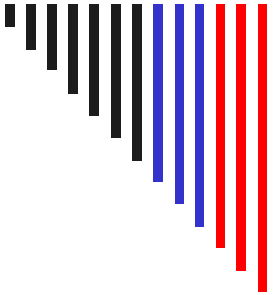


Chapter 8 Arrays

Objectives

- To understand the basic concepts and uses of arrays
- To be able to define C arrays
- To be able to pass arrays and array elements to functions
- To understand the classical approaches to sorting arrays: selection, bubble, and insertion sorting
- To write programs that sort data using the three classical algorithms
- To be able to analyze the efficiency of a sort algorithm
- To understand the two classical search algorithms: sequential and binary
- To write programs that search arrays
- To be able to design test cases for sorting and searching algorithms
- To be able to analyze the efficiency of searching algorithms

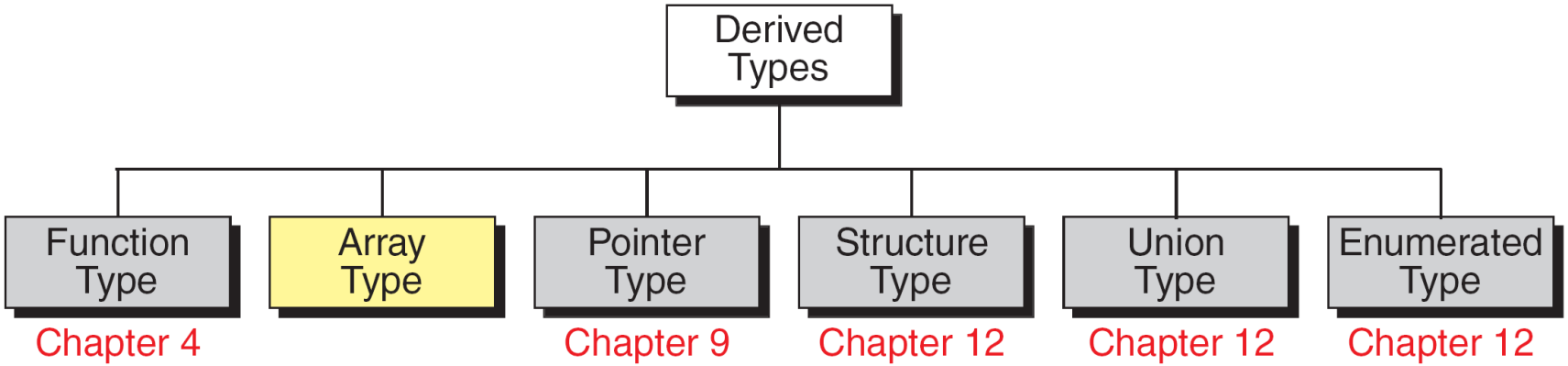


FIGURE 8-1 Derived Types

8-1 Concepts

Imagine we have a problem that requires us to read, process, and print a large number of integers. We must also keep the integers in memory for the duration of the program.

To process large amounts of data we need a powerful data structure, the array. An array is a collection of elements of the same data type.

Since an array is a sequenced collection, we can refer to the elements in the array as the first element, the second element, and so forth until we get to the last element.

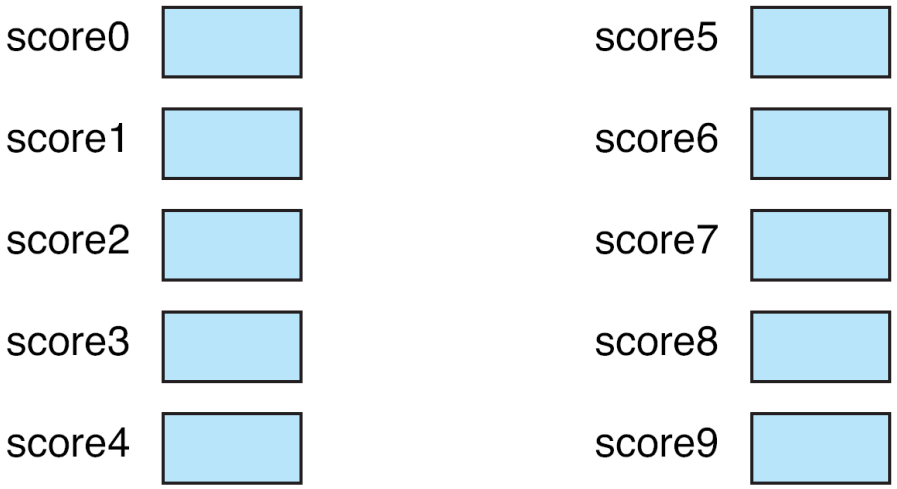


FIGURE 8-2 Ten Variables

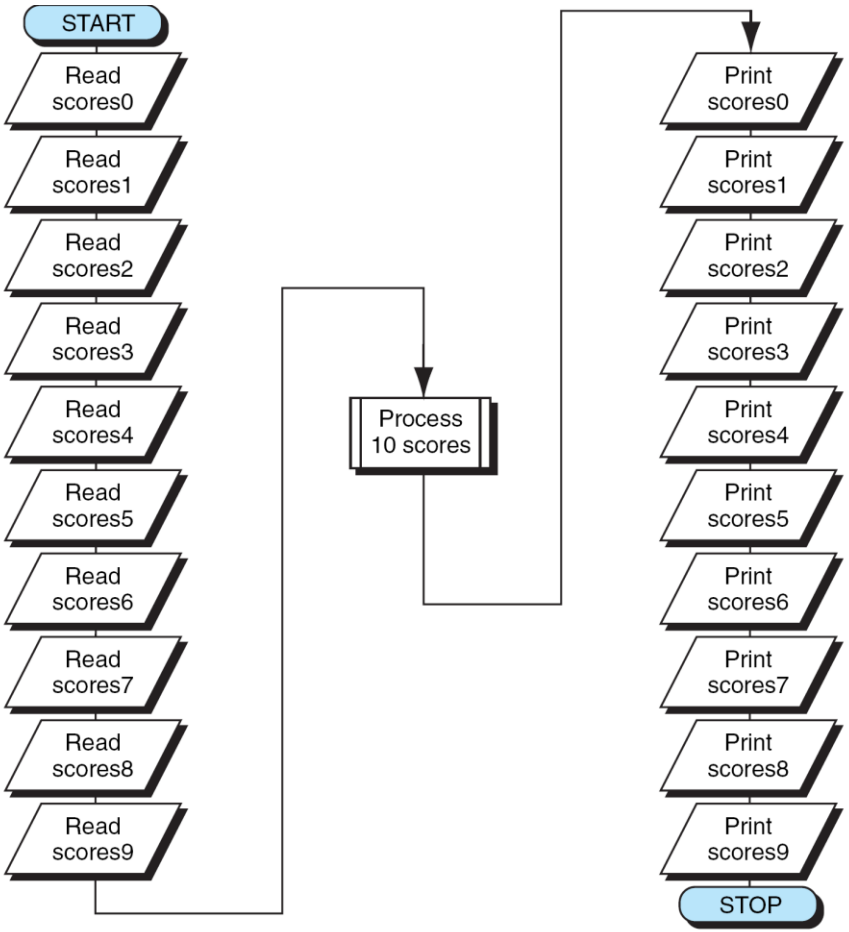


FIGURE 8-3 Process 10 variables

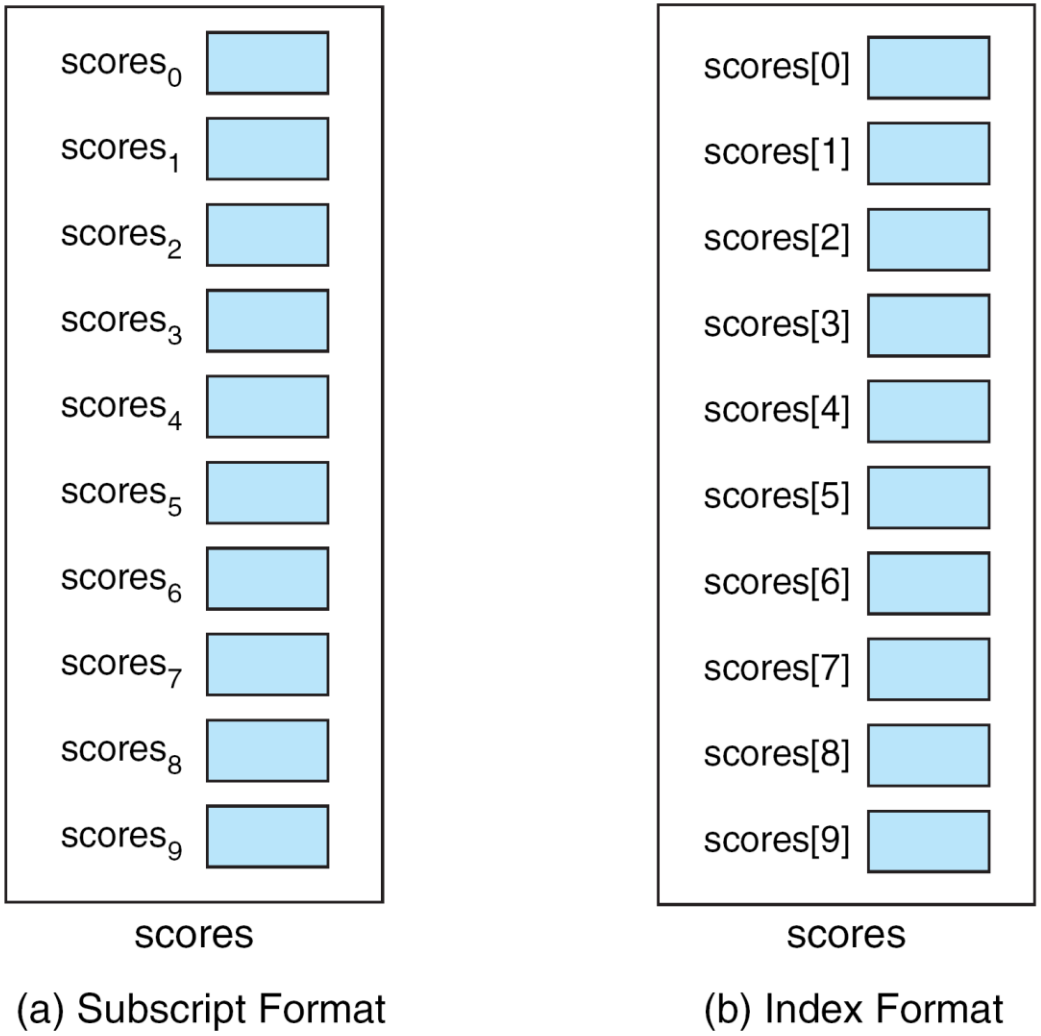


FIGURE 8-4 An Array of Scores

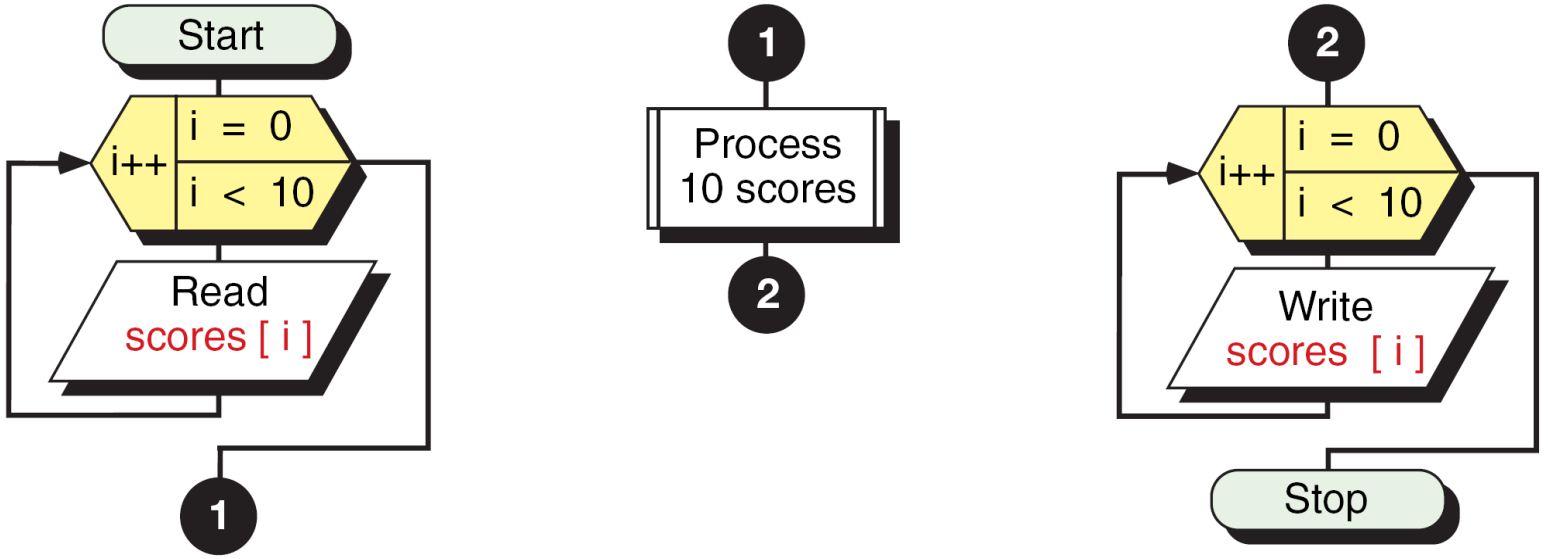


FIGURE 8-5 **Loop for 10 Scores**

8-2 Using Arrays in C

In this section, we first show how to declare and define arrays. Then we present several typical applications using arrays including reading values into arrays, accessing and exchanging elements in arrays, and printing arrays.

Topics discussed in this section:

Declaration and Definition
Accessing Elements in Arrays
Storing Values in Arrays
Index Range Checking

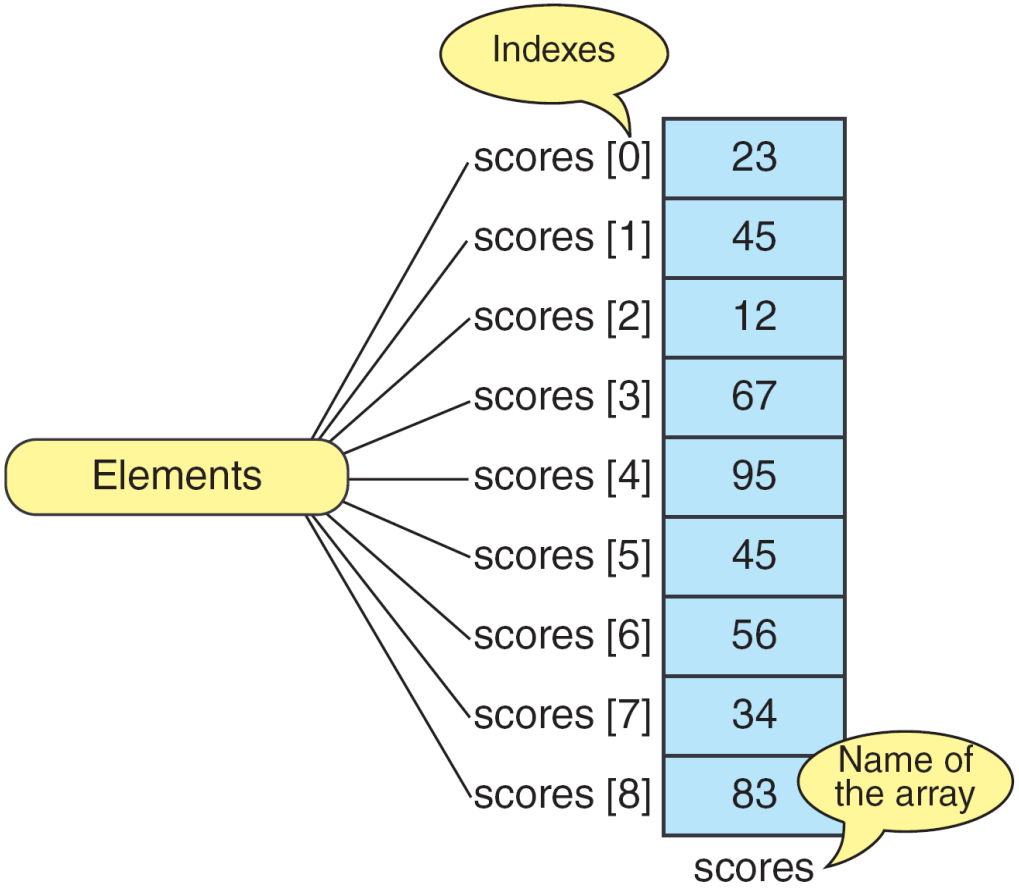


FIGURE 8-6 The Scores Array

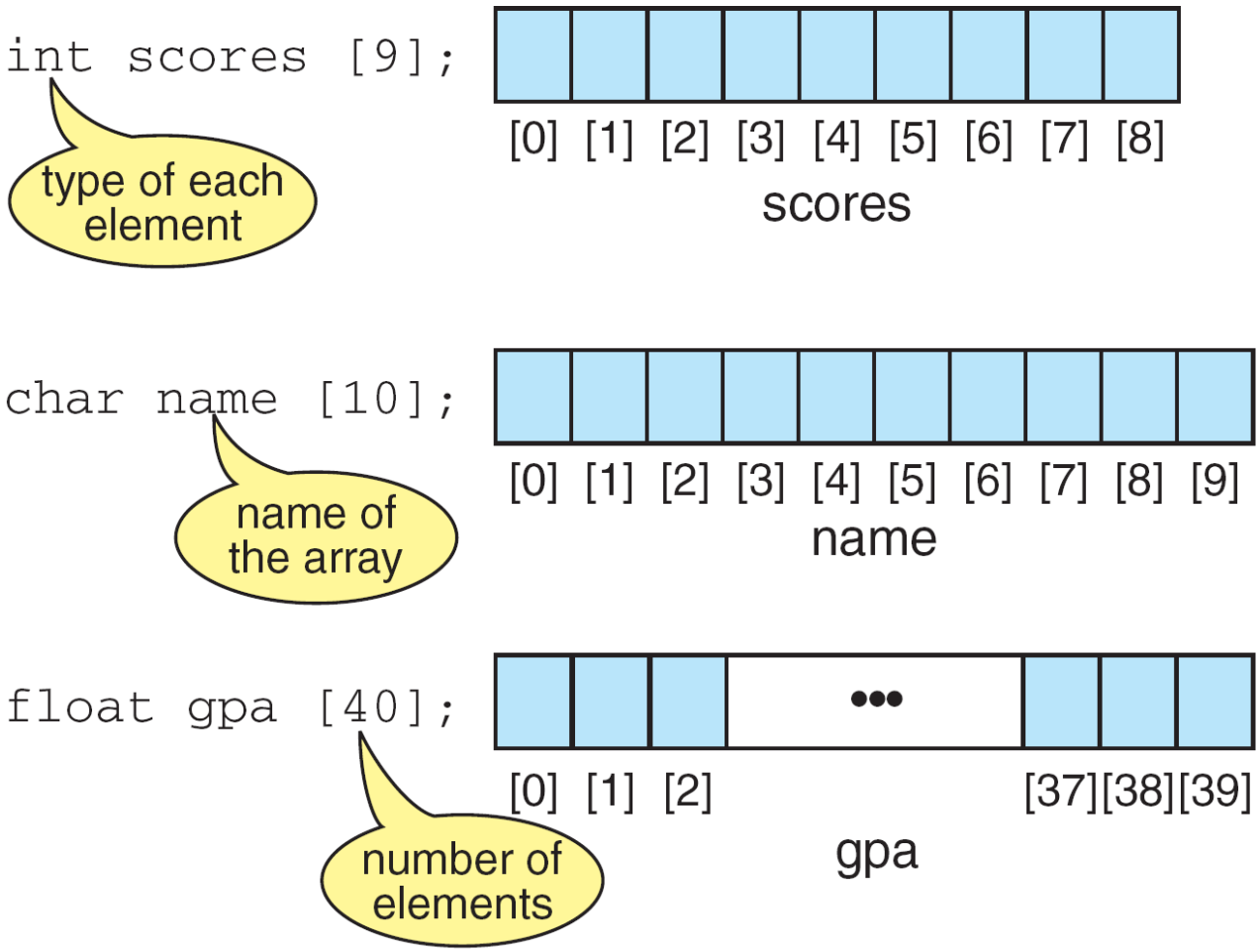


FIGURE 8-7 Declaring and Defining Arrays

Note

Only fixed-length arrays can be initialized when they are defined. Variable length arrays must be initialized by inputting or assigning the values.

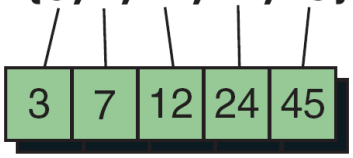
Accessing Elements in Arrays

- scores[0]
- scores[i]

- element address = array address
  + (sizeof (element) * index)

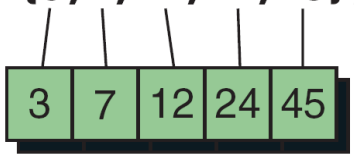
(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```



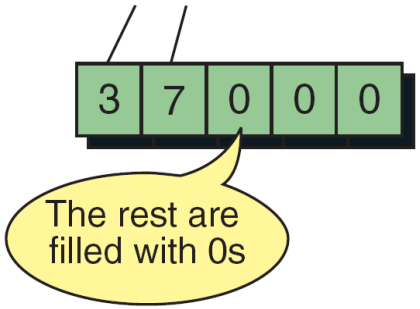
(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```



(c) Partial Initialization

```
int numbers[5] = {3,7};
```



(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```

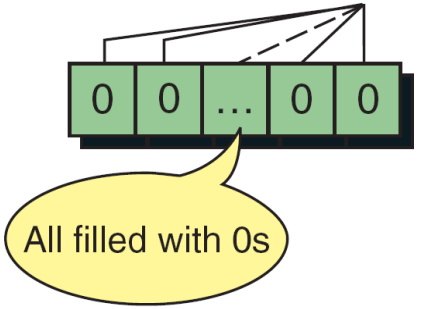


FIGURE 8-8 Initializing Arrays

Inputting & Assigning Values

```
for (i = 0; i < 9; i++)
    scanf ("%d", &scores[i]);

scores[4] = 23;
```

Note

One array cannot be copied to another using assignment.

PROGRAM 8-1 Print Ten Numbers per Line

```
1  // a program fragment
2  #define MAX_SIZE 25
3
4  // Local Declarations
5     int list [MAX_SIZE];
6
7  // Statements
8     …
9     numPrinted = 0;
10    for (int i = 0; i < MAX_SIZE; i++)
11       {
12        printf("%3d", list[i]);
13        if (numPrinted < 9)
14            numPrinted++;
15        else
16           {
17            printf("\n");
18            numPrinted = 0;
19            } // else
20       } // for
```

Precedence of Array References

- numbers[3] = numbers[4] + 15;

PROGRAM 8-2 Squares Array

```
1  /* Initialize array with square of index and print it.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #define ARY_SIZE 5
7
8  int main (void)
9  {
10 // Local Declarations
11    int sqrAry[ARY_SIZE];
12
13 // Statements
14    for (int i = 0; i < ARY_SIZE; i++)
15       sqrAry[i] = i * i;
16
17    printf("Element\tSquare\n");
18    printf("=======\t======\n");
```

PROGRAM 8-2 Squares Array

```
19      for (int i = 0; i < ARY_SIZE; i++)
20          printf("%5d\t%4d\n", i, sqrAry[i]);
21      return 0;
22  }   // main
```

```
Results:
Element Square
======= ======
      0      0
      1      1
      2      4
      3      9
      4     16
```

PROGRAM 8-3 Print Input Reversed

```
 1  /* Read a number series and print it reversed.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int readNum;
11     int numbers[50];
12
13  // Statements
14     printf("You may enter up to 50 integers:\n");
15     printf("How many would you like to enter? ");
16     scanf ("%d", &readNum);
17
18     if (readNum > 50)
19         readNum = 50;
```

PROGRAM 8-2 Print Input Reversed

```
20
21      // Fill the array
22      printf("\nEnter your numbers: \n");
23      for (int i = 0; i < readNum; i++)
24           scanf("%d", &numbers[i]);
25
26      // Print the array
27      printf("\nYour numbers reversed are: \n");
28      for (int i = readNum - 1, numPrinted = 0;
29               i >= 0;
30               i--)
31         {
32          printf("%3d", numbers[i]);
33          if (numPrinted < 9)
34              numPrinted++;
35          else
36             {
37              printf("\n");
38              numPrinted = 0;
39             } // else
```

PROGRAM 8-2 Print Input Reversed

```
40          } // for
41       return 0;
42    }  // main
```

```
Results:
You may enter up to 50 integers:
How many would you like to enter? 12

Enter your numbers:
1 2 3 4 5 6 7 8 9 10 11 12

Your numbers reversed are:
 12 11 10  9  8  7  6  5  4  3
  2  1
```

8-3 Inter-function Communication

To process arrays in a large program, we have to be able to pass them to functions. We can pass arrays in two ways: pass individual elements or pass the whole array. In this section we discuss first how to pass individual elements and then how to pass the whole array.

Topics discussed in this section:

Passing Individual Elements
Passing the Whole Array

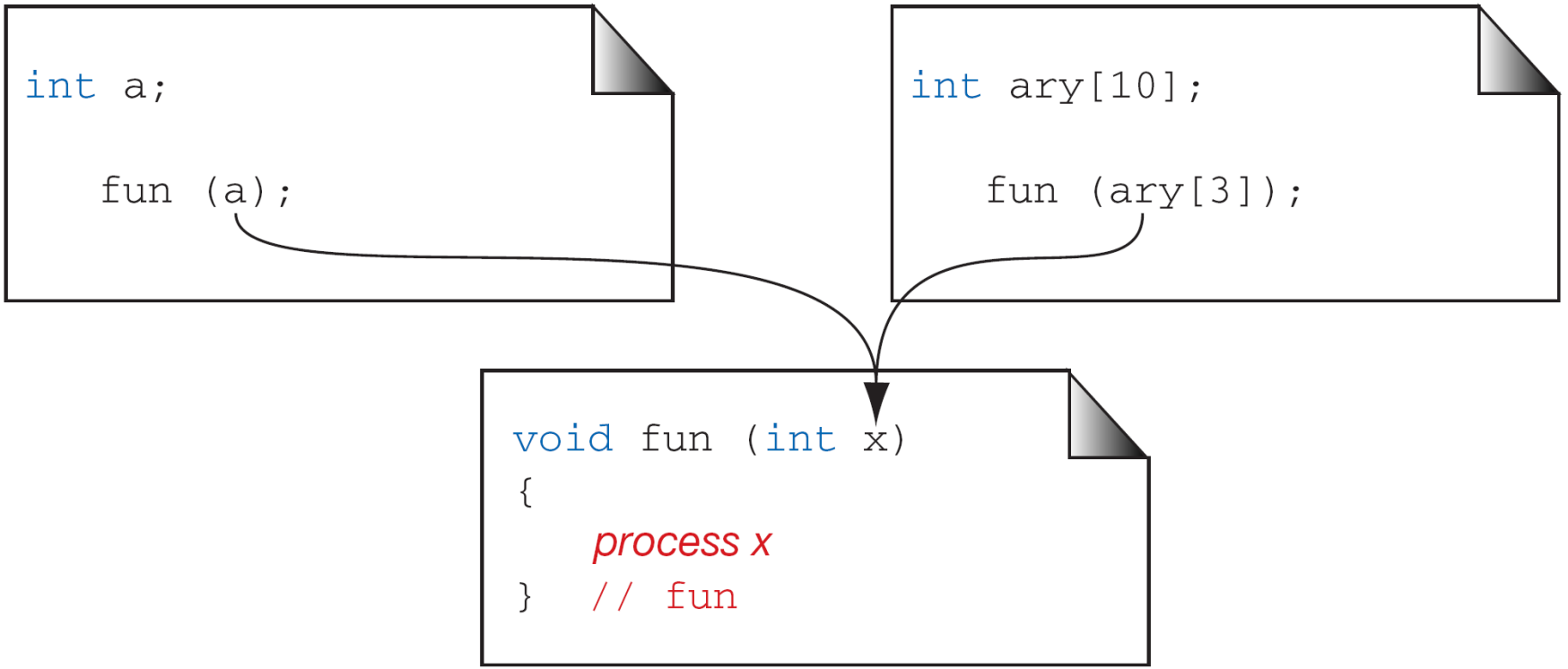


FIGURE 8-11 Passing Array Elements

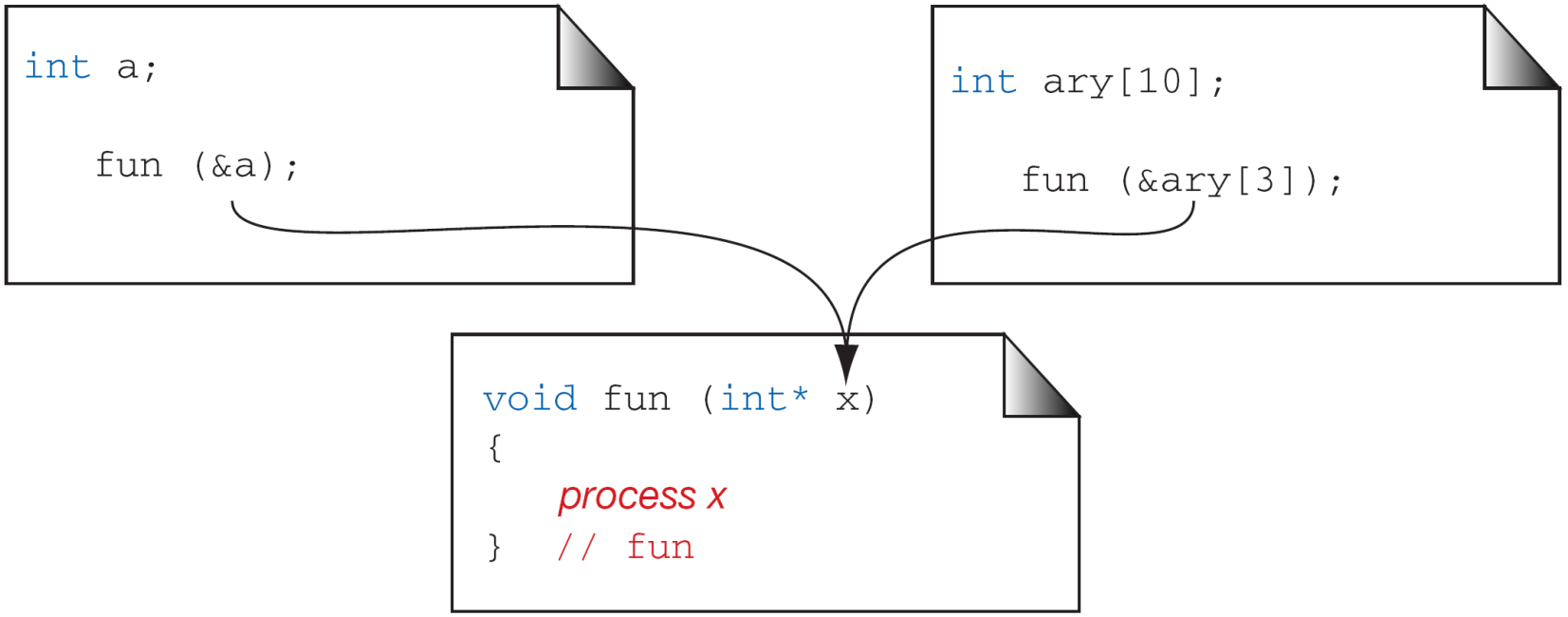


FIGURE 8-12 Passing the Address of an Array Element

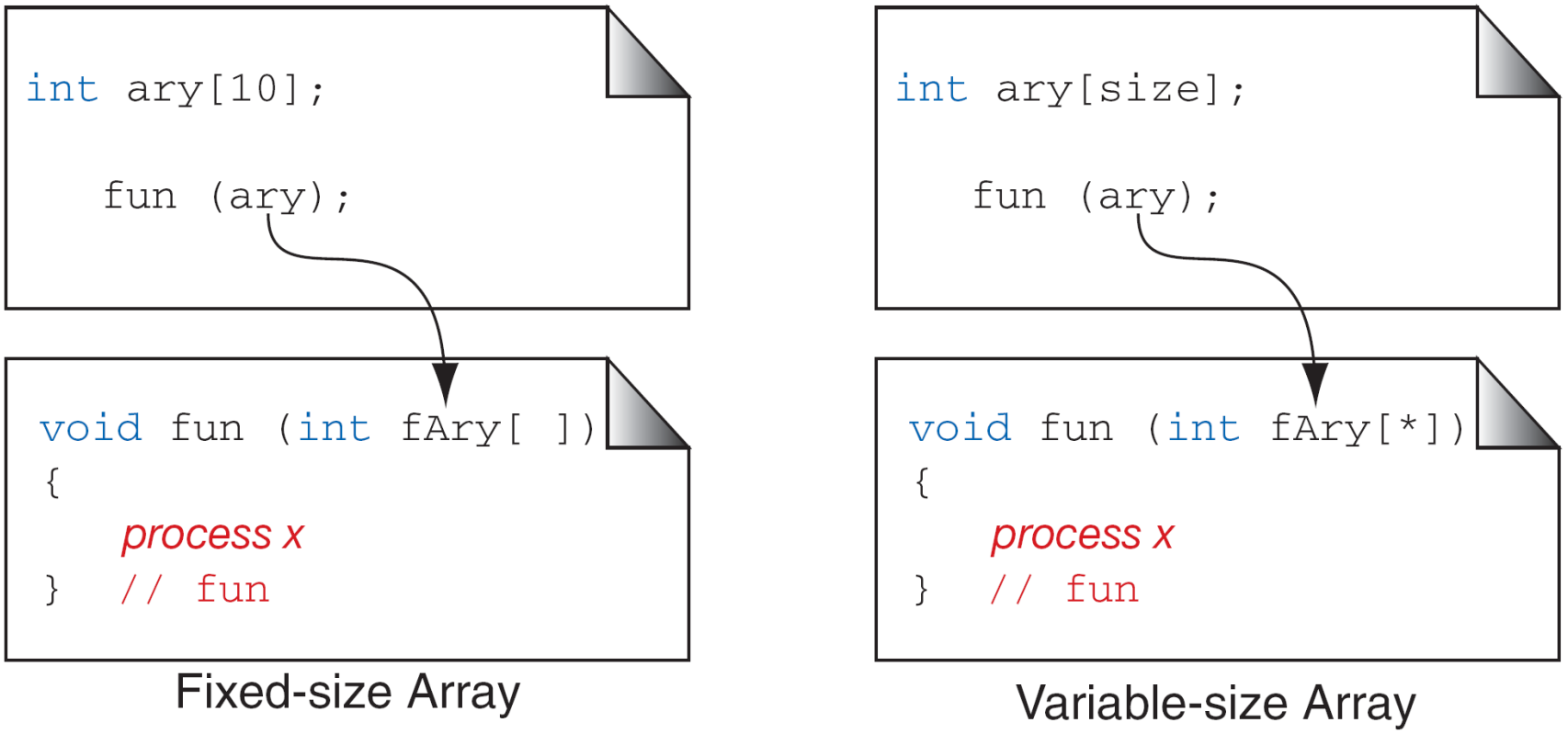


FIGURE 8-13 Passing the Whole Array

PROGRAM 8-4 Calculate Array Average

```
1  /* Calculate the average of the number in an array.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6
7  // Function Declaration
8  double average (int ary[ ]);
9
10 int main (void)
11 {
12 // Local Declarations
13    double ave;
14    int    base[5] = {3, 7, 2, 4, 5};
15
16 // Statement
17    ave = average(base);
18    printf("Average is: %lf\n", ave);
19    return 0;
20 }  // main
```

PROGRAM 8-4 Calculate Array Average

```
21
22  /* ===================== average ====================
23     Calculate and return average of values in array.
24        Pre  Array contains values
25        Post Average calculated and returned
26  */
27  double average (int ary[ ])
28  {
29  // Local Declarations
30     int sum = 0;
31
32  // Statement
33     for (int i = 0; i < 5; i++)
34           sum += ary[i];
35
36     return (sum / 5.0);
37  }  // average
```

PROGRAM 8-6 Change Values in an Array

```
 1  /* Multiply each element in an array by 2.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  // Function Declaration
 7  void multiply2 (int x[ ]);
 8
 9  int main (void)
10  {
11  // Local Declarations
12     int base[5] = {3, 7, 2, 4, 5};
13
14  // Statements
15     multiply2 (base);
16
17     printf("Array now contains: ");
```

PROGRAM 8-6 Change Values in an Array

```
18      for (int i = 0; i < 5; i++)
19         printf("%3d", base[i]);
20      printf("\n");
21      return 0;
22  }  // main
23
24  /* ==================== multiply2 ====================
25      Multiply each element in array by 2.
26         Pre  array contains data
27         Post each element doubled
28  */
29  void multiply2 (int ary[ ])
30  {
31  // Statements
32      for (int i = 0; i < 5; i++)
33           ary[i] *= 2;
34      return;
35  }  // multiply2
```

Passing an Array as a Constant

- double average (const int ary[], int size);

8-4 Array Applications

In this section we study two array applications: frequency arrays with their graphical representations and random number permutations.

Topics discussed in this section:

Frequency Arrays
Histograms
Random Number Permutations

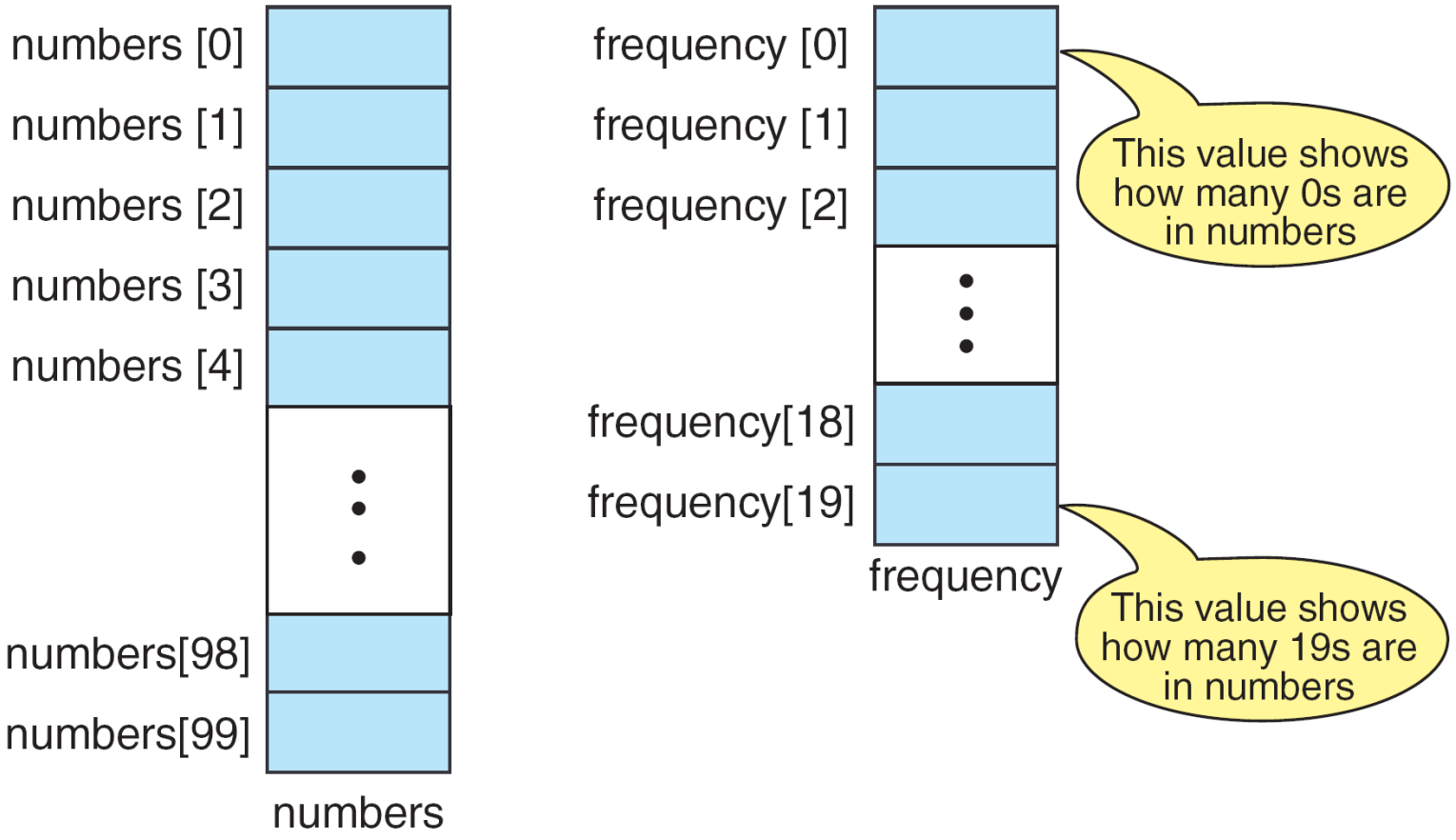


FIGURE 8-14 Frequency Array

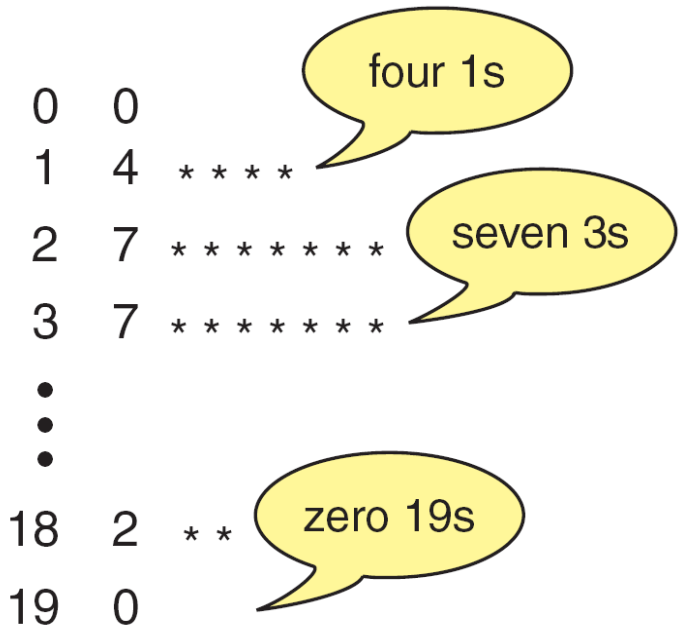


FIGURE 8-15 Frequency Histogram

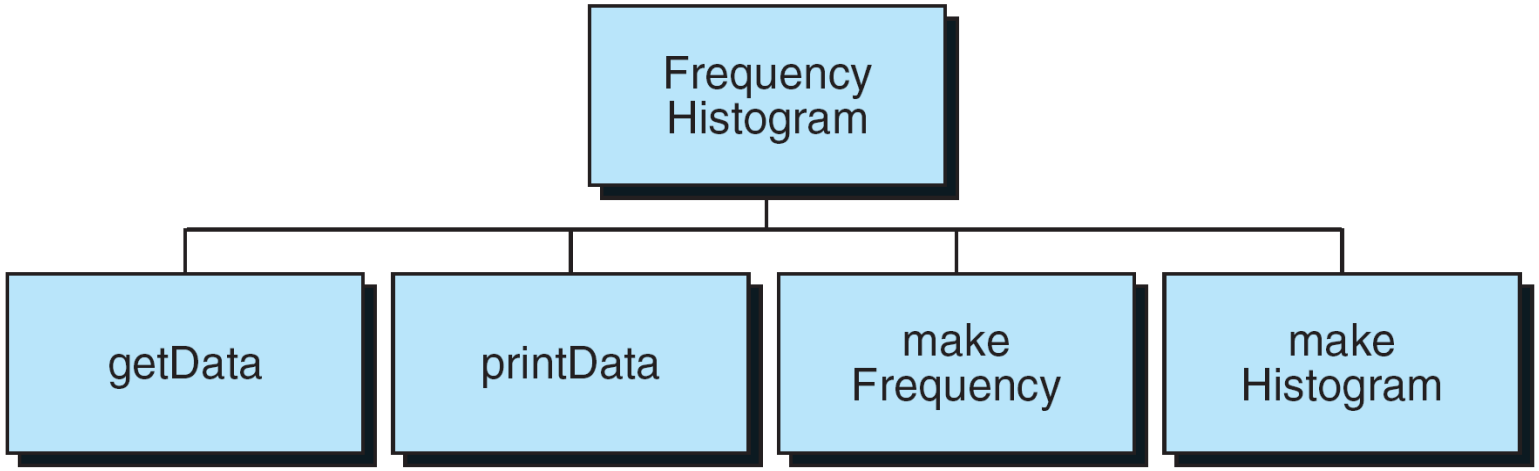


FIGURE 8-16 Histogram Program Design

PROGRAM 8-7 Frequency and Histogram

```
1  /* Read data from a file into an array.
2     Build frequency array & print data in histogram.
3        Written by:
4        Data:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  #define MAX_ELMNTS 100
10 #define ANLYS_RNG 20
11
12 // Function Declarations
13    int getData (int numbers[], int size, int range);
14
15    void printData     (int numbers[],
16                        int size,         int lineSize);
17    void makeFrequency (int numbers[],    int size,
18                        int frequency[], int range);
19    void makeHistogram (int frequency[], int range);
20
```

PROGRAM 8-7 Frequency and Histogram

```
21  int main (void)
22  {
23  // Local Declarations
24     int size:
25     int nums      [MAX_ELMNTS];
26     int frequency [ANLYS_RNG];
27
28  // Statements
29     size = getData (nums, MAX_ELMNTS, ANLYS_RNG);
30     printData (nums, size, 10);
31
32     makeFrequency(nums, size, frequency, ANLYS_RNG);
33     makeHistogram(frequency, ANLYS_RNG);
34     return 0;
35  }  // main
36
```

PROGRAM 8-7 Frequency and Histogram

```
37 /* ===================== getData ====================
38    Read data from file into array. The array
39    does not have to be completely filled.
40       Pre   data is an empty array
41             size is maximum elements in array
42             range is highest value allowed
43       Post  array is filled. Return number of elements
44 */
45 int getData (int data [], int size, int range)
46 {
47 // Local Declarations
48    int   dataIn;
49    int   loader = 0;
50    FILE* fpData;
51
52 // Statements
53 // Adjust range for zero value
54    range--;
55    if (!(fpData = fopen ("P08-07.dat", "r")))
56       printf("Error opening file\a\a\n") , exit (100);
```

PROGRAM 8-7 Frequency and Histogram

```
57
58      while (loader < size
59          && fscanf(fpData, "%d", &dataIn) != EOF)
60          if (dataIn >= 0 && dataIn <= range)
61              data[loader++] = dataIn;
62          else
63              printf("\nData point %d invalid. Ignored. \n",
64                      dataIn);
65
66  // Test to see what stopped while
67      if (loader == size)
68          printf("\nToo much data. Process what read.\n");
69      return loader;
70  }   // getData
71
72  /* ==================== printData ====================
73      Prints the data as a two-dimensional array.
74          Pre   data: a filled array
75                size: number of elements in array
76                lineSize: number of elements printed/line
77          Post  the data have been printed
78  */
```

PROGRAM 8-7 Frequency and Histogram

```
79  void printData (int data[], int size, int lineSize)
80  {
81  // Local Declarations
82     int numPrinted = 0;
83
84  // Statements
85     printf("\n\n");
86     for (int i = 0; i < size; i++)
87        {
88         numPrinted++;
89         printf("%2d ", data[i]);
90         if (numPrinted >= lineSize)
91            {
92             printf("\n");
93             numPrinted = 0;
94            } // if
95        } // for
96     printf("\n\n");
97     return;
98  }  // printData
99
```

PROGRAM 8-7 Frequency and Histogram

```
100 /* ================= makeFrequency ==================
101    analyze the data in nums and build their frequency
102    distribution
103       Pre   nums: array of validated data to be analyzed
104             last: number of elements in array
105             frequency: array for accumulation.
106             range: maximum index/value for frequency
107       Post  Frequency array has been built.
108 */
109 void makeFrequency (int nums[],      int last,
110                     int frequency[], int range)
111 {
112 // Statements
113    // First initialize the frequency array
114    for (int f = 0; f < range; f++)
115         frequency [f] = 0;
116
```

PROGRAM 8-7 Frequency and Histogram

```
117     // Scan numbers and build frequency array
118     for (int i = 0; i < last; i++)
119          frequency [nums [i]]++;
120     return;
121 }   // makeFrequency
122
123 /* ================= makeHistogram ==================
124     Print a histogram representing analyzed data.
125        Pre   freq contains times each value occurred
126              size represents elements in frequency array
127        Post  histogram has been printed
128 */
129 void makeHistogram (int freq[], int range)
130 {
131 // Statements
132     for (int i = 0; i < range; i++)
133        {
134         printf ("%2d %2d ", i, freq[i]);
135         for (int j = 1; j <= freq[i]; j++)
```

PROGRAM 8-7 Frequency and Histogram

```
136                printf ("*");
137             printf ("\n");
138           } // for i...
139       return;
140   }  // makeHistogram
141   // ================= End of Program ================
```

```
Results:
Data point 20 invalid. Ignored.

Data point 25 invalid. Ignored.

 1  2  3  4  5  6  7  8  7 10
 2 12 13 13 15 16 17 18 17  7
 3  4  6  8 10  2  4  6  8 10
 4  3  5  7  1  3  7  7 11 13
 5 10 11 12 13 16 18 11 12  7
 6  1  2  2  3  3  3  4  4  4
 7  7  8  7  6  5  4  1  2  2
 8 11 11 13 13 13 17 17  7  7
13 17 17 15 15
```

PROGRAM 8-7 Frequency and Histogram

```
 0   0
 1   4 ****
 2   7 *******
 3   7 *******
 4   8 ********
 5   4 ****
 6   5 *****
 7  12 ************
 8   5 *****
 9   0
10   4 ****
11   5 *****
12   3 ***
13   8 ********
14   0
15   3 ***
16   2 **
17   6 ******
18   2 **
19   0
```

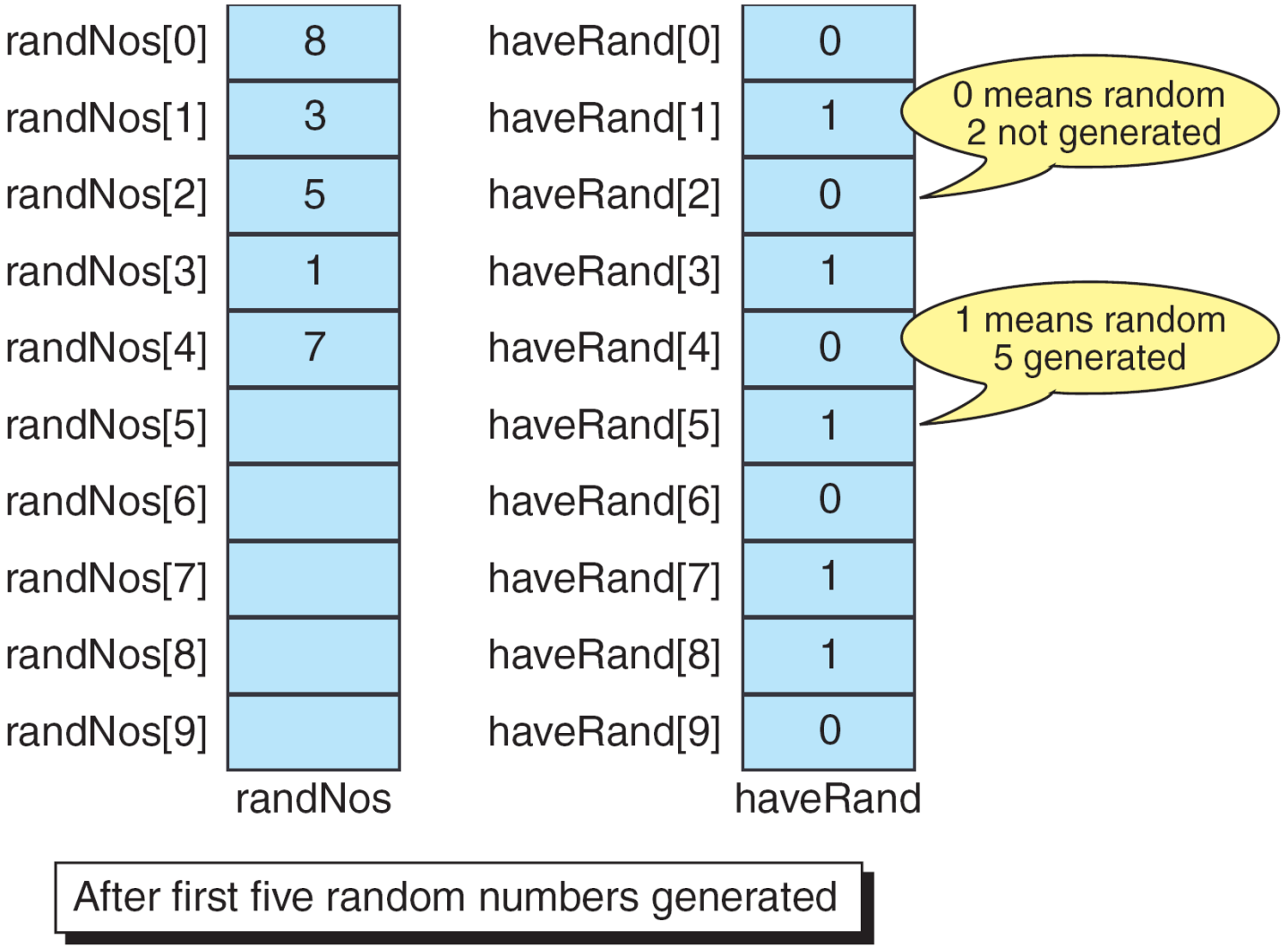


FIGURE 8-17 Design for Random Number Permutations

PROGRAM 8-8 Generate a Permutation

```
1  /* Generate a random number permutation.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  #define  ARY_SIZE  20
9
10 // Function Declarations
11 void bldPerm   (int randNos[]);
12 void printData (int data[], int size, int lineSize);
13
14 int main (void)
15 {
16 // Local Declarations
17    int randNos [ARY_SIZE];
18
```

PROGRAM 8-8 Generate a Permutation

```
19  // Statements
20     printf("Begin Random Permutation Generation\n");
21
22     bldPerm   (randNos);
23     printData (randNos, ARY_SIZE, 10);
24
25     return 0;
26  }  // main
27
28  /* ==================== bldPerm ====================
29     Generate a random number permutation in array.
30        Pre   randNos is array to receive permutations
31        Post  randNos filled
32  */
33  void bldPerm (int randNos[])
34  {
35  // Local Declarations
36     int oneRandNo;
37     int haveRand[ARY_SIZE] = {0};
38
```

PROGRAM 8-8 Generate a Permutation

```
39  // Statements
40     for (int i = 0; i < ARY_SIZE; i++)
41          {
42           do
43             {
44              oneRandNo = rand() % ARY_SIZE;
45             } while (haveRand[oneRandNo] == 1);
46           haveRand[oneRandNo] = 1;
47           randNos[i] = oneRandNo;
48          } // for
49     return;
50  }  // bldPerm
51
52  /* ==================== printData ====================
53     Prints the data as a two-dimensional array.
54        Pre   data: a filled array
55              last: index to last element to be printed
56              lineSize: number of elements on a line
57        Post  data printed
58  */
```

PROGRAM 8-8 Generate a Permutation

```
59  void printData (int data[], int size, int lineSize)
60  {
61  // Local Declarations
62     int numPrinted = 0;
63
64  // Statements
65     printf("\n");
66     for (int i = 0; i < size; i++)
67        {
68         numPrinted++;
69         printf("%2d ", data[i]);
70         if (numPrinted >= lineSize)
71            {
72             printf("\n");
73             numPrinted = 0;
74            } // if
75        } // for
76     printf("\n");
77     return;
78  }  // printData
```

PROGRAM 8-8 Generate a Permutation

```
Results:
Begin Random Permutation Generation

18 13 15 11  7 10 19 12  6  9
 4  0  5  3 17 14  2 16  1  8
```

8-5 Sorting

One of the most common applications in computer science is sorting—the process through which data are arranged according to their values. We are surrounded by data. If the data are not ordered, we would spend hours trying to find a single piece of information.

Topics discussed in this section:

Selection Sort
Bubble Sort
Insertion Sort
Testing Sorts

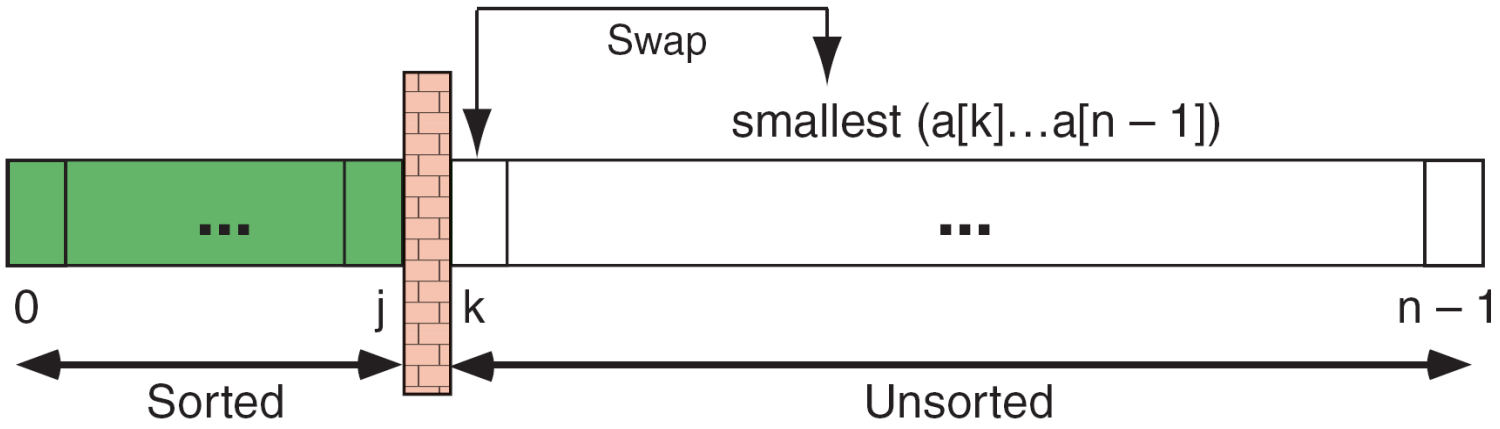


FIGURE 8-18 Selection Sort Concept

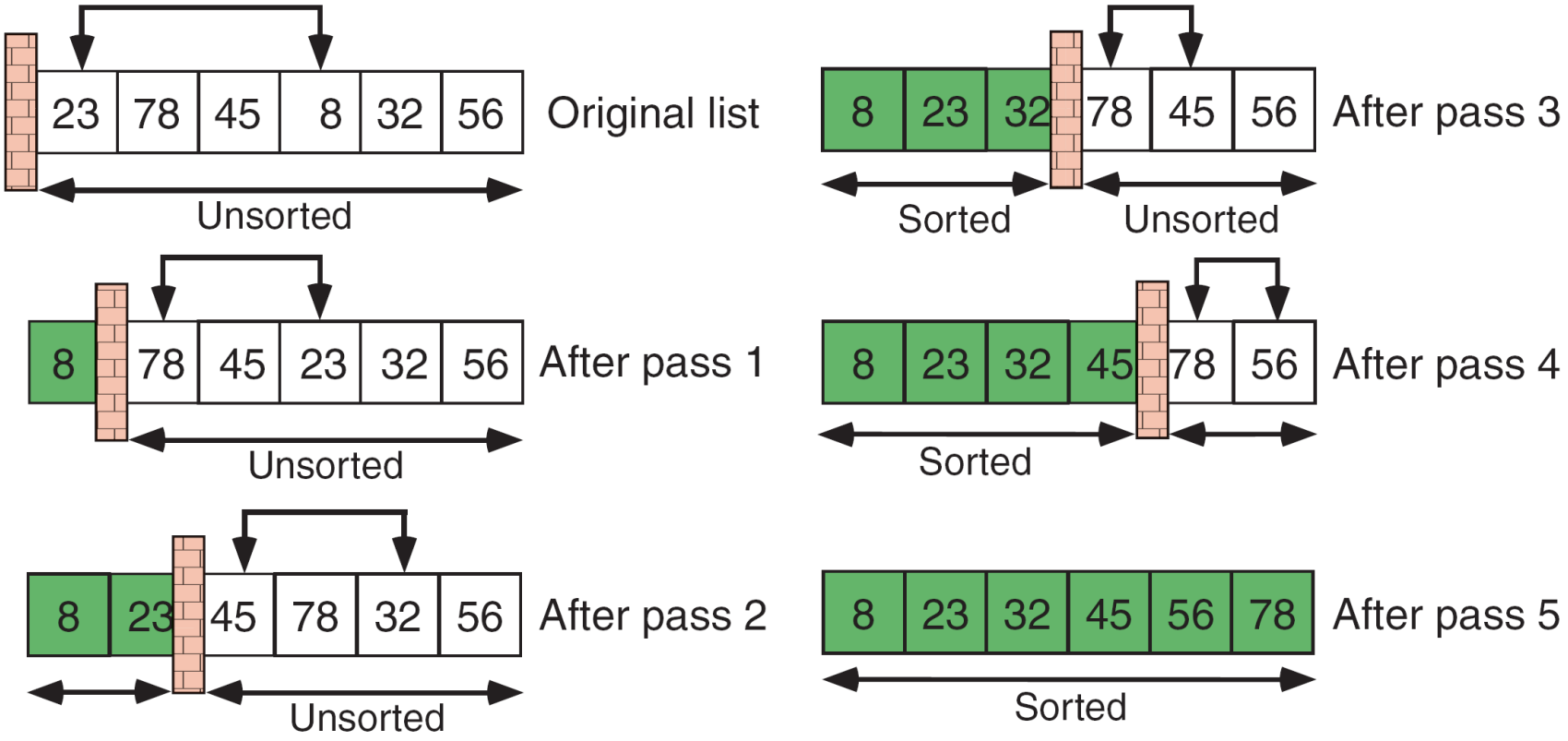


FIGURE 8-19 Selection Sort Example

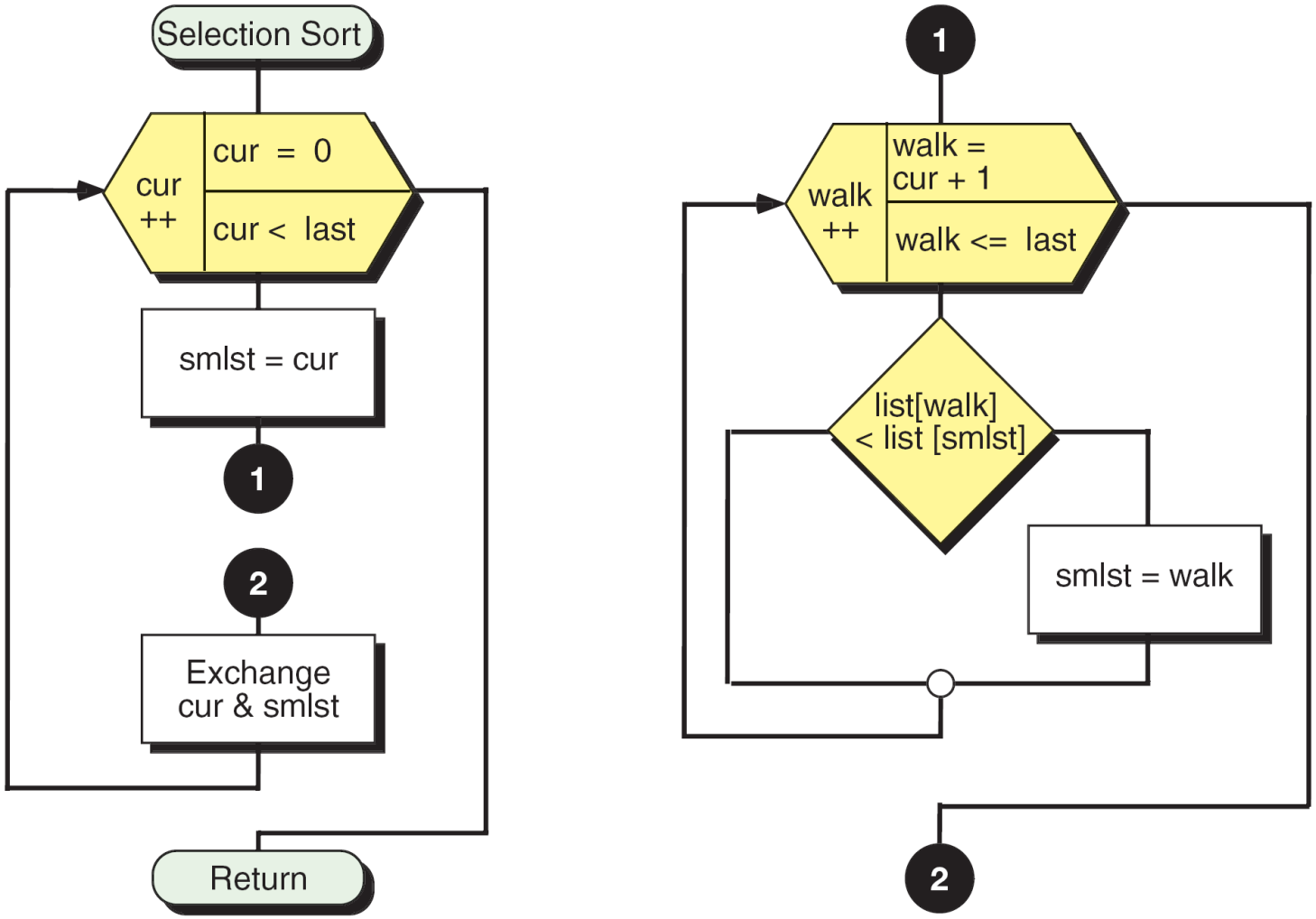


FIGURE 8-20 Design for Selection Sort

PROGRAM 8-9 Selection Sort

```
1  /* ================== selectionSort ====================
2     Sorts by selecting smallest element in unsorted
3     portion of array and exchanging it with element at
4     the beginning of the unsorted list.
5        Pre   list must contain at least one item
6              last contains index to last element in list
7        Post  list rearranged smallest to largest
8  */
9  void selectionSort (int list[], int last)
10 {
11 // Local Declarations
12    int smallest;
13    int tempData;
14
15 // Statements
16    // Outer Loop
17    for (int current = 0; current < last; current++)
18        {
19         smallest = current;
```

PROGRAM 8-9 Selection Sort

```
20          // Inner Loop: One sort pass each loop
21          for (int walk = current + 1;
22                   walk <= last;
23                   walk++)
24              if (list[walk] < list[smallest])
25                  smallest = walk;
26          // Smallest selected: exchange with current
27          tempData        = list[current];
28          list[current]   = list[smallest];
29          list[smallest] = tempData;
30          } // for current
31      return;
32  }  // selectionSort
```

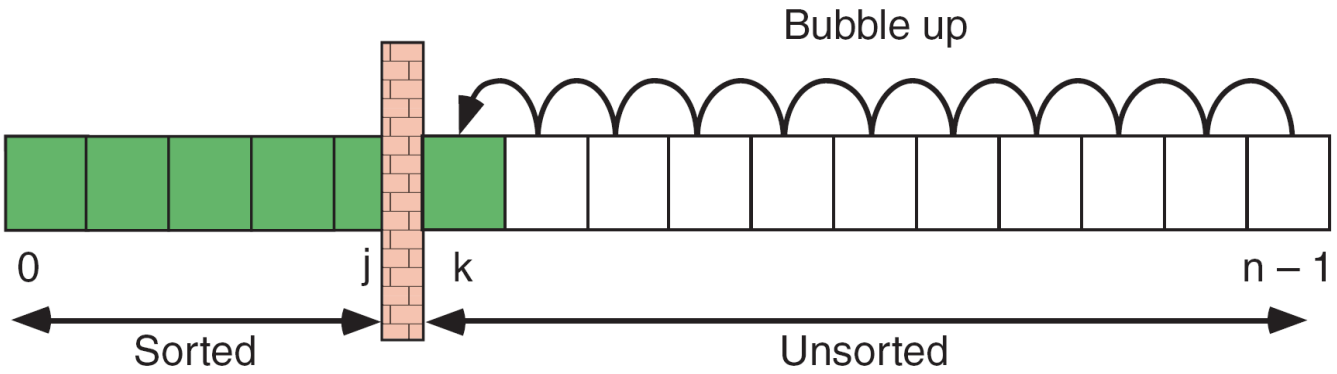


FIGURE 8-21 Bubble Sort Concept

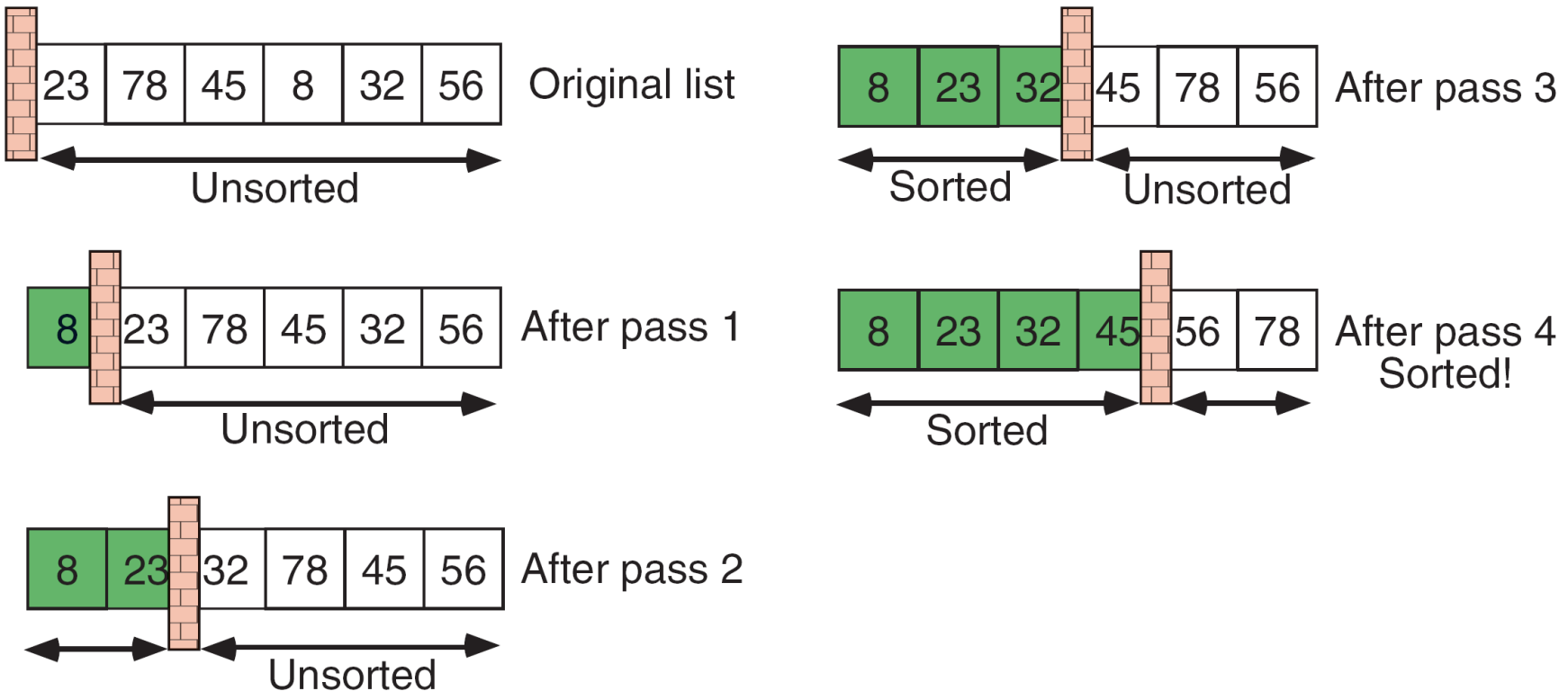


FIGURE 8-22 Bubble Sort Example

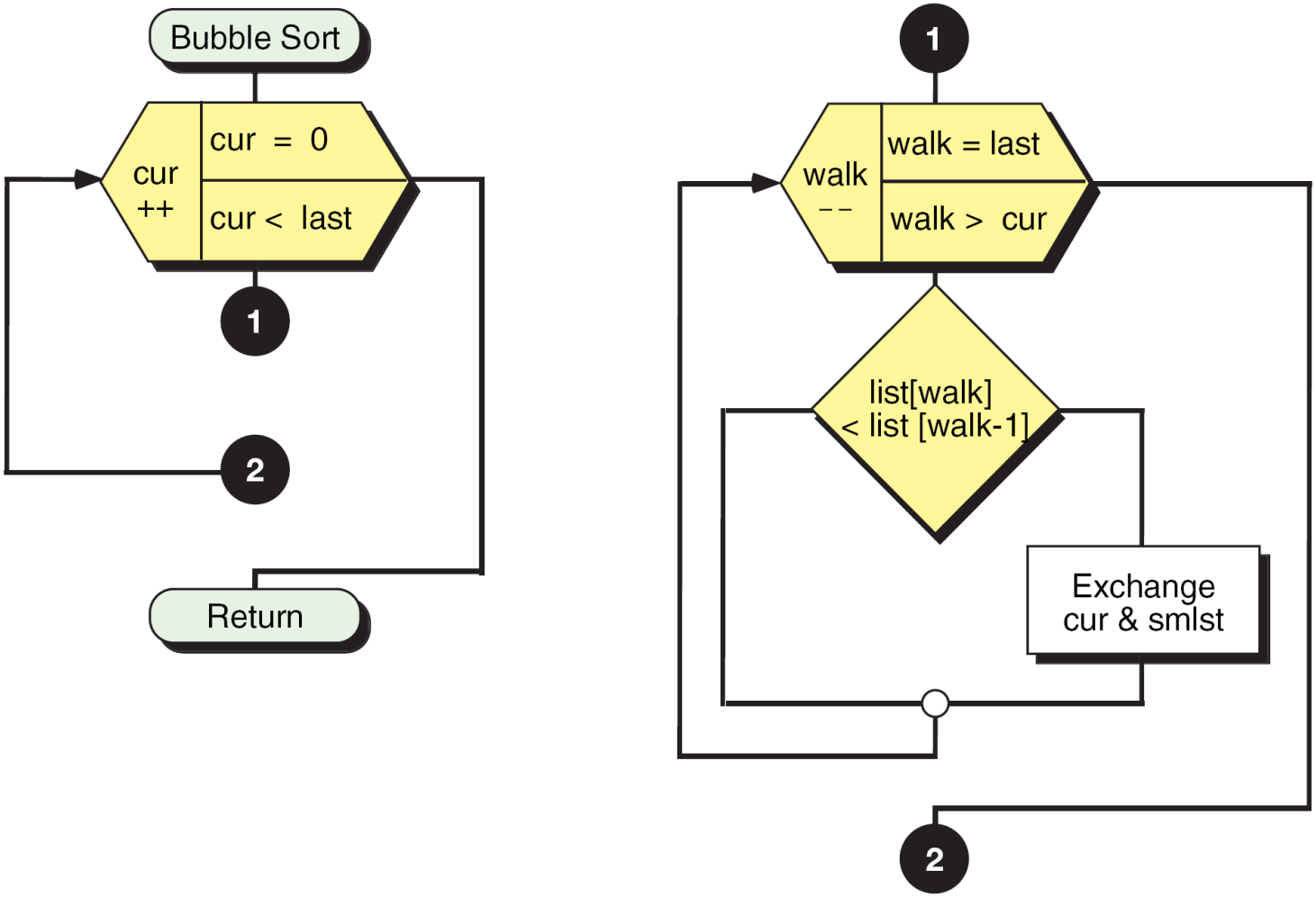


FIGURE 8-23 Bubble Sort Design

PROGRAM 8-10 Bubble Sort

```
1  /* ==================== bubbleSort ====================
2     Sort list using bubble sort. Adjacent elements are
3     compared and exchanged until list is ordered.
4        Pre  the list must contain at least one item
5             last contains index to last element in list
6        Post list rearranged in sequence low to high
7  */
8  void bubbleSort (int list [], int last)
9  {
10 // Local Declarations
11    int temp;
12
13 // Statements
14    // Outer loop
15    for(int current = 0; current < last; current++)
16       {
17        // Inner loop: Bubble up one element each pass
```

PROGRAM 8-10 Bubble Sort

```
18          for (int walker = last;
19                    walker > current;
20                    walker--)
21              if (list[walker]  < list[walker - 1])
22                 {
23                  temp              = list[walker];
24                  list[walker]      = list[walker - 1];
25                  list[walker - 1] = temp;
26                 } // if
27          } // for current
28      return;
29  }  // bubbleSort
```

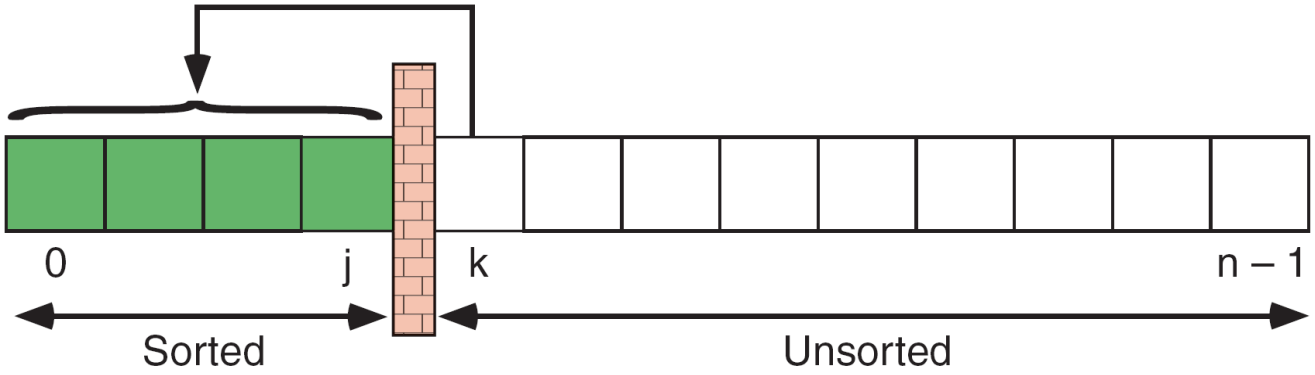


FIGURE 8-24 Insertion Sort Concept

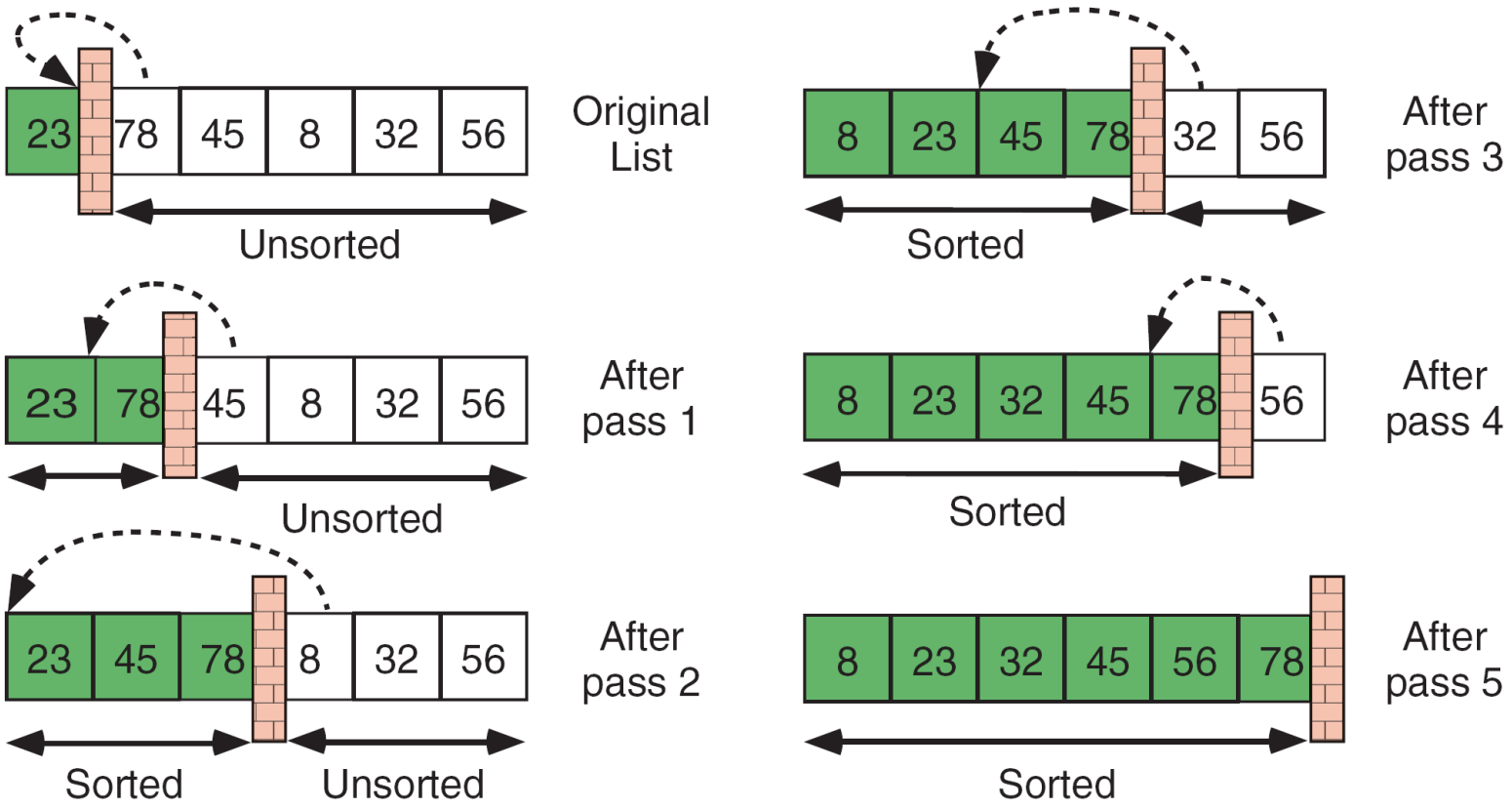


FIGURE 8-25 Insertion Sort Example

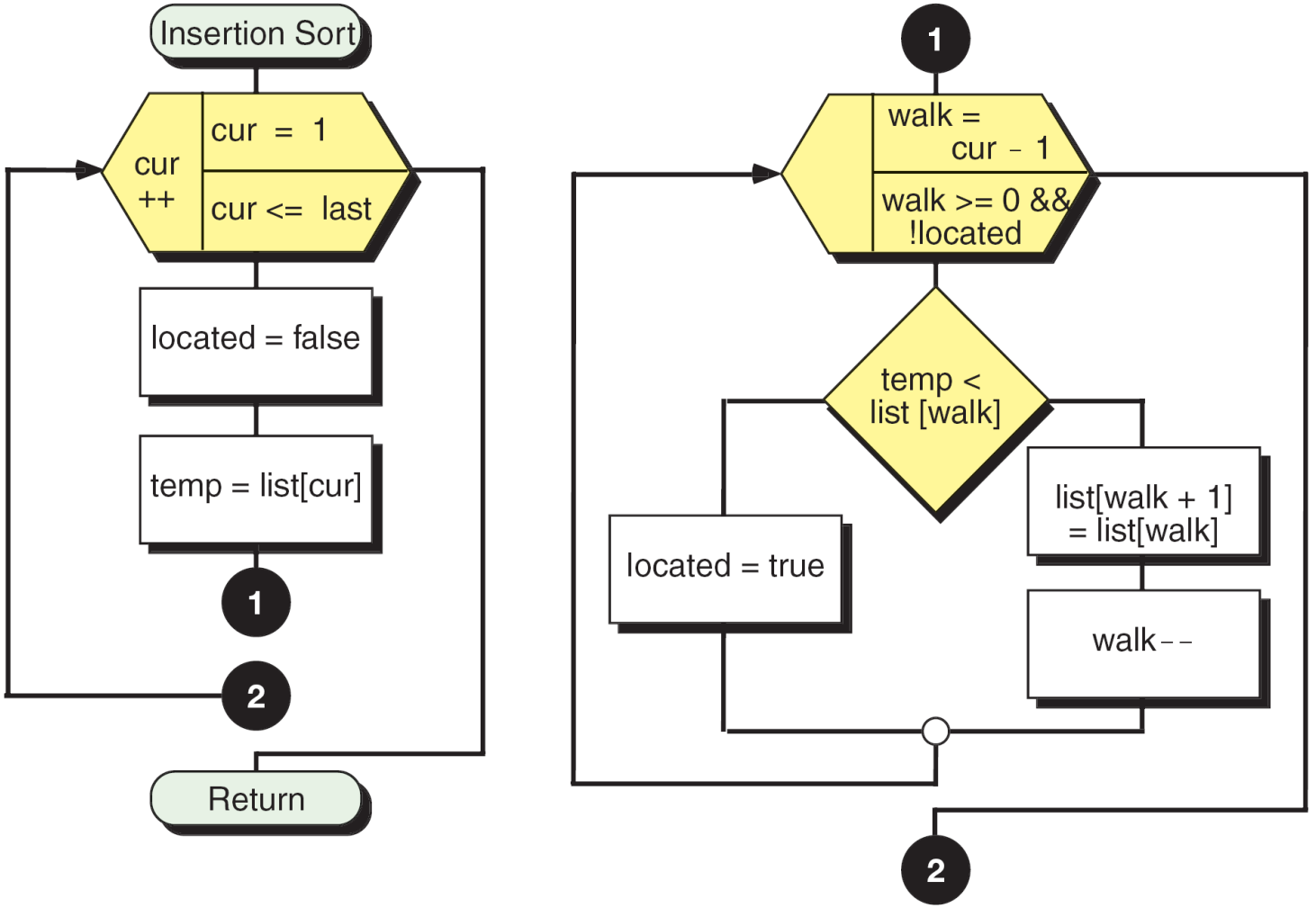


FIGURE 8-26 Insertion Sort Design

PROGRAM 8-11 Insertion Sort

```
1  /* ================== insertionSort ==================
2     Sort list using Insertion Sort. The list is divided
3     into sorted and unsorted lists. With each pass, first
4     element in unsorted list is inserted into sorted list.
5        Pre  list must contain at least one element
6             last contains index to last element in list
7        Post list has been rearranged
8  */
9  void insertionSort (int list[], int last)
10 {
11 // Local Declarations
12    int  walk;
13    int  temp;
14    bool located;
15
16 // Statements
17    // Outer loop
18    for (int current = 1; current <= last; current++)
19       {
```

PROGRAM 8-11 Insertion Sort

```
20          // Inner loop: Select and move one element
21          located = false;
22          temp = list[current];
23          for (walk = current - 1; walk >= 0 && !located;)
24             if (temp < list[walk])
25                {
26                 list[walk + 1] = list[walk];
27                 walk--;
28                } // if
29             else
30                 located = true;
31          list [walk + 1] = temp;
32        } // for
33     return;
34  }  // insertionSort
```

PROGRAM 8-12 Testing Sorts

```
1  /* Test driver for insertion sort.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <stdbool.h>
7  #include "P08-11.c"
8
9  #define MAX_ARY_SIZE 15
10
11 // Function Declarations
12 void insertionSort  (int list[], int last);
13
14 int main (void)
15 {
16 // Local Declarations
17    int ary[MAX_ARY_SIZE] = { 89, 72,  3, 15, 21,
18                              57, 61, 44, 19, 98,
19                               5, 77, 39, 59, 61 };
```

PROGRAM 8-12 Testing Sort

```
20  // Statements
21     printf("Unsorted: ");
22     for (int i = 0; i < MAX_ARY_SIZE; i++)
23          printf("%3d", ary[i]);
24
25     insertionSort (ary, MAX_ARY_SIZE - 1);
26
27     printf("\nSorted  : ");
28     for (int i = 0; i < MAX_ARY_SIZE; i++)
29          printf("%3d", ary[i]);
30     printf("\n");
31     return 0;
32  }  // main
```

```
Results:
Unsorted:  89 72  3 15 21 57 61 44 19 98  5 77 39 59 61
Sorted  :   3  5 15 19 21 39 44 57 59 61 61 72 77 89 98
```

Sort	Exchanges	Shifts
Selection	Once for each pass	
Bubble	Several in each pass	
Insertion	Only partially in loop	Zero or more in each pass

Table 8-1 Sort Exchanges and Passes

8-6 Searching

Another common operation in computer science is searching, which is the process used to find the location of a target among a list of objects. In the case of an array, searching means that given a value, we want to find the location (index) of the first element in the array that contains that value.

Topics discussed in this section:

Sequential Search
Binary Search

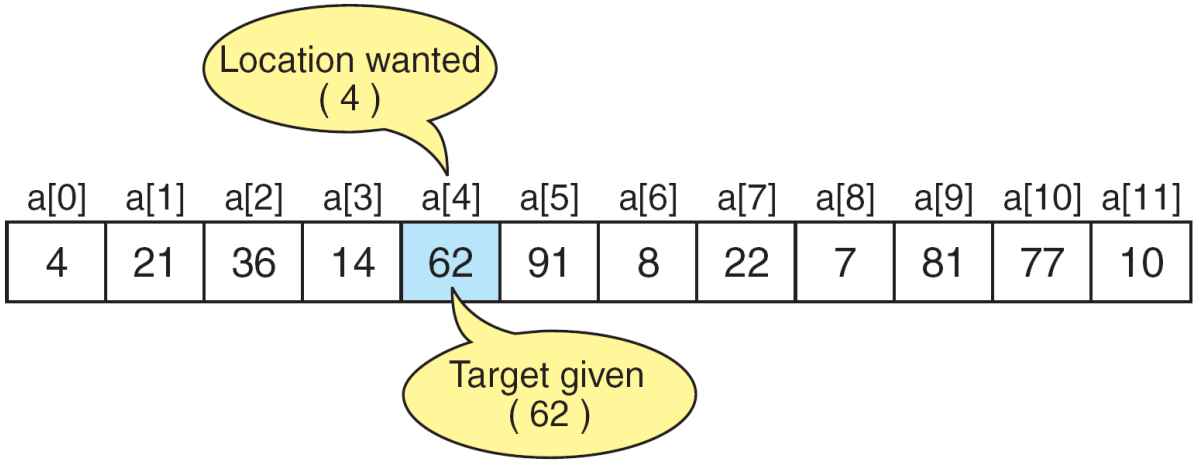


FIGURE 8-27 Search Concept

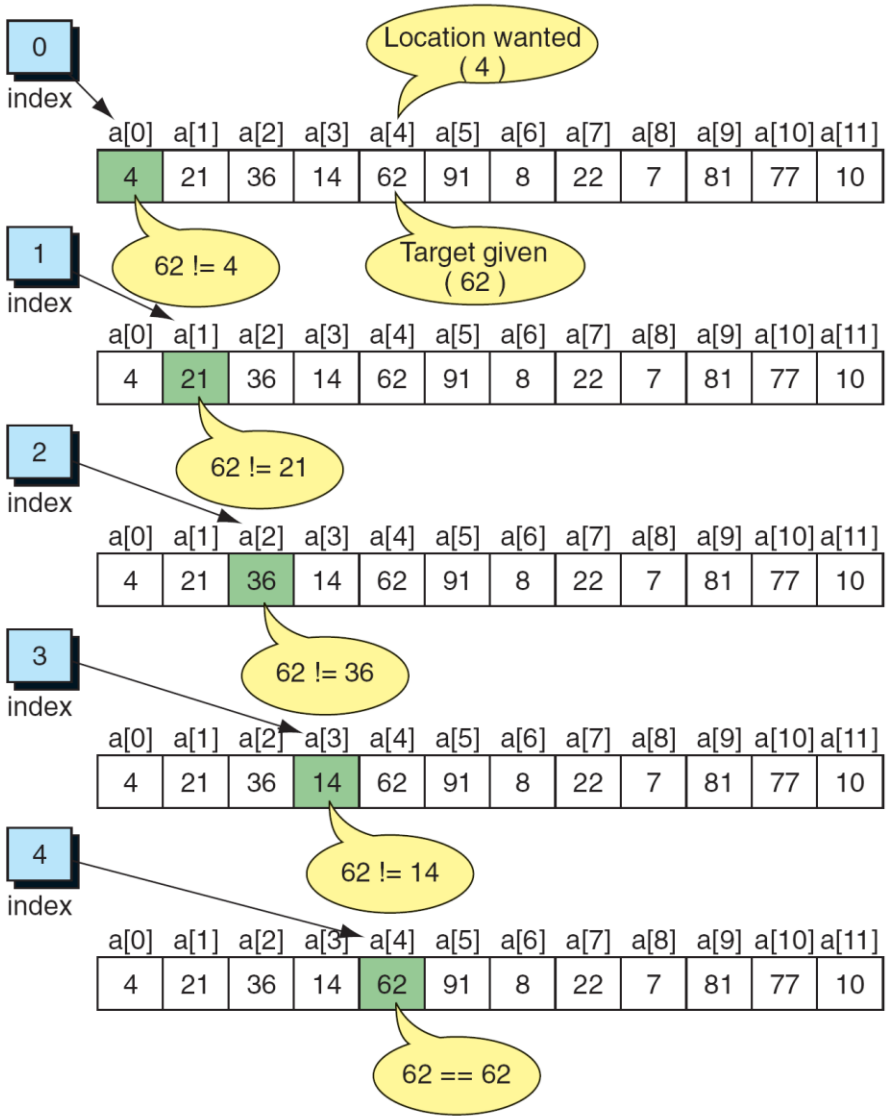


FIGURE 8-28 Locating Data in Unordered List

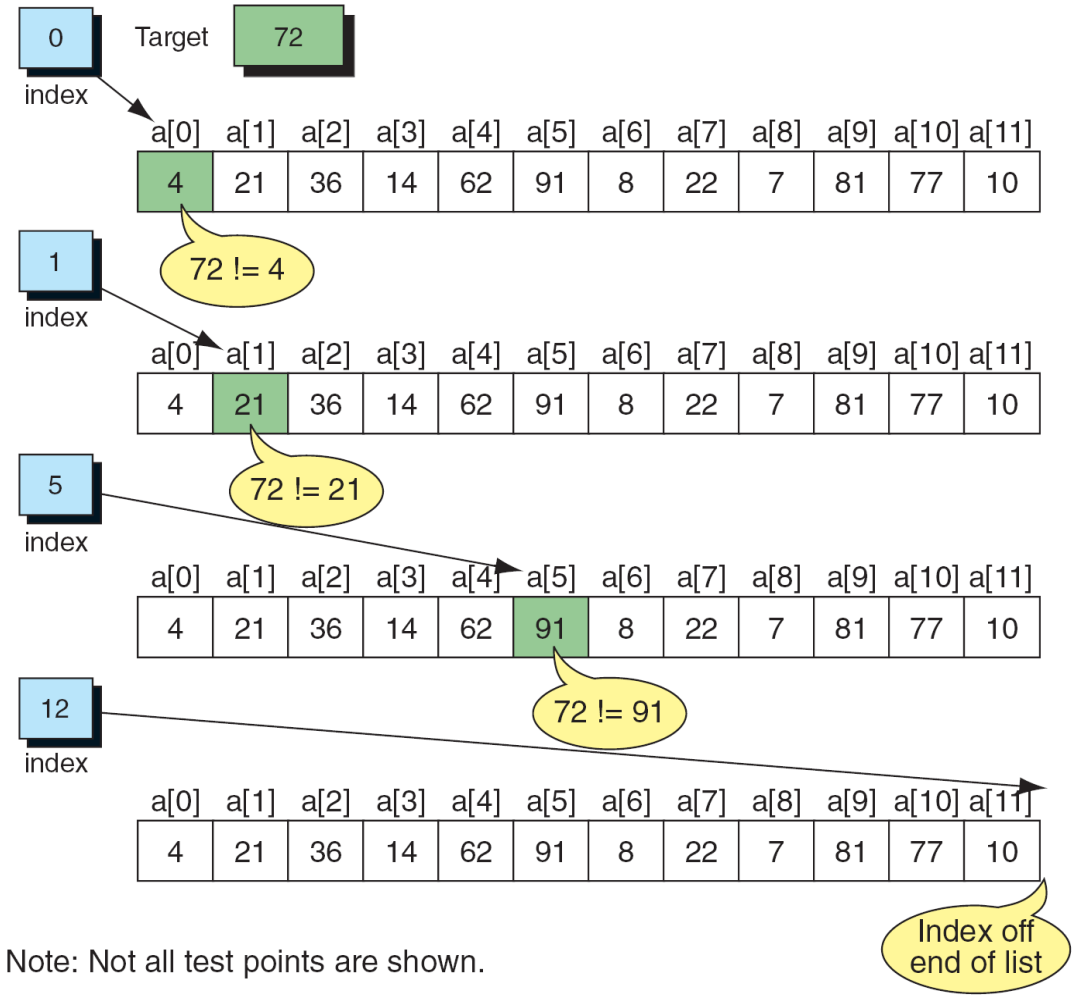


FIGURE 8-29 Unsuccessful Search in Unordered List

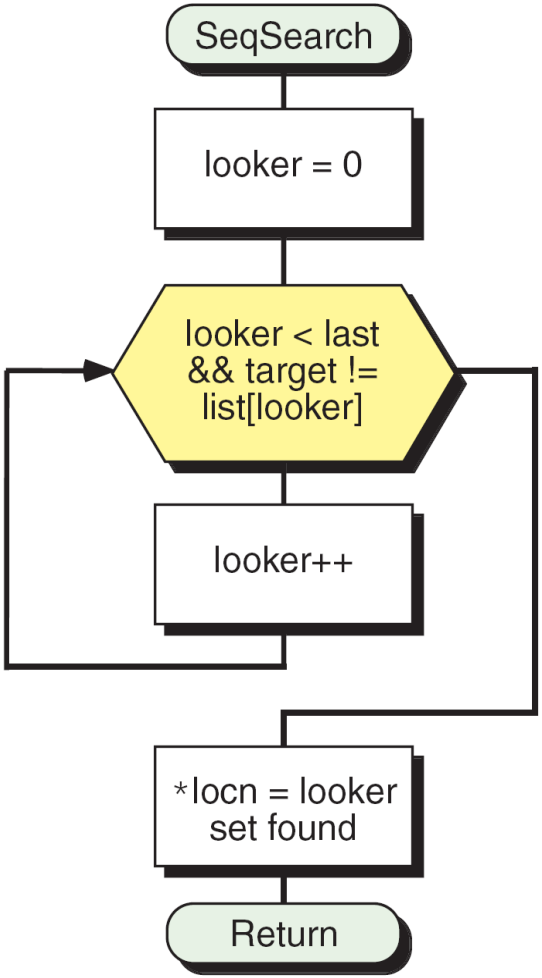


FIGURE 8-30 Sequential Search Design

PROGRAM 8-13 Sequential Search

```
1  /* ==================== seqSearch ====================
2     Locate target in an unordered list of size elements.
3        Pre   list must contain at least one item
4              last is index to last element in list
5              target contains the data to be located
6              locn is address for located target index
7        Post  Found: matching index stored in locn
8                     return true (found)
9              Not Found: last stored in locn
10                           return false (not found)
11 */
12 bool seqSearch (int list[], int  last,
13                  int target, int* locn)
14 {
15 // Local Declarations
16    int  looker;
17    bool found;
18
```

PROGRAM 8-13 Sequential Search

```
19  // Statements
20     looker = 0;
21     while (looker < last && target != list[looker])
22        looker++;
23
24     *locn = looker;
25     found = (target == list[looker]);
26     return found;
27  }  // seqSearch
```

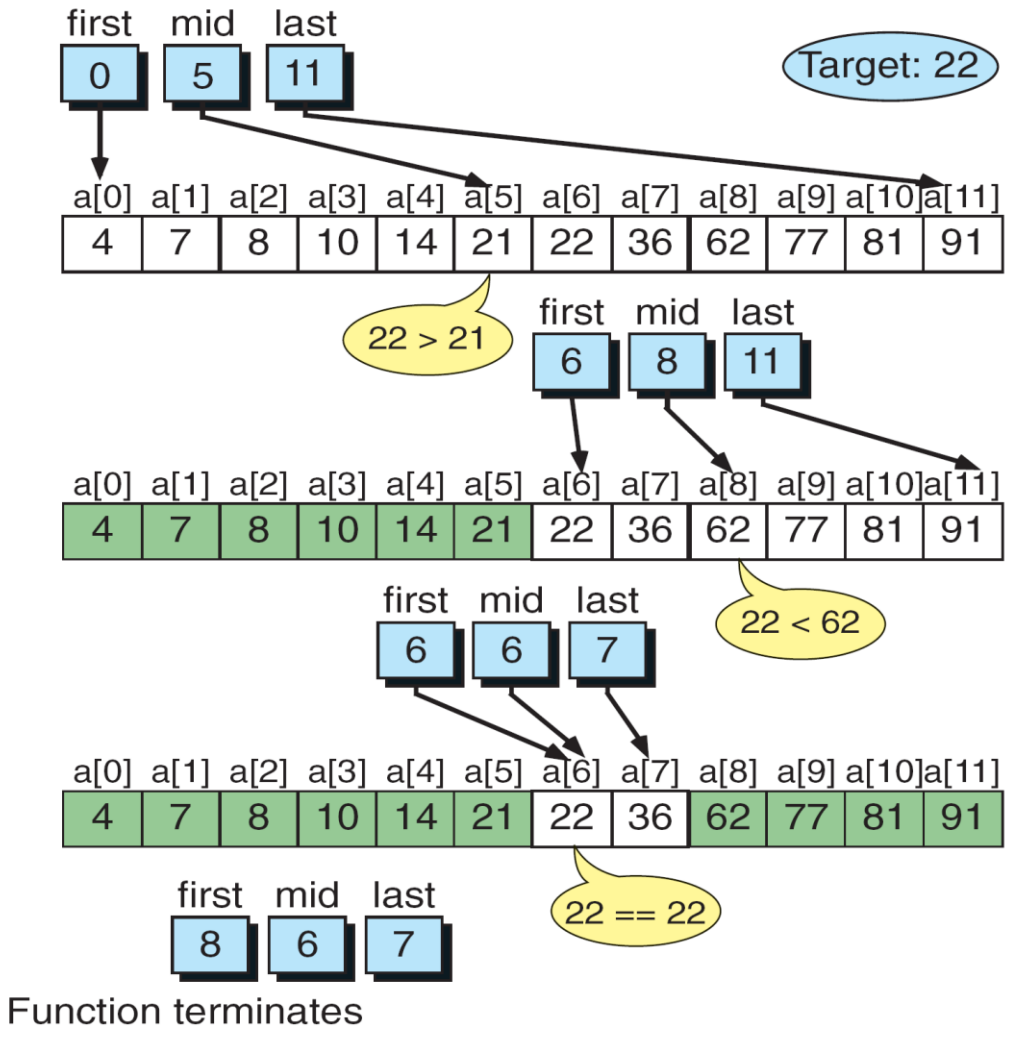


FIGURE 8-31 Binary Search Example

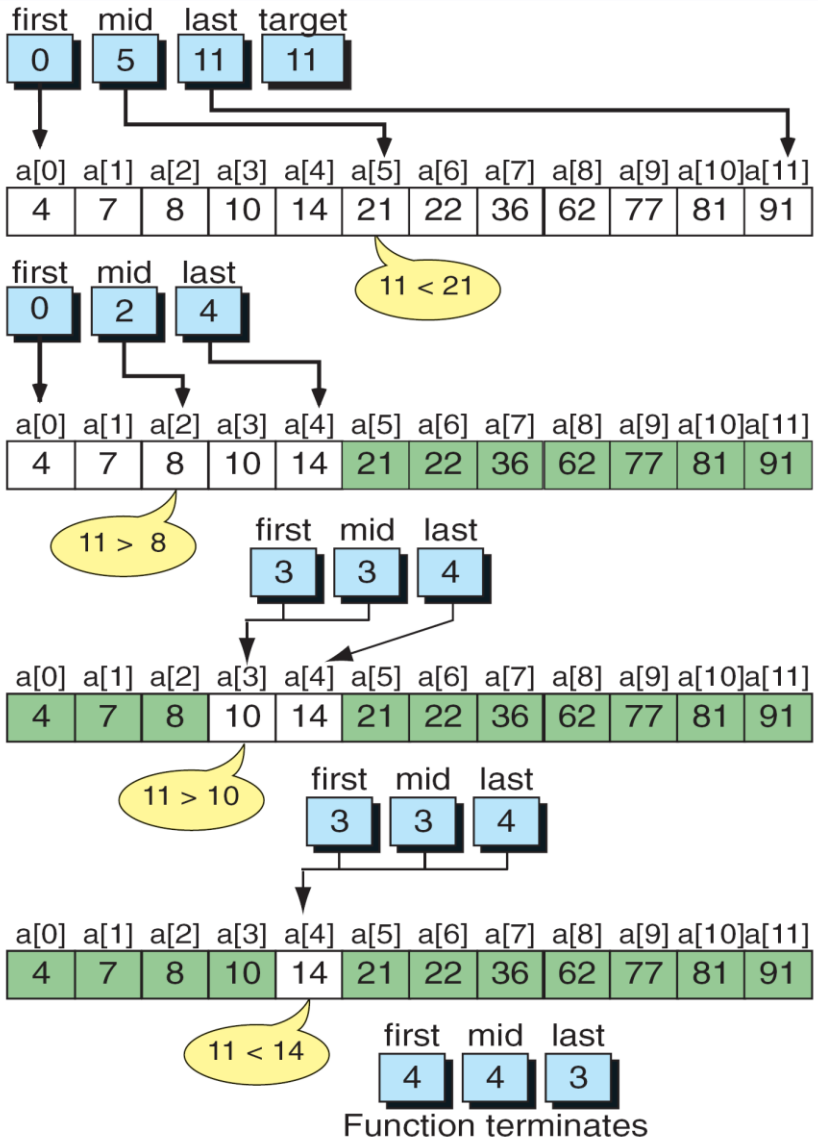


FIGURE 8-32 Unsuccessful Binary Search Example

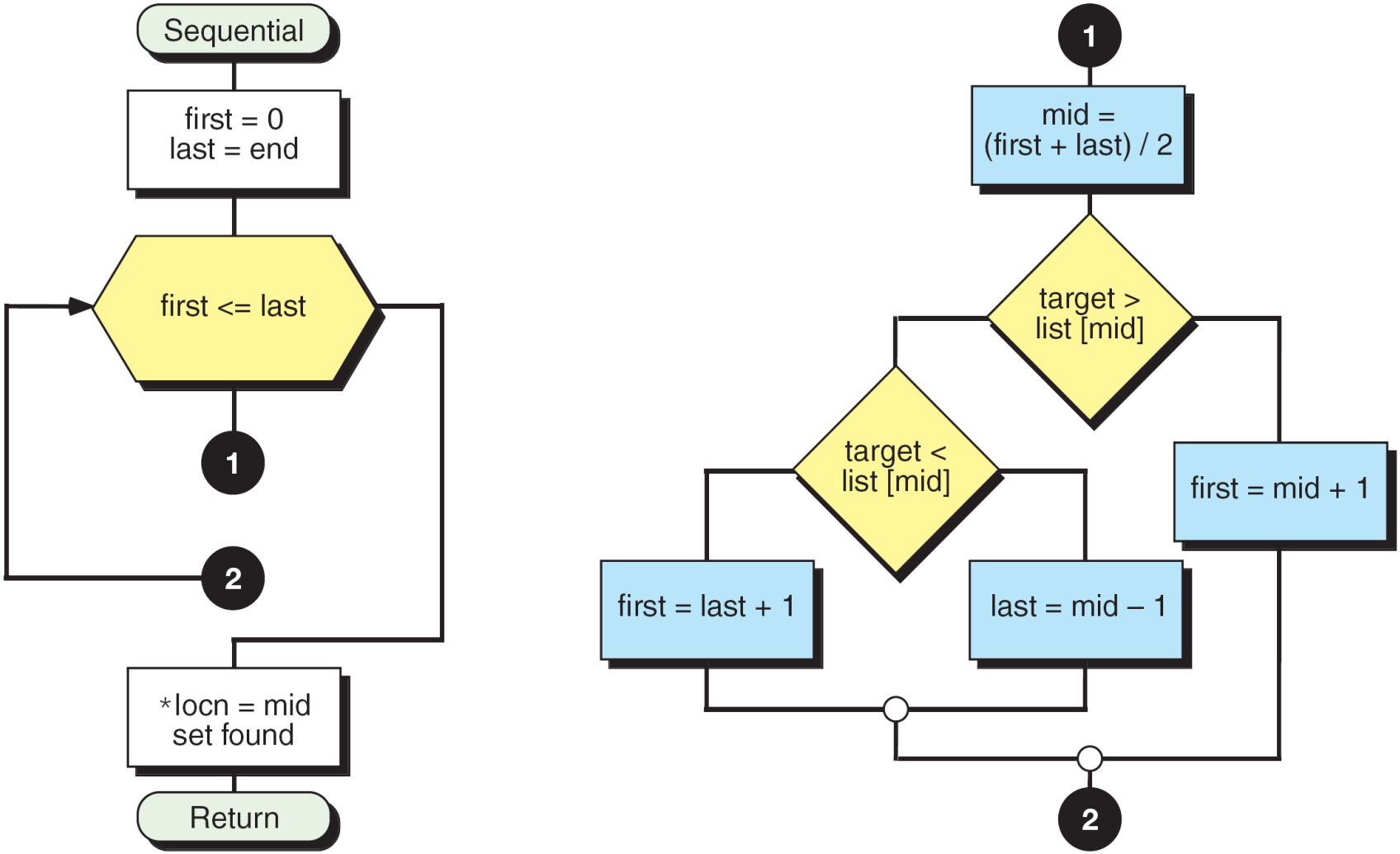


FIGURE 8-33 Design for Binary Search

PROGRAM 8-14 Binary Search

```
1  /* ==================== binarySearch ====================
2     Search an ordered list using Binary Search
3        Pre   list must contain at least one element
4              end is index to the largest element in list
5              target is the value of element being sought
6              locn is address for located target index
7        Post  Found: locn = index to target element
8                     return 1 (found)
9           Not Found: locn = element below or above target
10                          return 0 (not found)
11 */
12 bool binarySearch (int list[], int  end,
13                     int target, int* locn)
14 {
15 // Local Declarations
16    int first;
17    int mid;
18    int last;
19
```

PROGRAM 8-14 Binary Search

```
20  // Statements
21      first = 0;
22      last = end;
23      while (first <= last)
24          {
25           mid = (first + last) / 2;
26           if (target > list[mid])
27              // look in upper half
28              first = mid + 1;
29           else if (target < list[mid])
30              // look in lower half
31              last = mid - 1;
32           else
33              // found equal: force exit
34              first = last + 1;
35          } // end while
36      *locn = mid;
37      return target == list [mid];
38  }   // binarySearch
```

8-7 Two-Dimensional Arrays

The arrays we have discussed so far are known as one-dimensional arrays because the data are organized linearly in only one direction. Many applications require that data be stored in more than one dimension. One common example is a table, which is an array that consists of rows and columns.

Topics discussed in this section:

Declaration

Passing A Two-Dimensional Array

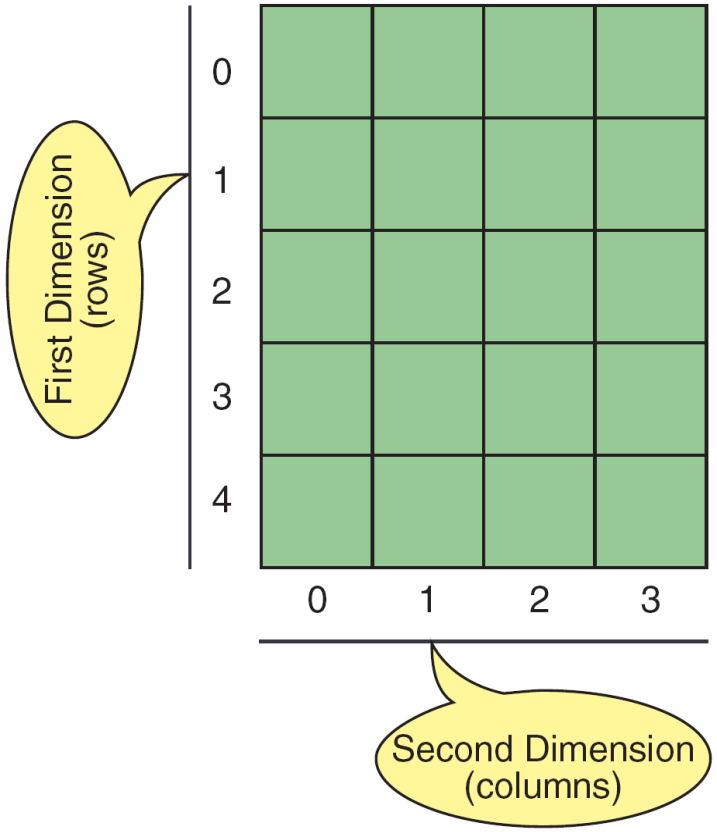


FIGURE 8-34 Two-dimensional Array

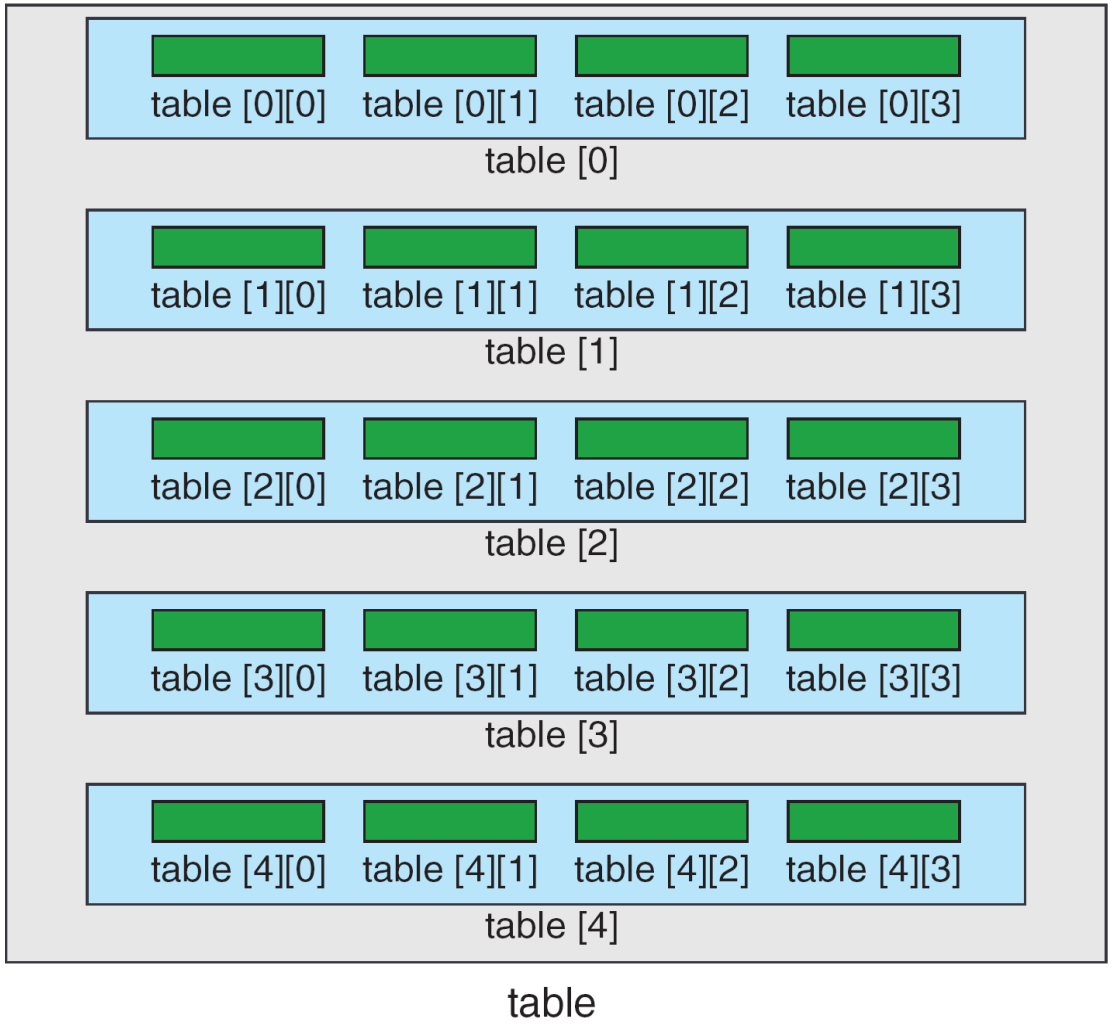


FIGURE 8-35 Array Of Arrays

Declaration

- int table[5][4];
- int table[3][2] = {0, 1, 2, 3, 4, 5};
- int table[3][2] = {
  {0, 1}, {2, 3}, {4, 5} };
- int table[][2] = {{0, 1}, {2, 3}, {4, 5} };
- int table[3][2] = {0};
- table[r][w]
- &table[r][w]

PROGRAM 8-15 Fill Two-dimensional Array

```
1  /* ==================== fillArray ====================
2     This function fills array such that each array element
3     contains a number that, when viewed as a two-digit
4     integer, the first digit is the row number and the
5     second digit is the column number.
6        Pre  table is array in memory
7             numRows is number of rows in array
8        Post array has been initialized
9  */
10 void fillArray (int table[][MAX_COLS], int numRows)
11 {
12 // Statements
13    for (int row = 0; row < numRows; row++)
14       {
15        table [row][0] = row * 10:
16        for (int col = 1; col < MAX_COLS; col++)
17            table [row][col] = table [row][col - 1] + 1;
18       } // for
19    return;
20 }  // fillArray
```

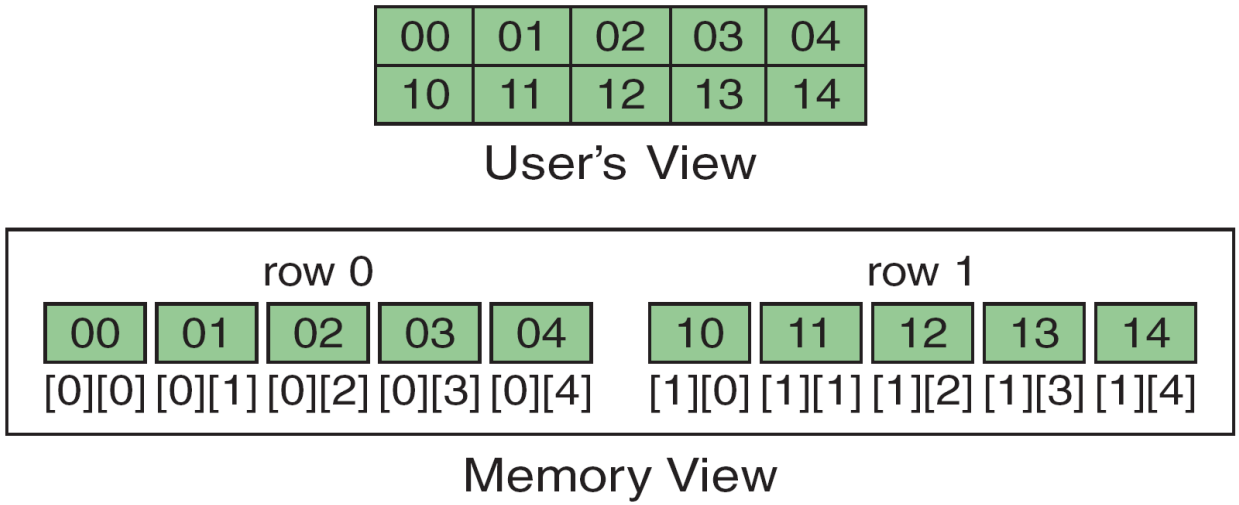


FIGURE 8-36 Memory Layout

PROGRAM 8-16 Convert Table to One-dimensional Array

```
1  /* This program changes a two-dimensional array to the
2     corresponding one-dimensional array.
3        Written by:
4        Date:
5  */
6  #include <stdio.h>
7  #define ROWS 2
8  #define COLS 5
9
10 int main (void)
11 {
12 // Local Declarations
13    int table [ROWS] [COLS] =
14       {
15        {00, 01, 02, 03, 04},
16        {10, 11, 12, 13, 14}
17       }; // table
18    int line [ROWS * COLS];
19
```

PROGRAM 8-16 Convert Table to One-dimensional Array

```
20  // Statements
21     for (int row = 0; row < ROWS; row++)
22        for (int column = 0; column < COLS; column++)
23           line[row * COLS + column] = table[row][column];
24
25     for (int row = 0; row < ROWS * COLS; row++)
26         printf(" %02d ", line[row]);
27
28     return 0;
29  }  // main
```

```
Results:
00   01   02   03   04   10   11   12   13   14
```

Passing a 2D Array

- Passing a row
  - void print_square (int []);
  - int table[MAX_ROWS][MAX_COLS];
  - print_square(table[row]);
- Passing the whole array
  - double average (int table[][MAX_COLS]);
  - int table[MAX_ROWS][MAX_COLS];
  - avg = average(table);

```c
#define MAX_ROWS   5
#define MAX_COLS   4
// Function Declarations
void print_square (int []);
int main (void)
{
  int table [MAX_ROWS][MAX_COLS] =
              {
               { 0, 1, 2, 3 },
               { 10, 11, 12, 13 },
               { 20, 21, 22, 23 },
               { 30, 31, 32, 33 },
               { 40, 41, 42, 43 }
              }; /* table */
  …
  for (int row = 0; row < MAX_ROWS; row++)
      print_square (table [row]);
  …
  return 0;
} // main
```

```c
void print_square (int x[])
{
   for (int col = 0; col < MAX_COLS; col++)
        printf("%6d", x[col] * x[col]);
   printf ("\n");
   return;
} // print_square
```

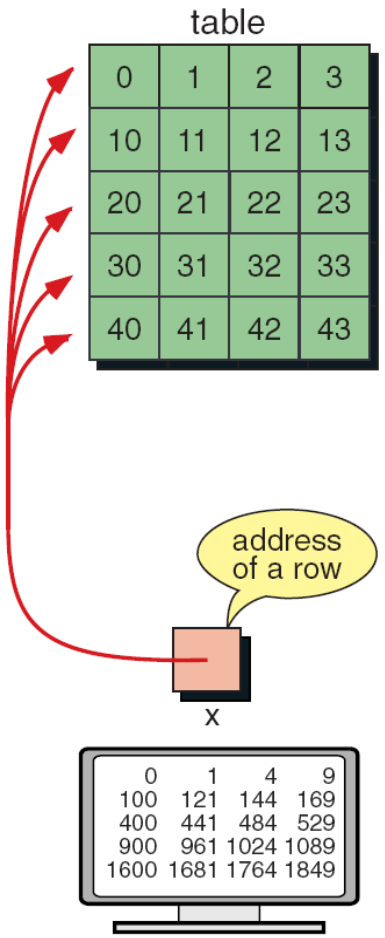


FIGURE 8-37 Passing a Row

0	1	1	1	1	1
-1	0	1	1	1	1
-1	-1	0	1	1	1
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	1
-1	-1	-1	-1	-1	0

FIGURE 8-39 Example of Filled Matrix

PROGRAM 8-17 Fill Matrix

```
1  /* This program fills the diagonal of a matrix (square
2     array) with 0, the lower left triangle with -1, and
3     the upper right triangle with 1.
4        Written by:
5        Date:
6  */
7  #include <stdio.h>
8
9  int main (void)
10 {
11 // Local Declarations
12    int table [6][6];
13
14 // Statements
15    for (int row = 0; row < 6; row++)
16       for (int column = 0; column < 6; column++)
```

PROGRAM 8-17 Fill Matrix

```
17            if (row ==  column)
18                table [row][column] = 0;
19            else if (row > column)
20                table [row][column] = -1;
21            else
22                table [row][column] = 1;
23
24      for (int row = 0; row < 6; row++)
25          {
26           for (int column = 0; column < 6; column++)
27              printf("%3d", table[row][column]);
28           printf("\n");
29          } // for row
30      return 0;
31  }  // main
```

8-8 Multidimensional Arrays

Multidimensional arrays can have three, four, or more dimensions. The first dimension is called a plane, which consists of rows and columns. The C language considers the three-dimensional array to be an array of two-dimensional arrays.

Topics discussed in this section:

Declaring Multidimensional Arrays

Declaring Multi-D Arrays

- int table[3][5][4];
- int table[planes][rows][cols];

- Initialization?

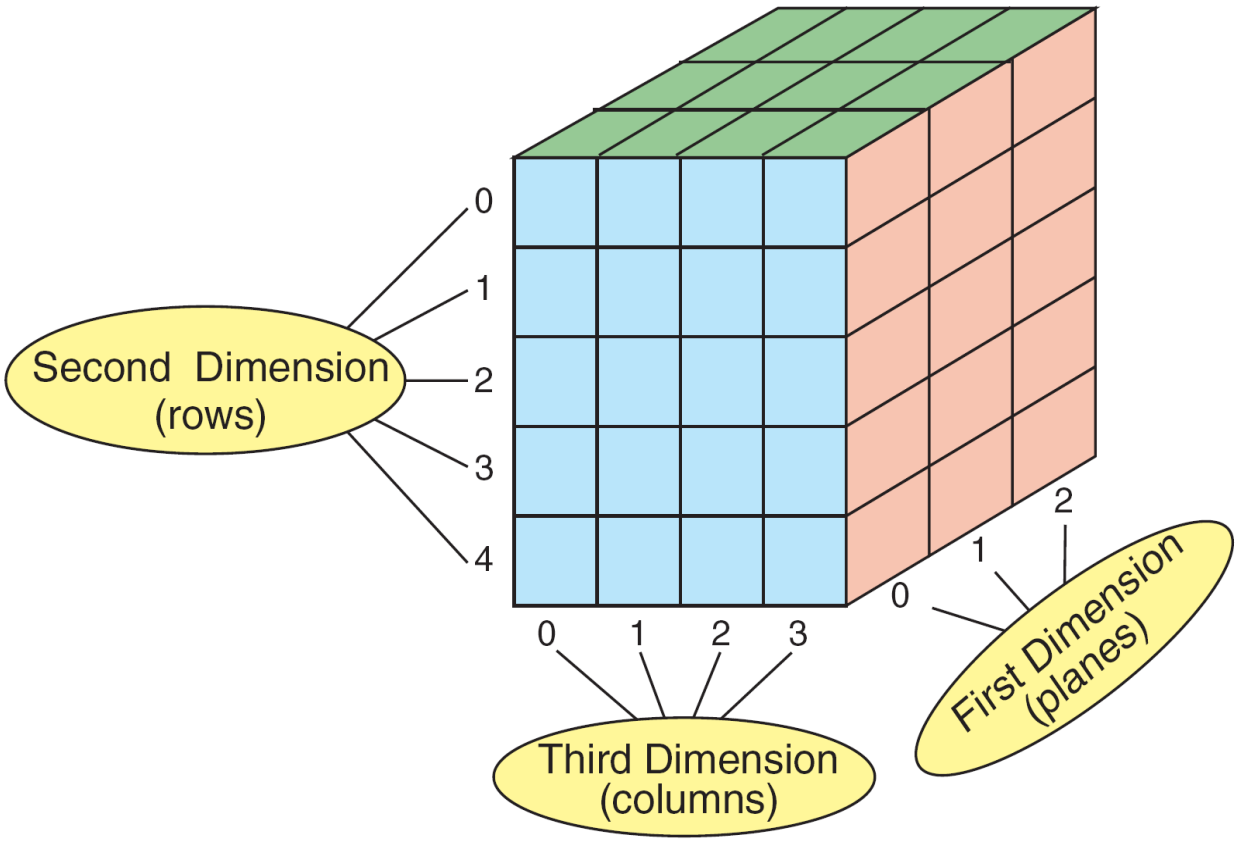


FIGURE 8-40 A Three-dimensional Array (3 x 5 x 4)

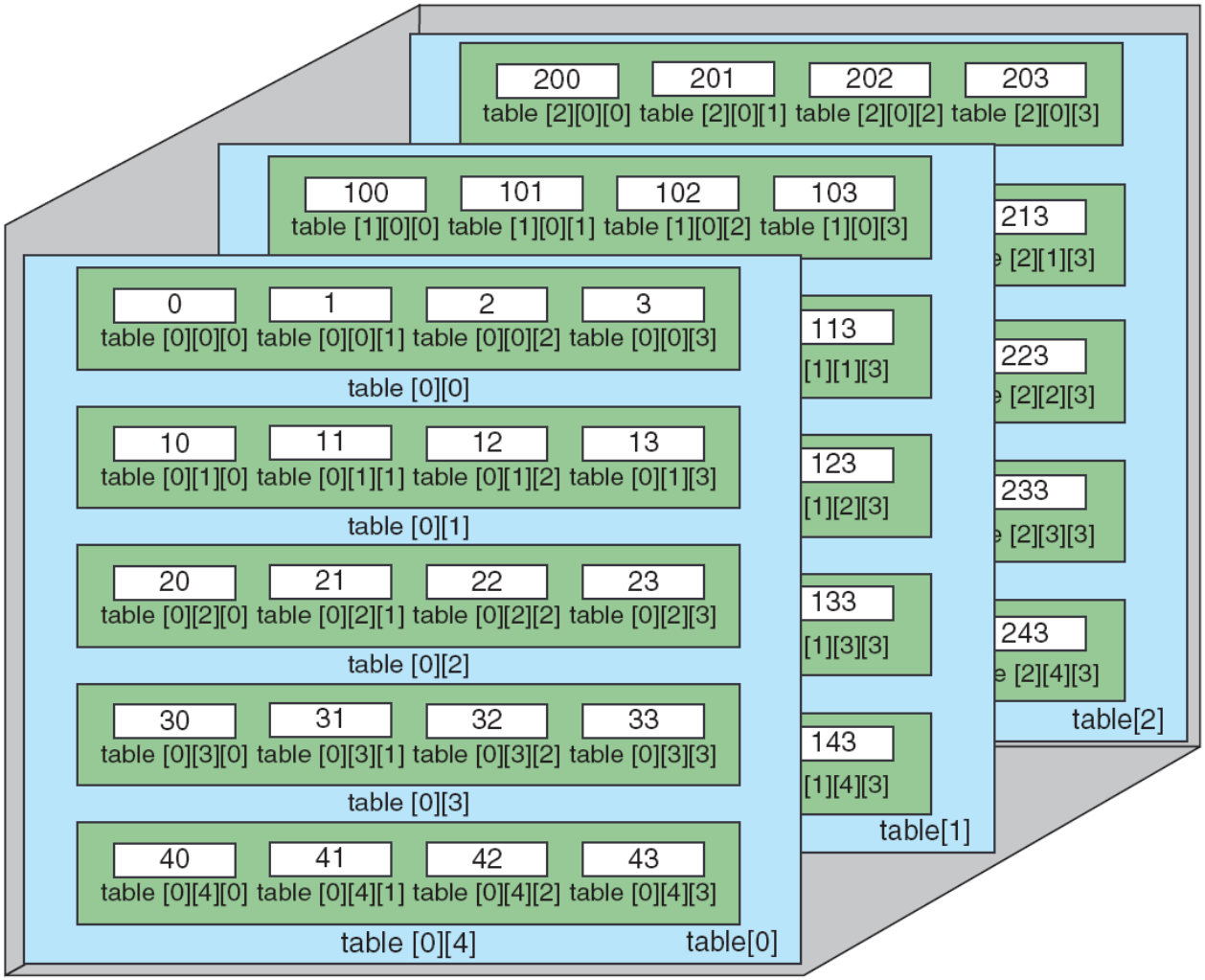


FIGURE 8-41 **C View of Three-dimensional Array**

8-9 Programming Example—Calculate Averages

We previously introduced the programming concept known as incremental development. In this chapter we develop an example—calculate average—that contains many of the programming techniques.

Topics discussed in this section:

First Increment: mainYour First C
Second Increment: Get Data
Third Increment: Calculate Row Averages
Fourth Increment: Calculate Column Averages
Fifth Increment: Print Tables

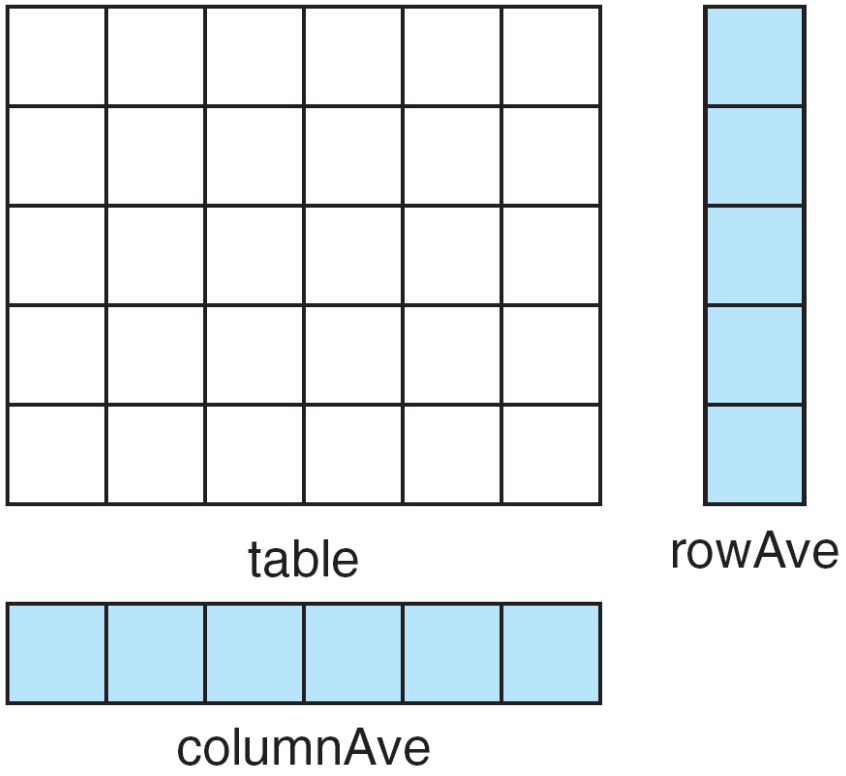


FIGURE 8-42 **Data Structures For Calculate Row–Column Averages**

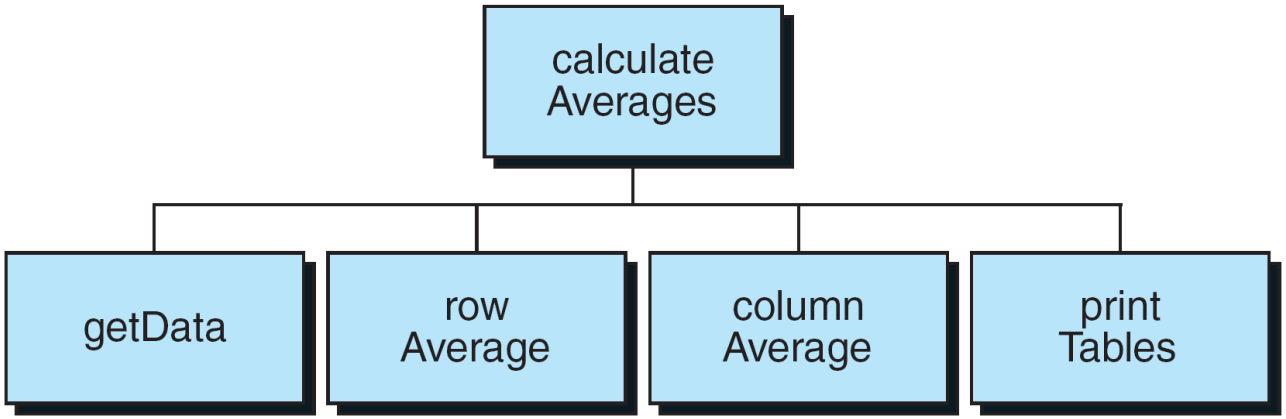


FIGURE 8-43 Calculate Row–Column Average Design

PROGRAM 8-18 Calculate Row and Column Averages: main

```
 1  /* Read values from keyboard into a two-dimensional
 2     array. Create two one-dimensional arrays that
 3     contain row and column averages.
 4        Written by:
 5        Date:
 6  */
 7  #include <stdio.h>
 8
 9  #define MAX_ROWS 5
10  #define MAX_COLS 6
11
12  int main (void)
13  {
14  // Local Declarations
15     int   table     [MAX_ROWS][MAX_COLS];
16
17     float rowAve    [MAX_ROWS] = {0};
18     float columnAve [MAX_COLS] = {0};
19
20  // Statements
21     return 0;
22  }  // main
```

PROGRAM 8-19 Calculate Row and Column Averages: Get Data

```
 1  /* Read values from keyboard into a two-dimensional
 2     array. Create two one-dimensional arrays that
 3     contain row and column averages.
 4        Written by:
 5        Date:
 6  */
 7  #include <stdio.h>
 8
 9  #define MAX_ROWS 5
10  #define MAX_COLS 6
11
12  // Function Declaration
13  void  getData       (int   table[][MAX_COLS]);
14
15  int main (void)
16  {
17  // Local Declarations
18     int   table     [MAX_ROWS][MAX_COLS];
```

PROGRAM 8-19 Calculate Row and Column Averages: Get Data

```
19
20      float rowAve    [MAX_ROWS] = {0};
21      float columnAve [MAX_COLS] = {0};
22
23  // Statements
24      getData         (table);
25
26      printf("\n**Tables built\n");
27      for (int i = 0; i < MAX_ROWS; i++)
28          {
29           for (int y = 0; y < MAX_COLS; y++)
30                printf("%4d", table[i][y]);
31           printf("\n");
32          } // for i
33
34      return 0;
35  }  // main
36
```

PROGRAM 8-19 Calculate Row and Column Averages: Get Data

```
37 /* ====================== getData ====================
38    Reads data and fills two-dimensional array.
39       Pre   table is empty array to be filled
40       Post  array filled
41 */
42 void getData (int table[][MAX_COLS])
43 {
44 // Statements
45    for (int row = 0; row < MAX_ROWS; row++)
46       for (int col = 0; col < MAX_COLS; col++)
47           {
48            printf("\nEnter integer and <return>: ");
49            scanf("%d", &table[row][col]);
50           } // for col
51    return;
52 } // getData
53 // ================ End of Program ================
```

PROGRAM 8-19 Calculate Row and Column Averages: Get Data

```
Results:
**Tables built
  10   12   14   16   18   20
  22   24   26   28   30   23
  25   27   29   31   33   35
  39   41   43   45   47   49
  51   53   55   57   59   61
```

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages

```
1  /* Read values from keyboard into a two-dimensional
2     array. Create two one-dimensional arrays that
3     contain row and column averages.
4        Written by:
5        Date:
6  */
7  #include <stdio.h>
8
9  #define MAX_ROWS 5
10 #define MAX_COLS 6
11
12 // Function Declaration
13 void getData    (int   table[][MAX_COLS]);
14 void rowAverage (int   table[][MAX_COLS],
15                  float rowAvrg []);
16
17 int main (void)
18 {
```

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages

```
19  // Local Declarations
20     int   table      [MAX_ROWS][MAX_COLS];
21
22     float rowAve     [MAX_ROWS] = {0};
23     float columnAve  [MAX_COLS] = {0};
24
25  // Statements
26     getData    (table);
27     rowAverage (table, rowAve);
28
29     printf("\n**Tables built\n");
30     for (int i = 0; i < MAX_ROWS; i++)
31         {
32          for (int y = 0; y < MAX_COLS; y++)
33               printf("%4d", table[i][y]);
34          printf("\n");
35         } // for i
36     printf("\n");
37
```

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages

```
38      printf("**Row averages\n");
39      for (int i = 0; i < MAX_ROWS; i++)
40           printf("%6.1f", rowAve[i]);
41      printf("\n");
42
43      return 0;
44  }   // main
45
46  /* ==================== getData ====================
47      Reads data and fills two-dimensional array.
48         Pre   table is empty array to be filled
49         Post  array filled
50  */
51  void getData (int table[][MAX_COLS])
52  {
53  // Statements
54      for (int row = 0; row < MAX_ROWS; row++)
55         for (int col = 0; col < MAX_COLS; col++)
56              {
57               printf("\nEnter integer and <return>: ");
58               scanf("%d", &table[row][col]);
59              } // for col
60      return;
61  } // getData
```

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages

```
62
63 /* ================== rowAverage ==================
64    This function calculates the row averages for a table
65       Pre   table has been filled with values
66       Post  averages calculated and in average array
67 */
68 void rowAverage (int    table[][MAX_COLS],
69                  float  rowAvrg [])
70 {
71 // Statements
72    for (int row = 0; row < MAX_ROWS; row++)
73        {
74         for (int col = 0; col < MAX_COLS; col++)
75             rowAvrg[row] += table [row][col];
76         rowAvrg [row] /=  MAX_COLS;
77        } // for row
78    return;
79 }  // rowAverage
80 // ================ End of Program ================
```

PROGRAM 8-20 Calculate Row and Column Averages: Row Averages

```
Results:
**Tables built
  10   12   14   16   18   20
  22   24   26   28   30   23
  25   27   29   31   33   35
  39   41   43   45   47   49
  51   53   55   57   59   61

**Row averages
  15.0   25.5   30.0   44.0   56.0
```

8-10 Software Engineering

In this section, we discuss two basic concepts: testing and algorithm efficiency. To be effective, testing must be clearly thought out. We provide some concepts for testing array algorithms by studying sorting and searching. We then continue the algorithm efficiency discussion started in Chapter 6.

Topics discussed in this section:

Testing Sorts
Testing Searches
Analyzing Sort Algorithms
Analyzing Search Algorithms

Test Case	Sample Data
Random data	5 23 7 78 22 6 19 33 51 11 93 31
Nearly ordered	5 6 7 21 19 22 23 31 29 33 51 93
Ordered — ascending	5 6 7 11 19 22 23 31 33 51 78 93
Ordered — descending	93 78 51 33 31 23 22 19 11 7 6 5

Table 8-2 Recommended Sort Test Cases

Note

When testing an array search, always access the first and last elements.

Expected Results	Index	Test	Search
found	0	target == list[0]	all
found	1	target == list[1]	binary only
found	*n* – 1	target == list[*n*–1]	all
found	0 < i < *n*	target == list[i]	all
not found	0	target < list[0]	binary only
not found	*n* – 1	target > list[n]	binary only
not found	0 < i < *n*	target != list[i]	all

Table 8-3 **Test cases for searches**

Note

The efficiency of the bubble sort is $O(n^2)$.

Note

The efficiency of the selection sort is $O(n^2)$.

Note

The efficiency of the insertion sort is O(n^2).

Note

The efficiency of the sequential search is O(*n*).

Note

The efficiency of the binary search is O(log*n*).

Size	Binary	Sequential (Average)	Sequential (Worst Case)
16	4	8	16
50	6	25	50
256	8	128	256
1,000	10	500	1,000
10,000	14	5,000	10,000
100,000	17	50,000	100,000
1,000,000	20	500,000	1,000,000

Table 8-4 **Comparison of binary and sequential searches**