C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 9: Arrays and Strings

Objectives

In this chapter, you will:

- Learn about arrays
- Explore how to declare and manipulate data into arrays
- Understand the meaning of “array index out of bounds”
- Become familiar with the restrictions on array processing
- Discover how to pass an array as a parameter to a function

Objectives (continued)

- Learn about `C`-strings
- Examine the use of string functions to process `C`-strings
- Discover how to input data into—and output data from—a `C`-string
- Learn about parallel arrays
- Discover how to manipulate data in a two-dimensional array
- Learn about multidimensional arrays

Data Types

- A data type is called simple if variables of that type can store only one value at a time
- A structured data type is one in which each data item is a collection of other data items

Arrays

- Array: a collection of a fixed number of components wherein all of the components have the same data type
- In a one-dimensional array, the components are arranged in a list form
- Syntax for declaring a one-dimensional array:

```
dataType arrayName[intExp];
```

`intExp` evaluates to a positive integer

Arrays (continued)

- Example:

```
int num[5];
```

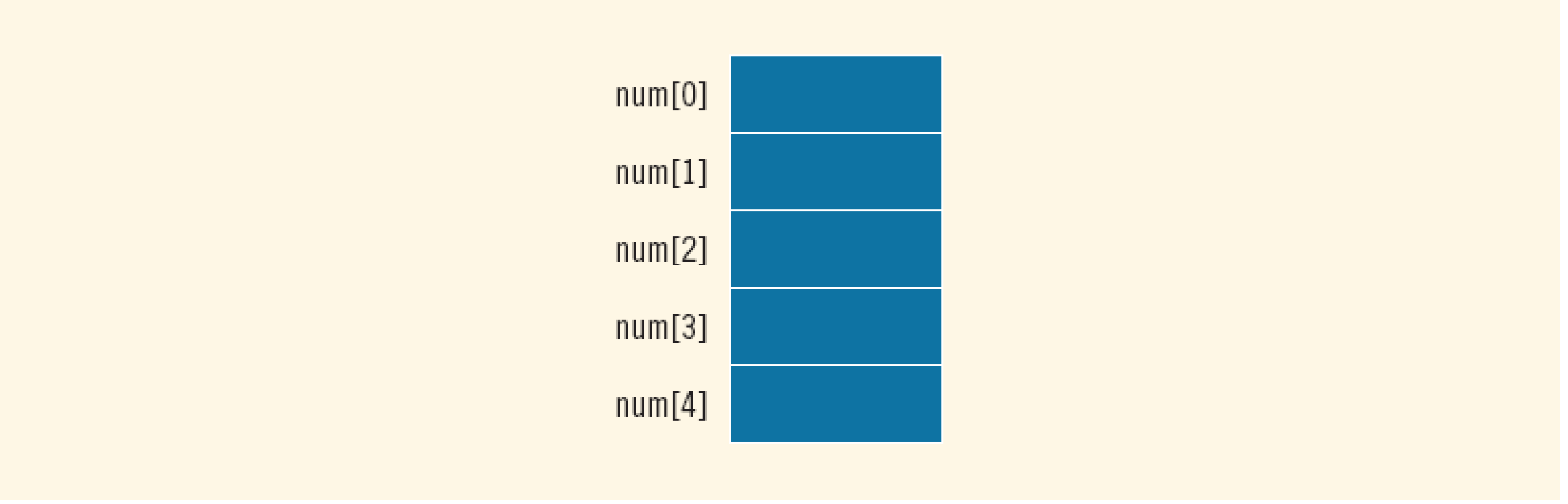

FIGURE 9-1 Array `num`

Accessing Array Components

- General syntax:

```
arrayName[indexExp]
```

  where `indexExp`, called an **index**, is any expression whose value is a nonnegative integer
- Index value specifies the position of the component in the array
- `[]` is the **array subscripting operator**
- The array index always starts at `0`

Accessing Array Components (continued)

```
int list[10];
```

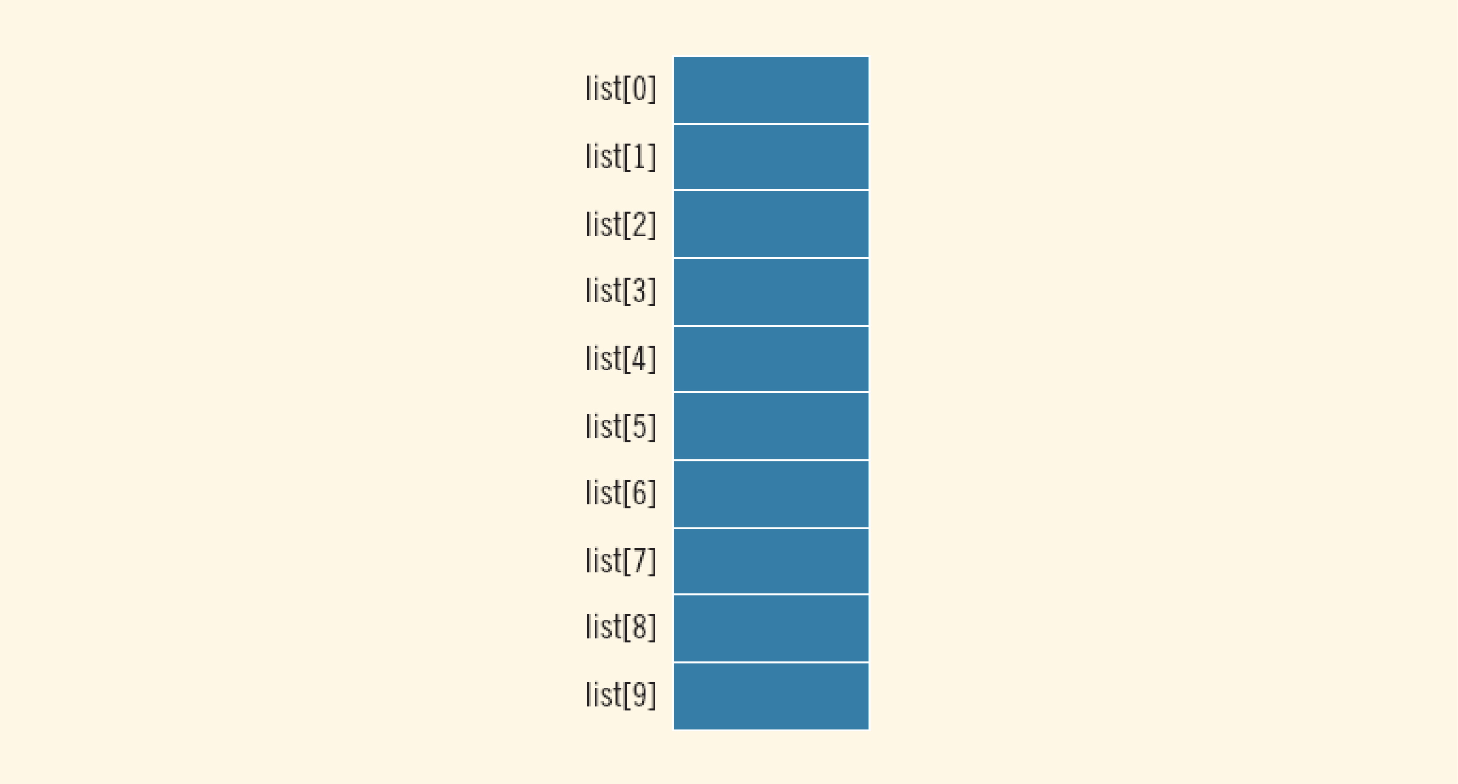

FIGURE 9-2 Array `list`

Accessing Array Components (continued)

```
list[5] = 34;
```

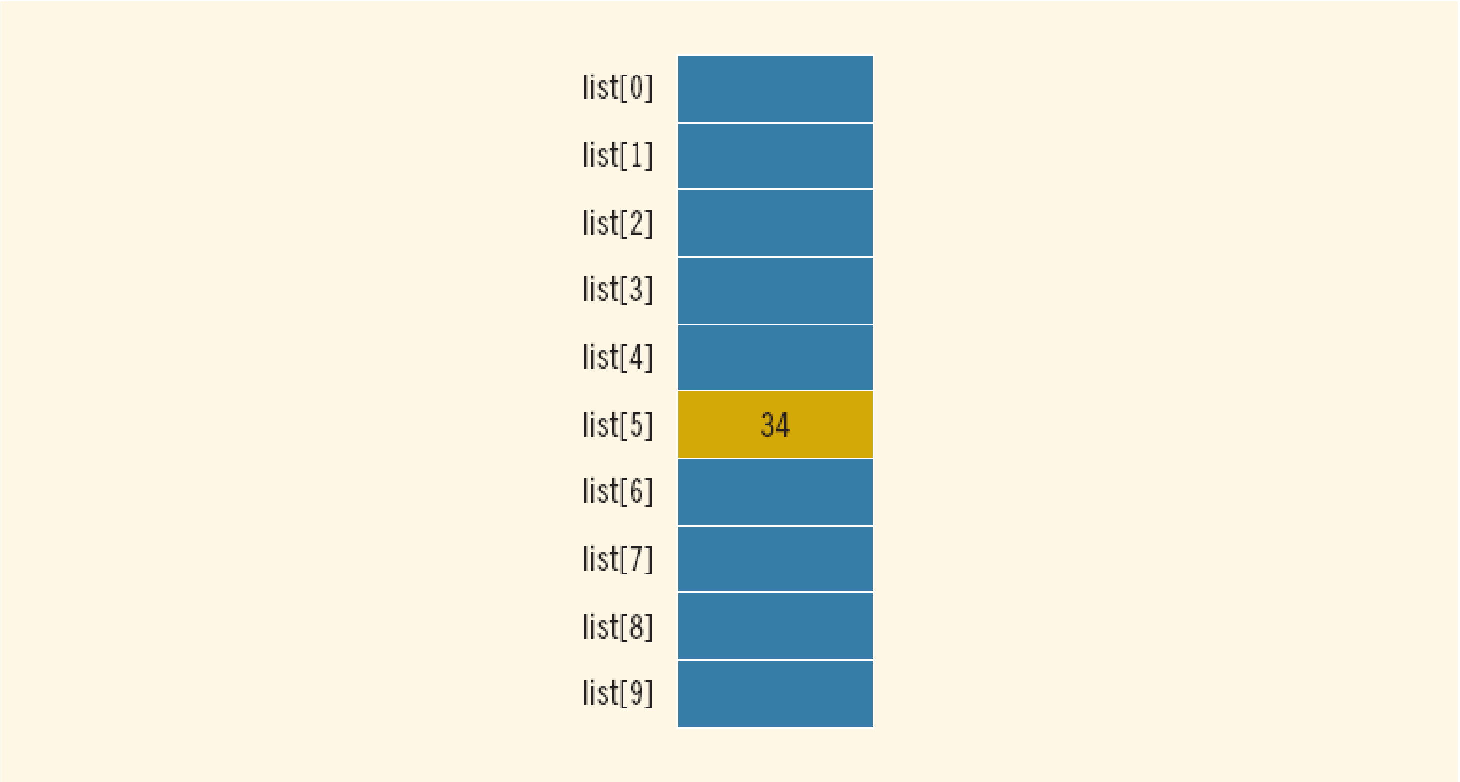

FIGURE 9-3 Array `list` after execution of the statement `list[5]= 34;`

Accessing Array Components (continued)

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

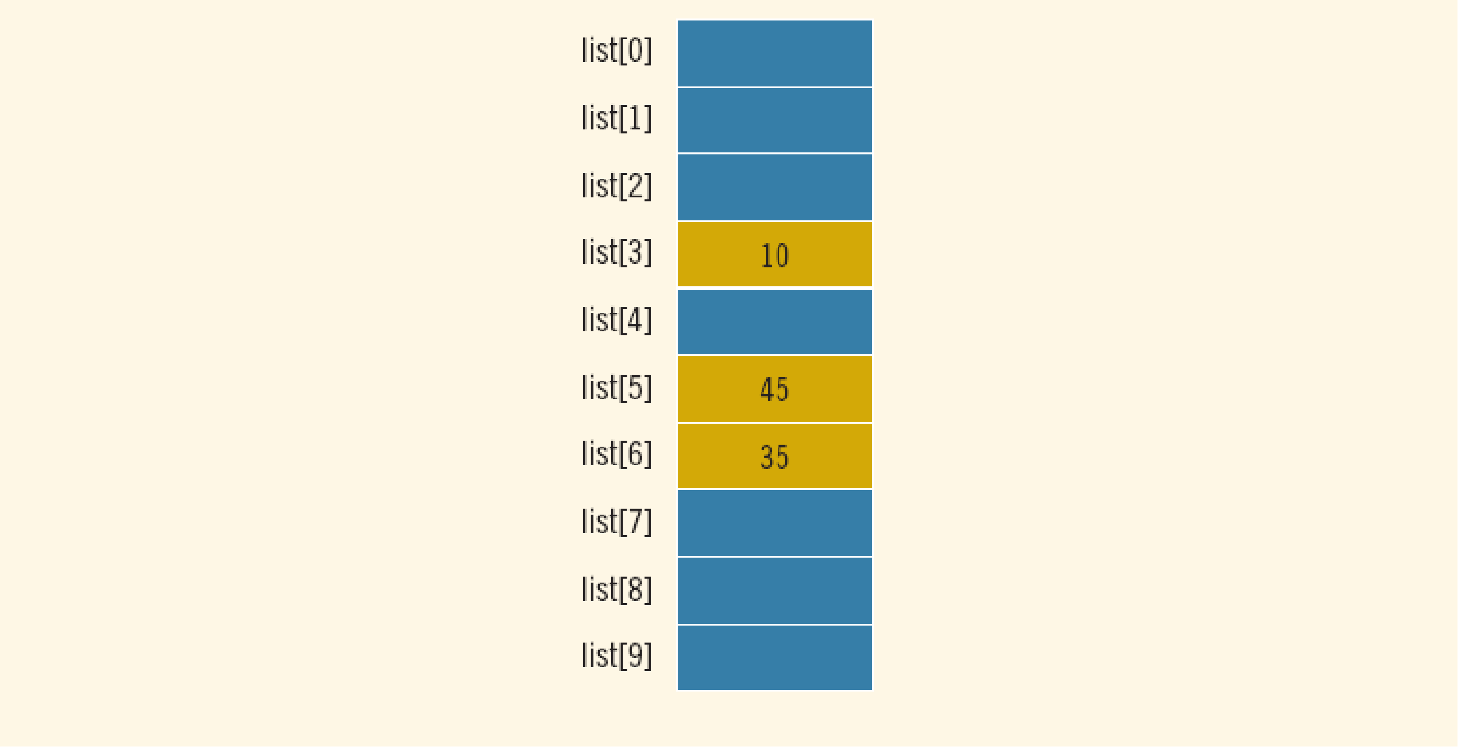

FIGURE 9-4 Array `list` after execution of the statements `list[3]= 10;`, `list[6]= 35;`, and `list[5] = list[3] + list[6];`

Accessing Array Components (continued)

EXAMPLE 9-2

You can also declare arrays as follows:

```
const int ARRAY_SIZE = 10;
int list[ARRAY_SIZE];
```

That is, you can first declare a named constant and then use the value of the named constant to declare an array and specify its size.

NOTE When you declare an array, its size must be known. For example, you cannot do the following:

```
int arraySize;                                //Line 1

cout << "Enter the size of the array: "; //Line 2
cin >> arraySize;                             //Line 3
cout << endl;                                 //Line 4

int list[arraySize];                          //Line 5; not allowed
```

Processing One-Dimensional Arrays

- Some basic operations performed on a one-dimensional array are:
  - Initializing
  - Inputting data
  - Outputting data stored in an array
  - Finding the largest and/or smallest element
- Each operation requires ability to step through the elements of the array
- Easily accomplished by a loop

Processing One-Dimensional Arrays (continued)

- Consider the declaration

```
int list[100];  //array of size 100
int i;
```

- Using `for` loops to access array elements:

```
for (i = 0; i < 100; i++) //Line 1
    //process list[i]     //Line 2
```

- Example:

```
for (i = 0; i < 100; i++) //Line 1
    cin >> list[i];       //Line 2
```

EXAMPLE 9-3

```
double sales[10];
int index;
double largestSale, sum, average;
```

Initializing an array:

```
for (index = 0; index < 10; index++)
    sales[index] = 0.0;
```

Reading data into an array:

```
for (index = 0; index < 10; index++)
    cin >> sales[index];
```

Printing an array:

```
for (index = 0; index < 10; index++)
    cout << sales[index] << " ";
```

Finding the sum and average of an array:

```
sum = 0;
for (index = 0; index < 10; index++)
    sum = sum + sales[index];

average = sum / 10;
```

Largest element in the array:

```
maxIndex = 0;
for (index = 1; index < 10; index++)
    if (sales[maxIndex] < sales[index])
        maxIndex = index;
largestSale = sales[maxIndex];
```

Array Index Out of Bounds

- If we have the statements:

```
double num[10];
int i;
```

- The component `num[i]` is valid if `i = 0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`
- The index of an array is in bounds if the `index >=0` and the `index <= ARRAY_SIZE-1`
  - Otherwise, we say the `index` is out of bounds
- In C++, there is no guard against indices that are out of bounds

Array Initialization During Declaration

- Arrays can be initialized during declaration
  - In this case, it is not necessary to specify the size of the array
    - Size determined by the number of initial values in the braces
- Example:

```
double sales[] = {12.25, 32.50, 16.90, 23, 45.68};
```

Partial Initialization of Arrays During Declaration

- The statement:

```
int list[10] = {0};
```

  declares `list` to be an array of `10` components and initializes all of them to zero
- The statement:

```
int list[10] = {8, 5, 12};
```

  declares `list` to be an array of `10` components, initializes `list[0]` to `8`, `list[1]` to `5`, `list[2]` to `12` and all other components are initialized to `0`

Partial Initialization of Arrays During Declaration (continued)

- The statement:

```
int list[] = {5, 6, 3};
```

declares `list` to be an array of `3` components and initializes `list[0]` to `5`, `list[1]` to `6`, and `list[2]` to `3`

- The statement:

```
int list[25]= {4, 7};
```

declares an array of `25` components; initializes `list[0]` to `4` and `list[1]` to `7`; all other components are initialized to `0`

Some Restrictions on Array Processing

- Consider the following statements:

```
int myList[5] = {0, 4, 8, 12, 16};  //Line 1
int yourList[5];  //Line 2
```

- C++ does not allow aggregate operations on an array:

```
yourList = myList;  //illegal
```

- Solution:

```
for (int index = 0; index < 5; index ++)
    yourList[index] = myList[index];
```

Some Restrictions on Array Processing (continued)

- The following is illegal too:

```
cin >> yourList; //illegal
```

- Solution:

```
for (int index = 0; index < 5; index ++)
    cin >> yourList[index];
```

- The following statements are legal, but do not give the desired results:

```
cout << yourList;

if (myList <= yourList)
.
.
```

Arrays as Parameters to Functions

- Arrays are passed by reference only
- The symbol `&` is *not* used when declaring an array as a formal parameter
- The size of the array is usually omitted
  - If provided, it is ignored by the compiler

EXAMPLE 9-5

Consider the following function:

```
void funcArrayAsParam(int listOne[], double listTwo[])
{
      .
      .
      .
}
```

Constant Arrays as Formal Parameters

EXAMPLE 9-6

```
    //Function to initialize an int array to 0.
    //The array to be initialized and its size are passed
    //as parameters. The parameter listSize specifies the
    //number of elements to be initialized.
void initializeArray(int list[], int listSize)
{
    int index;

    for (index = 0; index < listSize; index++)
        list[index] = 0;
}
    //Function to print the elements of an int array.
    //The array to be printed and the number of elements
    //are passed as parameters. The parameter listSize
    //specifies the number of elements to be printed.
void printArray(const int list[], int listSize)
{
    int index;

    for (index = 0; index < listSize; index++)
        cout << list[index] << " ";
}
```

Base Address of an Array and Array in Computer Memory

- The base address of an array is the address, or memory location of the first array component
- If `list` is a one-dimensional array, its base address is the address of `list[0]`
- When we pass an array as a parameter, the base address of the actual array is passed to the formal parameter

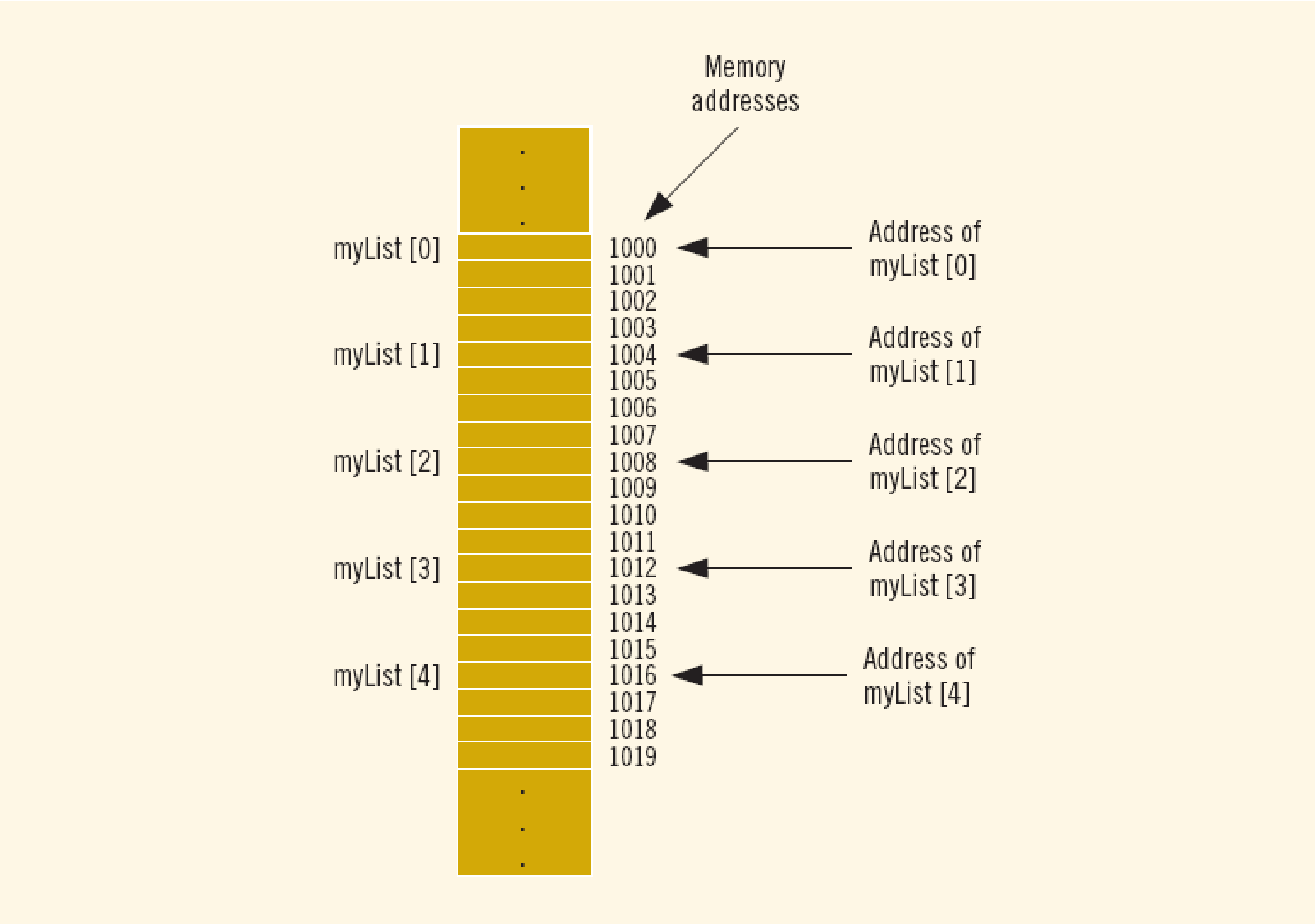

FIGURE 9-6 Array `myList` and the addresses of its components

Functions Cannot Return a Value of the Type Array

- C++ does not allow functions to return a value of the type array

Integral Data Type and Array Indices

- C++ allows any integral type to be used as an array index
- Example:

```
enum paintType {GREEN, RED, BLUE, BROWN, WHITE, ORANGE, YELLOW};
double paintSale[7];
paintType paint;

for (paint = GREEN; paint <= YELLOW;
                    paint = static_cast<paintType>(paint + 1))
    paintSale[paint] = 0.0;

paintSale[RED] = paintSale[RED] + 75.69;
```

Other Ways to Declare Arrays

```
const int NO_OF_STUDENTS = 20;
int testScores[NO_OF_STUDENTS];

const int SIZE = 50;                    //Line 1
typedef double list[SIZE];              //Line 2

list yourList;                          //Line 3
list myList;                            //Line 4
```

C–Strings (Character Arrays)

- Character array: an array whose components are of type `char`
- C-strings are null-terminated (`'\0'`) character arrays
- Example:
  - `'A'` is the character `A`
  - `"A"` is the C-string `A`
    - `"A"` represents two characters, `'A'` and `'\0'`

`C`–Strings (Character Arrays) (continued)

- Consider the statement

```
char name[16];
```

- Since `C`-strings are null terminated and `name` has 16 components, the largest string that it can store has 15 characters
- If you store a string of length, say 10 in `name`
  - The first 11 components of `name` are used and the last five are left unused

C-Strings (Character Arrays) (continued)

- The statement

```
char name[16] = "John";
```

declares an array `name` of length 16 and stores the C-string `"John"` in it

- The statement

```
char name[] = "John";
```

declares an array `name` of length 5 and stores the C-string `"John"` in it

C–Strings (Character Arrays) (continued)

TABLE 9-1 strcpy, strcmp, and strlen functions

Function	Effect
`strcpy(s1, s2)`	Copies the string s2 into the string variable s1 The length of s1 should be at least as large as s2
`strcmp(s1, s2)`	Returns a value < 0 if s1 is less than s2 Returns 0 if s1 and s2 are the same Returns a value > 0 if s1 is greater than s2
`strlen(s)`	Returns the length of the string s, excluding the null character

String Comparison

- `C`-strings are compared character by character using the collating sequence of the system
- If we are using the ASCII character set
  - `"Air" < "Boat"`
  - `"Air" < "An"`
  - `"Bill" < "Billy"`
  - `"Hello" < "hello"`

EXAMPLE 9-8

Suppose you have the following statements:

```
char studentName[21];
char myname[16];
char yourname[16];
```

The following statements show how string functions work:

Statement	Effect
`strcpy(myname, "John Robinson");`	`Myname = "John Robinson"`
`strlen("John Robinson");`	Returns 13, the length of the string `"John Robinson"`
`int len;` `len = strlen("Sunny Day");`	Stores 9 into `len`
`strcpy(yourname, "Lisa Miller");` `strcpy(studentName, yourname);`	`yourname = "Lisa Miller"` `studentName = "Lisa Miller"`
`strcmp("Bill", "Lisa");`	Returns a value < 0

`strcpy(yourname, "Kathy Brown");`
`strcpy(myname, "Mark G. Clark");`
`strcmp(myname, yourname);`
	`yourname = "Kathy Brown"`
	`myname = "Mark G. Clark"`
	Returns a value > 0

Reading and Writing Strings

- Most rules that apply to arrays apply to C-strings as well
- Aggregate operations, such as assignment and comparison, are not allowed on arrays
- Even the input/output of arrays is done component-wise
- The one place where C++ allows aggregate operations on arrays is the input and output of C-strings (that is, character arrays)

String Input

- `cin >> name;` stores the next input `C`-string into `name`
- To read strings with blanks, use `get`:

```
cin.get(str, m+1);
```

  - Stores the next `m` characters into `str` but the newline character is not stored in `str`
  - If the input string has fewer than `m` characters, the reading stops at the newline character

String Output

- `cout << name;` outputs the content of `name` on the screen
  - `<<` continues to write the contents of name until it finds the null character
  - If `name` does not contain the null character, then we will see strange output
    - `<<` continues to output data from memory adjacent to `name` until `'\0'` is found

Specifying Input/Output Files at Execution Time

- You can let the user specify the name of the input and/or output file at execution time:

```
ifstream infile;
ofstream outfile;

char fileName[51];     //assume that the file name is at most
                       //50 characters long

cout << "Enter the input file name: ";
cin >> fileName;

infile.open(fileName);   //open the input file
.
.
.

cout << "Enter the output file name: ";
cin >> fileName;

outfile.open(fileName);  //open the output file
```

`string` Type and Input/Output Files

- Argument to the function `open` must be a null-terminated string (a `C`-string)
- If we use a variable of type `string` to read the name of an I/O file, the value must first be converted to a `C`-string before calling `open`
- Syntax:

```
strVar.c_str()
```

where `strVar` is a variable of type `string`

Parallel Arrays

- Two (or more) arrays are called parallel if their corresponding components hold related information
- Example:

```
int studentId[50];
char courseGrade[50];
```

```
23456 A
86723 B
22356 C
92733 B
11892 D
.
.
.
```

Two-Dimensional Arrays

- Two-dimensional array: collection of a fixed number of components (of the same type) arranged in two dimensions
  - Sometimes called matrices or tables
- Declaration syntax:

```
dataType  arrayName[intExp1][intExp2];
```

where `intexp1` and `intexp2` are expressions yielding positive integer values, and specify the number of rows and the number of columns, respectively, in the array

Two-Dimensional Arrays (continued)

```
double sales[10][5];
```

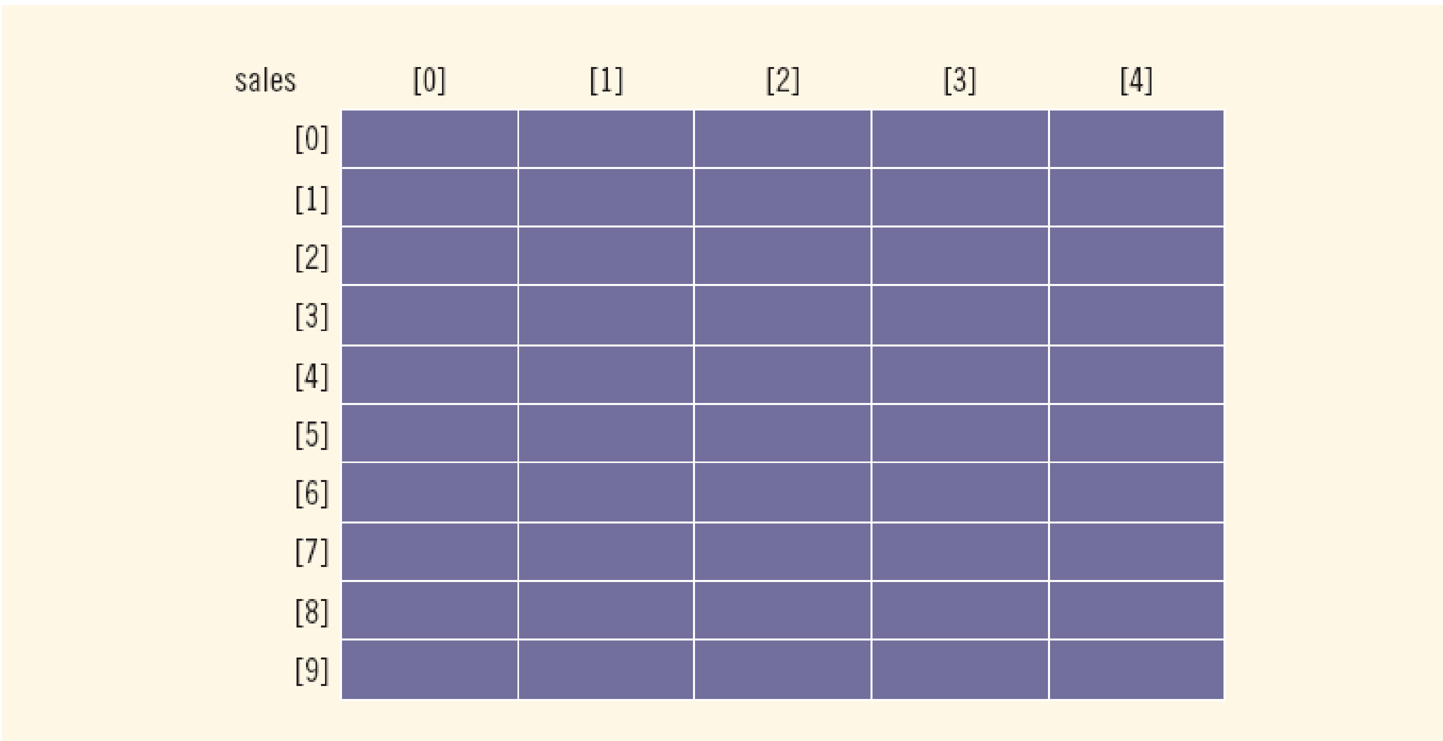

FIGURE 9-8 Two-dimensional array `sales`

Accessing Array Components

- Syntax:

```
arrayName[indexExp1][indexExp2]
```

where `indexexp1` and `indexexp2` are expressions yielding nonnegative integer values, and specify the row and column position

Accessing Array Components (continued)

```
sales[5][3] = 25.75;
```

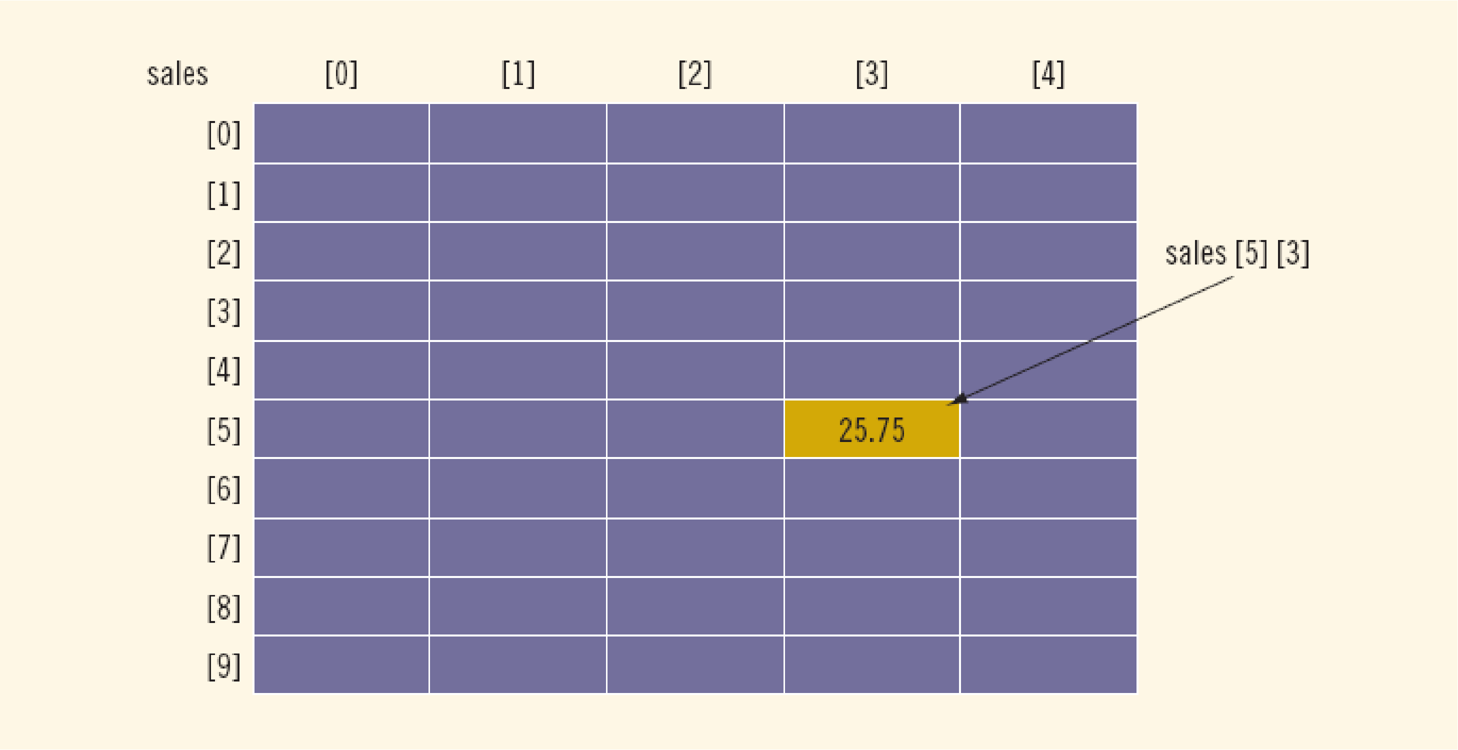

FIGURE 9-9 `sales[5][3]`

Two-Dimensional Array Initialization During Declaration

- Two-dimensional arrays can be initialized when they are declared:

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                   {20, 4, 7},
                   {11, 18, 14}};
```

  - Elements of each row are enclosed within braces and separated by commas
  - All rows are enclosed within braces
  - For number arrays, if all components of a row aren’t specified, unspecified ones are set to 0

Two-Dimensional Arrays and Enumeration Types

```
enum carType {GM, FORD, TOYOTA, BMW, NISSAN, VOLVO};
enum colorType {RED, BROWN, BLACK, WHITE, GRAY};

int inStock[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
inStock[FORD][WHITE] = 15;
```

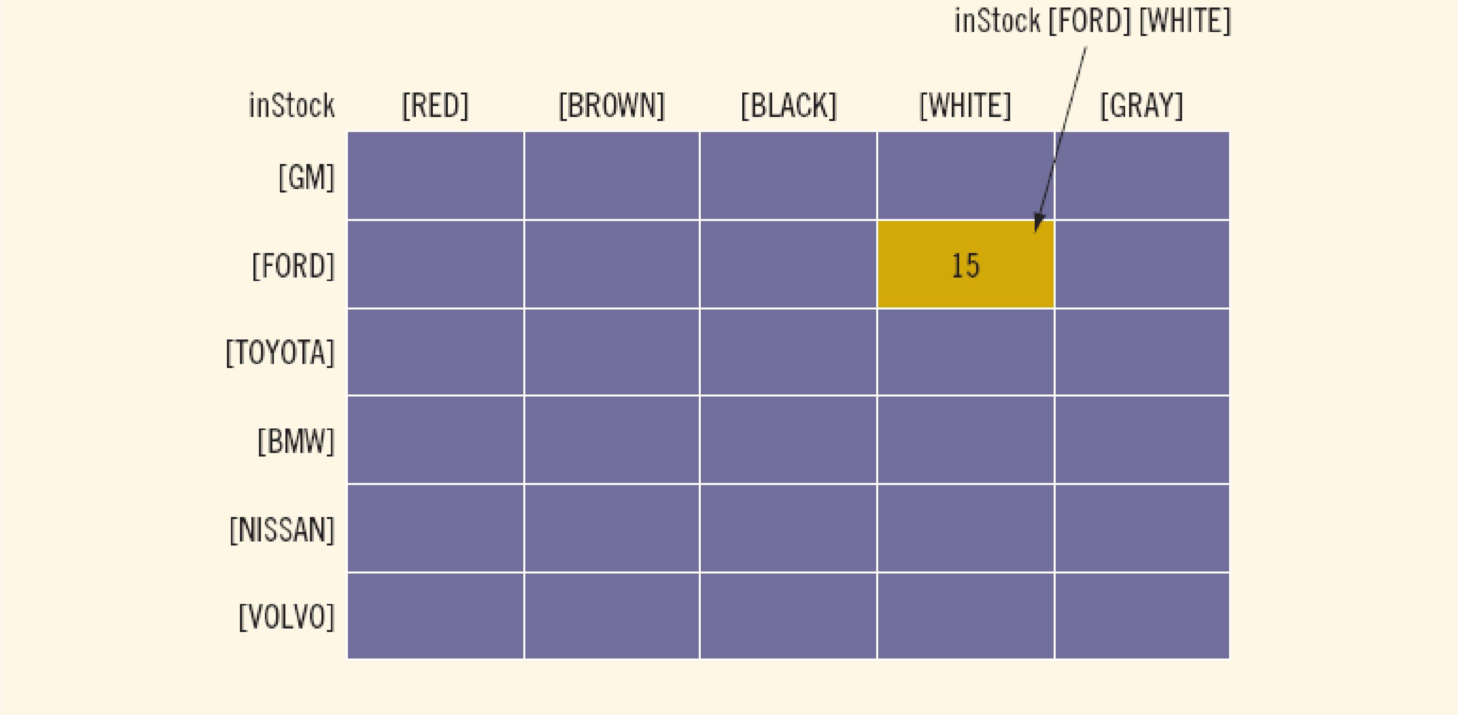

FIGURE 9-12 inStock[FORD][WHITE]

Processing Two-Dimensional Arrays

- Ways to process a two-dimensional array:
  - Process the entire array
  - Process a particular row of the array, called row processing
  - Process a particular column of the array, called column processing
- Each row and each column of a two-dimensional array is a one-dimensional array
  - To process, use algorithms similar to processing one-dimensional arrays

Processing Two-Dimensional Arrays (continued)

```
const int NUMBER_OF_ROWS = 7;    //This can be set to any number.
const int NUMBER_OF_COLUMNS = 6; //This can be set to any number.

int matrix[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
int row;
int col;
int sum;
int largest;
int temp;
```

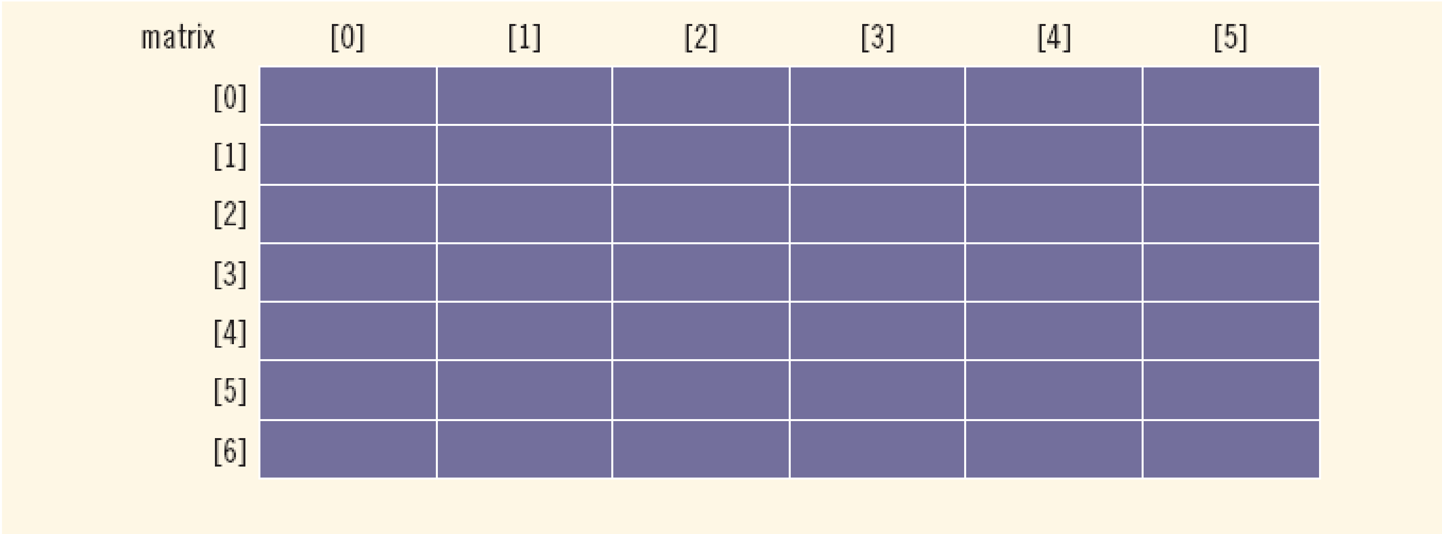

FIGURE 9-13 Two-dimensional array `matrix`

Initialization

- To initialize row number 4 (i.e., fifth row) to 0

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    matrix[row][col] = 0;
```

- To initialize the entire matrix to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

- To output the components of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cout << setw(5) << matrix[row][col] << " ";

    cout << endl;
}
```

Input

- To input data into each component of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cin >> matrix[row][col];
```

Sum by Row

- To find the sum of row number 4 of `matrix`:

```
sum = 0;
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

- To find the sum of each individual row:

```
  //Sum of each individual row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    sum = 0;
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        sum = sum + matrix[row][col];

    cout << "Sum of row " << row + 1 << " = " << sum << endl;
}
```

Sum by Column

- To find the sum of each individual column:

```
  //Sum of each individual column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
    sum = 0;
    for (row = 0; row < NUMBER_OF_ROWS; row++)
        sum = sum + matrix[row][col];

    cout << "Sum of column " << col + 1 << " = " << sum
         << endl;
}
```

Largest Element in Each Row and Each Column

```
  //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    largest = matrix[row][0]; //Assume that the first element
                              //of the row is the largest.
    for (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in row " << row + 1 << " = "
         << largest << endl;
}

  //Largest element in each column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
    largest = matrix[0][col]; //Assume that the first element
                              //of the column is the largest.
    for (row = 1; row < NUMBER_OF_ROWS; row++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in column " << col + 1
         << " = " << largest << endl;
}
```

Reversing Diagonal

- Before:

matrix	[0]	[1]	[2]	[3]
[0]	1	8	10	11
[1]	34	2	12	45
[2]	0	13	3	20
[3]	14	35	56	4

FIGURE 9-14 Two-dimensional array `matrix`

Reversing Diagonal (continued)

- To reverse both the diagonals:

```
  //Reverse the main diagonal
for (row = 0; row < NUMBER_OF_ROWS / 2; row++)
{
    temp = matrix[row][row];
    matrix[row][row] =
      matrix[NUMBER_OF_ROWS - 1 - row][NUMBER_OF_ROWS - 1 - row];
    matrix[NUMBER_OF_ROWS - 1 - row][NUMBER_OF_ROWS - 1 - row]
          = temp;
}

  //Reverse the opposite diagonal
for (row = 0; row < NUMBER_OF_ROWS / 2; row++)
{
    temp = matrix[row][NUMBER_OF_ROWS - 1 - row];
    matrix[row][NUMBER_OF_ROWS - 1 - row] =
                    matrix[NUMBER_OF_ROWS - 1 - row][row];
    matrix[NUMBER_OF_ROWS - 1 - row][row] = temp;
}
```

Reversing Diagonal (continued)

- After:

matrix	[0]	[1]	[2]	[3]
[0]	4	8	10	14
[1]	34	3	13	45
[2]	0	12	2	20
[3]	11	35	56	1

FIGURE 9-15 The array `matrix` after reversing diagonals

Passing Two-Dimensional Arrays as Parameters to Functions

- Two-dimensional arrays can be passed as parameters to a function
  - Pass by reference
    - Base address (address of first component of the actual parameter) is passed to formal parameter
- Two-dimensional arrays are stored in row order
- When declaring a two-dimensional array as a formal parameter, can omit size of first dimension, but not the second

Arrays of Strings

- Strings in C++ can be manipulated using either the data type string or character arrays (`C`-strings)
- On some compilers, the data type `string` may not be available in Standard C++ (i.e., non-ANSI/ISO Standard C++)

Arrays of Strings and the `string` Type

- To declare an array of 100 components of type `string`:

```
string list[100];
```

- Basic operations, such as assignment, comparison, and input/output, can be performed on values of the `string` type
- The data in `list` can be processed just like any one-dimensional array

Arrays of Strings and C-Strings (Character Arrays)

```
char list[100][16];
strcpy(list[1], "Snow White");
```

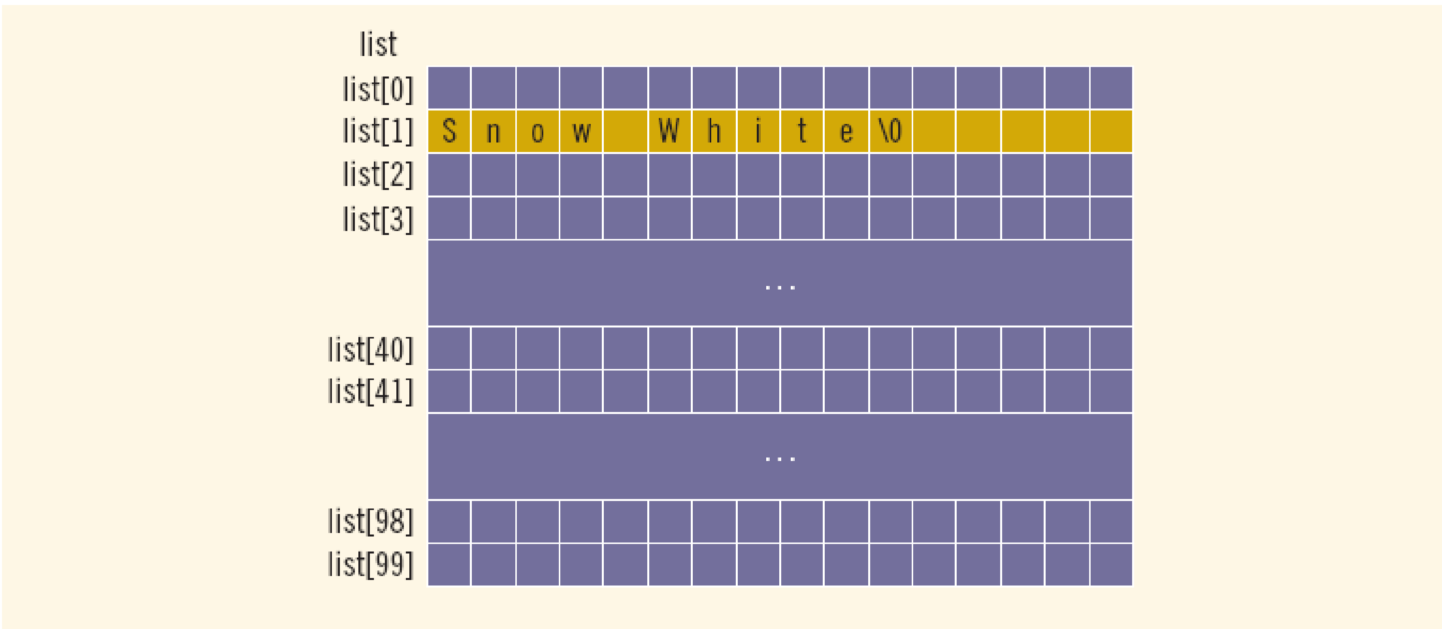

FIGURE 9-17 Array `list`, showing `list[1]`

```
for (j = 0; j < 100; j++)
    cin.get(list[j], 16);
```

Another Way to Declare a Two-Dimensional Array

- Consider the following:

```
const int NUMBER_OF_ROWS = 20;
const int NUMBER_OF_COLUMNS = 10;

typedef int tableType[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
```

- To declare an array of 20 rows and 10 columns:

```
tableType matrix;
```

Multidimensional Arrays

- Multidimensional array: collection of a fixed number of elements (called components) arranged in *n* dimensions (n >= 1)
  - Also called an *n*-dimensional array
- Declaration syntax:

```
dataType arrayName[intExp1][intExp2] ... [intExpn];
```

- To access a component:

```
arrayName[indexExp1][indexExp2] ... [indexExpn]
```

Multidimensional Arrays (continued)

- When declaring a multidimensional array as a formal parameter in a function
  - Can omit size of first dimension but not other dimensions
- As parameters, multidimensional arrays are passed by reference only
- A function cannot return a value of the type array
- There is no check if the array indices are within bounds

Programming Example: Code Detection

- When a message is transmitted in secret code over a transmission channel, it is usually transmitted as a sequence of bits (0s and 1s)
- Due to noise in the transmission channel, the transmitted message may become corrupted
  - Message received at destination is not the same as the message transmitted
  - Some of the bits may have been changed

Programming Example: Code Detection (continued)

- Several techniques to check the validity of the transmitted message at the destination
- One technique is to transmit the same message twice
  - At the destination, both copies of the message are compared bit by bit
  - If the corresponding bits are the same, the message received is error-free

Programming Example: Code Detection (continued)

- We write a program to check if the message received at the destination is error-free
- For simplicity, assume that:
  - The secret code representing the message is a sequence of digits (0 to 9)
  - The maximum length of the message is 250 digits
- The first number in the message is the length of the message

Programming Example: Code Detection (continued)

- If the secret code is

```
7 9 2 7 8 3 5 6
```

  then the message is seven digits long
- The above message is transmitted (twice) as

```
7 9 2 7 8 3 5 6 7 9 2 7 8 3 5 6
```

- Input: a file containing the secret code and its copy
- Output: the secret code, its copy, and a message if the received code is error-free

Programming Example: Code Detection (continued)

- The results are output in the following form:

```
Code Digit          Code Digit Copy
      9                 9
      2                 2
      7                 7
      8                 8
      3                 3
      5                 5
      6                 6
```

- Message transmitted OK

Programming Example: Problem Analysis

- Because we have to compare digits of the secret code and its copy:
  - First, read the secret code and store it in an array
  - Next, read first digit of the copy and compare it with the first digit of the code, and so on
  - If any corresponding digits are not the same, print a message next to the digits
- The first number in the secret code, and in the copy, indicates the length of the code

Programming Example: Algorithm Design

- Open the input and output files
- If the input file does not exist, exit the program
- Read the length of the secret code
- If the length of the secret code is greater than 250, terminate the program because the maximum length of the code in this program is 250
- Read and store the secret code into an array

Programming Example: Algorithm Design (continued)

- Read the length of the copy
- If the length of the secret code and its copy are the same, compare the codes; otherwise, print an error message
- Note: To simplify function `main`, write a function, `readCode`, to read the secret code and another function, `compareCode`, to compare the codes

Programming Example: `readCode`

- First, read length of secret code
- If length of secret code is greater than 250
  - Set `lenCodeOk` (a reference parameter) to `false` and the function terminates
- Value of `lenCodeOk` is passed to calling function to indicate if secret code was read successfully
- If length of code is less than 250, `readCode` reads and stores secret code into an array

Programming Example: `readCode` (continued)

```
void readCode(ifstream& infile, int list[], int& length,
              bool& lenCodeOk)
{
    int count;

    lenCodeOk = true;

    infile >> length;  //get the length of the secret code

    if (length > MAX_CODE_SIZE)
    {
        lenCodeOk = false;
        return;
    }

        //Get the secret code.
    for (count = 0; count < length; count++)
        infile >> list[count];
}
```

Programming Example: `compareCode`

- Set a `bool` variable `codeOk` to true
- If length of code and copy are not equal
  - Output error message and terminate function
- For each digit in input file
  - Read the next digit of secret code copy
  - Output digits from code and copy
  - If corresponding digits are not equal, output error message and set `codeOk` to `false`
- If `codeOk`, output message indicating code transmitted OK, else output an error message

Programming Example: `compareCode` (continued)

```
void compareCode(ifstream& infile, ofstream& outfile,
                 int list[], int length)
{
        //Step a
    int length2;
    int digit;
    bool codeOk;
    int count;

    codeOk = true;                                    //Step b

    infile >> length2;                                //Step c

    if (length != length2)                            //Step d
    {
        cout << "The original code and its copy "
             << "are not of the same length."
             << endl;
        return;
    }

    outfile << "Code Digit    Code Digit Copy"
            << endl;
```

Programming Example: `compareCode` (continued)

```
    for (count = 0; count < length; count++)            //Step e
    {
        infile >> digit;                                //Step e.1
        outfile << setw(5) << list[count]
                << setw(17) << digit;                   //Step e.2

        if (digit != list[count])                       //Step e.3
        {
            outfile << "  code digits are not the same"
                    << endl;
            codeOk = false;
        }
        else
            outfile << endl;
    }

    if (codeOk)                                         //Step f
        outfile << "Message transmitted OK."
                << endl;
    else
        outfile << "Error in transmission. "
                << "Retransmit!!" << endl;
}
```

Programming Example: Main Algorithm

- Declare variables
- Open the files
- Call `readCode` to read the secret code
- If (length of the secret code <= 250)
  - Call `compareCode` to compare the codes

  else
  - Output an appropriate error message

Summary

- Array: structured data type with a fixed number of components of the same type
  - Components are accessed using their relative positions in the array
- Elements of a one-dimensional array are arranged in the form of a list
- An array index can be any expression that evaluates to a nonnegative integer
  - Must always be less than the size of the array

Summary (continued)

- The base address of an array is the address of the first array component
- When passing an array as an actual parameter, you use only its name
  - Passed by reference only
- A function cannot return a value of the type array
- In C++, C-strings are null terminated and are stored in character arrays

Summary (continued)

- Commonly used `C`-string manipulation functions include:
  - `strcpy`, `strcmp`, and `strlen`
- Parallel arrays are used to hold related information
- In a two-dimensional array, the elements are arranged in a table form

Summary

- To access an element of a two-dimensional array, you need a pair of indices:
  - One for the row position
  - One for the column position
- In row processing, a two-dimensional array is processed one row at a time
- In column processing, a two-dimensional array is processed one column at a time