Chapter 7: Arrays

C++ FOR ENGINEERS AND SCIENTISTS

Objectives

In this chapter, you will learn about:

- One-dimensional arrays
- Array initialization
- Declaring and processing two-dimensional arrays
- Arrays as arguments
- Statistical analysis

Objectives (continued)

- The Standard Template Library (STL)
- Searching and sorting
- Common programming errors

One-Dimensional Arrays

- **One-dimensional array:** A list of related values with the same data type, stored using a single group name (called the **array name**)
  - Syntax:

```
dataType arrayName[number-of-items]
```

- By convention, the number of items is first declared as a constant, and the constant is used in the array declaration

One-Dimensional Arrays (continued)

```
const int NUMELS = 6;
int volts[NUMELS];

const int ARRAYSIZE = 4;
char code[ARRAYSIZE];

const int SIZE = 100;
double amount[SIZE];
```

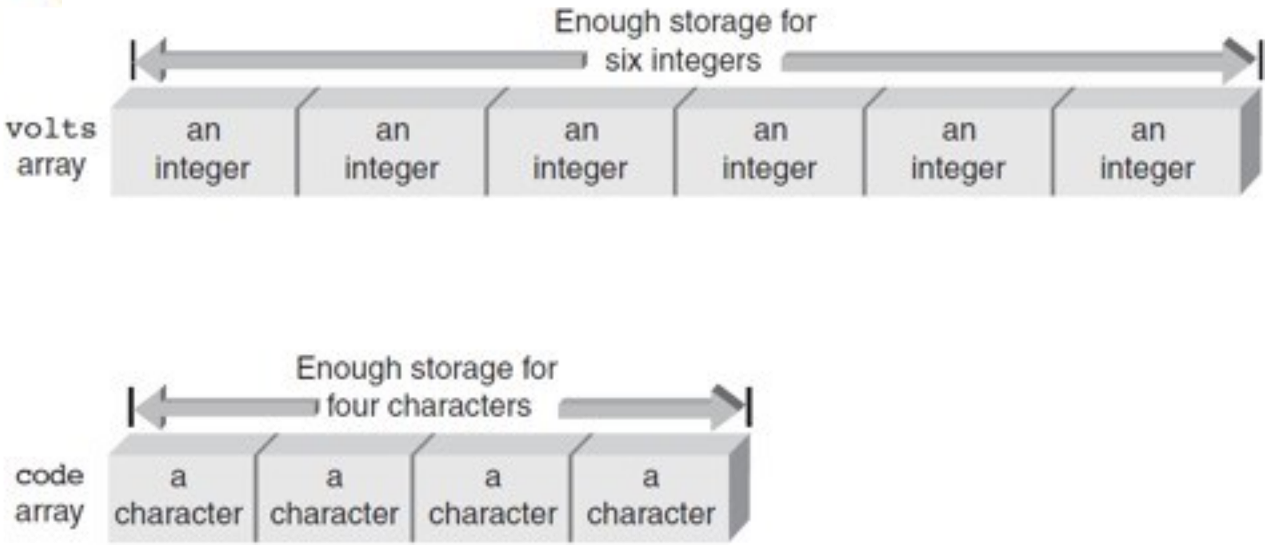


Figure 7.1 The `volts` and `code` arrays in memory

One-Dimensional Arrays (continued)

- **Element**: An item in the array
  - Array storage of elements is contiguous
- **Index** (or **subscript**) of an element: The position of the element within the array
  - Indexes are zero-relative
- To reference an element, use the array name and the index of the element

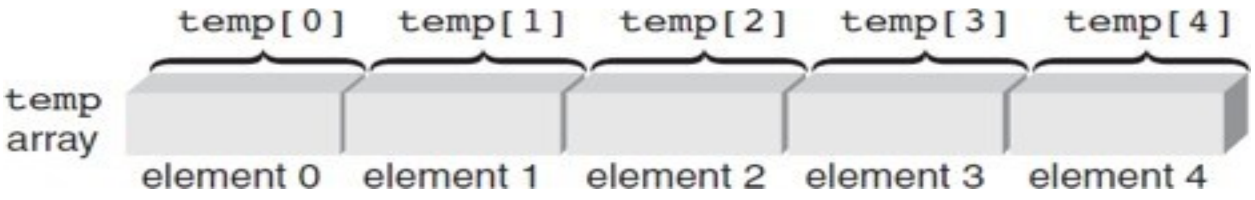


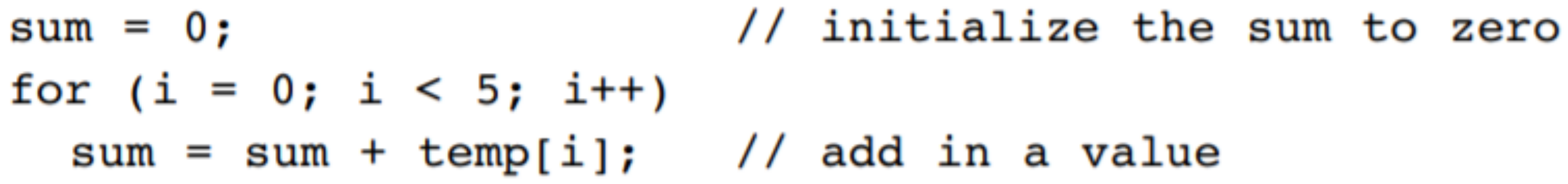
Figure 7.2 Identifying array elements

One-Dimensional Arrays (continued)

- Index represents the offset from the start of the array
- Element is also called **indexed variable** or **subscripted variable**
- Subscripted variable can be used anywhere that a variable can be used
- Expressions can be used within the brackets if the value of the expression
  - Yields an integer value
  - Is within the valid range of subscripts

One-Dimensional Arrays (continued)

- All of the elements of an array can be processed by using a loop
- The loop counter is used as the array index to specify the element
- Example:

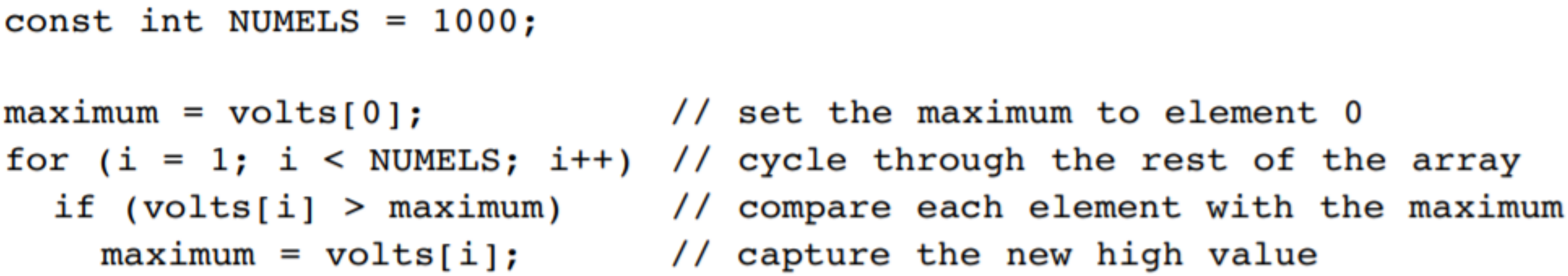


```
sum = 0;                     // initialize the sum to zero
for (i = 0; i < 5; i++)
  sum = sum + temp[i];       // add in a value
```

Refer to page 388 for more explanations and examples

One-Dimensional Arrays (continued)

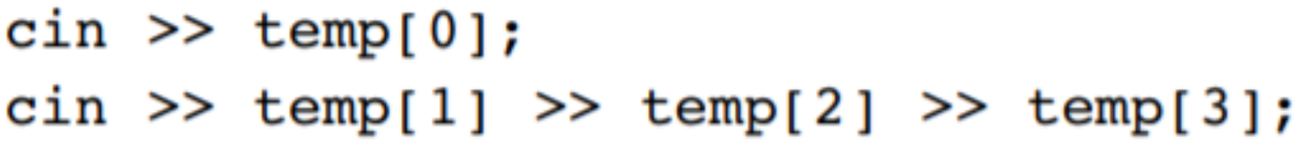
- Locate the maximum value in an array of 1000 elements named volts



```
const int NUMELS = 1000;

maximum = volts[0];            // set the maximum to element 0
for (i = 1; i < NUMELS; i++)  // cycle through the rest of the array
  if (volts[i] > maximum)     // compare each element with the maximum
    maximum = volts[i];       // capture the new high value
```

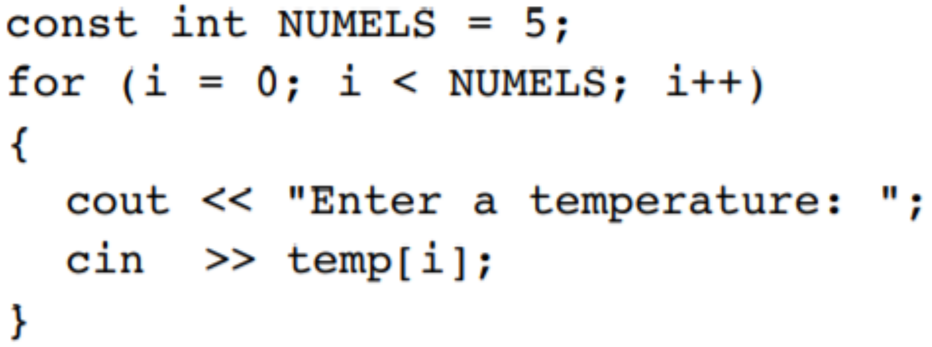
Input and Output of Array Values

- Array elements can be assigned values interactively using a **`cin`** stream object



```
cin >> temp[0];
cin >> temp[1] >> temp[2] >> temp[3];
```

- Alternatively, a for loop can be used to cycle through the array for interactive data input. For example, the following code prompts the user for five temperatures:



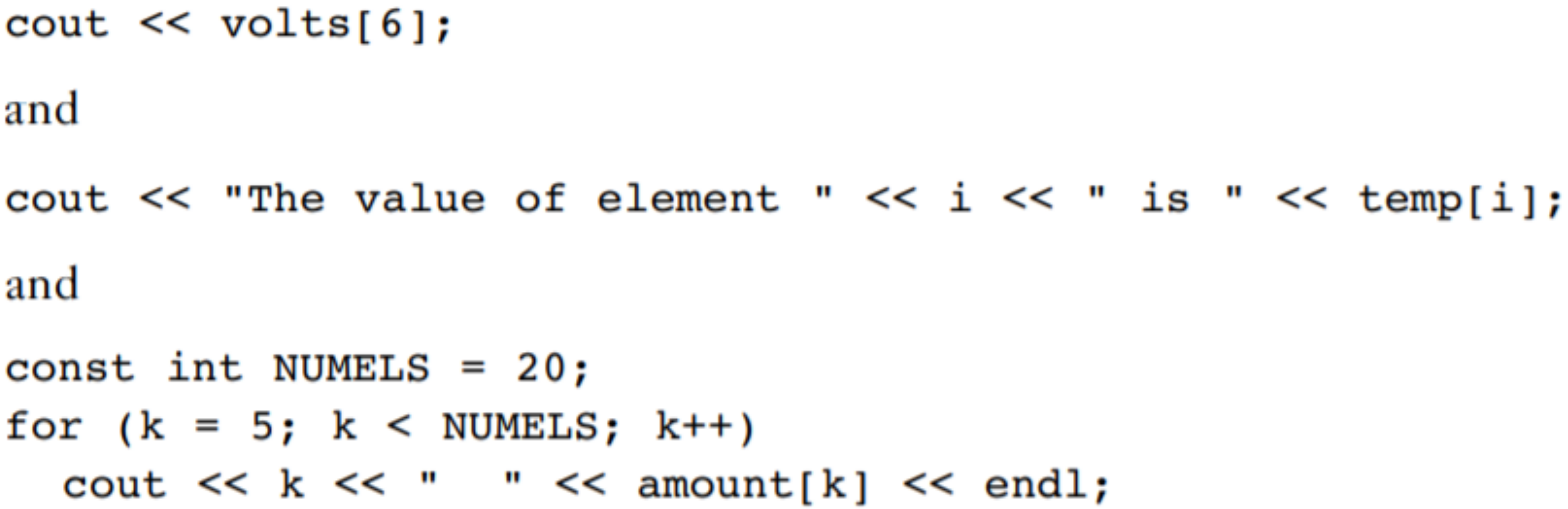
```
const int NUMELS = 5;
for (i = 0; i < NUMELS; i++)
{
  cout << "Enter a temperature: ";
  cin  >> temp[i];
}
```

Input and Output of Array Values

- Out of range array indexes are not checked at compile-time
  - May produce run-time errors
  - May overwrite a value in the referenced memory location and cause other errors

Input and Output of Array Values

- Array elements can be displayed using the **`cout`** stream object



```
cout << volts[6];
```

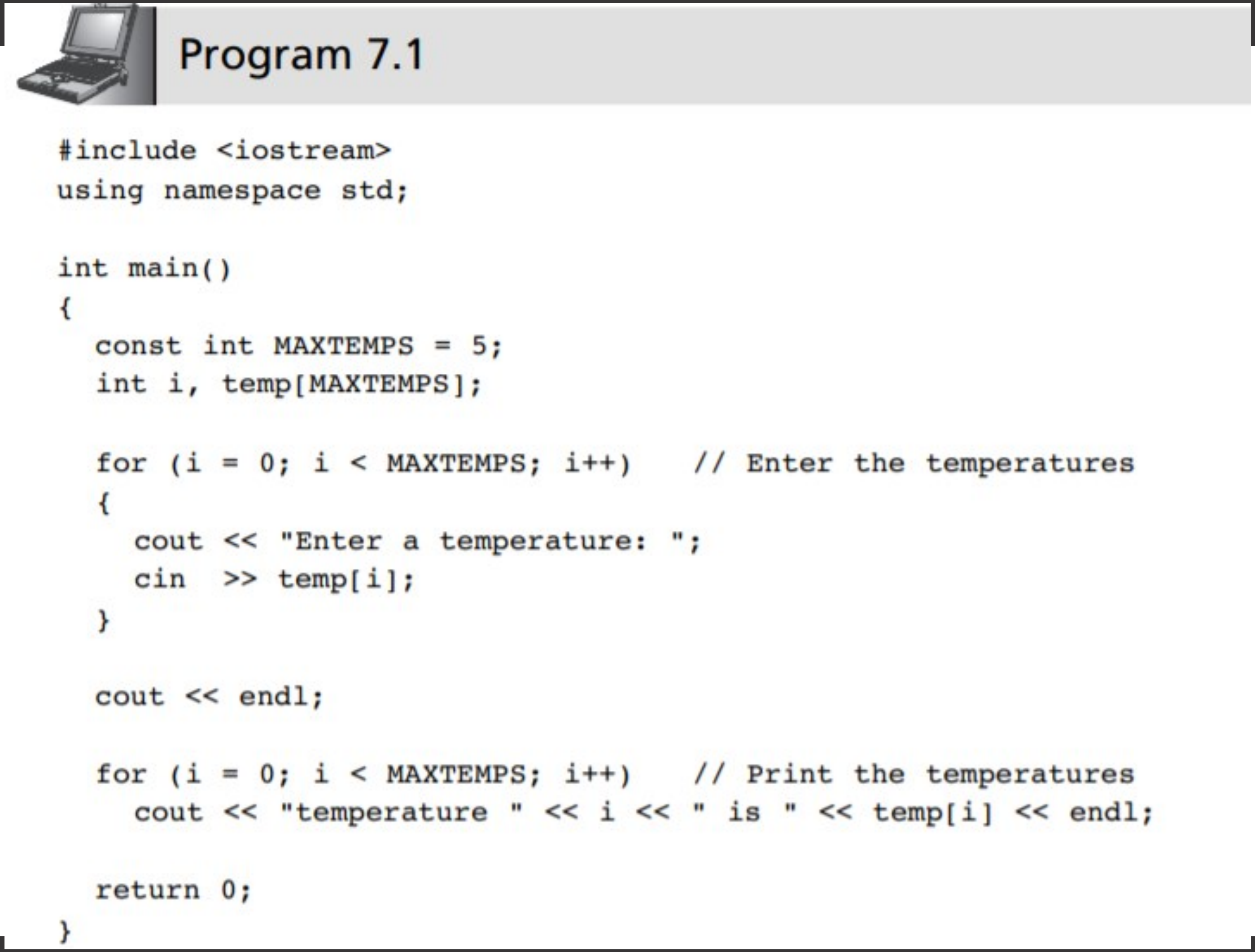
and

```
cout << "The value of element " << i << " is " << temp[i];
```

and

```
const int NUMELS = 20;
for (k = 5; k < NUMELS; k++)
  cout << k << "   " << amount[k] << endl;
```

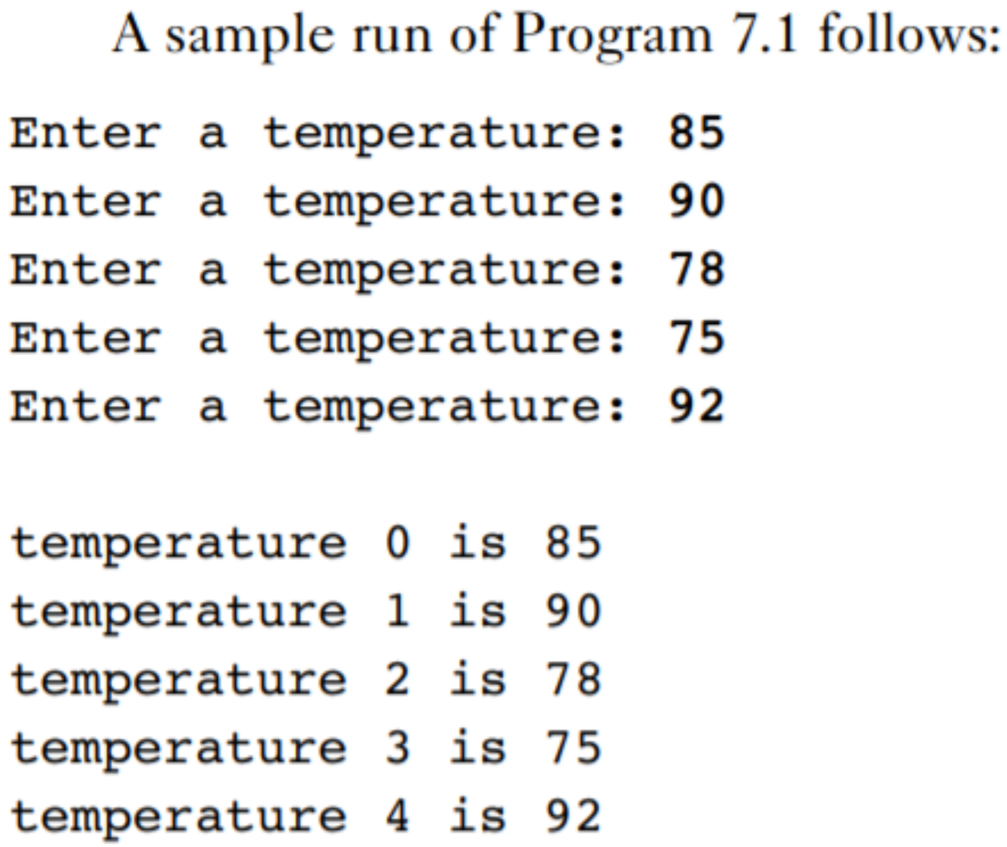
Input and Output of Array Values



Program 7.1

```
#include <iostream>
using namespace std;

int main()
{
  const int MAXTEMPS = 5;
  int i, temp[MAXTEMPS];

  for (i = 0; i < MAXTEMPS; i++)    // Enter the temperatures
  {
    cout << "Enter a temperature: ";
    cin  >> temp[i];
  }

  cout << endl;

  for (i = 0; i < MAXTEMPS; i++)    // Print the temperatures
    cout << "temperature " << i << " is " << temp[i] << endl;

  return 0;
}
```

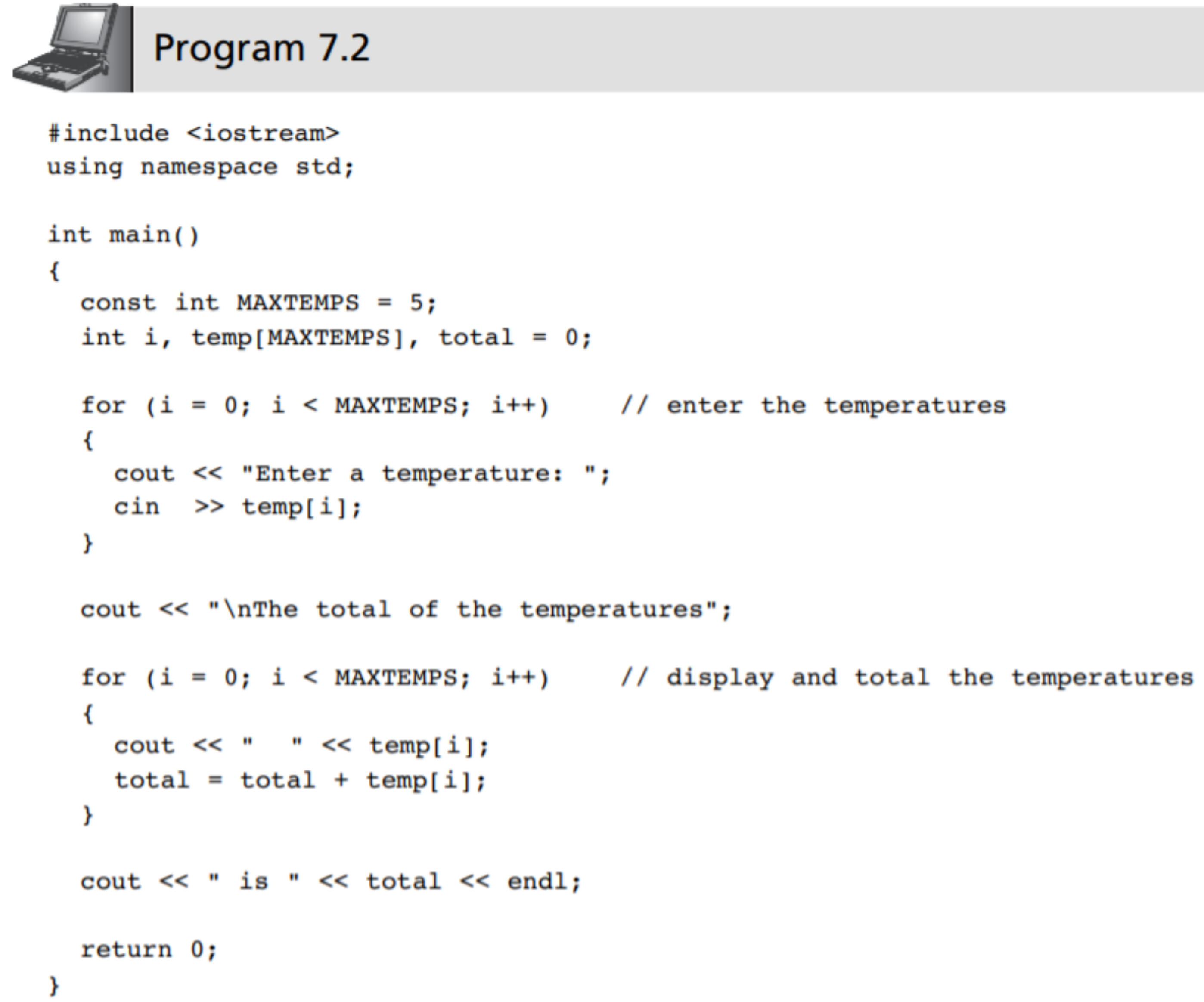
Input and Output of Array Values



A sample run of Program 7.1 follows:

```
Enter a temperature: 85
Enter a temperature: 90
Enter a temperature: 78
Enter a temperature: 75
Enter a temperature: 92

temperature 0 is 85
temperature 1 is 90
temperature 2 is 78
temperature 3 is 75
temperature 4 is 92
```

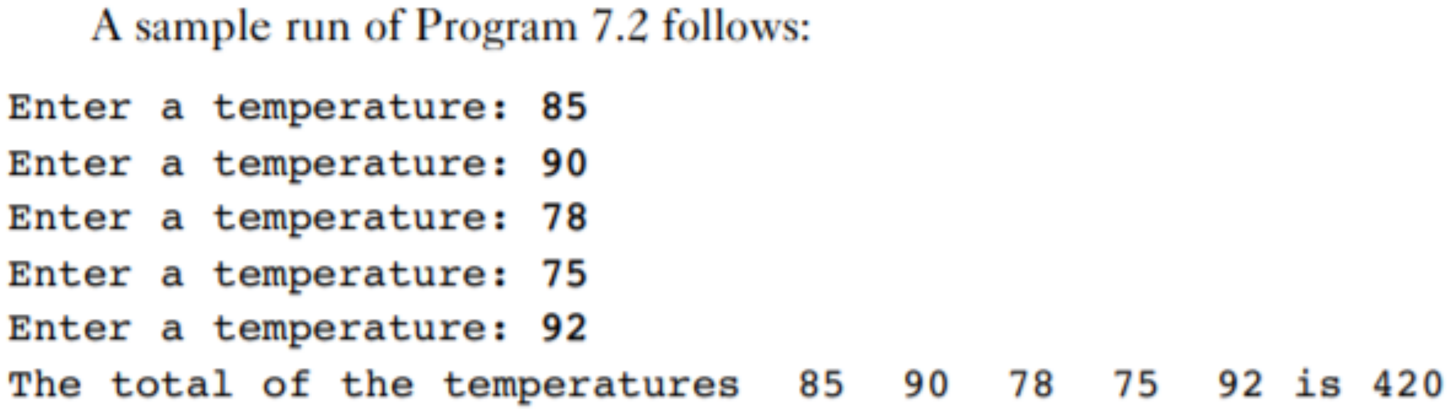
Input and Output of Array Values



Program 7.2

```
#include <iostream>
using namespace std;

int main()
{
  const int MAXTEMPS = 5;
  int i, temp[MAXTEMPS], total = 0;

  for (i = 0; i < MAXTEMPS; i++)    // enter the temperatures
  {
    cout << "Enter a temperature: ";
    cin  >> temp[i];
  }

  cout << "\nThe total of the temperatures";

  for (i = 0; i < MAXTEMPS; i++)    // display and total the temperatures
  {
    cout << "   " << temp[i];
    total = total + temp[i];
  }

  cout << " is " << total << endl;

  return 0;
}
```

Input and Output of Array Values



A sample run of Program 7.2 follows:

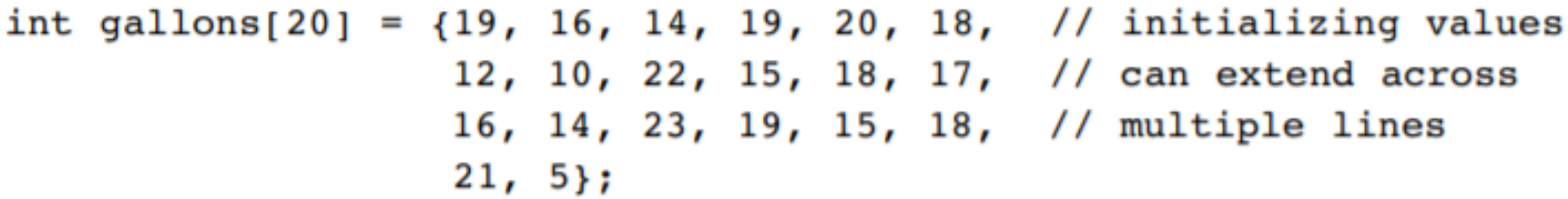
```
Enter a temperature: 85
Enter a temperature: 90
Enter a temperature: 78
Enter a temperature: 75
Enter a temperature: 92
The total of the temperatures  85  90  78  75  92 is 420
```

Array Initialization

- Array elements can be initialized in the array declaration statement
- Example:

```
int temp[5] = {98, 87, 92, 79, 85};
```

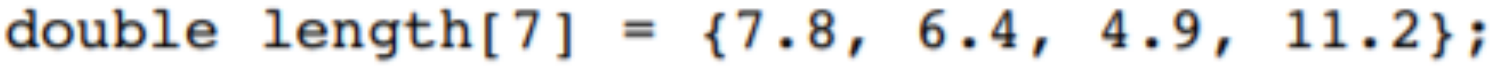
- Initialization:
  - —Can span multiple lines, because white space is ignored in C++



```
int gallons[20] = {19, 16, 14, 19, 20, 18,  // initializing values
                   12, 10, 22, 15, 18, 17,  // can extend across
                   16, 14, 23, 19, 15, 18,  // multiple lines
                   21, 5};
```

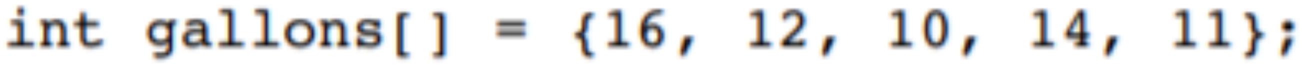
Array Initialization

- Initialization:
  - Starts with array element 0 if an insufficient number of values is specified



```
double length[7] = {7.8, 6.4, 4.9, 11.2};
```

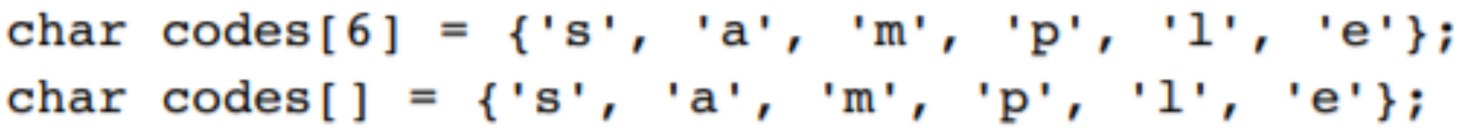
- If initializing in the declaration, the size may be omitted



```
int gallons[] = {16, 12, 10, 14, 11};
```

Array Initialization

- Similarly, the following two declarations are equivalent:



```
char codes[6] = {'s', 'a', 'm', 'p', 'l', 'e'};
char codes[] = {'s', 'a', 'm', 'p', 'l', 'e'};
```

Both these declarations set aside **six** character locations for an array named codes.

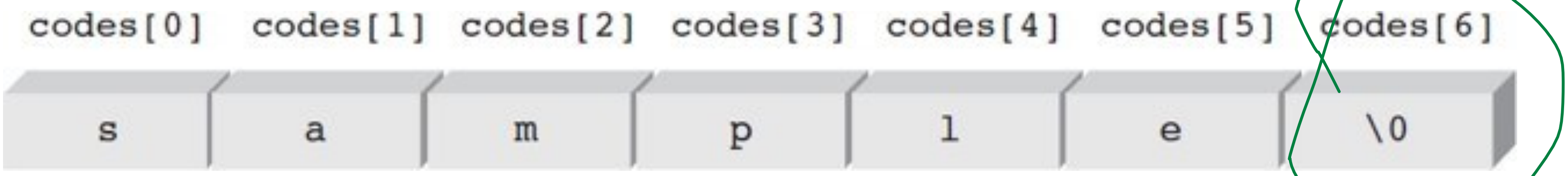
Array Initialization (continued)

- An interesting and useful simplification can also be used when initializing character arrays. For example, the following declaration uses the string "sample" to initialize the codes array:

```
    char codes[] = “sample”;
```

- char array will contain an extra null character at the end of the string

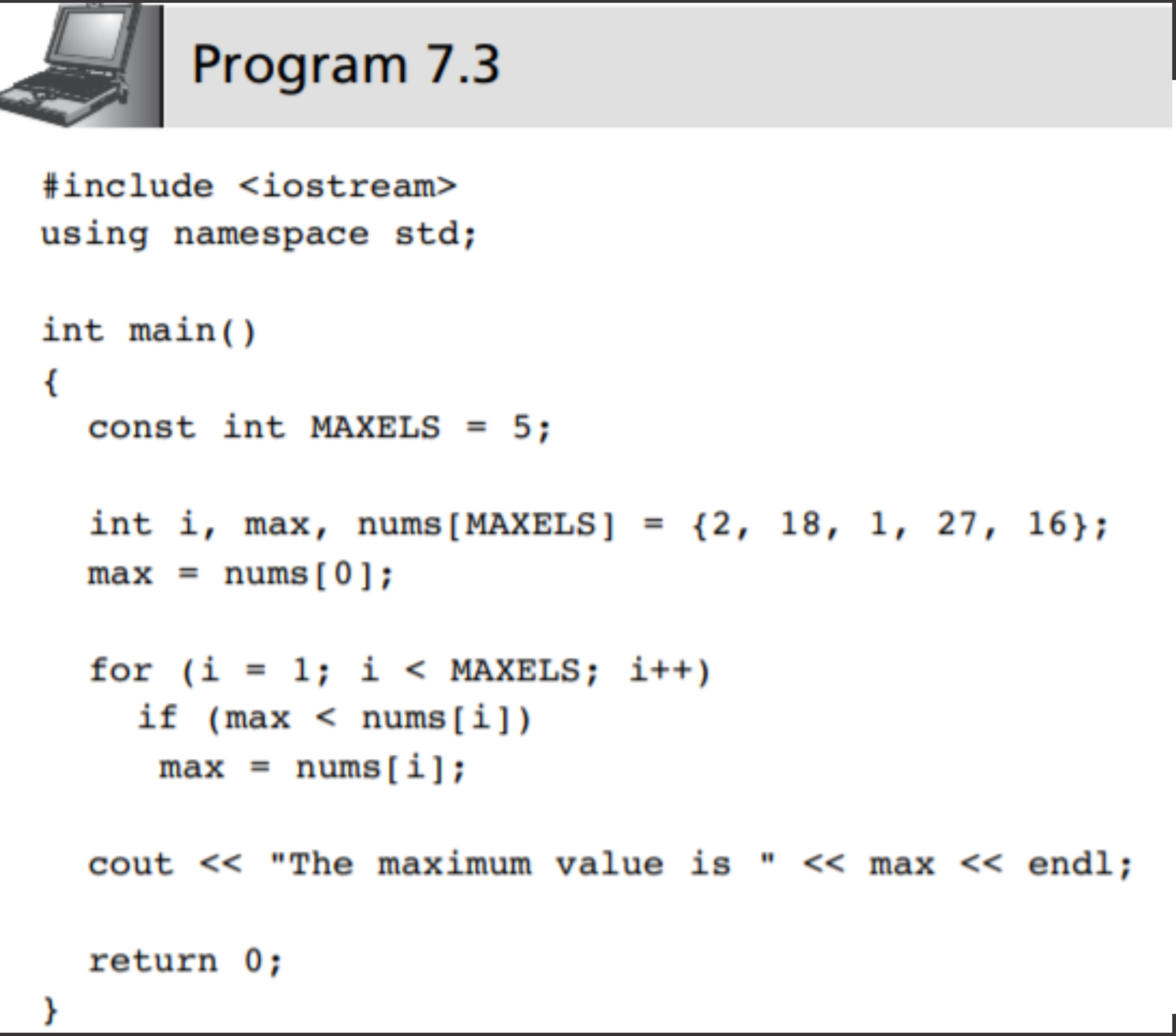
7char的空间



codes[0]	codes[1]	codes[2]	codes[3]	codes[4]	codes[5]	codes[6]
s	a	m	p	l	e	\0

Figure 7.4 Initializing a character array with a string adds a terminating \0 character

Array Initialization



Program 7.3

```
#include <iostream>
using namespace std;

int main()
{
  const int MAXELS = 5;

  int i, max, nums[MAXELS] = {2, 18, 1, 27, 16};
  max = nums[0];

  for (i = 1; i < MAXELS; i++)
    if (max < nums[i])
     max = nums[i];

  cout << "The maximum value is " << max << endl;

  return 0;
}
```

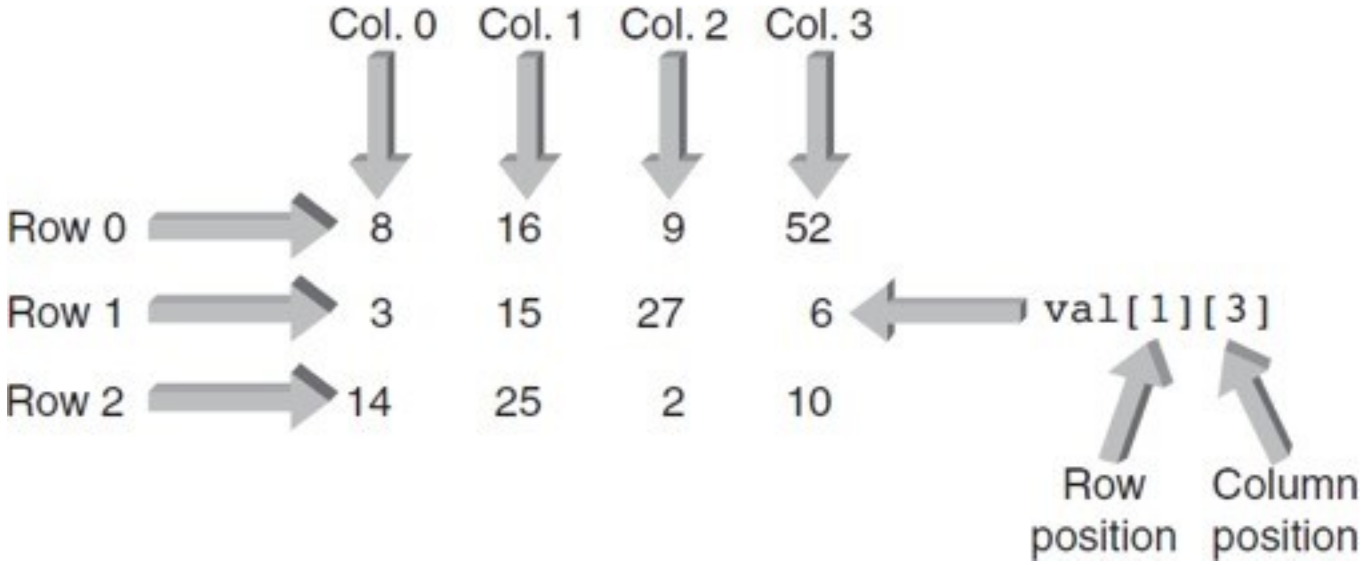
Declaring and Processing Two-Dimensional Arrays

- **Two-dimensional array:** Has both rows and columns
  - Also called a **table**
- Both dimensions must be specified in the array declaration
  - Row is specified first, then column
- Both dimensions must be specified when referencing an array element

Declaring and Processing Two-Dimensional Arrays (cont'd)

- Example:

```
int val[3][4];
```

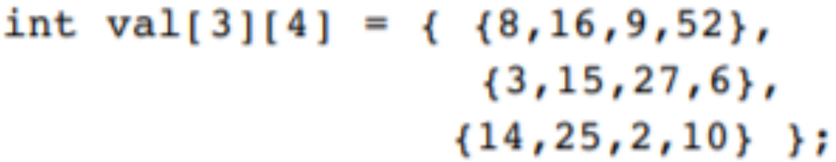


	Col. 0	Col. 1	Col. 2	Col. 3
Row 0	8	16	9	52
Row 1	3	15	27	6
Row 2	14	25	2	10

Figure 7.5 Each array element is identified by its row and column position

Declaring and Processing Two-Dimensional Arrays (cont’d)

- Two-dimensional arrays can be initialized in the declaration by listing values within braces, separated by commas

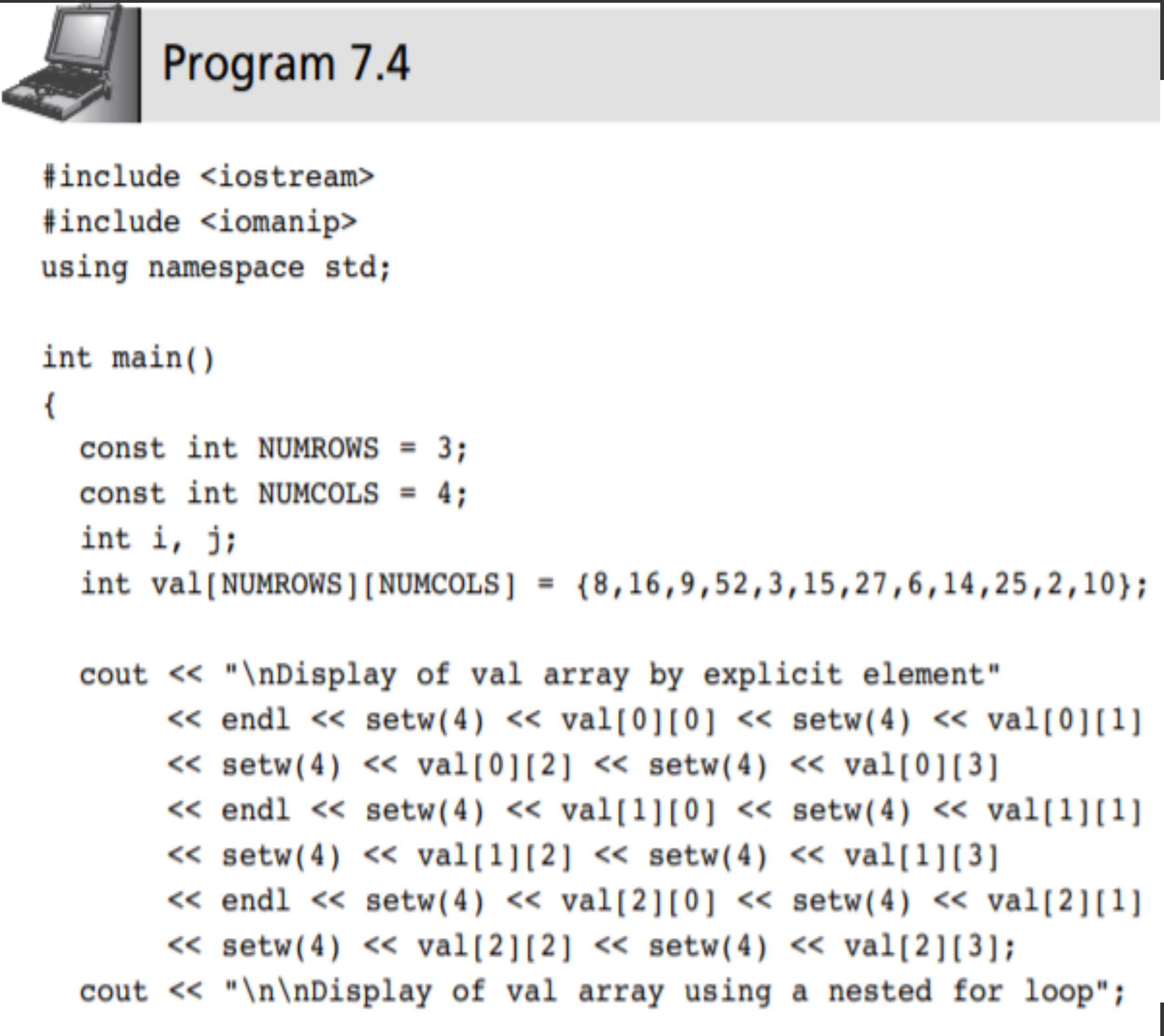


```
int val[3][4] = { {8,16,9,52},
                  {3,15,27,6},
                 {14,25,2,10} };
```

- Braces can be used to distinguish rows, but are not required
- Nested **`for`** loops are used to process two-dimensional arrays
  - Outer loop controls the rows
  - Inner loop controls the columns

Refer to pages 400-401 for more explanations and examples

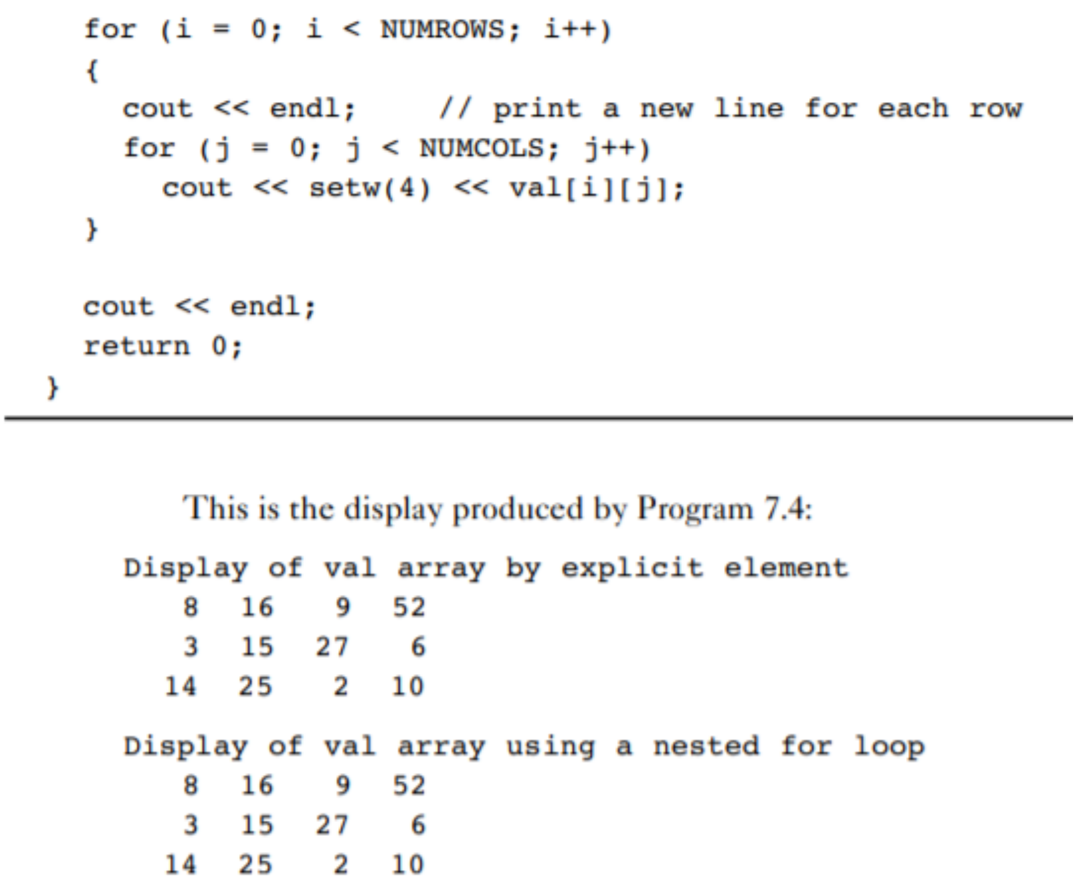
Declaring and Processing Two-Dimensional Arrays (cont'd)



Program 7.4

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int NUMROWS = 3;
  const int NUMCOLS = 4;
  int i, j;
  int val[NUMROWS][NUMCOLS] = {8,16,9,52,3,15,27,6,14,25,2,10};

  cout << "\nDisplay of val array by explicit element"
       << endl << setw(4) << val[0][0] << setw(4) << val[0][1]
       << setw(4) << val[0][2] << setw(4) << val[0][3]
       << endl << setw(4) << val[1][0] << setw(4) << val[1][1]
       << setw(4) << val[1][2] << setw(4) << val[1][3]
       << endl << setw(4) << val[2][0] << setw(4) << val[2][1]
       << setw(4) << val[2][2] << setw(4) << val[2][3];
  cout << "\n\nDisplay of val array using a nested for loop";
```

Declaring and Processing Two-Dimensional Arrays (cont'd)

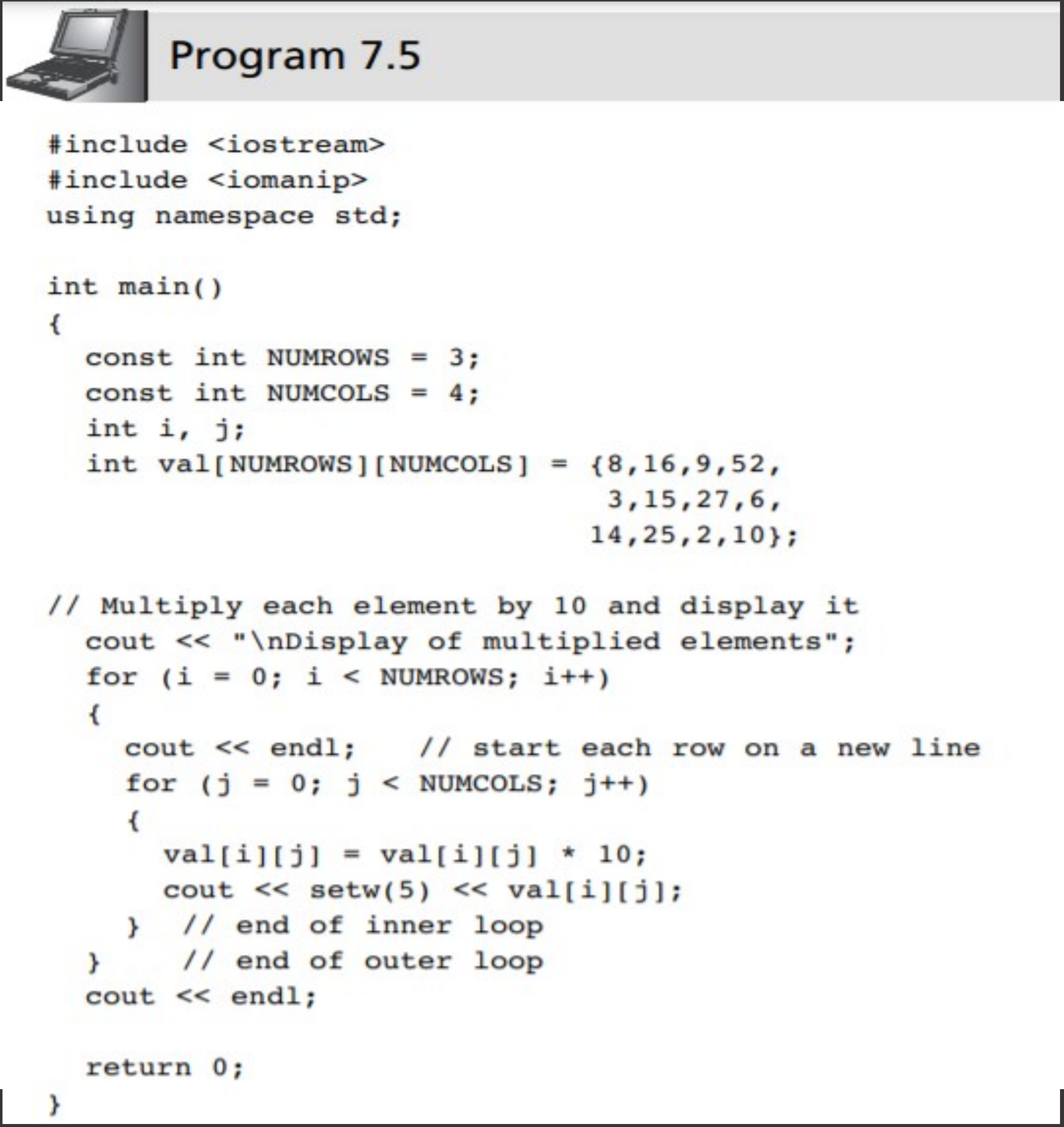


```
  for (i = 0; i < NUMROWS; i++)
  {
    cout << endl;     // print a new line for each row
    for (j = 0; j < NUMCOLS; j++)
      cout << setw(4) << val[i][j];
  }

  cout << endl;
  return 0;
}
```

This is the display produced by Program 7.4:

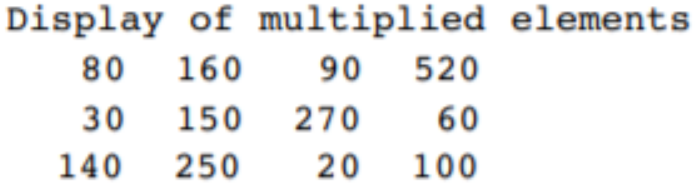
```
Display of val array by explicit element
   8  16   9  52
   3  15  27   6
  14  25   2  10

Display of val array using a nested for loop
   8  16   9  52
   3  15  27   6
  14  25   2  10
```

Declaring and Processing Two-Dimensional Arrays (cont'd)



Program 7.5

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int NUMROWS = 3;
  const int NUMCOLS = 4;
  int i, j;
  int val[NUMROWS][NUMCOLS] = {8,16,9,52,
                               3,15,27,6,
                              14,25,2,10};

// Multiply each element by 10 and display it
  cout << "\nDisplay of multiplied elements";
  for (i = 0; i < NUMROWS; i++)
  {
    cout << endl;   // start each row on a new line
    for (j = 0; j < NUMCOLS; j++)
    {
      val[i][j] = val[i][j] * 10;
      cout << setw(5) << val[i][j];
    }  // end of inner loop
  }    // end of outer loop
  cout << endl;

  return 0;
}
```



```
Display of multiplied elements
   80  160   90  520
   30  150  270   60
  140  250   20  100
```

Refer to page 402 for more explanations and examples

Larger Dimensional Arrays

- Arrays with more than two dimensions can be created, but are not commonly used
- Think of a three-dimensional array as a book of data tables

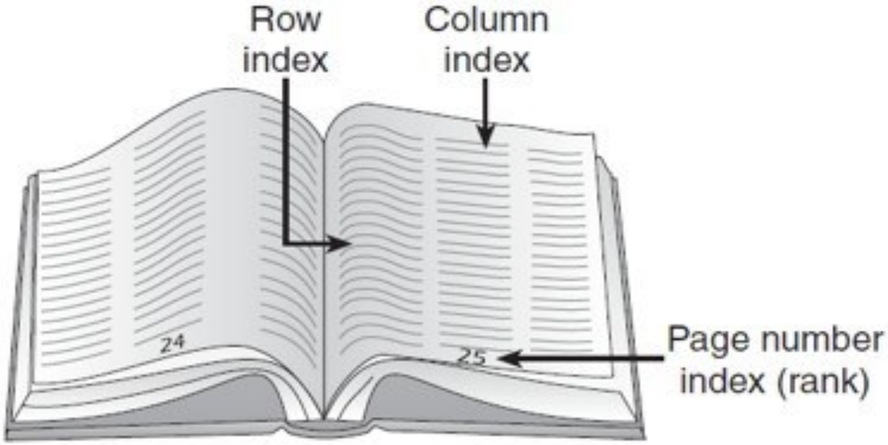
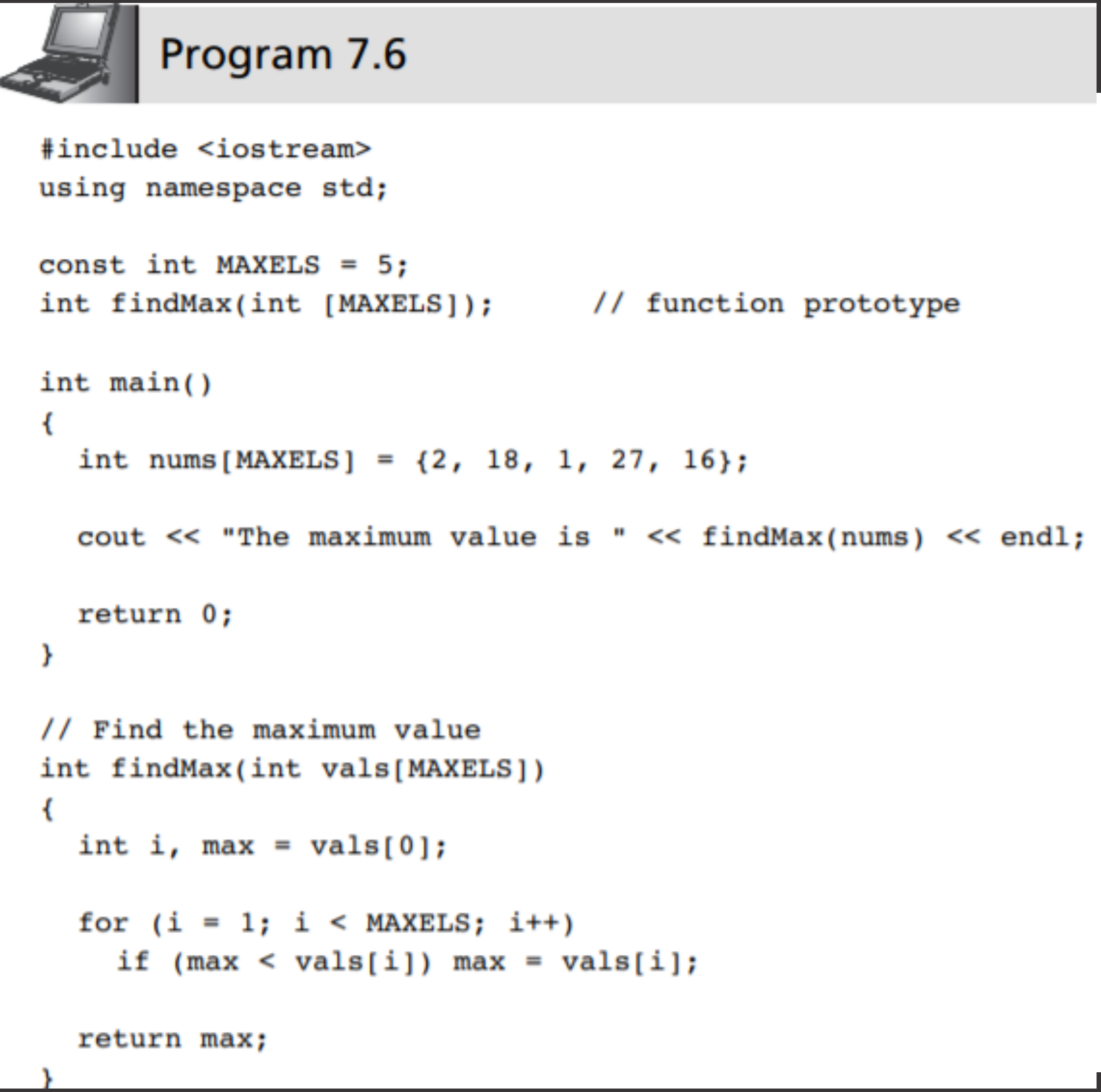


Figure 7.7 Representation of a three-dimensional array

Arrays as Arguments

- An individual array element can be passed as an argument just like any individual variable
- The called function receives a copy of the array element’s value
- Passing an entire array to a function causes the function to receive a reference to the array, not a copy of its element values
- The function must be declared with an array as the argument
- Single element of array is obtained by adding an offset to the array’s starting location

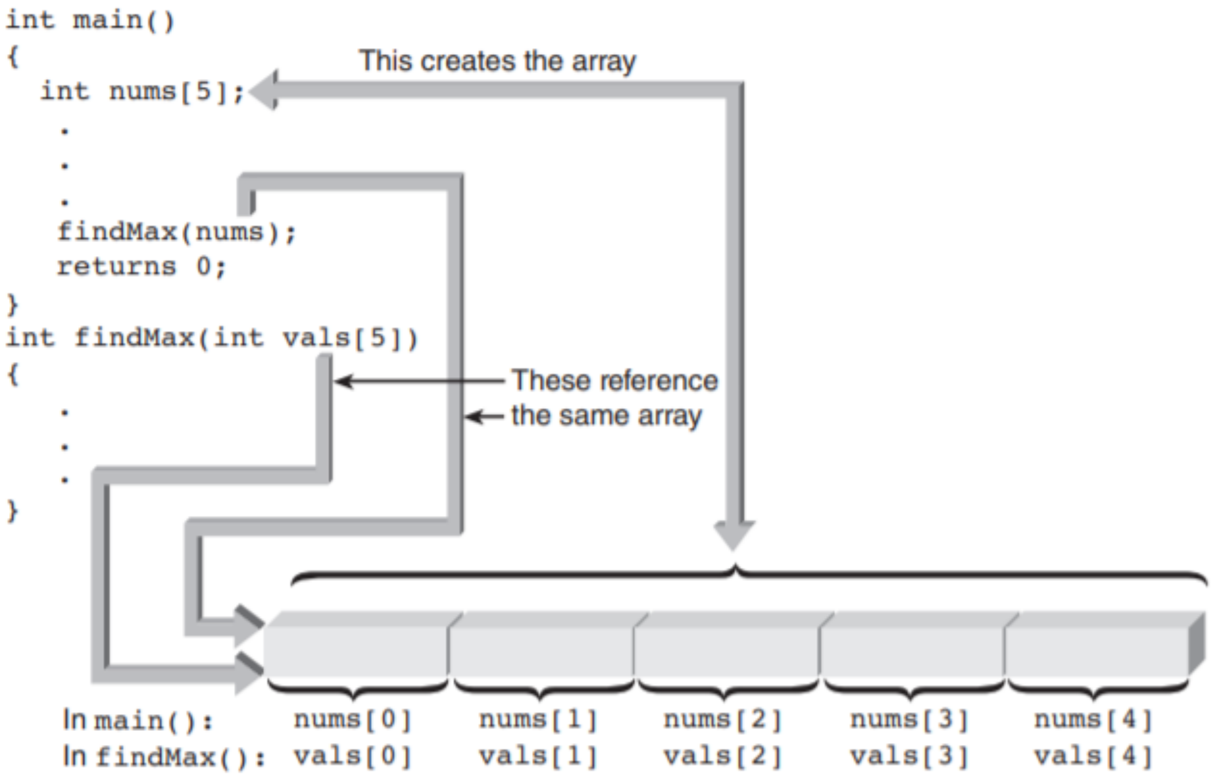
Arrays as Arguments



Program 7.6

```
#include <iostream>
using namespace std;

const int MAXELS = 5;
int findMax(int [MAXELS]);      // function prototype

int main()
{
  int nums[MAXELS] = {2, 18, 1, 27, 16};

  cout << "The maximum value is " << findMax(nums) << endl;

  return 0;
}

// Find the maximum value
int findMax(int vals[MAXELS])
{
  int i, max = vals[0];

  for (i = 1; i < MAXELS; i++)
    if (max < vals[i]) max = vals[i];

  return max;
}
```

Arrays as Arguments



```
int main()
{
  int nums[5];
   .
   .
   .
   findMax(nums);
   returns 0;
}
int findMax(int vals[5])
{
   .
   .
   .
}
```

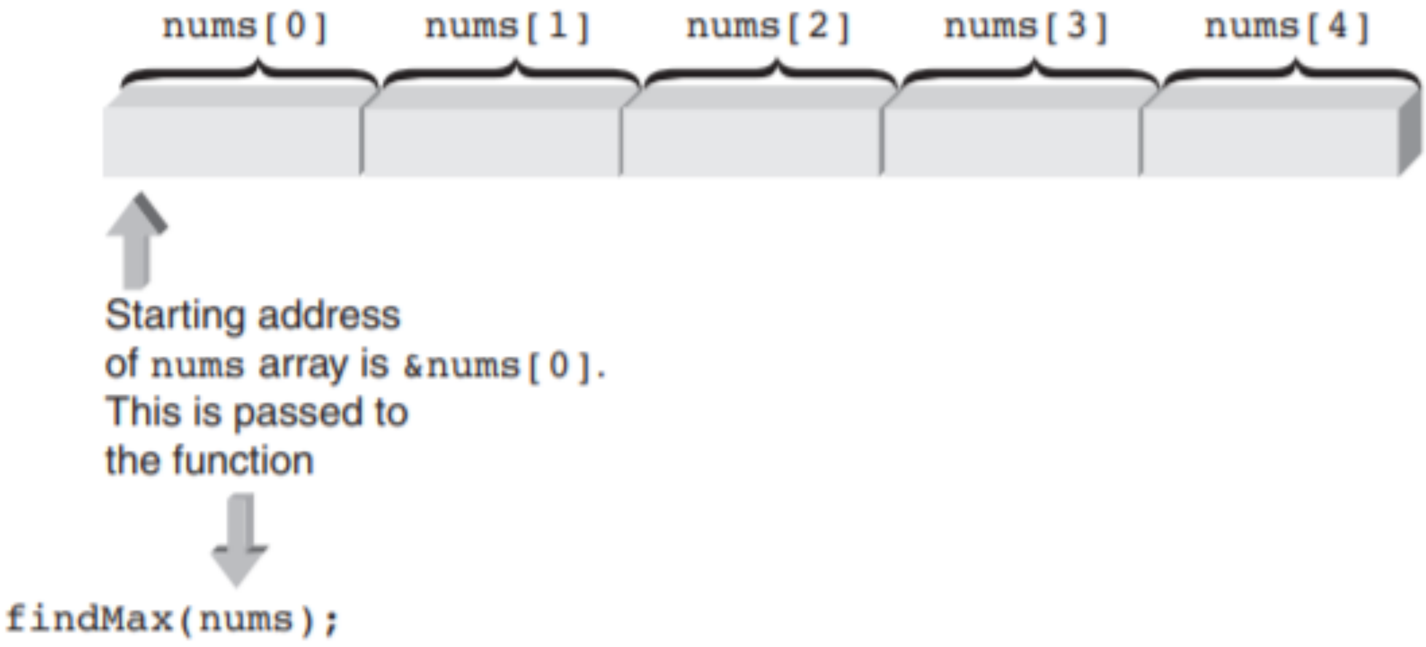
This creates the array

These reference the same array

In main(): nums[0] nums[1] nums[2] nums[3] nums[4]
In findMax(): vals[0] vals[1] vals[2] vals[3] vals[4]

Figure 7.8 Only one array is created

Arrays as Arguments

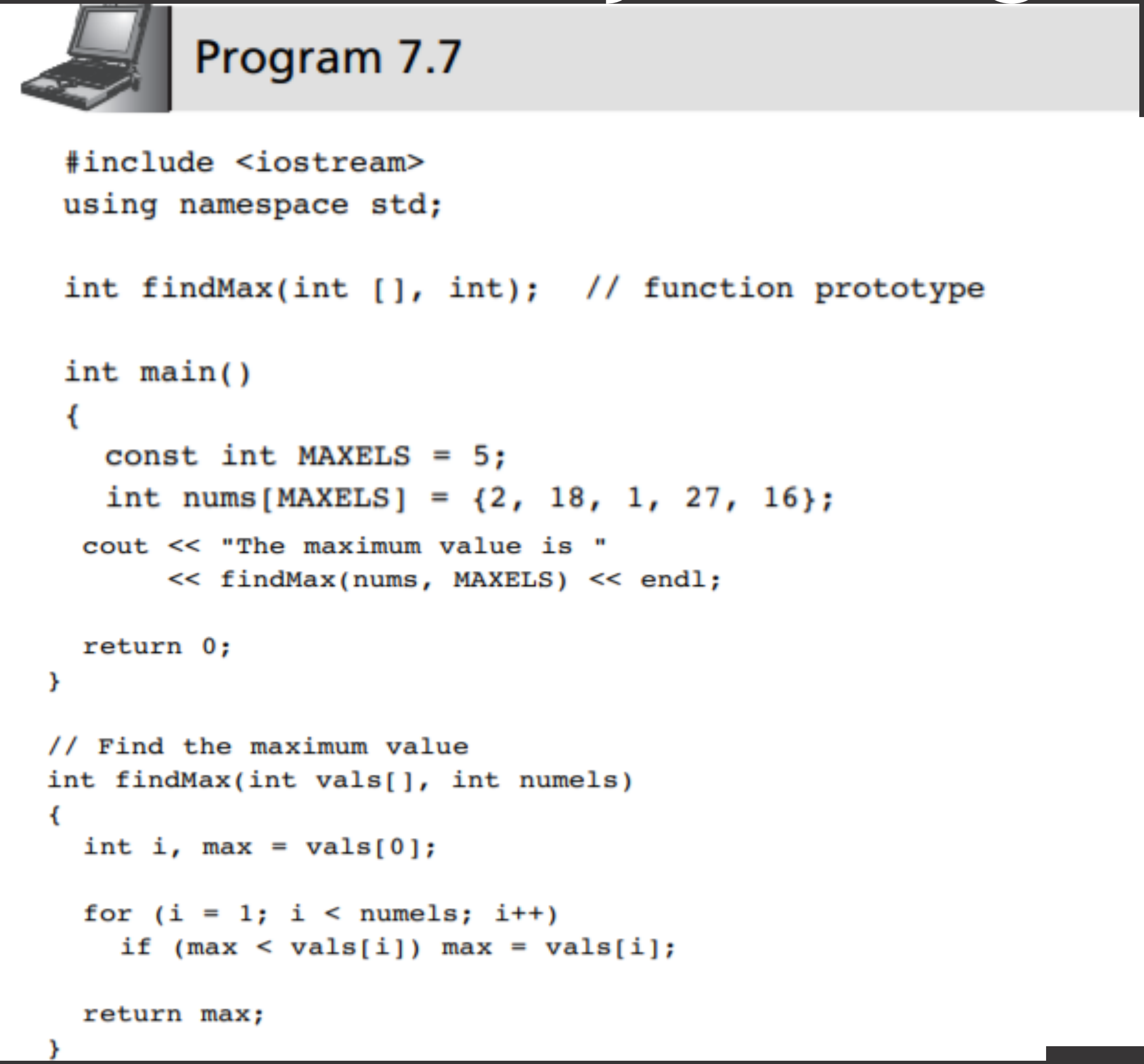


Starting address of nums array is &nums[0]. This is passed to the function

```
findMax(nums);
```

Figure 7.9 The array's starting address is passed

Arrays as Arguments



Program 7.7

```
#include <iostream>
using namespace std;

int findMax(int [], int);   // function prototype

int main()
{
  const int MAXELS = 5;
  int nums[MAXELS] = {2, 18, 1, 27, 16};
  cout << "The maximum value is "
       << findMax(nums, MAXELS) << endl;

  return 0;
}

// Find the maximum value
int findMax(int vals[], int numels)
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
    if (max < vals[i]) max = vals[i];

  return max;
}
```

Refer to page 409 for more explanations and examples

Arrays as Arguments (continued)

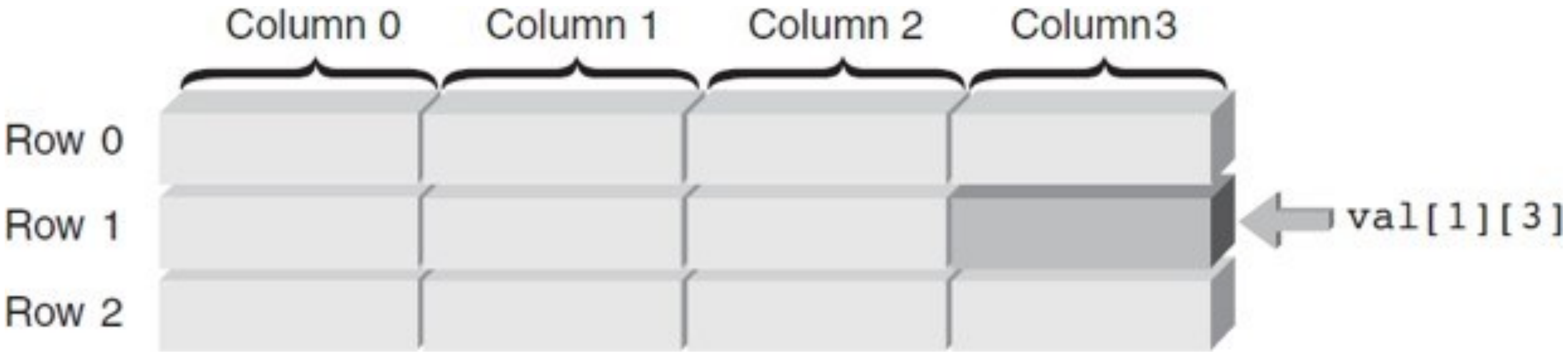
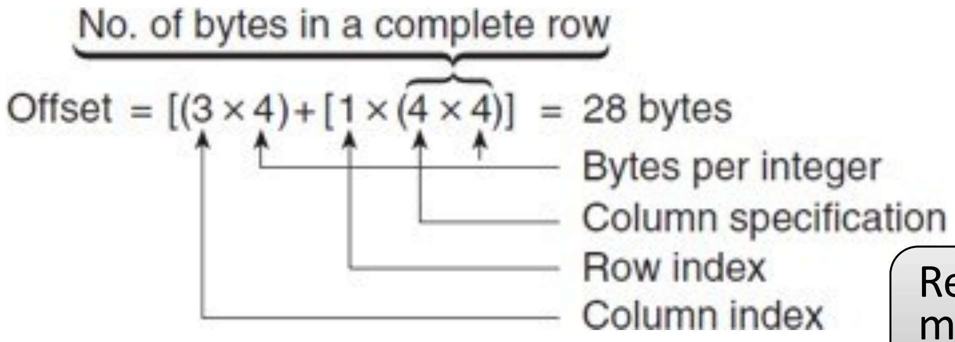


Figure 7.10 Storage of the `val` array



No. of bytes in a complete row

Offset = [(3 × 4)+[1 × (4 × 4)] = 28 bytes

Bytes per integer

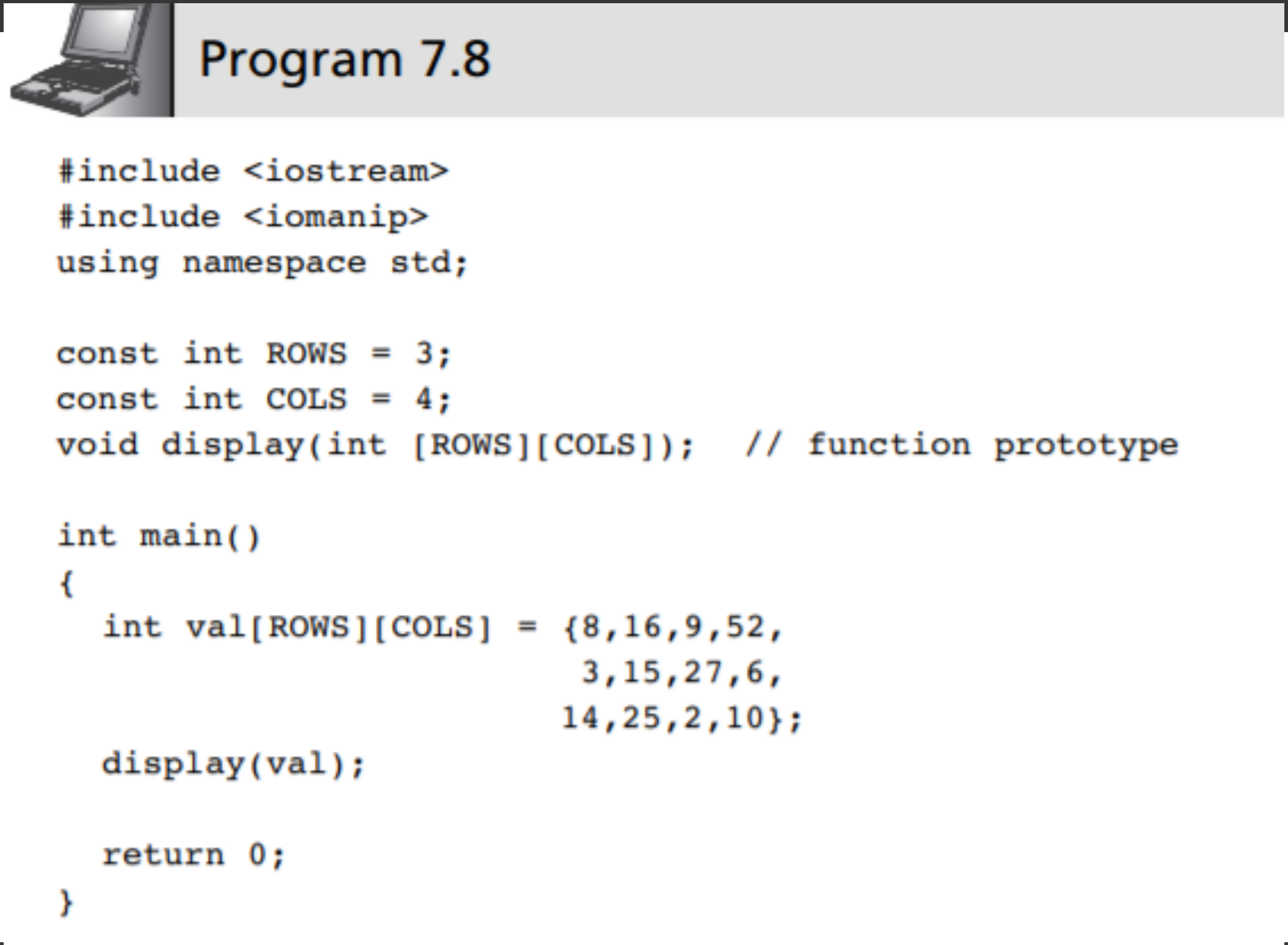
Column specification

Row index

Column index

Refer to page 412 for more explanations and examples

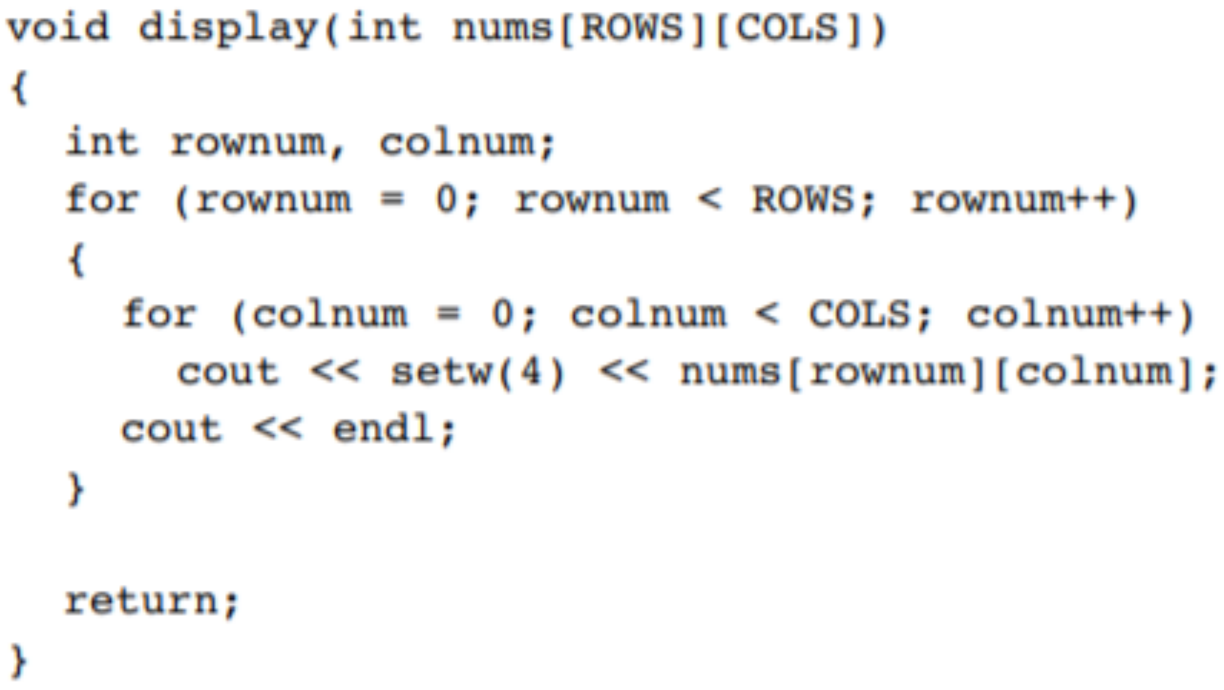
Arrays as Arguments (continued)



Program 7.8

```
#include <iostream>
#include <iomanip>
using namespace std;

const int ROWS = 3;
const int COLS = 4;
void display(int [ROWS][COLS]);  // function prototype

int main()
{
  int val[ROWS][COLS] = {8,16,9,52,
                         3,15,27,6,
                        14,25,2,10};
  display(val);

  return 0;
}
```

Arrays as Arguments (continued)



```
void display(int nums[ROWS][COLS])
{
  int rownum, colnum;
  for (rownum = 0; rownum < ROWS; rownum++)
  {
    for (colnum = 0; colnum < COLS; colnum++)
      cout << setw(4) << nums[rownum][colnum];
    cout << endl;
  }

  return;
}
```

Internal Array Element Location Algorithm

- Each element of an array is obtained by adding an offset to the starting address of the array:
  - *Address of element i = starting array address + the offset*
- Offset for one dimensional arrays:
  - *Offset = i * the size of the element*
- Offset for two dimensional arrays:
  - *Offset = column index value * the size of an element + row index value * number of bytes in a complete row*

Internal Array Element Location Algorithm (continued)

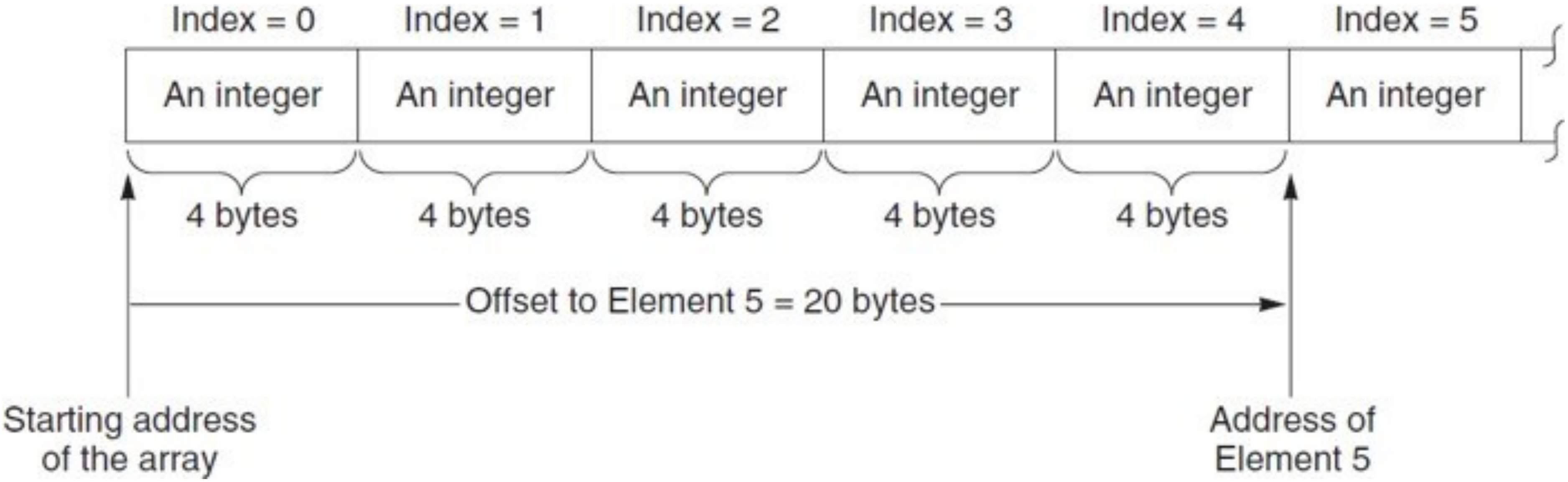
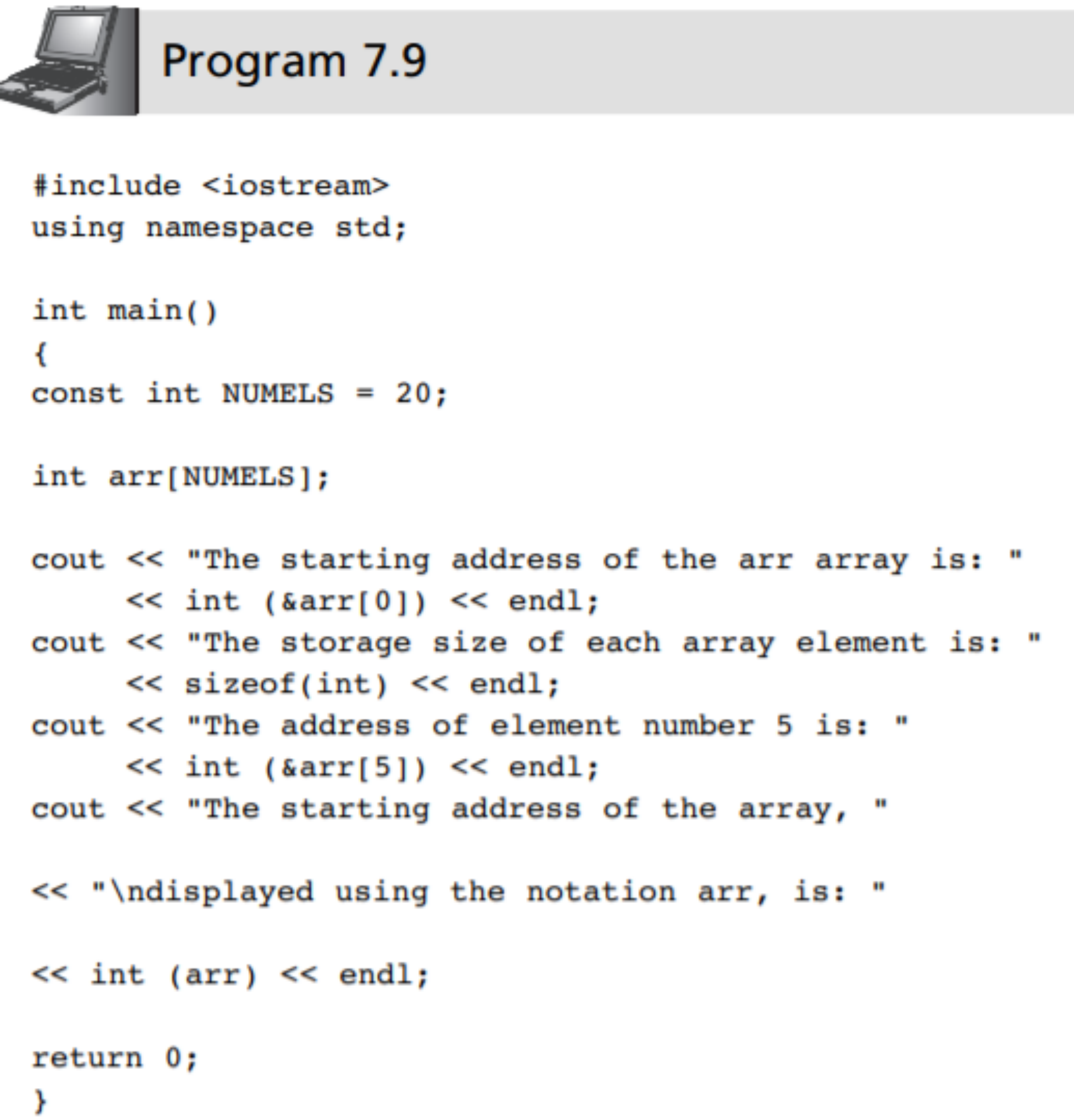


Figure 7.11 The offset to the element with an index value of 5

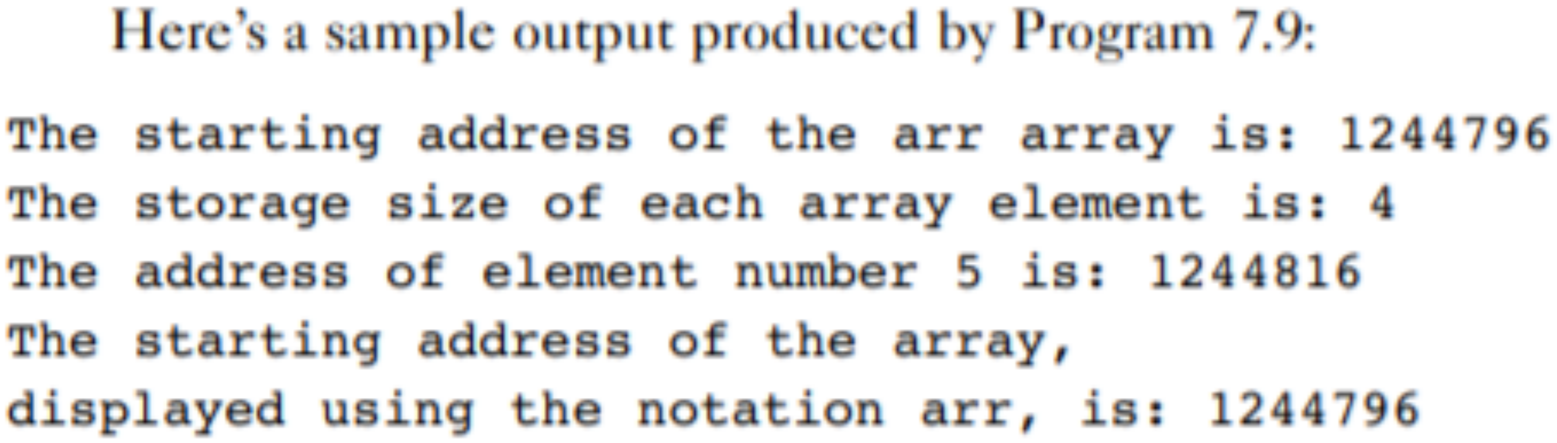
Internal Array Element Location Algorithm (continued)



Program 7.9

```
#include <iostream>
using namespace std;

int main()
{
const int NUMELS = 20;

int arr[NUMELS];

cout << "The starting address of the arr array is: "
     << int (&arr[0]) << endl;
cout << "The storage size of each array element is: "
     << sizeof(int) << endl;
cout << "The address of element number 5 is: "
     << int (&arr[5]) << endl;
cout << "The starting address of the array, "

<< "\ndisplayed using the notation arr, is: "

<< int (arr) << endl;

return 0;
}
```

Internal Array Element Location Algorithm (continued)



Here's a sample output produced by Program 7.9:

```
The starting address of the arr array is: 1244796
The storage size of each array element is: 4
The address of element number 5 is: 1244816
The starting address of the array,
displayed using the notation arr, is: 1244796
```

Case Studies

- Arrays are useful in applications that require multiple passes through the same set of data elements
  - Case Study 1: Statistical Analysis
  - Case Study 2: Curve Plotting
- Use the four step method to implement these problems

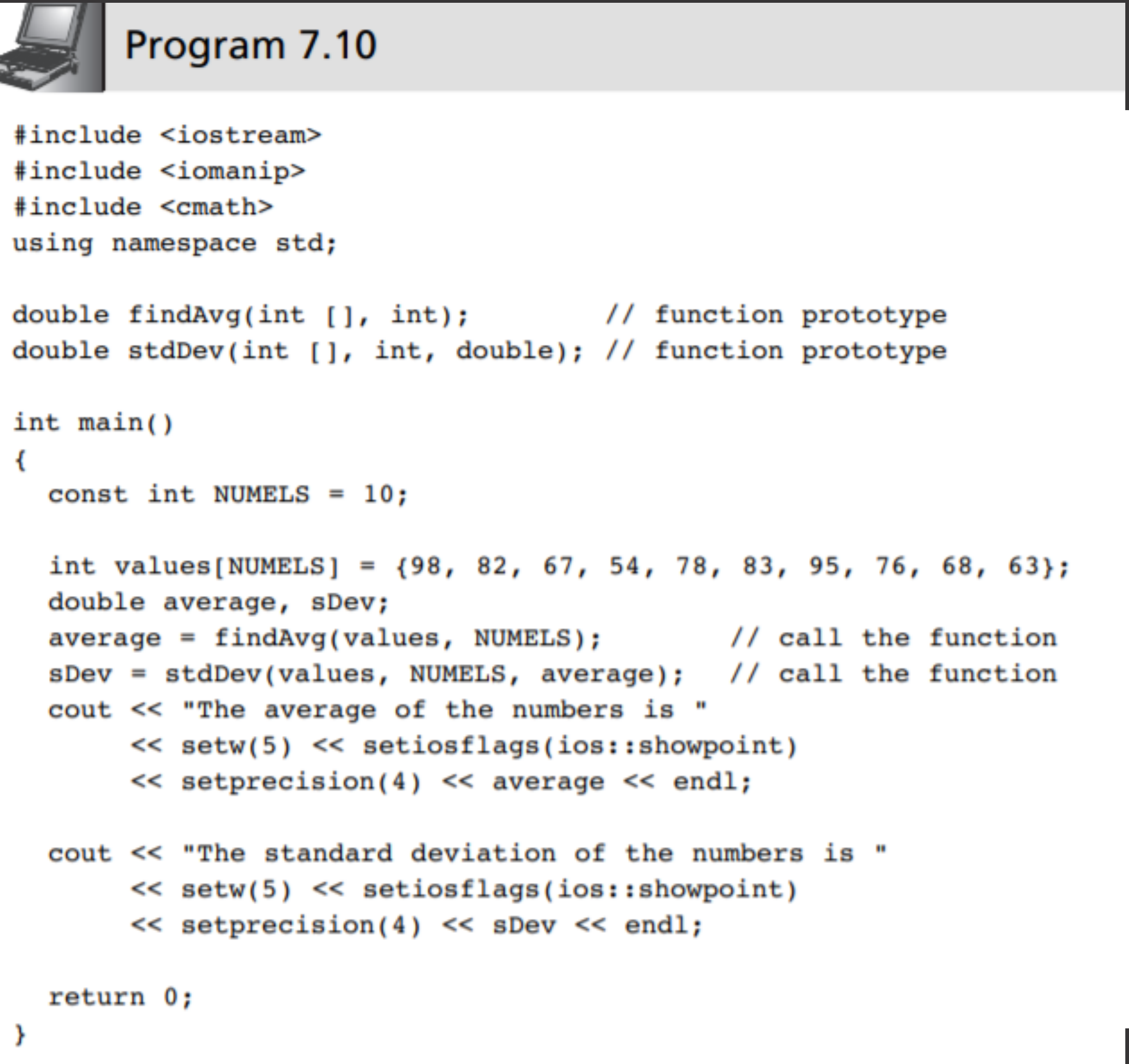
Case Studies

– Case Study 1: Statistical Analysis

A program is to be developed that accepts a list of a maximum of 100 numbers as input, determines both the average and standard deviation of the numbers, and then displays the result.

- Step 1: Analyze the Problem
- Step 2: Develop a Solution
- Step 3: Code the Solution
- Step 4: Test and Correct the Program

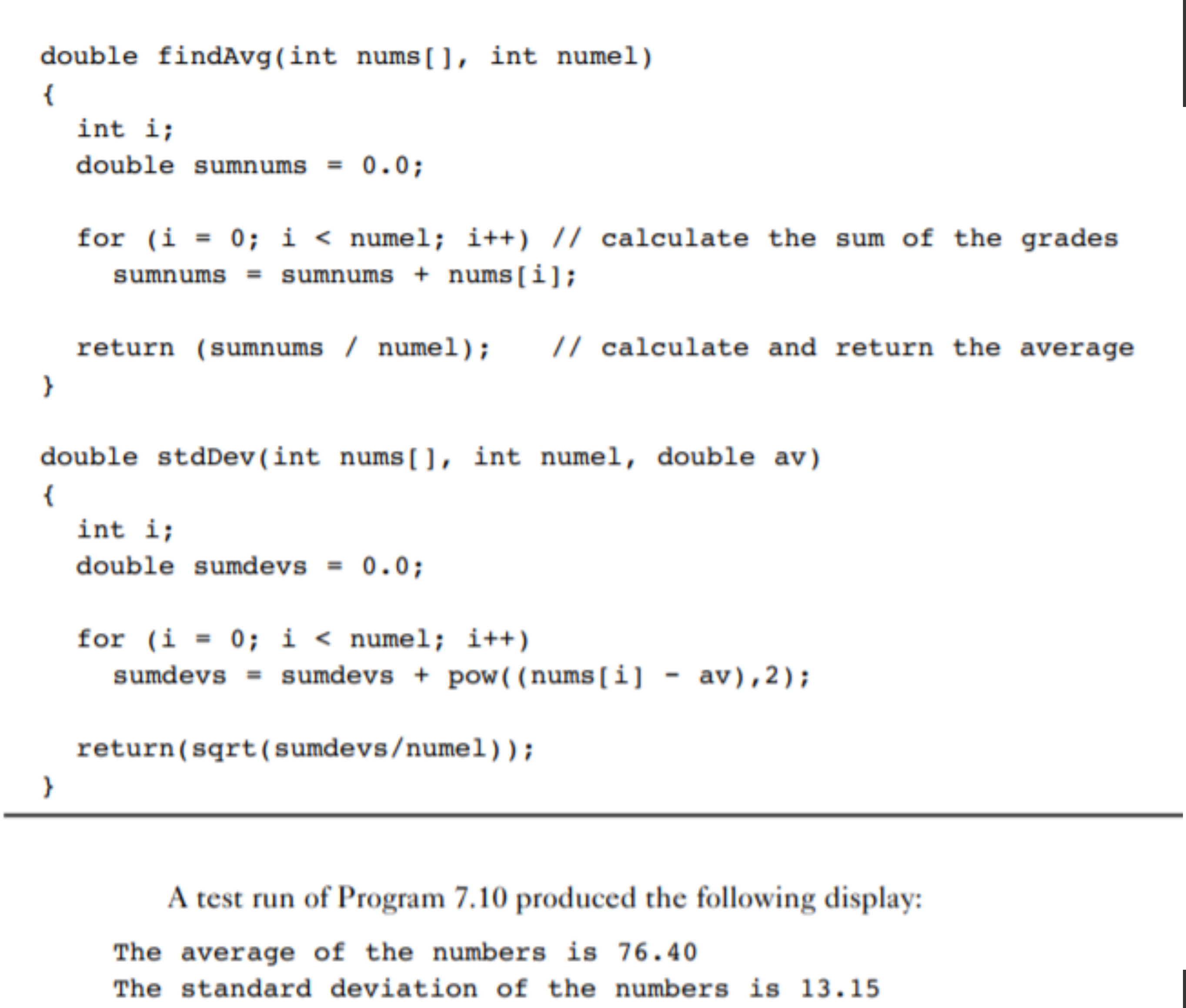
Case Studies



Program 7.10

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

double findAvg(int [], int);         // function prototype
double stdDev(int [], int, double); // function prototype

int main()
{
  const int NUMELS = 10;

  int values[NUMELS] = {98, 82, 67, 54, 78, 83, 95, 76, 68, 63};
  double average, sDev;
  average = findAvg(values, NUMELS);        // call the function
  sDev = stdDev(values, NUMELS, average);   // call the function
  cout << "The average of the numbers is "
       << setw(5) << setiosflags(ios::showpoint)
       << setprecision(4) << average << endl;

  cout << "The standard deviation of the numbers is "
       << setw(5) << setiosflags(ios::showpoint)
       << setprecision(4) << sDev << endl;

  return 0;
}
```

Case Studies



```
double findAvg(int nums[], int numel)
{
  int i;
  double sumnums = 0.0;

  for (i = 0; i < numel; i++) // calculate the sum of the grades
    sumnums = sumnums + nums[i];

  return (sumnums / numel);   // calculate and return the average
}

double stdDev(int nums[], int numel, double av)
{
  int i;
  double sumdevs = 0.0;

  for (i = 0; i < numel; i++)
    sumdevs = sumdevs + pow((nums[i] - av),2);

  return(sqrt(sumdevs/numel));
}
```

A test run of Program 7.10 produced the following display:

```
The average of the numbers is 76.40
The standard deviation of the numbers is 13.15
```

Case Studies

– Case Study 2: Curve Plotting

- Step 1: Store an asterisk in the desired array element
- Step 2: Display the array
- Step 3: Reset the asterisk element to a blank space
- Step 4: Repeat Steps 1 through 3 until the required number of lines have been displayed

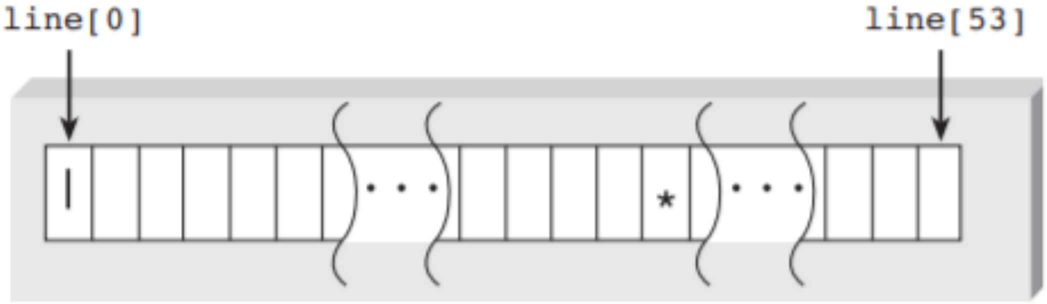
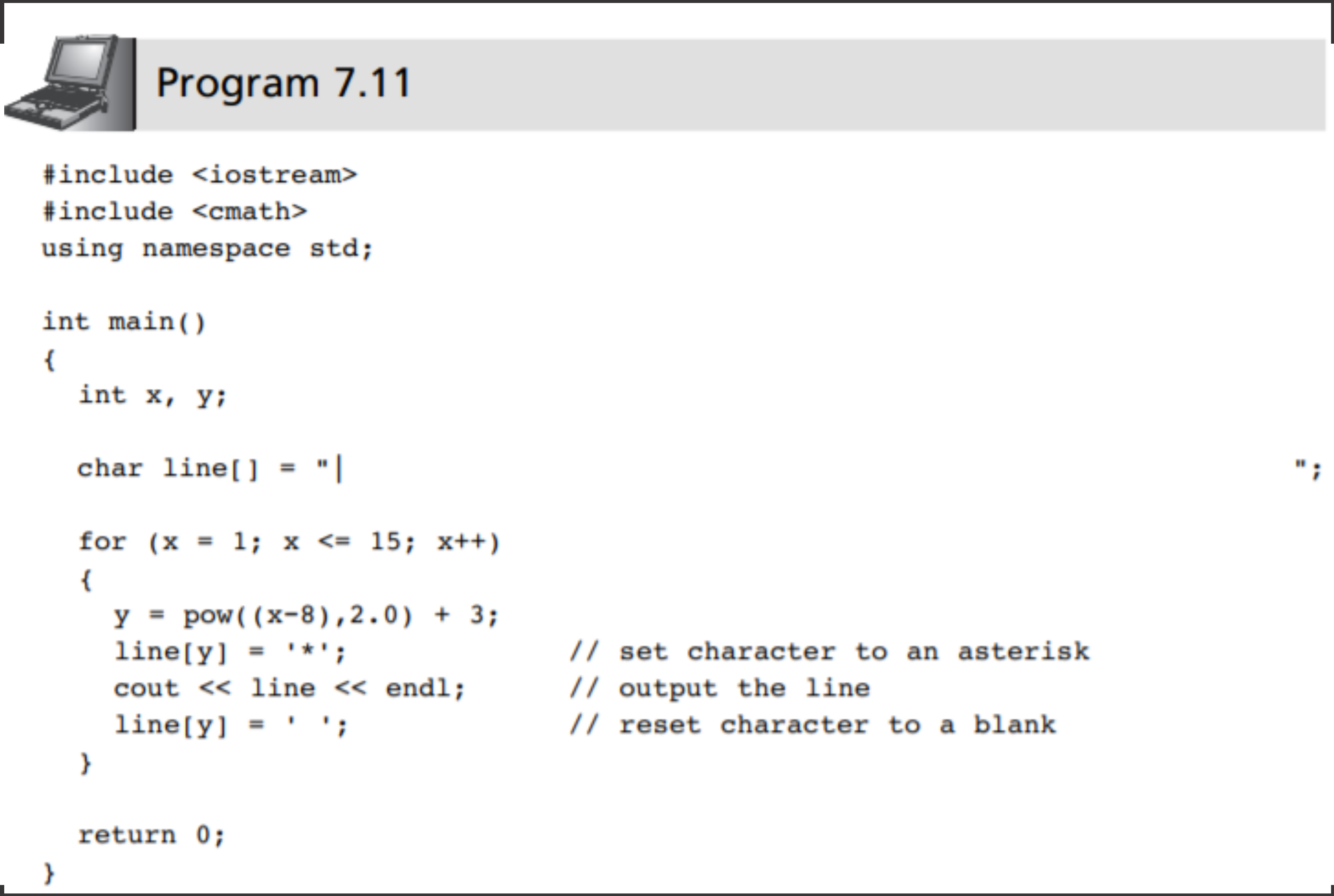


Figure 7.12 The line array

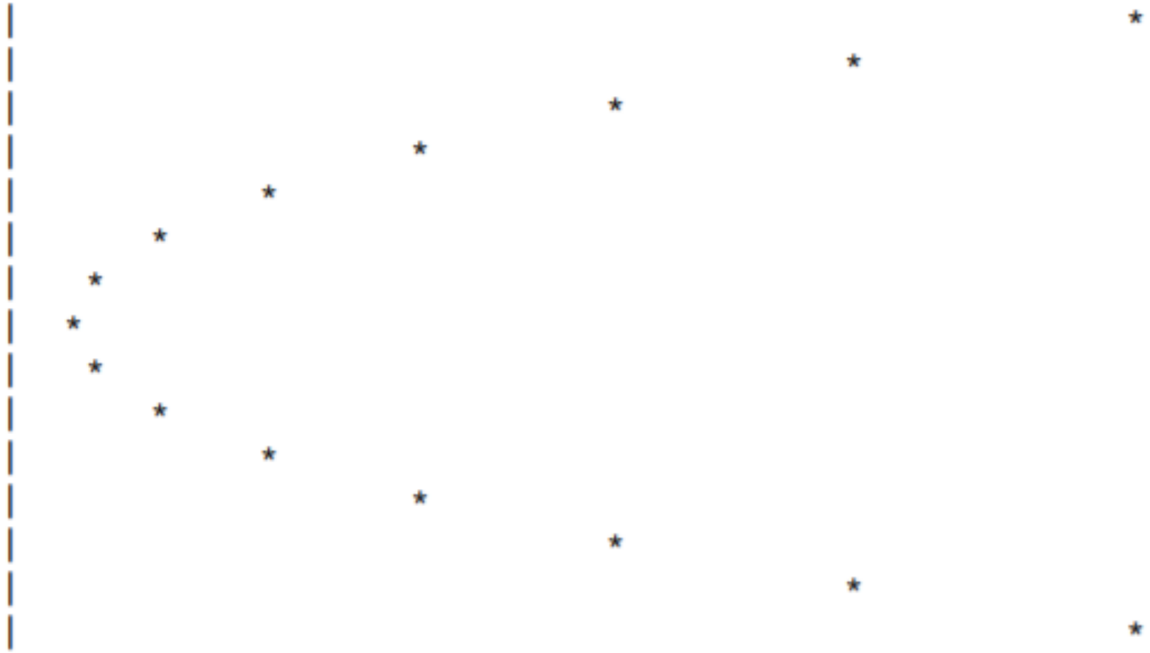
Case Studies



Program 7.11

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  int x, y;

  char line[] = "|                                                         ";

  for (x = 1; x <= 15; x++)
  {
    y = pow((x-8),2.0) + 3;
    line[y] = '*';              // set character to an asterisk
    cout << line << endl;       // output the line
    line[y] = ' ';              // reset character to a blank
  }

  return 0;
}
```

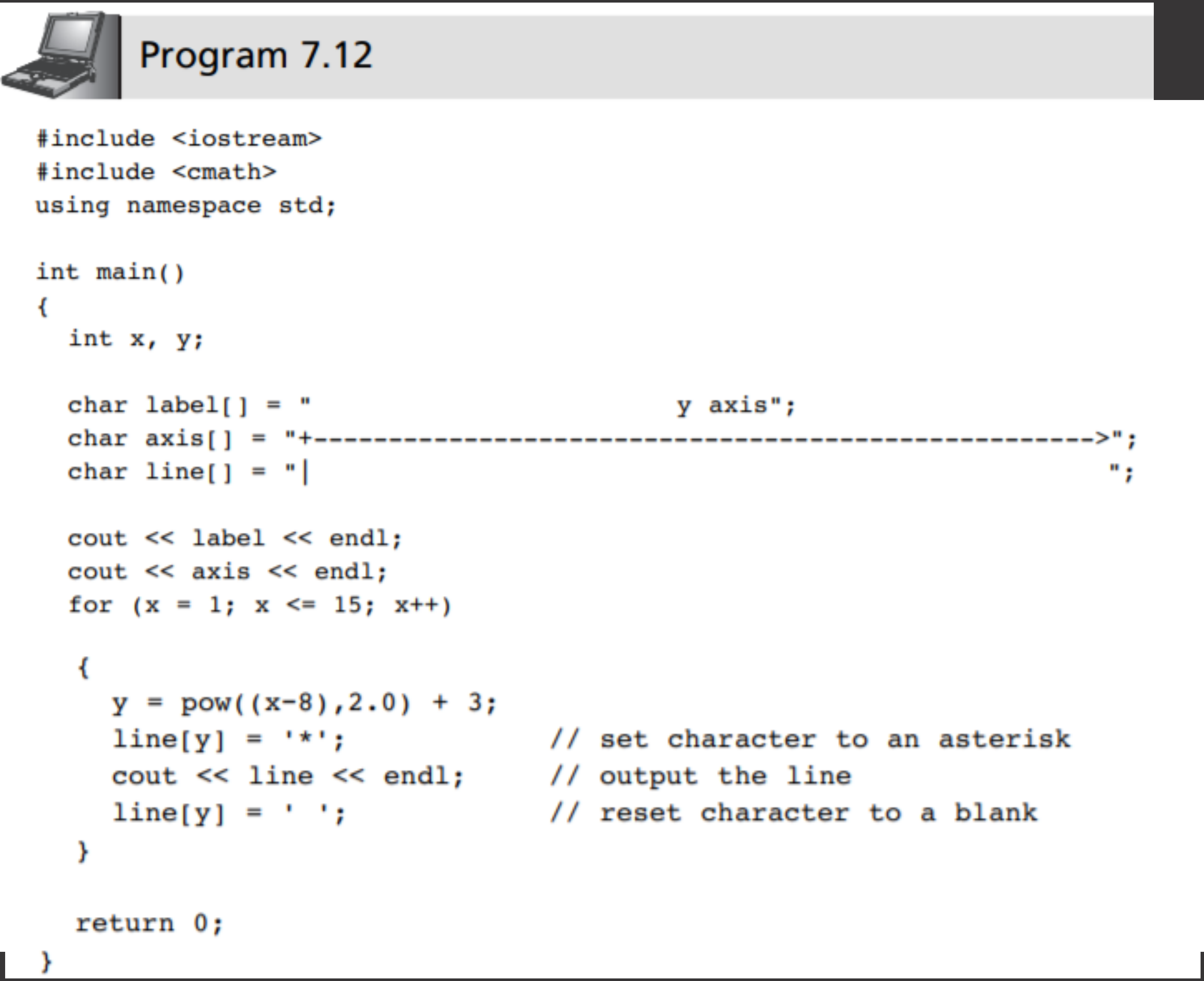
Case Studies



```
|                                                   *
|                                      *
|                           *
|                  *
|           *
|      *
|   *
|  *
|   *
|      *
|           *
|                  *
|                           *
|                                      *
|                                                   *
```

In reviewing Program 7.11, notice that a y-axis hasn't been explicitly included in the output. This minor omission is corrected in Program 7.12.

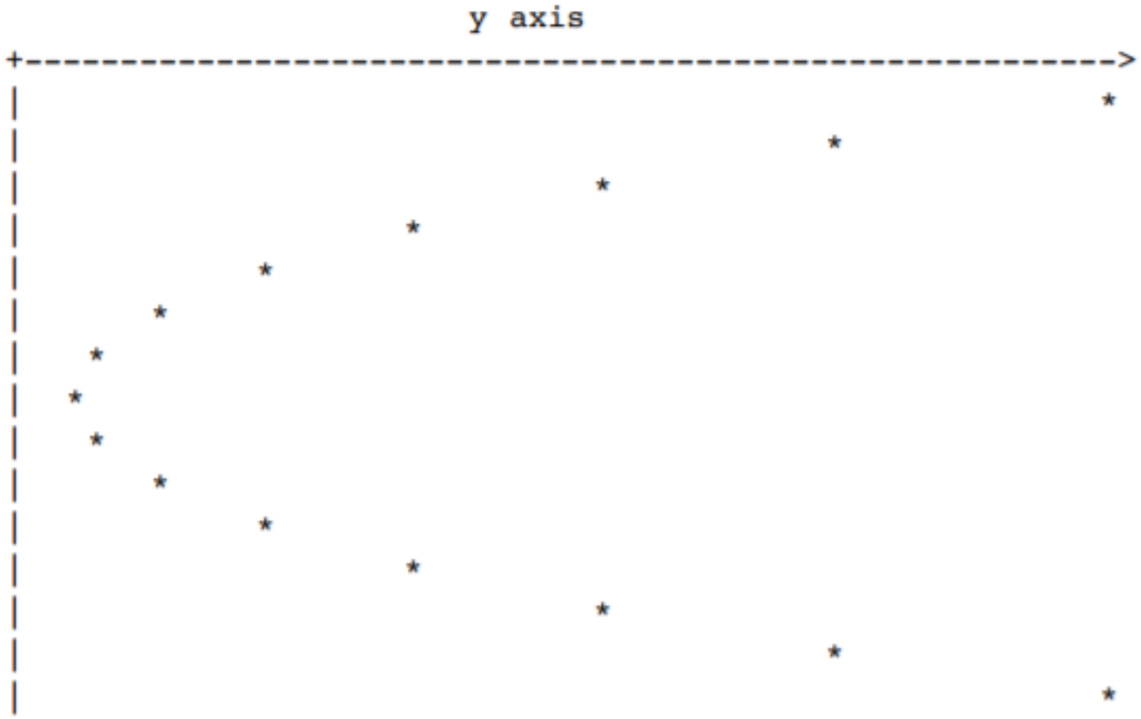
Case Studies



Program 7.12

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  int x, y;

  char label[] = "                        y axis";
  char axis[] = "+---------------------------------------------------->";
  char line[] = "|                                                     ";

  cout << label << endl;
  cout << axis << endl;
  for (x = 1; x <= 15; x++)

   {
     y = pow((x-8),2.0) + 3;
     line[y] = '*';            // set character to an asterisk
     cout << line << endl;     // output the line
     line[y] = ' ';            // reset character to a blank
   }

   return 0;
}
```

Case Studies



```
                      y axis
+---------------------------------------------------->
|                                                   *
|                                      *
|                           *
|                  *
|           *
|      *
|   *
|  *
|   *
|      *
|           *
|                  *
|                           *
|                                      *
|                                                   *
```

Program 7.12 is essentially the same as Program 7.11, with the addition of two array declarations for the y-axis and two cout statements to display the label and y-axis character arrays.

The Standard Template Library

- **Standard Template Library (STL):** Generic set of data structures that can be modified, expanded, and contracted

标准模板库

- Each STL class is coded as a template to permit the construction of a container

The Standard Template Library

- **Container:** A generic data structure, referring to a set of data items that form a natural group

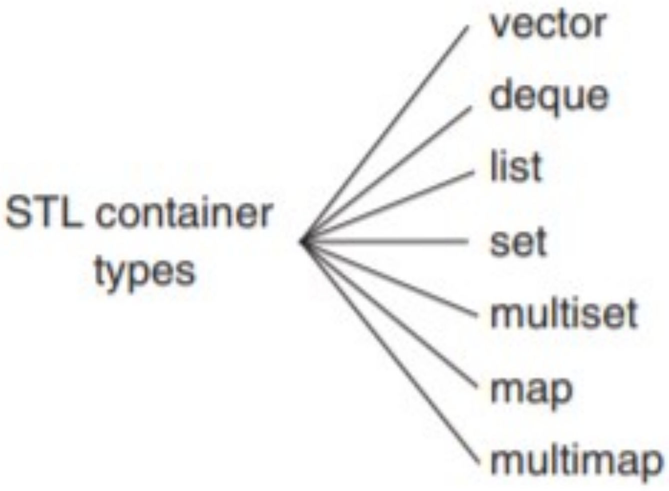


Figure 7.17 The collection of STL container types

The Standard Template Library

- **Vector:** Similar to an array
  - Uses a zero-relative index, but automatically expands as needed
- STL **`Vector`** class provides many useful methods (functions) for vector manipulation:
  - **`insert(pos, elem)`**: inserts **`elem`** at position **`pos`**
  - **`name.push_back(elem)`**: appends **`elem`** at the end of the vector
  - **`name.size`**: returns the size of the vector
- STL also provides generic functions called **algorithms**

The STL (continued)

- Must include the header files for **`vector`** and **`algorithm`**, with the namespace **`std`**
- Syntax:
  - To create and initialize a vector:

    ```
    vector<dataType> vectorName(start,end);
    ```
  - To modify a specific element:

    ```
    vectorName[index] = newValue;
    ```
  - To insert a new element:

    ```
    vectorName.insert(index, newValue);
    ```
- STL provides other containers, algorithms, and iterators

Refer to pages 433-439 for more explanations and examples

A Closer Look: Searching & Sorting

- Sorting: Arranging data in ascending or descending order for some purpose
- Searching: Scanning through a list of data to find a particular item

Search Algorithms

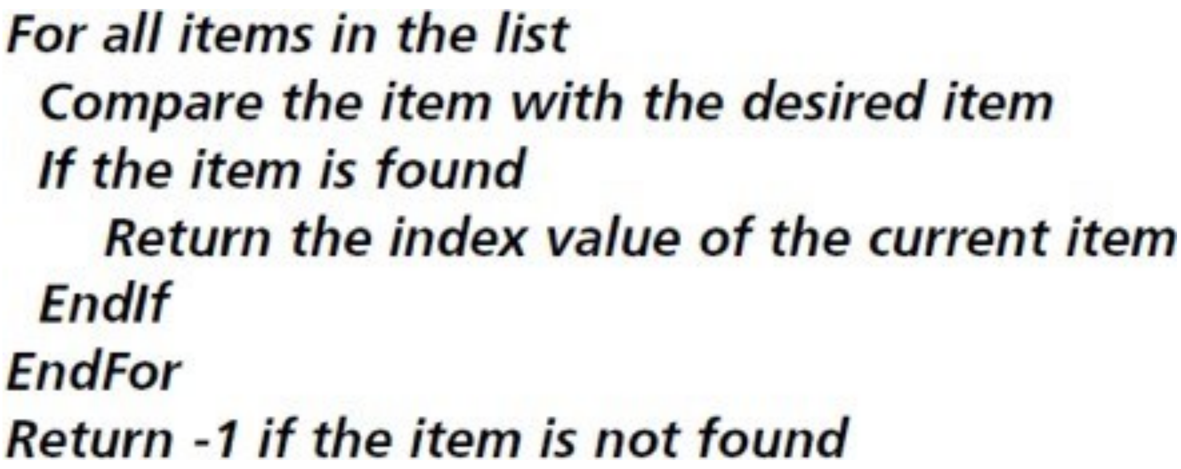
- Searches can be faster if the data is in sorted order
- Two common methods for searching:
  - Linear search
  - Binary search
- Linear search is a sequential search
  - Each item is examined in the order it occurs in the list
- Average number of comparisons required to find the desired item is $n/2$ for a list of n items

Linear Search

- Each item in the list is examined in the order in which it occurs
- Not a very efficient method for searching
- Advantage is that the list does not have to be in sorted order
- On average, the number of required comparisons is $n/2$, where *n* is the number of elements in the list

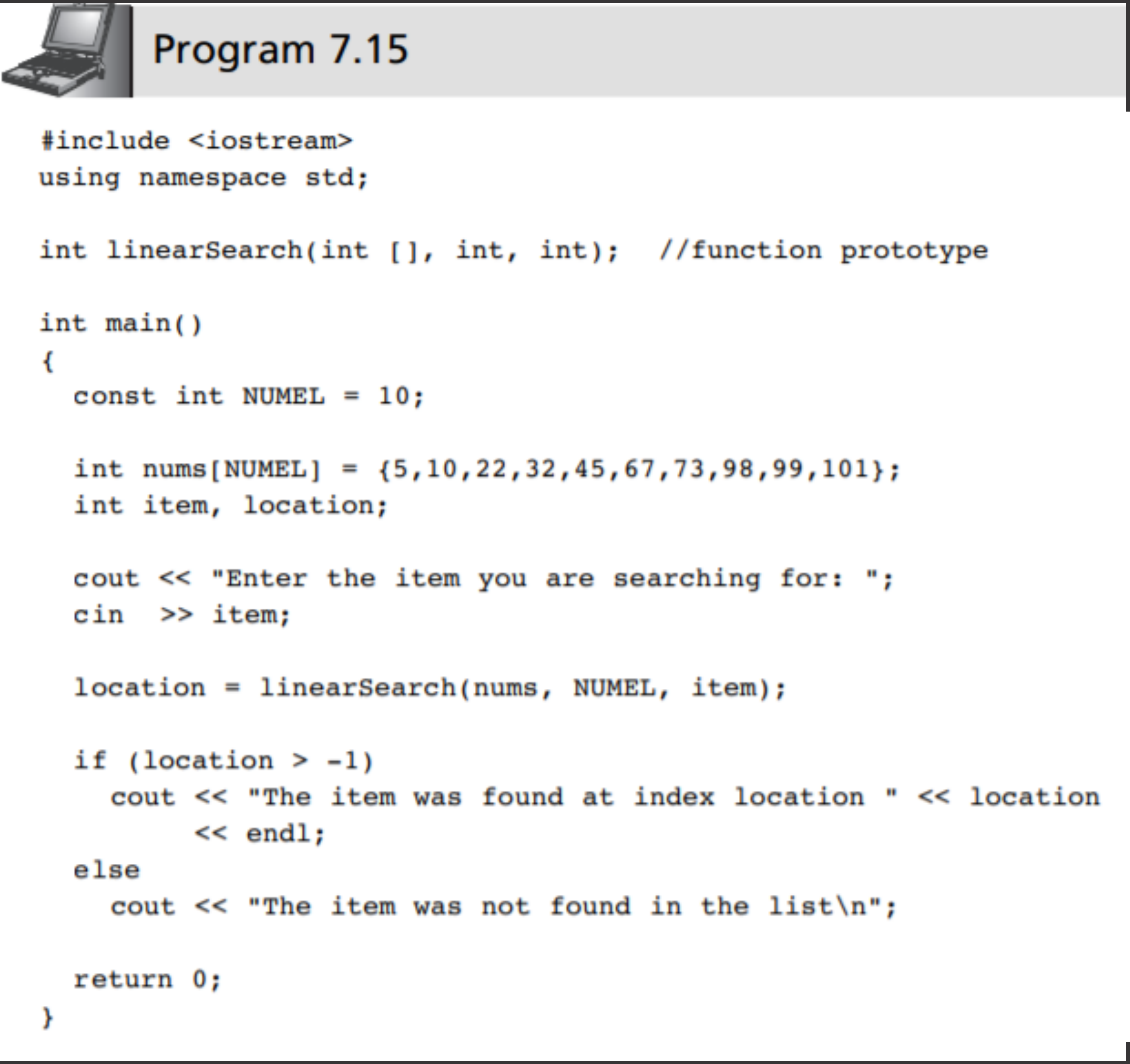
Linear Search (continued)

- Pseudocode for a linear search



```
For all items in the list
  Compare the item with the desired item
  If the item is found
      Return the index value of the current item
  EndIf
EndFor
Return -1 if the item is not found
```

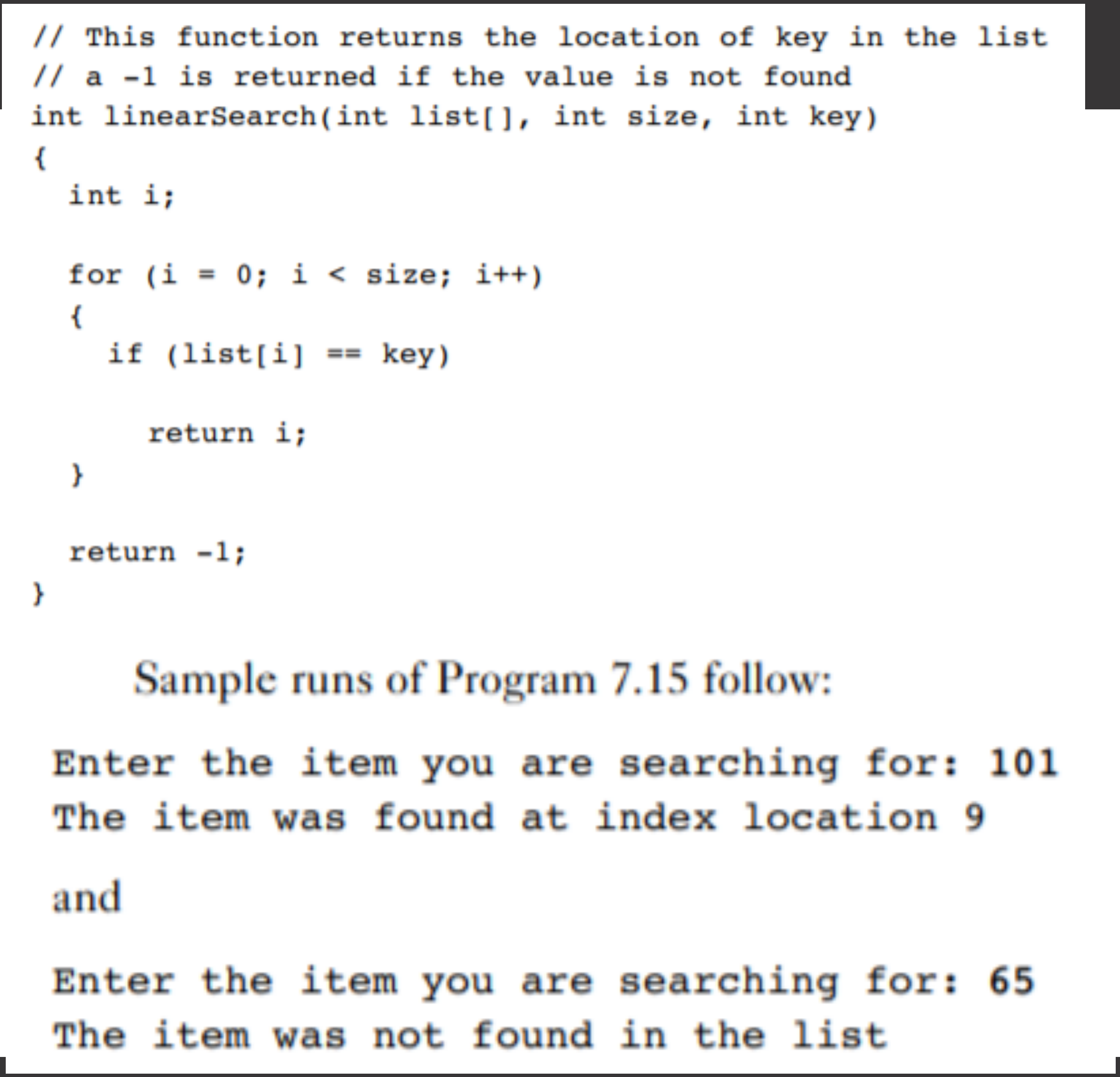
Linear Search (continued)



Program 7.15

```
#include <iostream>
using namespace std;

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

int main()
{
  const int NUMEL = 10;

  int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
  int item, location;

  cout << "Enter the item you are searching for: ";
  cin  >> item;

  location = linearSearch(nums, NUMEL, item);

  if (location > -1)
    cout << "The item was found at index location " << location
         << endl;
  else
    cout << "The item was not found in the list\n";

  return 0;
}
```

Linear Search (continued)



```
// This function returns the location of key in the list
// a -1 is returned if the value is not found
int linearSearch(int list[], int size, int key)
{
  int i;

  for (i = 0; i < size; i++)
  {
    if (list[i] == key)

      return i;
  }

  return -1;
}
```

Sample runs of Program 7.15 follow:

```
Enter the item you are searching for: 101
The item was found at index location 9
```

and

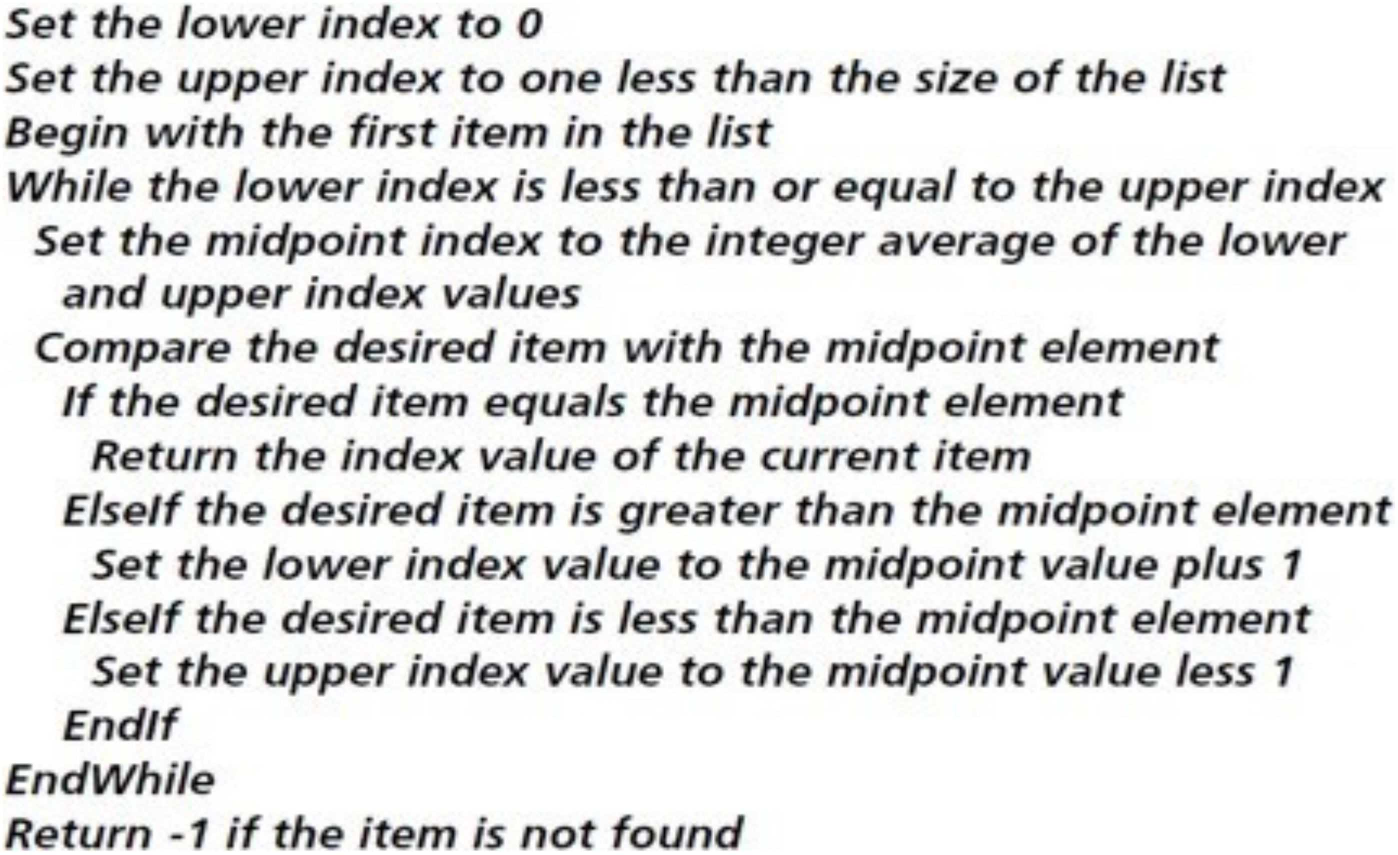
```
Enter the item you are searching for: 65
The item was not found in the list
```

Binary Search

- Binary search requires that the list is stored in sorted order
- Desired item is compared to the middle element, with three possible outcomes:
  - Desired element was found: finished
  - Desired element is greater than the middle element, so discard all elements below
  - Desired element is less than the middle element, so discard all elements above

Binary Search (continued)

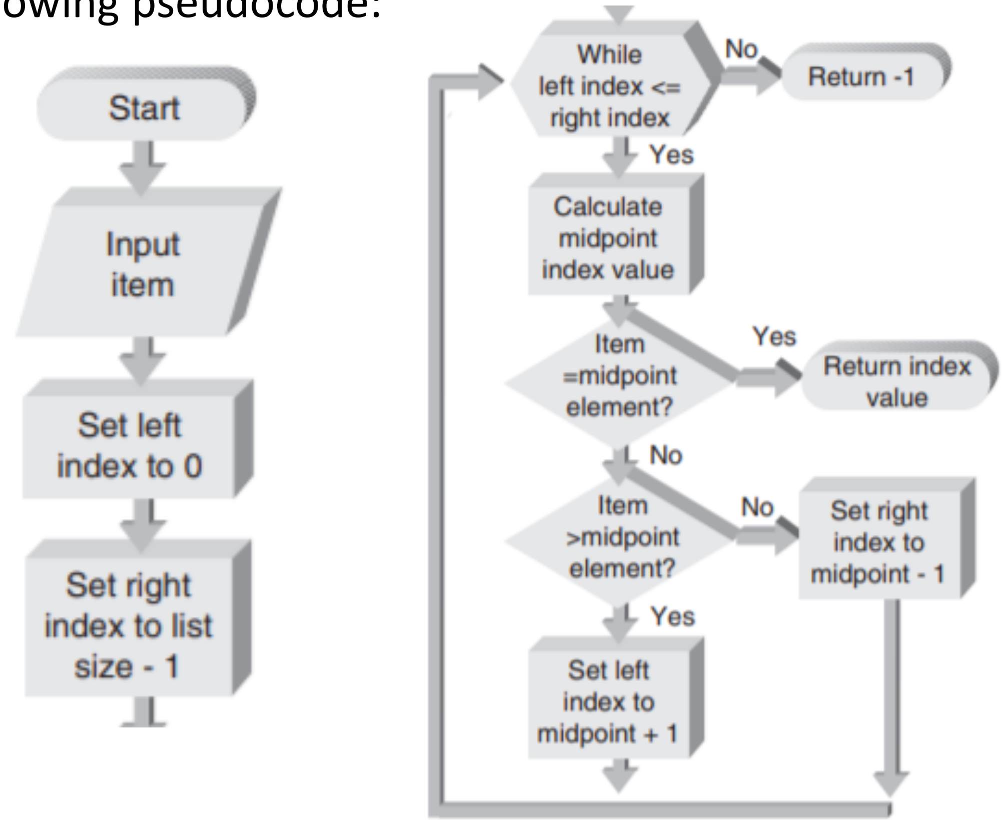
- Pseudocode for a binary search



```
Set the lower index to 0
Set the upper index to one less than the size of the list
Begin with the first item in the list
While the lower index is less than or equal to the upper index
  Set the midpoint index to the integer average of the lower
    and upper index values
  Compare the desired item with the midpoint element
    If the desired item equals the midpoint element
      Return the index value of the current item
    ElseIf the desired item is greater than the midpoint element
      Set the lower index value to the midpoint value plus 1
    ElseIf the desired item is less than the midpoint element
      Set the upper index value to the midpoint value less 1
    EndIf
EndWhile
Return -1 if the item is not found
```

Binary Search (continued)

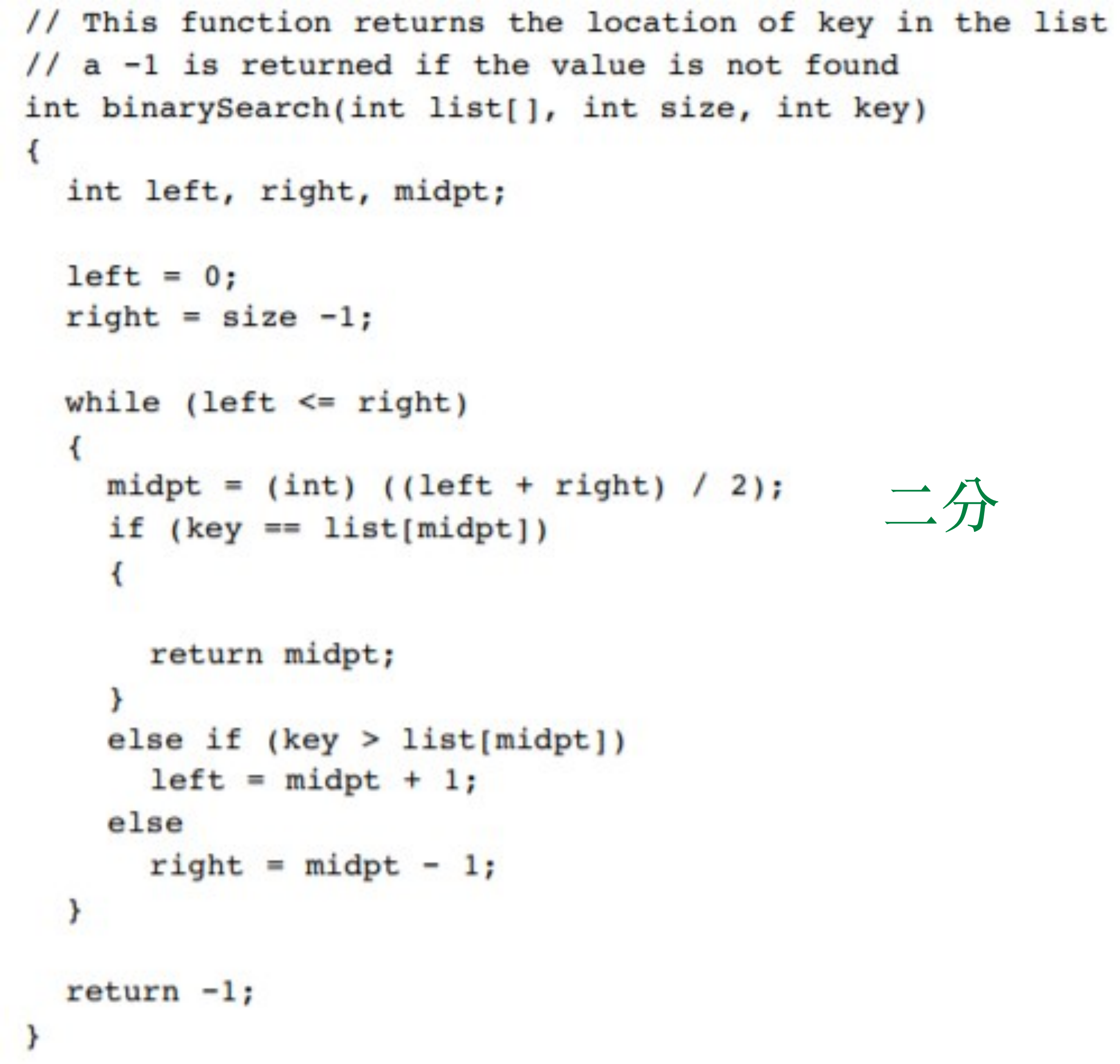
The algorithm for this search strategy is shown in Figure 7.18 and defined by the following pseudocode:



Start
Input item
Set left index to 0
Set right index to list size - 1
While left index <= right index
No
Return -1
Yes
Calculate midpoint index value
Item =midpoint element?
Yes
Return index value
No
Item >midpoint element?
No
Set right index to midpoint - 1
Yes
Set left index to midpoint + 1

Figure 7.18 The binary search algorithm

Binary Search (continued)



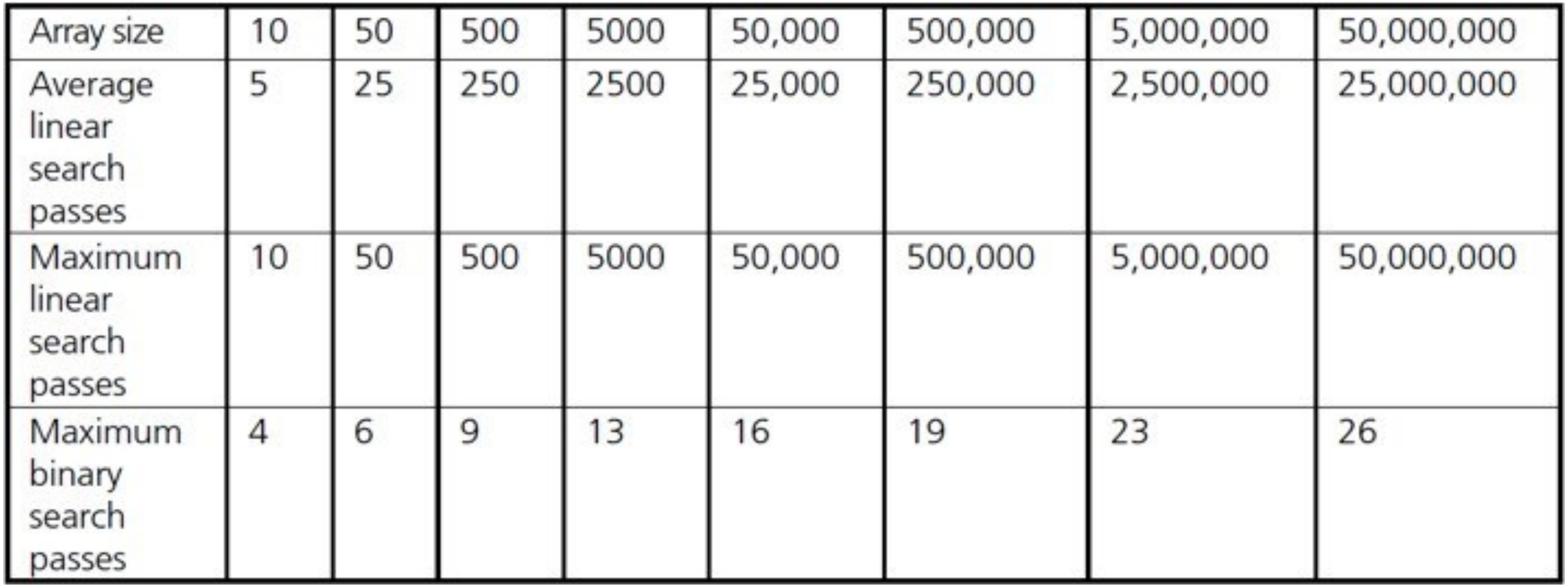
```
// This function returns the location of key in the list
// a -1 is returned if the value is not found
int binarySearch(int list[], int size, int key)
{
  int left, right, midpt;

  left = 0;
  right = size -1;

  while (left <= right)
  {
    midpt = (int) ((left + right) / 2);
    if (key == list[midpt])
    {

      return midpt;
    }
    else if (key > list[midpt])
      left = midpt + 1;
    else
      right = midpt - 1;
  }

  return -1;
}
```

二分

Binary Search (continued)

- On each pass of binary search, the number of items to be searched is cut in half
- After *p* passes through the loop, there are $n/(2^p)$ elements left to search

Linear and Binary Search



Array size	10	50	500	5000	50,000	500,000	5,000,000	50,000,000
Average linear search passes	5	25	250	2500	25,000	250,000	2,500,000	25,000,000
Maximum linear search passes	10	50	500	5000	50,000	500,000	5,000,000	50,000,000
Maximum binary search passes	4	6	9	13	16	19	23	26

Table 7.4 A Comparison of `while` Loop Passes for Linear and Binary Searches

Big O Notation

- Big O Notation
  - Represents “the order of magnitude of”
- Sort algorithms come in two major categories:
  - Internal sort: entire list can be resident in memory at one time
  - External sort: for very large lists that cannot be totally in memory at one time

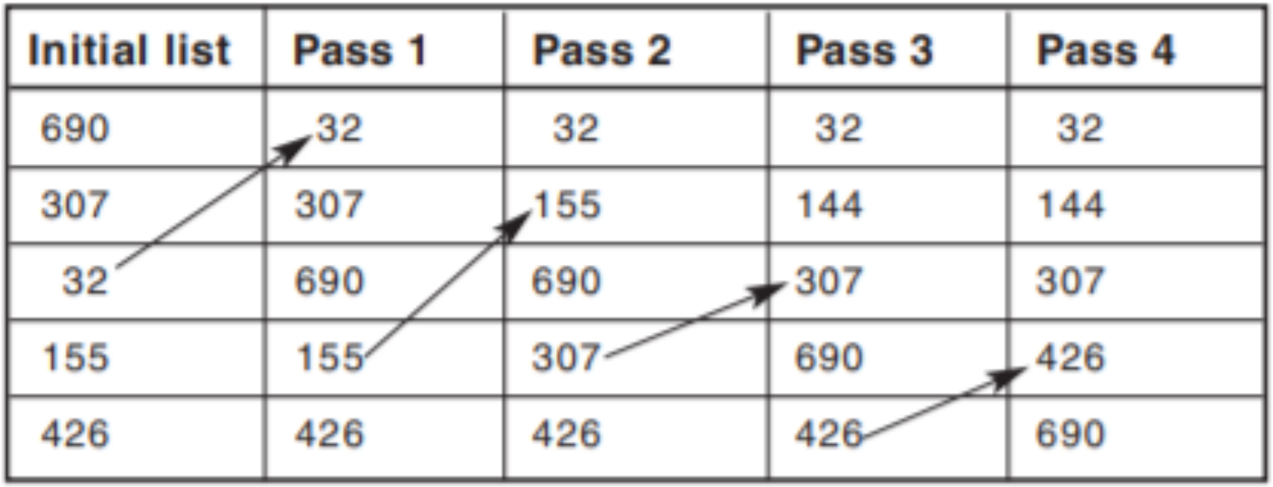
Sort Algorithms

- Two major categories of sorting techniques exist
  - **Internal sort**: Use when data list is small enough to be stored in the computer’s memory
  - **External sort**: Use for larger data sets stored on external disk
- Internal sort algorithms
  - Selection sort
  - Exchange sort

Selection Sort

- Smallest element is found and exchanged with the first element
- Next smallest element is found and exchanged with the second element
- Process continues *n*-1 times, with each pass requiring one less comparison

Selection Sort (continued)

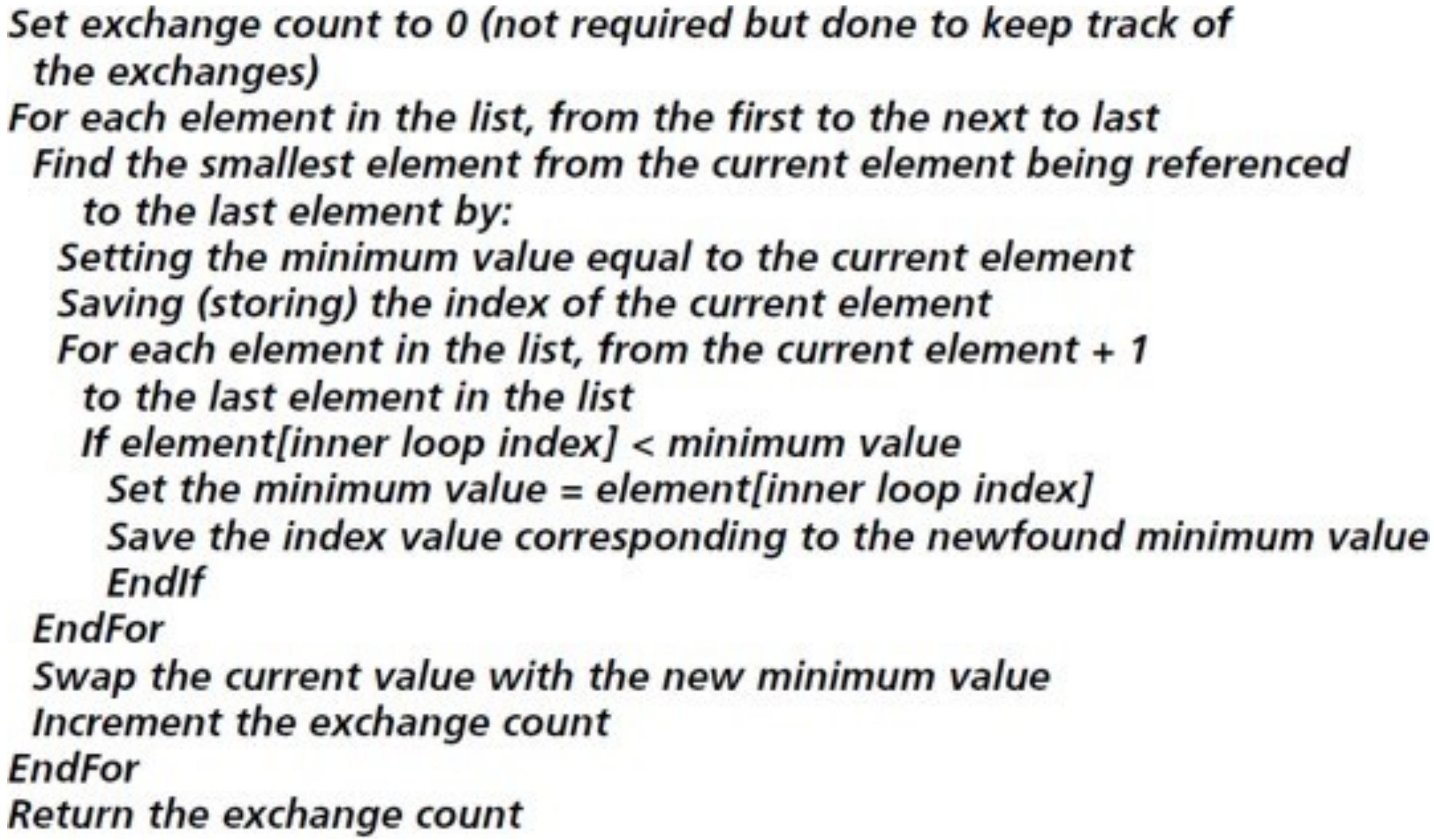


Initial list	Pass 1	Pass 2	Pass 3	Pass 4
690	32	32	32	32
307	307	155	144	144
32	690	690	307	307
155	155	307	690	426
426	426	426	426	690

Figure 7.19 A sample selection sort

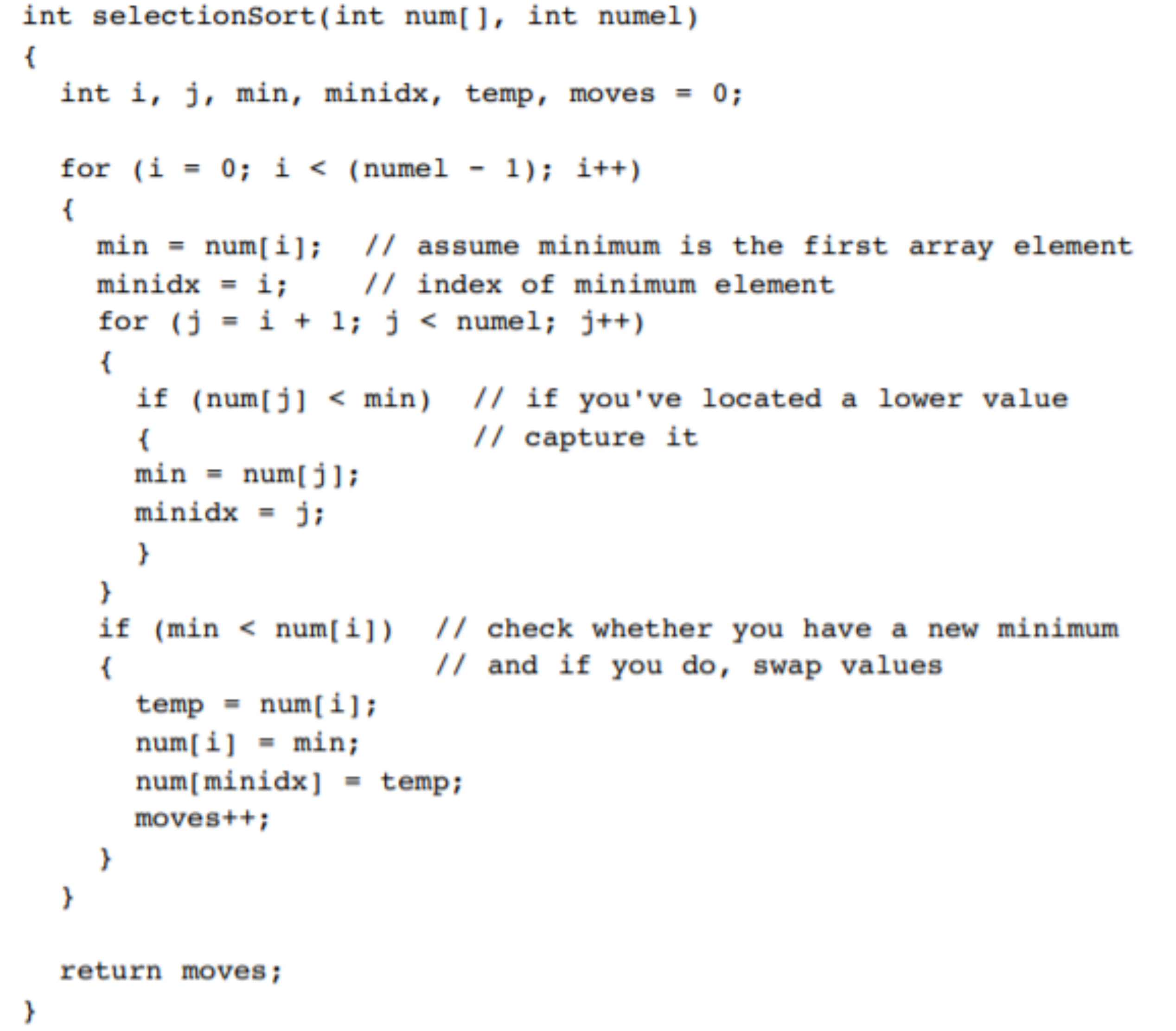
Selection Sort (continued)

- Pseudocode for a selection sort



```
Set exchange count to 0 (not required but done to keep track of
  the exchanges)
For each element in the list, from the first to the next to last
  Find the smallest element from the current element being referenced
      to the last element by:
    Setting the minimum value equal to the current element
    Saving (storing) the index of the current element
    For each element in the list, from the current element + 1
      to the last element in the list
      If element[inner loop index] < minimum value
        Set the minimum value = element[inner loop index]
        Save the index value corresponding to the newfound minimum value
        EndIf
  EndFor
  Swap the current value with the new minimum value
  Increment the exchange count
EndFor
Return the exchange count
```

Selection Sort (continued)



```
int selectionSort(int num[], int numel)
{
  int i, j, min, minidx, temp, moves = 0;

  for (i = 0; i < (numel - 1); i++)
  {
    min = num[i];  // assume minimum is the first array element
    minidx = i;    // index of minimum element
    for (j = i + 1; j < numel; j++)
    {
      if (num[j] < min)  // if you've located a lower value
      {                  // capture it
      min = num[j];
      minidx = j;
      }
    }
    if (min < num[i])  // check whether you have a new minimum
    {                  // and if you do, swap values
      temp = num[i];
      num[i] = min;
      num[minidx] = temp;
      moves++;
    }
  }

  return moves;
}
```

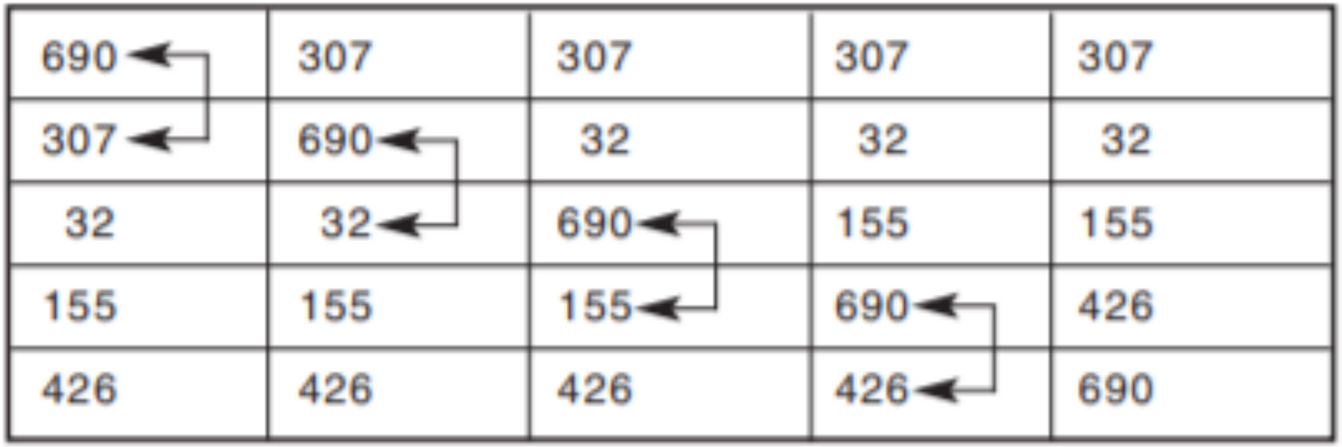
Selection Sort (continued)

- Selection sort advantages :
  - Maximum number of required moves is **`n-1`**
  - Each move is a final move
- Selection sort disadvantages:
  - $n(n\text{-}1)/2$ comparisons are always required
  - Order of magnitude of selection sort: O(n^2)

Exchange (Bubble) Sort

- Successive values in the list are compared
- Each pair is interchanged if needed to place them in sorted order
- If sorting in ascending order, the largest value will “bubble up” to the last position in the list
- Second pass through the list stops comparing at second-to-last element
- Process continues until an entire pass through the list results in no exchanges

Exchange (Bubble) Sort (continued)

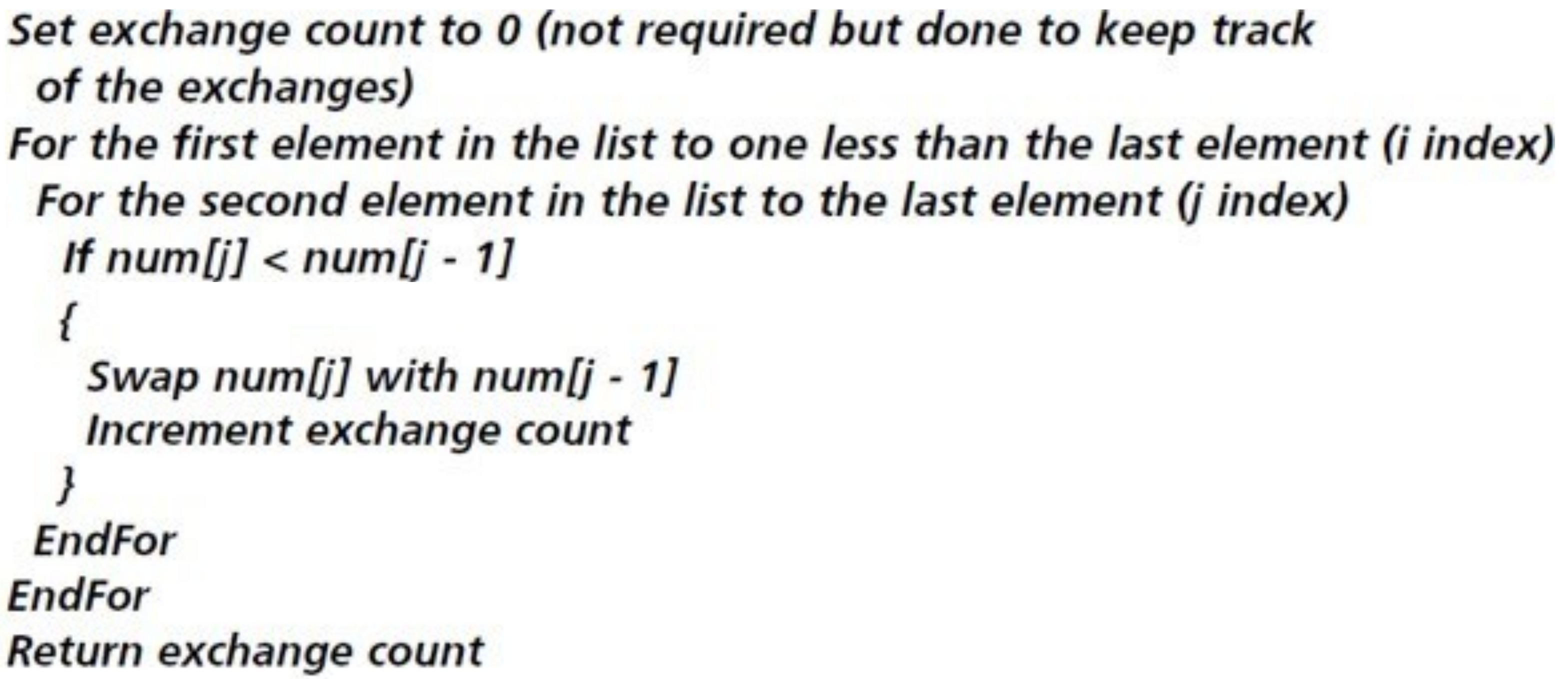


690	307	307	307	307
307	690	32	32	32
32	32	690	155	155
155	155	155	690	426
426	426	426	426	690

Figure 7.20 The first pass of an exchange sort

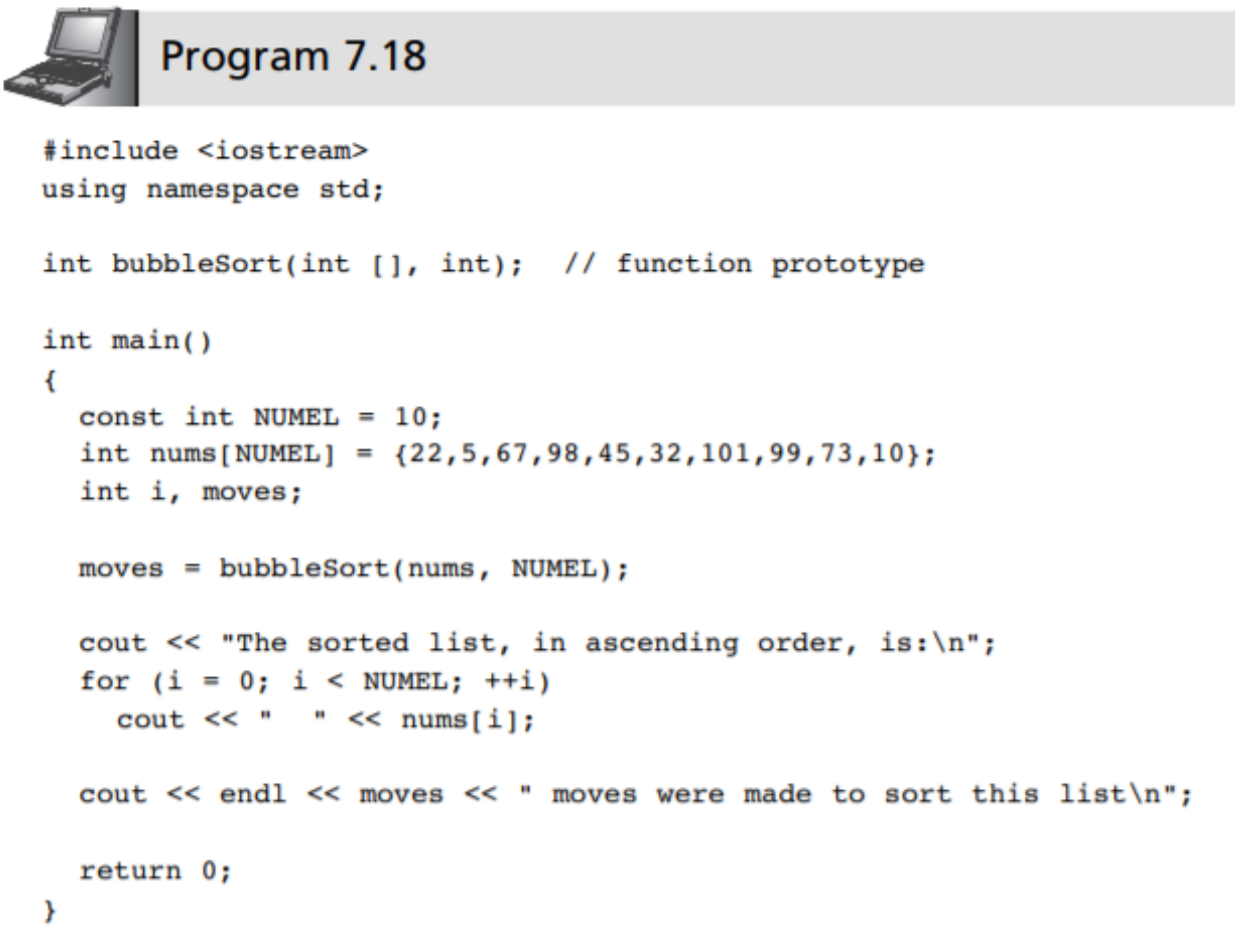
Exchange (Bubble) Sort (continued)

- Pseudocode for an exchange sort



```
Set exchange count to 0 (not required but done to keep track
  of the exchanges)
For the first element in the list to one less than the last element (i index)
  For the second element in the list to the last element (j index)
    If num[j] < num[j - 1]
    {
      Swap num[j] with num[j - 1]
      Increment exchange count
    }
  EndFor
EndFor
Return exchange count
```

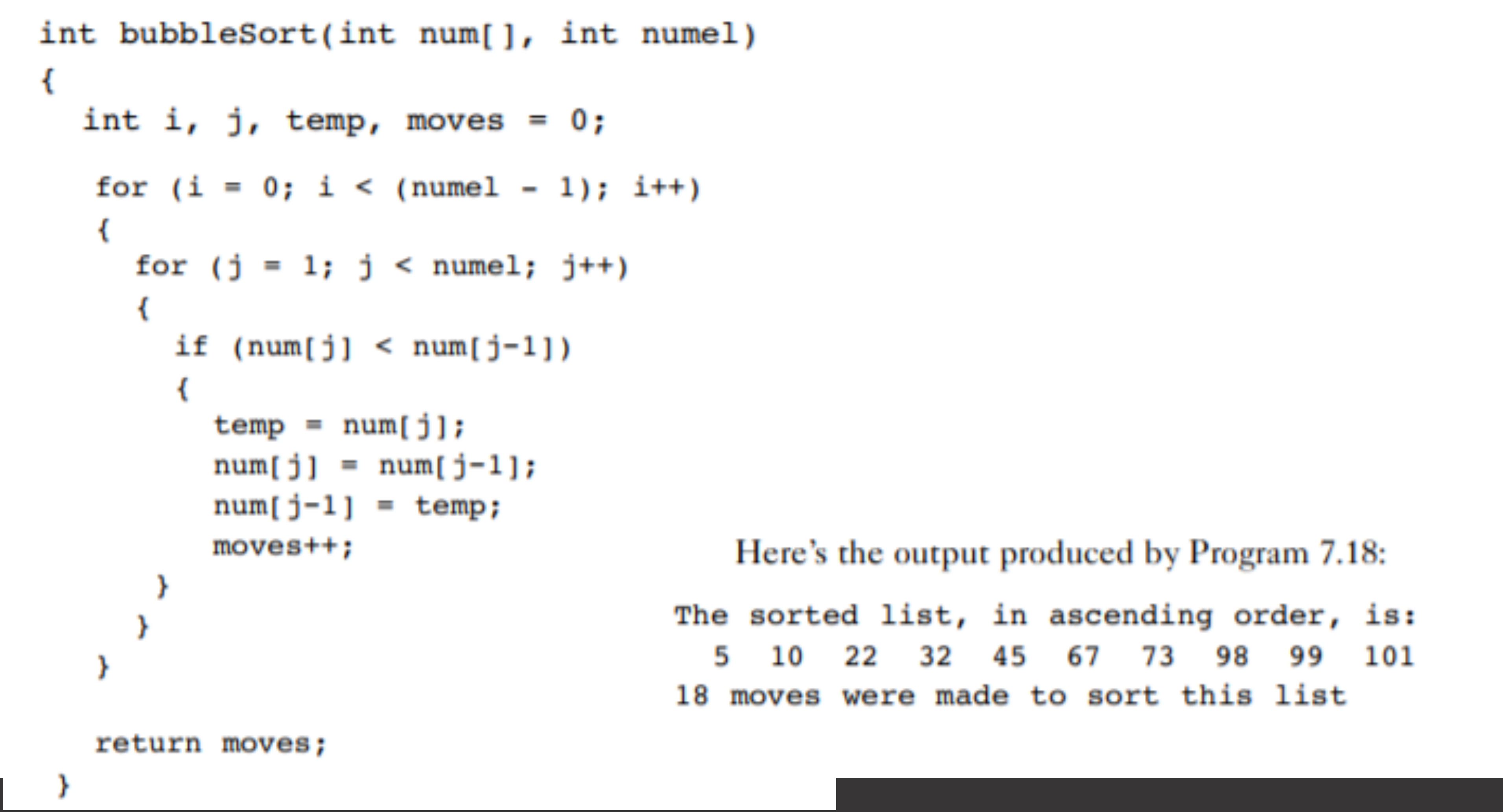
Exchange (Bubble) Sort (continued)



Program 7.18

```
#include <iostream>
using namespace std;

int bubbleSort(int [], int);  // function prototype

int main()
{
  const int NUMEL = 10;
  int nums[NUMEL] = {22,5,67,98,45,32,101,99,73,10};
  int i, moves;

  moves = bubbleSort(nums, NUMEL);

  cout << "The sorted list, in ascending order, is:\n";
  for (i = 0; i < NUMEL; ++i)
    cout << "  " << nums[i];

  cout << endl << moves << " moves were made to sort this list\n";

  return 0;
}
```

Exchange (Bubble) Sort (continued)



```
int bubbleSort(int num[], int numel)
{
  int i, j, temp, moves = 0;

   for (i = 0; i < (numel - 1); i++)
   {
     for (j = 1; j < numel; j++)
     {
       if (num[j] < num[j-1])
       {
         temp = num[j];
         num[j] = num[j-1];
         num[j-1] = temp;
         moves++;
        }
      }
   }

   return moves;
 }
```

Here's the output produced by Program 7.18:

```
The sorted list, in ascending order, is:
  5  10  22  32  45  67  73  98  99  101
18 moves were made to sort this list
```

Exchange (Bubble) Sort (continued)

- Number of comparisons = O(n^2)
- Maximum number of comparisons: $n(n-1)/2$
- Maximum number of moves: $n(n-1)/2$
- Many moves are not final moves

Common Programming Errors

- Forgetting to declare the array
- Using a subscript that references a non-existent array element (out of bounds)
- Failing to use a counter value in a loop that is large enough to cycle through all array elements
- Forgetting to initialize the array

Summary

- An array is a data structure that stores a list of values having the same data type
  - Array elements: stored in contiguous memory locations; referenced by array name/index position
  - Two-dimensional arrays have rows and columns
  - Arrays may be initialized when they are declared
  - Arrays may be passed to a function by passing the name of the array as the argument
    - Arrays passed as arguments are passed by reference
    - Individual array elements as arguments are passed by value

Homework

1. P456, exercise 1
2. P458, exercise 11