // found or until end of array is reached; return subscript of
36 // element if key or -1 if key not found
37 int linearSearch( const int array[], int key, int sizeOfArray )
38 {
39    for ( int j = 0; j < sizeOfArray; j++ )
40       if ( array[ j ] == key ) // if found,
41          return j; // return location of key
42
43    return -1; // key not found
44 } // end function linearSearch
```

Search through entire array

Return location if current value equals key value

```
Enter integer search key: 36
Found value in element 18
```

```
Enter integer search key: 37
Value not found
```

7.8 Sorting Arrays with Insertion Sort

- **Sorting data**
  - **One of the most important computing applications**
    - **Virtually every organization must sort some data**
- **Insertion sort**
  - **Simple but inefficient**
  - **First iteration takes second element**
    - **If it is less than the first element, swap it with first element**
  - **Second iteration looks at the third element**
    - **Insert it into the correct position with respect to first two elements**
  - **…**
  - **At the i^{th} iteration of this algorithm, the first i elements in the original array will be sorted**

Performance Tip 7.4

Sometimes, simple algorithms perform poorly. Their virtue is that they are easy to write, test and debug. More complex algorithms are sometimes needed to realize optimal performance.

Outline

fig07_20.cpp

(1 of 2)

```
1  // Fig. 7.20: fig07_20.cpp
2  // This program sorts an array's values into ascending order.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    const int arraySize = 10; // size of array a
13    int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
14    int insert; // temporary variable to hold element to insert
15
16    cout << "Unsorted array:\n";
17
18    // output original array
19    for ( int i = 0; i < arraySize; i++ )
20       cout << setw( 4 ) << data[ i ];
21
22    // insertion sort
23    // loop over the elements of the array
24    for ( int next = 1; next < arraySize; next++ )
25    {
26       insert = data[ next ]; // store the value in the current element
27
28       int moveItem = next; // initialize location to place element
```

For each array element

Outline

```
29
30       // search for the location in which to put the current element
31       while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
32       {
33          // shift element one slot to the right
34          data[ moveItem ] = data[ moveItem - 1 ];
35          moveItem--;
36       } // end while
37
38       data[ moveItem ] = insert; // place inserted element into the array
39    } // end for
40
41    cout << "\nSorted array:\n";
42
43    // output sorted array
44    for ( int i = 0; i < arraySize; i++ )
45       cout << setw( 4 ) << data[ i ];
46
47    cout << endl;
48    return 0; // indicates successful termination
49 } // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Find location where current element should reside

Place element in proper location

(2 of 2)

7.9 Multidimensional Array

- **Multidimensional arrays with two dimensions**
  - **Called two dimensional or 2-D arrays**
  - **Represent tables of values with rows and columns**
  - **Elements referenced with two subscripts (`[ x ][ y ]`)**
  - **In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array**
- **Multidimensional arrays can have more than two dimensions**

Common Programming Error 7.12

Referencing a two-dimensional array element `a[ x ][ y ]` incorrectly as `a[ x, y ]` is an error. Actually, `a[ x, y ]` is treated as `a[ y ]`, because C++ evaluates the expression `x, y` (containing a comma operator) simply as `y` (the last of the comma-separated expressions).

7.9 Multidimensional Array (Cont.)

- **Declaring and initializing two-dimensional arrays**
  - **Declaring two-dimensional array `b`**
    - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
      - **`1` and `2` initialize `b[ 0 ][ 0 ]` and `b[ 0 ][ 1 ]`**
      - **`3` and `4` initialize `b[ 1 ][ 0 ]` and `b[ 1 ][ 1 ]`**
    - `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`
      - **Row `0` contains values `1` and `0` (implicitly initialized to zero)**
      - **Row `1` contains values `3` and `4`**

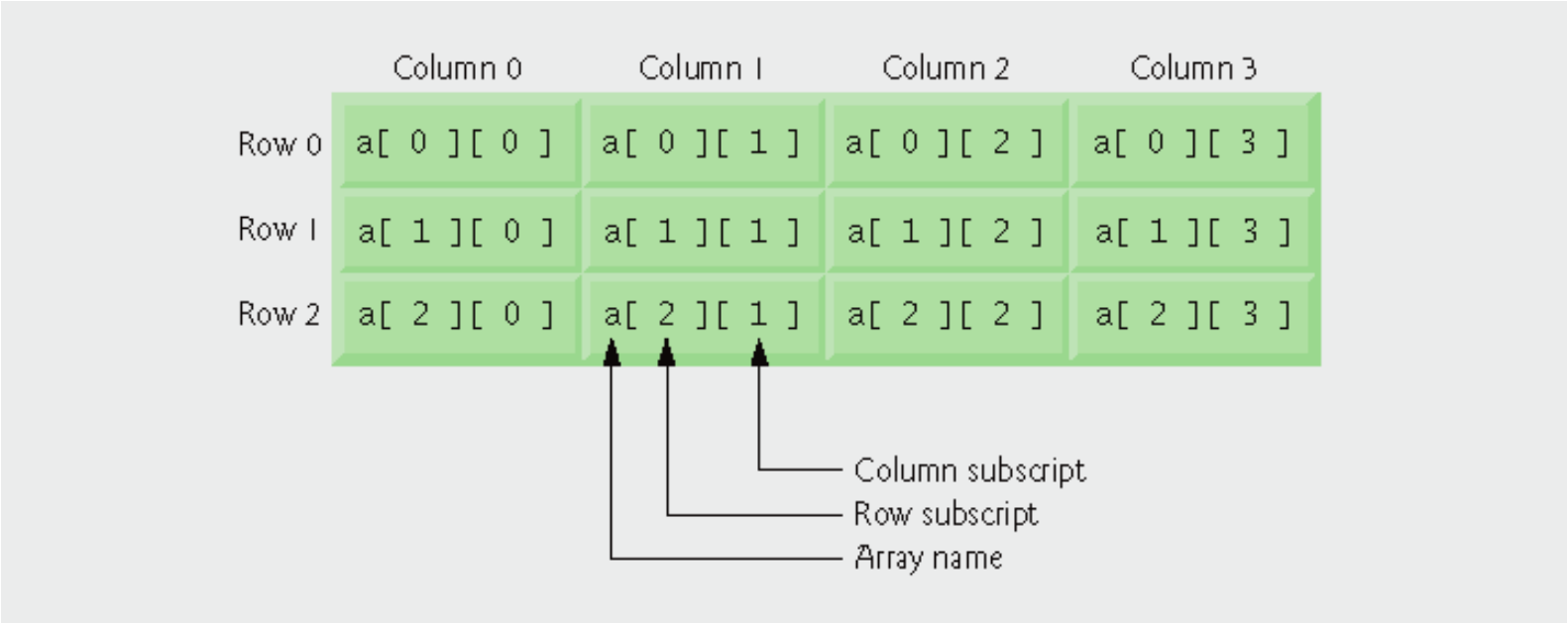

Fig.7.21 | Two-dimensional array with three rows and four columns.

Outline

fig07_22.cpp

(1 of 2)

```
1  // Fig. 7.22: fig07_22.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void printArray( const int [][ 3 ] ); // prototype
8
9  int main()
10 {
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    cout << "Values in array1 by row are:" << endl;
16    printArray( array1 );
17
18    cout << "\nValues in array2 by row are:" << endl;
19    printArray( array2 );
20
21    cout << "\nValues in array3 by row are:" << endl;
22    printArray( array3 );
23    return 0; // indicates successful termination
24 } // end main
```

Use nested array initializers to initialize arrays

Outline

fig07_22.cpp

```
25
26 // output array with two rows and three columns
27 void printArray( const int a[][ 3 ] )
28 {
29    // loop through array's rows
30    for ( int i = 0; i < 2; i++ )
31    {
32       // loop through columns of current row
33       for ( int j = 0; j < 3; j++ )
34          cout << a[ i ][ j ] << ' ';
35
36       cout << endl; // start new line of output
37    } // end outer for
38 } // end function printArray
```

Use nested **for** loops to print array

```
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
```

7.9 Multidimensional Array (Cont.)

- **Multidimensional array parameters**
  - **Size of first dimension is not required**
    - **As with a one-dimensional array**
  - **Size of subsequent dimensions are required**
    - **Compiler must know how many elements to skip to move to the second element in the first dimension**
  - **Example**
    - `void printArray( const int a[][ 3 ] );`
      - **Function will skip row `0`'s `3` elements to access row `1`'s elements (`a[ 1 ][ x ]`)**

7.9 Multidimensional Array (Cont.)

- **Multidimensional-array manipulations**
  - **Commonly performed with `for` statements**
    - **Example**
      - **Modify all elements in a row**
        - ```
          for ( int col = 0; col < 4; col++ )
             a[ 2 ][ col ] = 0;
          ```
    - **Example**
      - **Total all elements**
        - ```
          total = 0;
          for ( row = 0; row < 3; row++ )
             for ( col = 0; col < 4; col++ )
                total += a[ row ][ col ];
          ```

7.10 Case Study: Class GradeBook Using a Two-Dimensional Array

- Class GradeBook
  - One-dimensional array
    - Store student grades on a single exam
  - Two-dimensional array
    - Store multiple grades for a single student and multiple students for the class as a whole
      - Each row represents a student’s grades
      - Each column represents all the grades the students earned for one particular exam

Outline

fig07_23.cpp

(1 of 2)

```
1  // Fig. 7.23: GradeBook.h
2  // Definition of class GradeBook that uses a
3  // two-dimensional array to store test grades.
4  // Member functions are defined in GradeBook.cpp
5  #include <string> // program uses C++ Standard Library string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // constants
13    const static int students = 10; // number of students
14    const static int tests = 3; // number of tests
15
16    // constructor initializes course name and array of grades
17    GradeBook( string, const int [][ tests ] );
```

GradeBook constructor accepts a **string** and a two-dimensional array

```
18
19    void setCourseName( string ); // function to set the course name
20    string getCourseName(); // function to retrieve the course name
21    void displayMessage(); // display a welcome message
22    void processGrades(); // perform various operations on the grade data
23    int getMinimum(); // find the minimum grade in the grade book
24    int getMaximum(); // find the maximum grade in the grade book
25    double getAverage( const int [], const int ); // find average of grades
26    void outputBarChart(); // output bar chart of grade distribution
27    void outputGrades(); // output the contents of the grades array
28 private:
29    string courseName; // course name for this grade book
30    int grades[ students ][ tests ]; // two-dimensional array of grades
31 }; // end class GradeBook
```

Declare two-dimensional array **grades**

Outline

fig07_23.cpp

(2 of 2)

Outline

fig07_24.cpp

(1 of 7)

```
1  // Fig. 7.24: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses a two-dimensional array to store grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip> // parameterized stream manipulators
11 using std::setprecision; // sets numeric output precision
12 using std::setw; // sets field width
13
14 // include definition of class GradeBook from GradeBook.h
15 #include "GradeBook.h"
16
17 // two-argument constructor initializes courseName and grades array
18 GradeBook::GradeBook( string name, const int gradesArray[][ tests ] )
19 {
20    setCourseName( name ); // initialize courseName
21
22    // copy grades from gradeArray to grades
23    for ( int student = 0; student < students; student++ )
24
25       for ( int test = 0; test < tests; test++ )
26          grades[ student ][ test ] = gradesArray[ student ][ test ];
27 } // end two-argument GradeBook constructor
28
```

Use nested **for** loops to copy elements from **gradesArray** to **grades**

Outline

fig07_24.cpp

(2 of 7)

```
29 // function to set the course name
30 void GradeBook::setCourseName( string name )
31 {
32    courseName = name; // store the course name
33 } // end function setCourseName
34
35 // function to retrieve the course name
36 string GradeBook::getCourseName()
37 {
38    return courseName;
39 } // end function getCourseName
40
41 // display a welcome message to the GradeBook user
42 void GradeBook::displayMessage()
43 {
44    // this statement calls getCourseName to get the
45    // name of the course this GradeBook represents
46    cout << "Welcome to the grade book for\n" << getCourseName() << "!"
47       << endl;
48 } // end function displayMessage
49
50 // perform various operations on the data
51 void GradeBook::processGrades()
52 {
53    // output grades array
54    outputGrades();
55
56    // call functions getMinimum and getMaximum
57    cout << "\nLowest grade in the grade book is " << getMinimum()
58       << "\nHighest grade in the grade book is " << getMaximum() << endl;
```

Outline

fig07_24.cpp

(3 of 7)

```
59
60    // output grade distribution chart of all grades on all tests
61    outputBarChart();
62 } // end function processGrades
63
64 // find minimum grade
65 int GradeBook::getMinimum()
66 {
67    int lowGrade = 100; // assume lowest grade is 100
68
69    // loop through rows of grades array
70    for ( int student = 0; student < students; student++ )
71    {
72       // loop through columns of current row
73       for ( int test = 0; test < tests; test++ )
74       {
75          // if current grade less than lowGrade, assign it to lowGrade
76          if ( grades[ student ][ test ] < lowGrade )
77             lowGrade = grades[ student ][ test ]; // new lowest grade
78       } // end inner for
79    } // end outer for
80
81    return lowGrade; // return lowest grade
82 } // end function getMinimum
83
```

Loop through rows and columns of **grades** to find the lowest grade of any student

Outline

fig07_24.cpp

(4 of 7)

```
84 // find maximum grade
85 int GradeBook::getMaximum()
86 {
87    int highGrade = 0; // assume highest grade is 0
88
89    // loop through rows of grades array
90    for ( int student = 0; student < students; student++ )
91    {
92       // loop through columns of current row
93       for ( int test = 0; test < tests; test++ )
94       {
95          // if current grade greater than lowGrade, assign it to highGrade
96          if ( grades[ student ][ test ] > highGrade )
97             highGrade = grades[ student ][ test ]; // new highest grade
98       } // end inner for
99    } // end outer for
100
101   return highGrade; // return highest grade
102} // end function getMaximum
103
104// determine average grade for particular set of grades
105double GradeBook::getAverage( const int setOfGrades[], const int grades )
106{
107   int total = 0; // initialize total
108
109   // sum grades in array
110   for ( int grade = 0; grade < grades; grade++ )
111      total += setOfGrades[ grade ];
112
```

Loop through rows and columns of **grades** to find the highest grade of any student

Outline

fig07_24.cpp

(5 of 7)

```
113   // return average of grades
114   return static_cast< double >( total ) / grades;
115} // end function getAverage
116
117// output bar chart displaying grade distribution
118void GradeBook::outputBarChart()
119{
120   cout << "\nOverall grade distribution:" << endl;
121
122   // stores frequency of grades in each range of 10 grades
123   const int frequencySize = 11;
124   int frequency[ frequencySize ] = { 0 };
125
126   // for each grade, increment the appropriate frequency
127   for ( int student = 0; student < students; student++ )
128
129      for ( int test = 0; test < tests; test++ )
130         ++frequency[ grades[ student ][ test ] / 10 ];
131
132   // for each grade frequency, print bar in chart
133   for ( int count = 0; count < frequencySize; count++ )
134   {
135      // output bar label ("0-9:", ..., "90-99:", "100:" )
136      if ( count == 0 )
137         cout << "  0-9: ";
138      else if ( count == 10 )
139         cout << "  100: ";
140      else
141         cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
142
```

Calculate the distribution of all student grades

Outline

fig07_24.cpp

(6 of 7)

```
143      // print bar of asterisks
144      for ( int stars = 0; stars < frequency[ count ]; stars++ )
145         cout << '*';
146
147      cout << endl; // start a new line of output
148   } // end outer for
149} // end function outputBarChart
150
151 // output the contents of the grades array
152 void GradeBook::outputGrades()
153 {
154    cout << "\nThe grades are:\n\n";
155    cout << "            "; // align column heads
156
157    // create a column heading for each of the tests
158    for ( int test = 0; test < tests; test++ )
159       cout << "Test " << test + 1 << "  ";
160
161    cout << "Average" << endl; // student average column heading
162
163    // create rows/columns of text representing array grades
164    for ( int student = 0; student < students; student++ )
165    {
166       cout << "Student " << setw( 2 ) << student + 1;
167
```

```
168      // output student's grades
169      for ( int test = 0; test < tests; test++ )
170         cout << setw( 8 ) << grades[ student ][ test ];
171
172      // call member function getAverage to calculate student's average;
173      // pass row of grades and the value of tests as the arguments
174      double average = getAverage( grades[ student ], tests );
175      cout << setw( 9 ) << setprecision( 2 ) << fixed << average << endl;
176   } // end outer for
177} // end function outputGrades
```

Outline

fig07_24.cpp

(7 of 7)

Outline

```
1  // Fig. 7.25: fig07_25.cpp
2  // Creates GradeBook object using a two-dimensional array of grades.
3
4  #include "GradeBook.h" // GradeBook class definition
5
6  // function main begins program execution
7  int main()
8  {
9     // two-dimensional array of student grades
10    int gradesArray[ GradeBook::students ][ GradeBook::tests ] =
11       { { 87, 96, 70 },
12         { 68, 87, 90 },
13         { 94, 100, 90 },
14         { 100, 81, 82 },
15         { 83, 65, 85 },
16         { 78, 87, 65 },
17         { 85, 75, 83 },
18         { 91, 94, 100 },
19         { 76, 72, 84 },
20         { 87, 93, 73 } };
21
22    GradeBook myGradeBook(
23       "CS101 Introduction to C++ Programming", gradesArray );
24    myGradeBook.displayMessage();
25    myGradeBook.processGrades();
26    return 0; // indicates successful termination
27 } // end main
```

Declare **gradesArray** as 3-by-10 array

Each row represents a student; each column represents an exam grade

fig07_25.cpp

(1 of 2)

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Outline

fig07_25.cpp

(2 of 2)

7.11 Introduction to C++ Standard Library Class Template vector

- **C-style pointer-based arrays**
  - **Have great potential for errors and several shortcomings**
    - **C++ does not check whether subscripts fall outside the range of the array**
    - **Two arrays cannot be meaningfully compared with equality or relational operators**
    - **One array cannot be assigned to another using the assignment operators**

7.11 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- Class template `vector`
  - Available to anyone building applications with C++
  - Can be defined to store any data type
    - Specified between angle brackets in `vector<` *type* `>`
    - All elements in a `vector` are set to `0` by default
  - Member function `size` obtains size of array
    - Number of elements as a value of type `size_t`
  - `vector` objects can be compared using equality and relational operators
  - Assignment operator can be used for assigning `vector`s

7.11 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- `vector` elements can be obtained as an unmodifiable *lvalue* or as a modifiable *lvalue*
  - Unmodifiable *lvalue*
    - Expression that identifies an object in memory, but cannot be used to modify that object
  - Modifiable *lvalue*
    - Expression that identifies an object in memory, can be used to modify the object

7.11 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- `vector` member function `at`
  - Provides access to individual elements
  - Performs bounds checking
    - Throws an exception when specified index is invalid
    - Accessing with square brackets does not perform bounds checking

Outline

```
1  // Fig. 7.26: fig07_26.cpp
2  // Demonstrating C++ Standard Library class template vector.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 #include <vector>
12 using std::vector;
13
14 void outputVector( const vector< int > & ); // display the vector
15 void inputVector( vector< int > & ); // input values into the vector
16
17 int main()
18 {
19    vector< int > integers1( 7 ); // 7-element vector< int >
20    vector< int > integers2( 10 ); // 10-element vector< int >
21
22    // print integers1 size and contents
23    cout << "Size of vector integers1 is " << integers1.size()
24       << "\nvector after initialization:" << endl;
25    outputVector( integers1 );
26
27    // print integers2 size and contents
28    cout << "\nSize of vector integers2 is " << integers2.size()
29       << "\nvector after initialization:" << endl;
30    outputVector( integers2 );
```

g07_26.cpp

of 6)

Using **const** prevents **outputVector** from modifying the **vector** passed to it

These **vectors** will store **ints**

Function **size** returns number of elements in the **vector**

Outline

fig07_26.cpp

(2 of 6)

```
31
32    // input and print integers1 and integers2
33    cout << "\nEnter 17 integers:" << endl;
34    inputVector( integers1 );
35    inputVector( integers2 );
36
37    cout << "\nAfter input, the vectors contain:\n"
38       << "integers1:" << endl;
39    outputVector( integers1 );
40    cout << "integers2:" << endl;
41    outputVector( integers2 );
42
43    // use inequality (!=) operator with vector objects
44    cout << "\nEvaluating: integers1 != integers2" << endl;
45
46    if ( integers1 != integers2 )
47       cout << "integers1 and integers2 are not equal" << endl;
48
49    // create vector integers3 using integers1 as an
50    // initializer; print size and contents
51    vector< int > integers3( integers1 ); // copy constructor
52
53    cout << "\nSize of vector integers3 is " << integers3.size()
54       << "\nvector after initialization:" << endl;
55    outputVector( integers3 );
56
57    // use overloaded assignment (=) operator
58    cout << "\nAssigning integers2 to integers1:" << endl;
59    integers1 = integers2; // integers1 is larger than integers2
```

Comparing **vector**s using **!=**

Copying data from one **vector** to another

Assigning data from one **vector** to another

Outline

fig07_26.cpp

(3 of 6)

```
60
61    cout << "integers1:" << endl;
62    outputVector( integers1 );
63    cout << "integers2:" << endl;
64    outputVector( integers2 );
65
66    // use equality (==) operator with vector objects
67    cout << "\nEvaluating: integers1 == integers2" << endl;
68
69    if ( integers1 == integers2 )
70       cout << "integers1 and integers2 are equal" << endl;
71
72    // use square brackets to create rvalue
73    cout << "\nintegers1[5] is " << integers1[ 5 ];
74
75    // use square brackets to create lvalue
76    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
77    integers1[ 5 ] = 1000;
78    cout << "integers1:" << endl;
79    outputVector( integers1 );
80
81    // attempt to use out-of-range subscript
82    cout << "\nAttempt to assign 1000 to integers1.at( 15 )" << endl;
83    integers1.at( 15 ) = 1000; // ERROR: out of range
84    return 0;
85 } // end main
```

Comparing **vector**s using **==**

Displaying a value in the **vector**

Updating a value in the **vector**

Function **at** provides bounds checking

Outline

fig07_26.cpp

(4 of 6)

```
86
87 // output vector contents
88 void outputVector( const vector< int > &array )
89 {
90    size_t i; // declare control variable
91
92    for ( i = 0; i < array.size(); i++ )
93    {
94       cout << setw( 12 ) << array[ i ];
95
96       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
97          cout << endl;
98    } // end for
99
100   if ( i % 4 != 0 )
101      cout << endl;
102} // end function outputVector
103
104// input vector contents
105void inputVector( vector< int > &array )
106{
107   for ( size_t i = 0; i < array.size(); i++ )
108      cin >> array[ i ];
109} // end function inputVector
```

Display each **vector** element

Input **vector** values using **cin**

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
           1           2           3           4
           5           6           7
```

(continued at top of next slide)

Outline

fig07_26.cpp

(5 of 6)

(continued from bottom of previous slide)

```
Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1.at( 15 )

abnormal program termination
```

Call to function **`at`** with an invalid subscript terminates the program

Outline

`fig07_26.cpp`

(6 of 6)

7.12 (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System

- **Collaborations**
  - **When objects communicate to accomplish task**
    - **One object sends a message to another object**
    - **Accomplished by invoking operations (functions)**
  - **Identifying the collaborations in a system**
    - **Read requirements document to find**
      - **What ATM should do to authenticate a user**
      - **What ATM should do to perform transactions**
    - **For each action, decide**
      - **Which objects must interact**
        - **Sending object**
        - **Receiving object**

An object of class…	sends the message…	to an object of class…
ATM	displayMessage	Screen
	getInput	Keypad
	authenticateUser	BankDatabase
	execute	BalanceInquiry
	execute	Withdrawal
	execute	Deposit
BalanceInquiry	getAvailableBalance	BankDatabase
	getTotalBalance	BankDatabase
	displayMessage	Screen
Withdrawal	displayMessage	Screen
	getInput	Keypad
	getAvailableBalance	BankDatabase
	isSufficientCashAvailable	CashDispenser
	debit	BankDatabase
	dispenseCash	CashDispenser
Deposit	displayMessage	Screen
	getInput	Keypad
	isEnvelopeReceived	DepositSlot
	credit	BankDatabase
BankDatabase	validatePIN	Account
	getAvailableBalance	Account
	getTotalBalance	Account
	debit	Account
	credit	Account

Fig.7.27 | Collaborations in the ATM system.

7.12 (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- **Interaction Diagrams**
  - **Model interactions using UML**
  - **Communication diagrams**
    - **Also called collaboration diagrams**
    - **Emphasize which objects participate in collaborations**
  - **Sequence diagrams**
    - **Emphasize when messages are sent between objects**

7.12 (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- **Communication diagrams**
  - **Objects**
    - **Modeled as rectangles**
    - **Contain names in the form `objectName : className`**
  - **Objects are connected with solid lines**

7.12 (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- **Communication diagrams (Cont.)**
  - **Messages are passed along these lines in the direction shown by arrows**
    - **Synchronous calls – solid arrowhead**
      - **Sending object may not proceed until control is returned from the receiving object**
    - **Asynchronous calls – stick arrowhead**
      - **Sending object does not have to wait for the receiving object**
    - **Name of message appears next to the arrow**

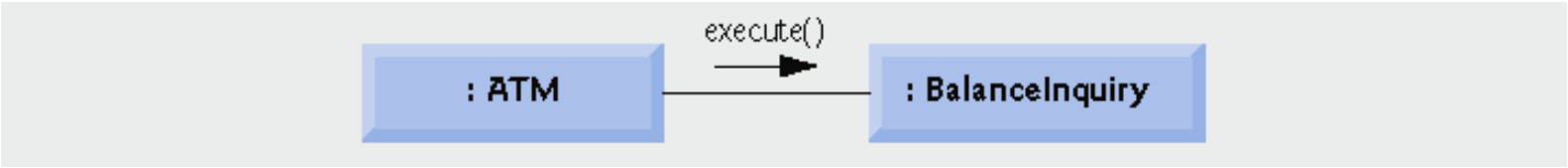

Fig.7.28 | Communication diagram of the ATM executing a balance inquiry.

7.12 (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- **Communication diagrams (Cont.)**
  - **Sequence of messages in a communication diagram**
    - **Indicated by the number to the left of a message name**
    - **Indicate the order in which the messages are passed**
    - **Process in numerical order from least to greatest**
    - **Nested messages are indicated by decimal numbering**
      - **Example**
        - **First message nested in message `1` is message `1.1`**

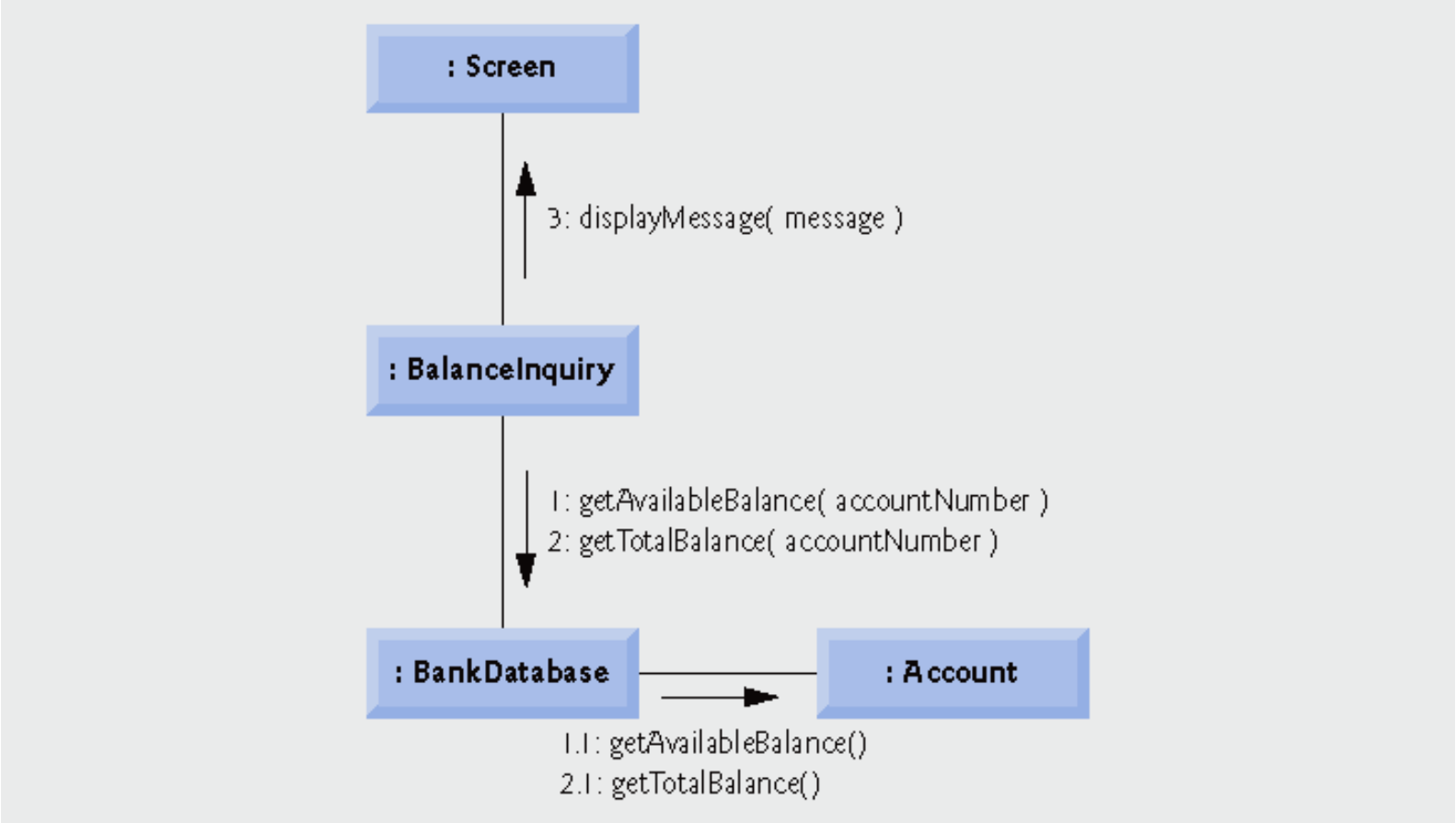

Fig.7.29 | Communication diagram of the ATM executing a balance inquiry.

7.12 (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- **Sequence diagrams**
  - **Help model the timing of collaborations**
  - **Lifeline**
    - **Dotted line extending down from an object's rectangle**
      - **Represents the progression of time (top to bottom)**
  - **Activation**
    - **Thin vertical rectangle on an object's lifeline**
      - **Indicates that the object is executing**

7.12 (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System (Cont.)

- **Sequence diagrams (Cont.)**
  - **Sending messages**
    - **Similar to communication diagrams**
    - **Solid arrow with filled arrowhead indicates a message**
      - **Points to the beginning of an activation**
    - **Dashed line with stick arrowhead indicates return of control**
      - **Extends from the end of an activation**

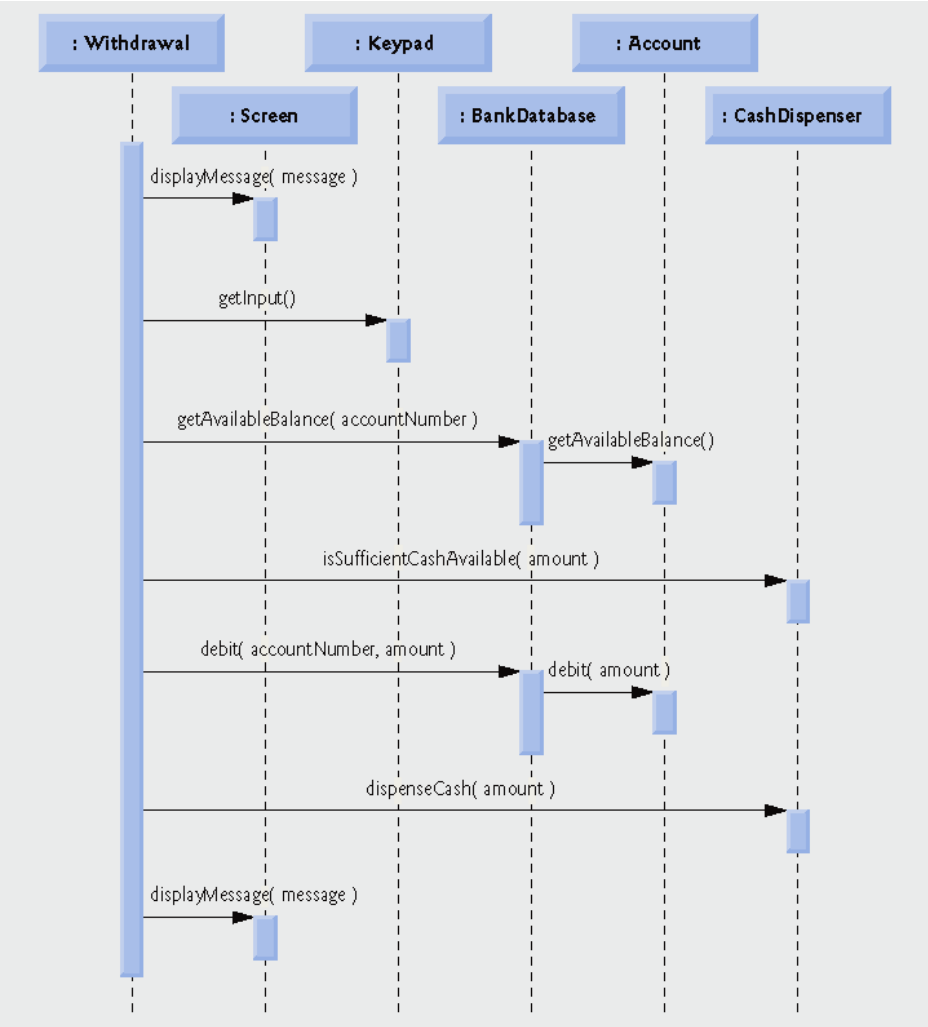

Fig.7.30 | Sequence diagram that models a `Withdrawal` executing.

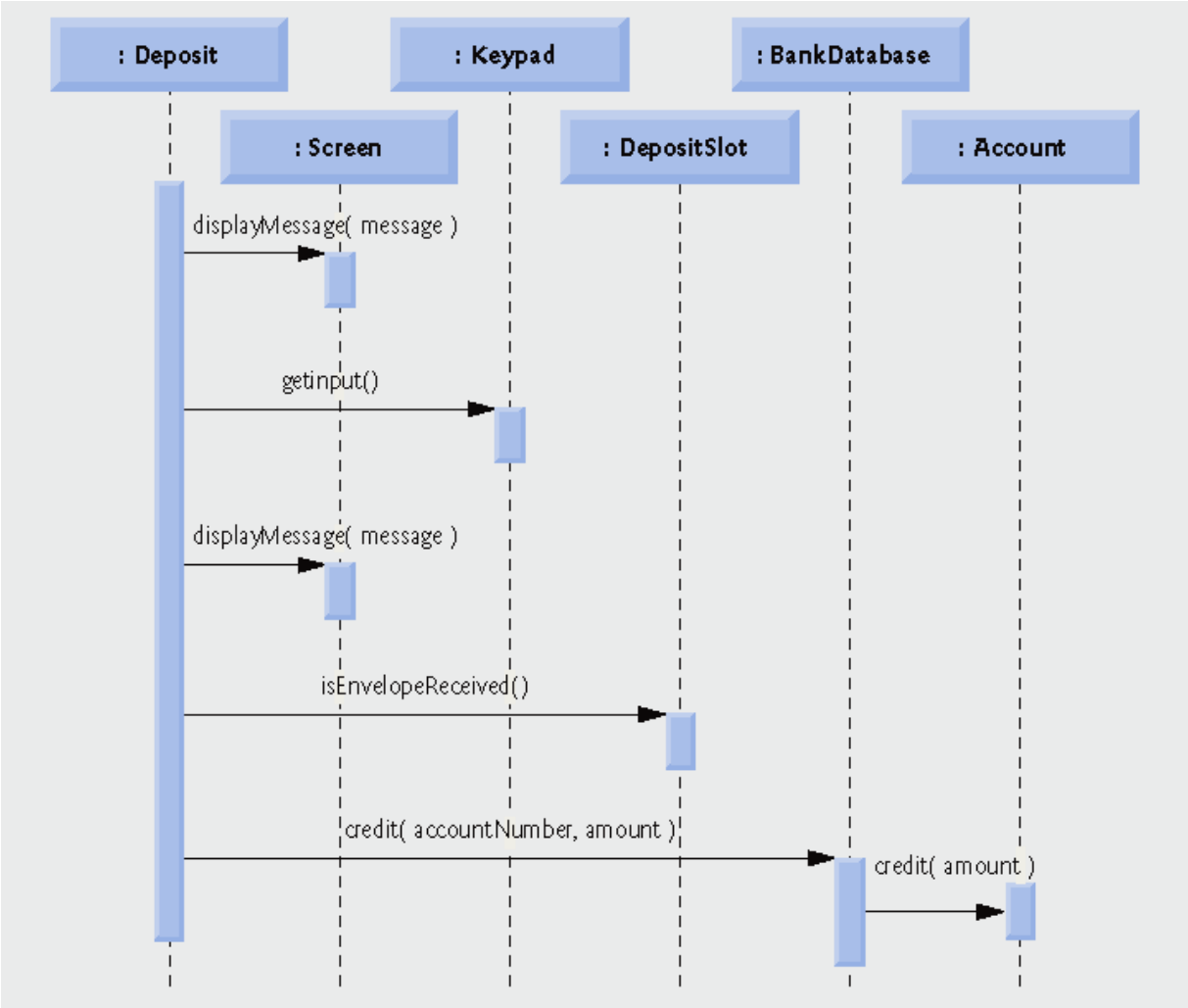

Fig.7.31 | Sequence diagram that models a `Deposit` executing.