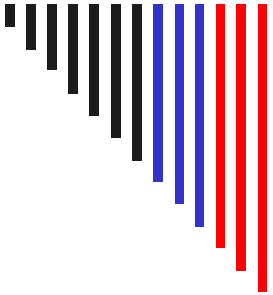


Chapter 10
Pointer Applications

Objectives

- To understand the relationship between arrays and pointers
- To understand the design and concepts behind pointer arithmetic
- To write programs using arrays and pointer arithmetic
- To better understand the design behind passing arrays to functions
- To understand the C implementation of dynamic memory
- To write programs using static and dynamic memory allocation
- To understand and implement ragged arrays (arrays of pointers)

10-1 Arrays and Pointers

The name of an array is a pointer constant to the first element. Because the array's name is a pointer constant, its value cannot be changed. Since the array name is a pointer constant to the first element, the address of the first element and the name of the array both represent the same location in memory.

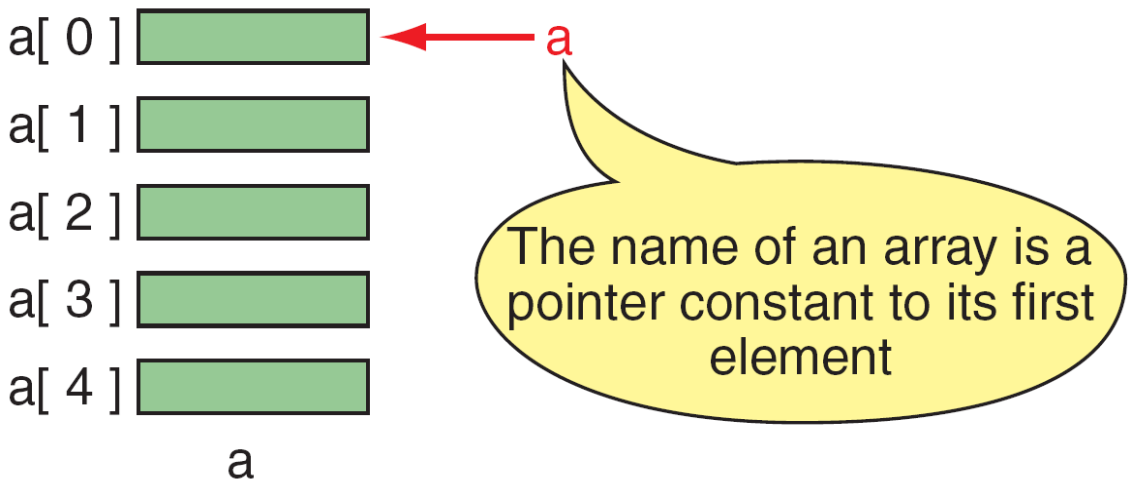


FIGURE 10-1 Pointers to Arrays

Note

same

a ⟷ &a[0]

***a* is a pointer only to the first element—not the whole array.**

Note

The name of an array is a pointer constant;
it cannot be used as an *lvalue*.

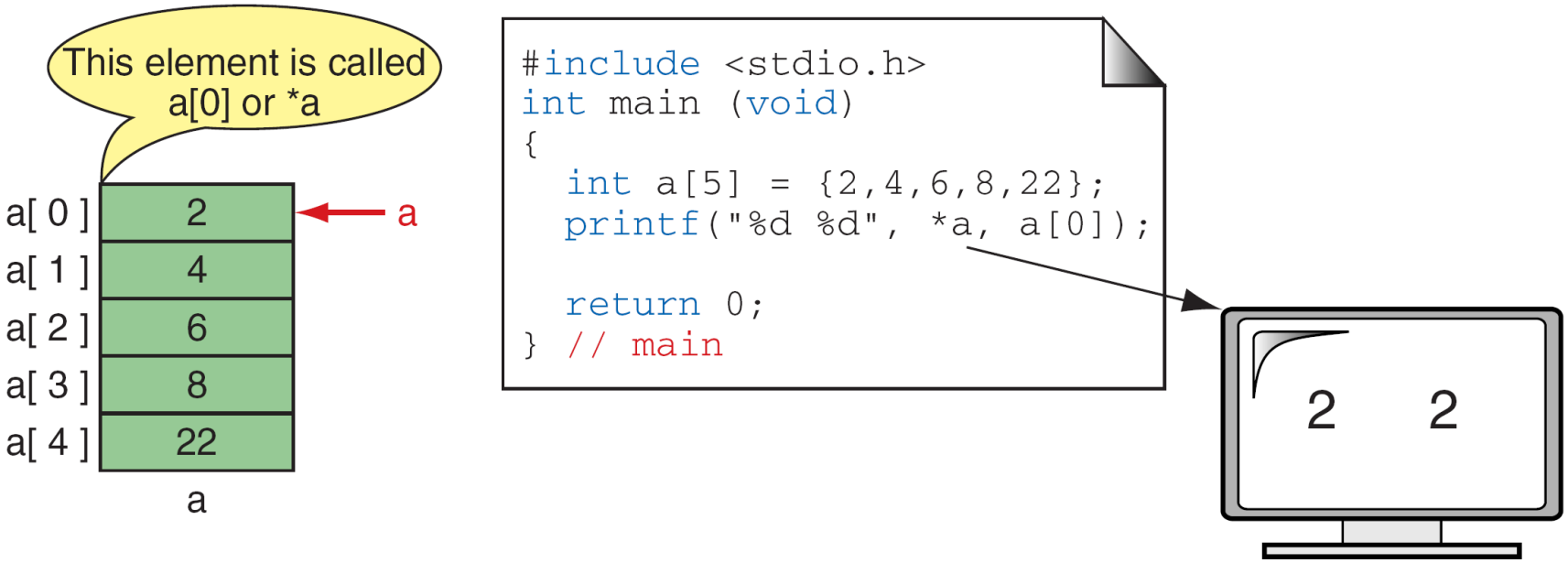


FIGURE 10-2 Dereference of Array Name

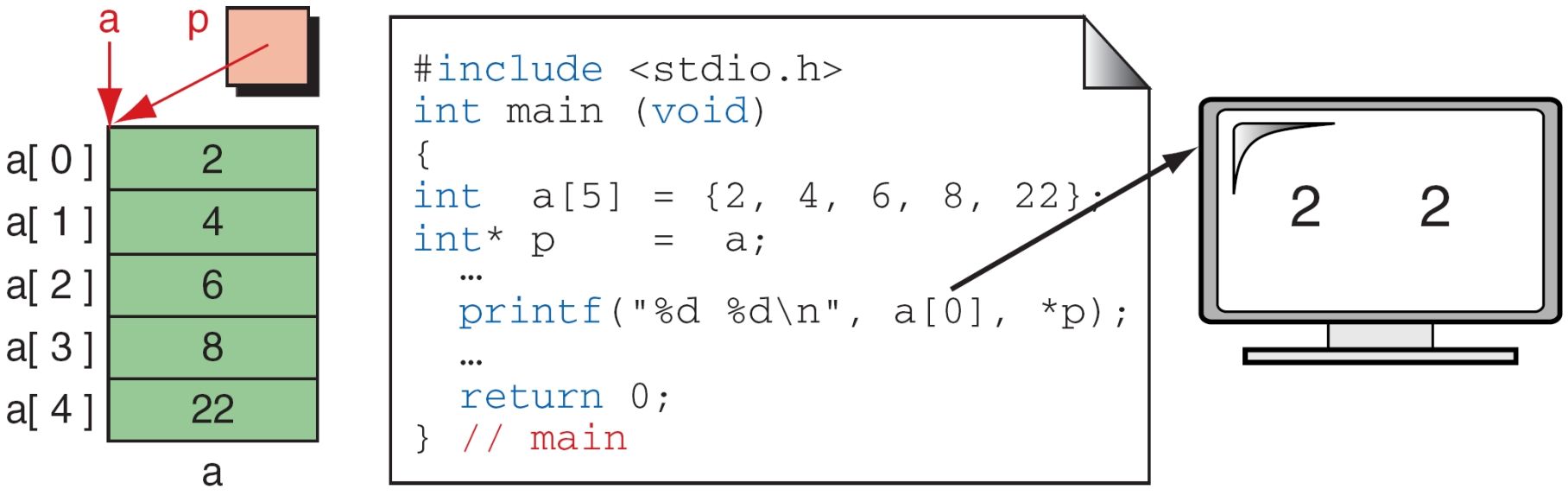


FIGURE 10-3 Array Names as Pointers

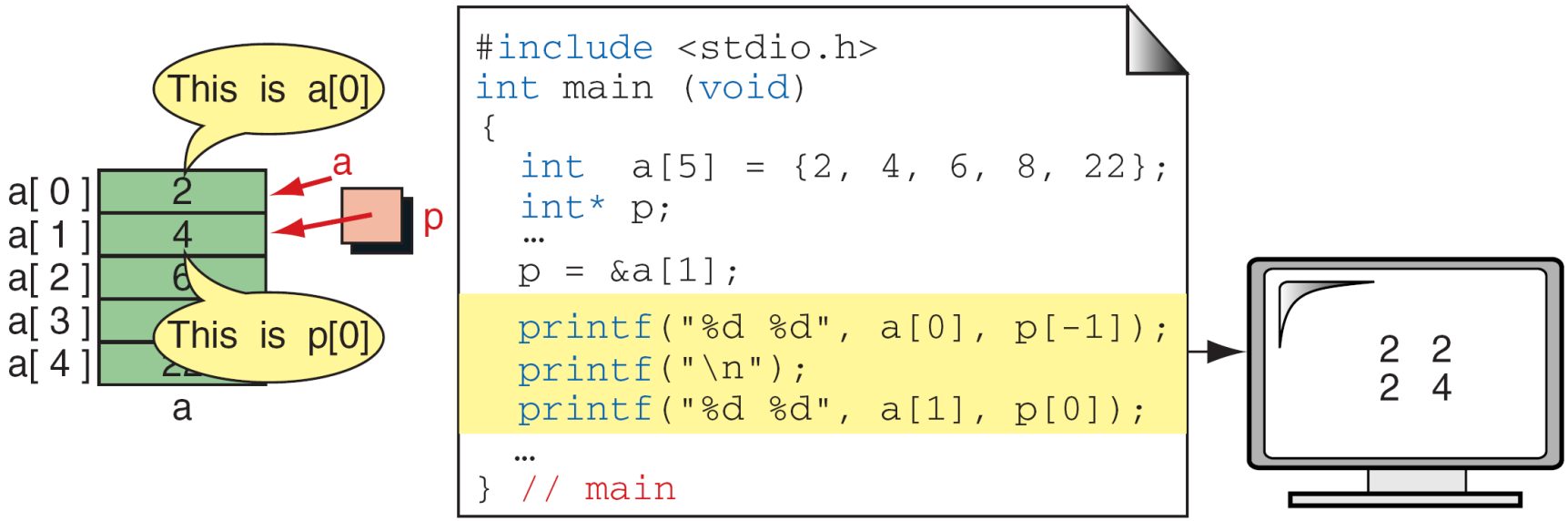


FIGURE 10-4 Multiple Array Pointers

Note

To access an array, any pointer to the first element can be used instead of the name of the array.

10-2 Pointer Arithmetic and Arrays

Besides indexing, programmers use another powerful method of moving through an array: pointer arithmetic. Pointer arithmetic offers a restricted set of arithmetic operators for manipulating the addresses in pointers.

Topics discussed in this section:

Pointers and One-Dimensional Arrays
Arithmetic Operations on Pointers
Using Pointer Arithmetic
Pointers and Two-Dimensional Arrays

Note

Given pointer, p, p ± n is a pointer to the value n elements away.

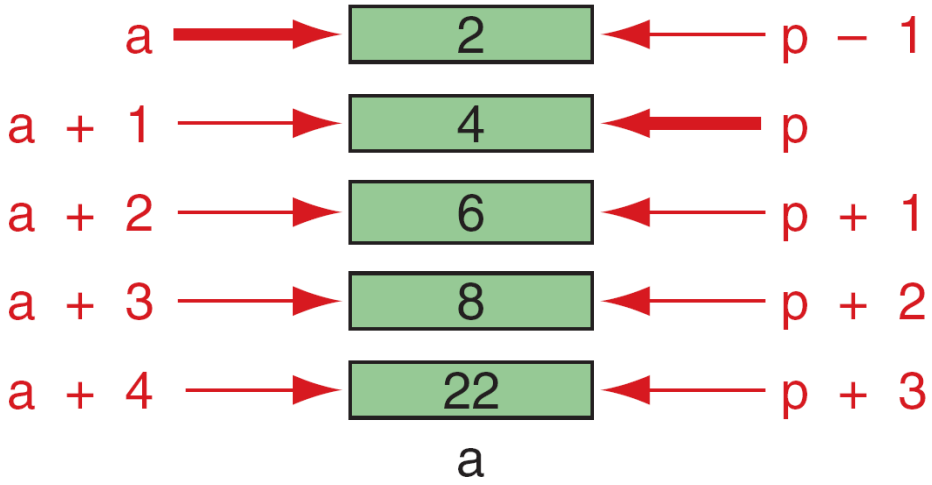
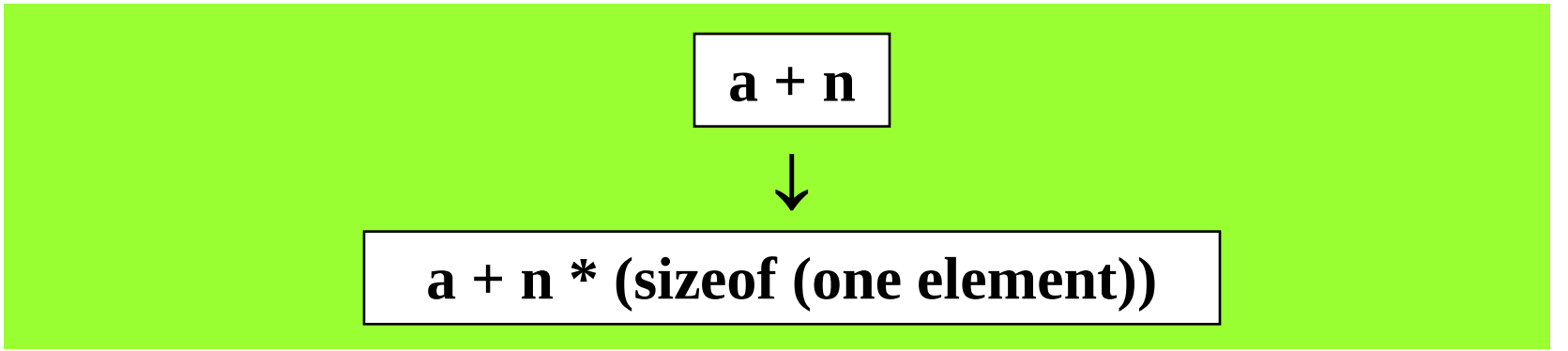


FIGURE 10-5 Pointer Arithmetic

Note



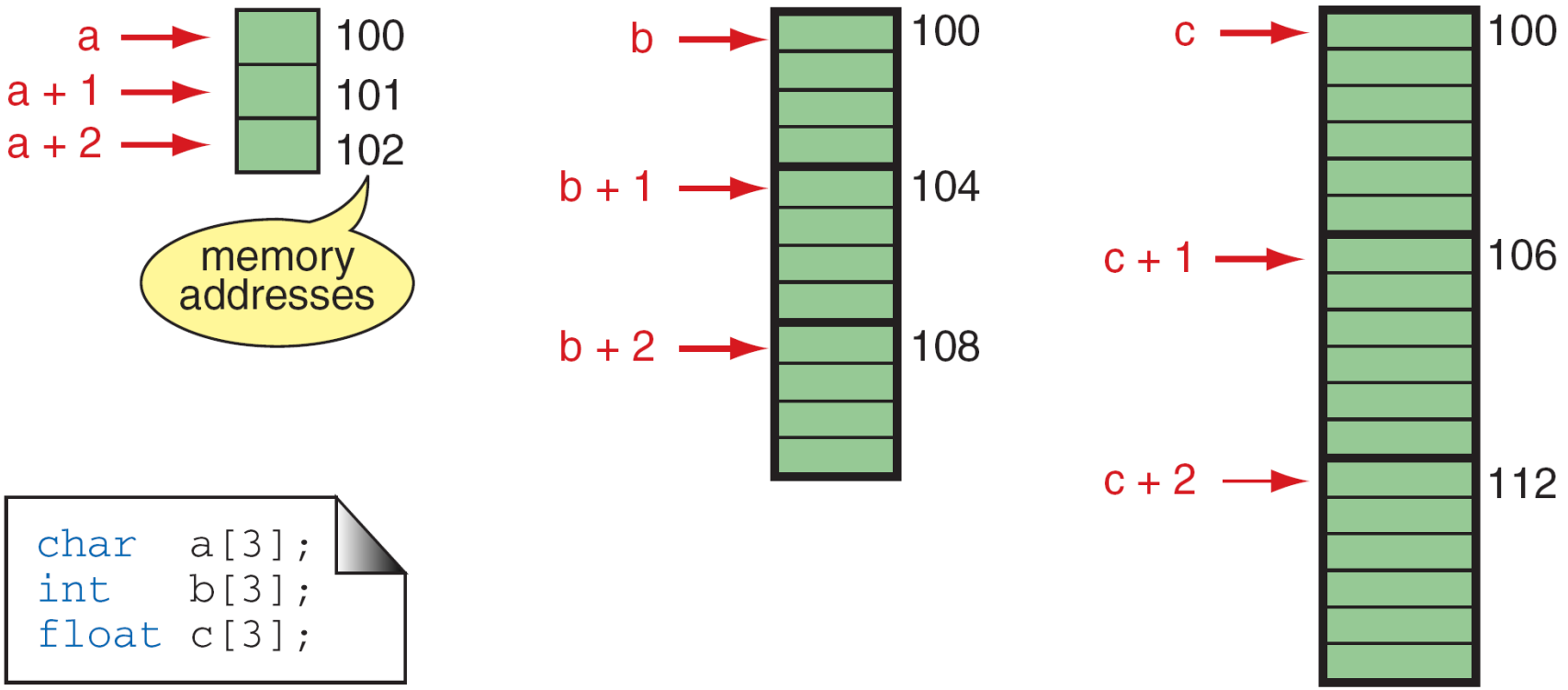


FIGURE 10-6 Pointer Arithmetic and Different Types

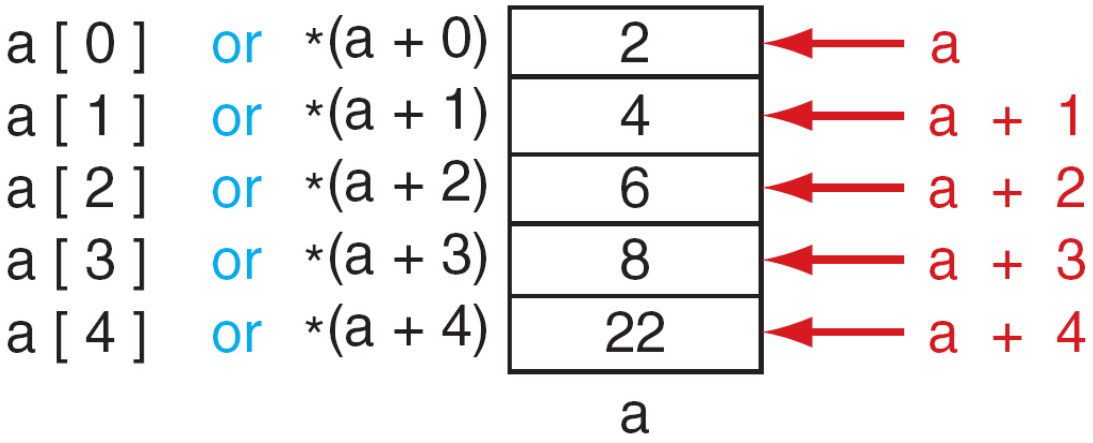


FIGURE 10-7 Dereferencing Array Pointers

Note

The following expressions are identical.
***(a + n) and a[n]**

Arithmetic Operations on Pointers

- p + 5, 5 + p, p – 5
- p1 – p2
- p++, --p
- p1 >= p2
- p1 != p2

Long Form	Short Form
if (ptr == NULL)	if (!ptr)
if (ptr != NULL)	if (ptr)

Table 10-1 Pointers and Relational Operators

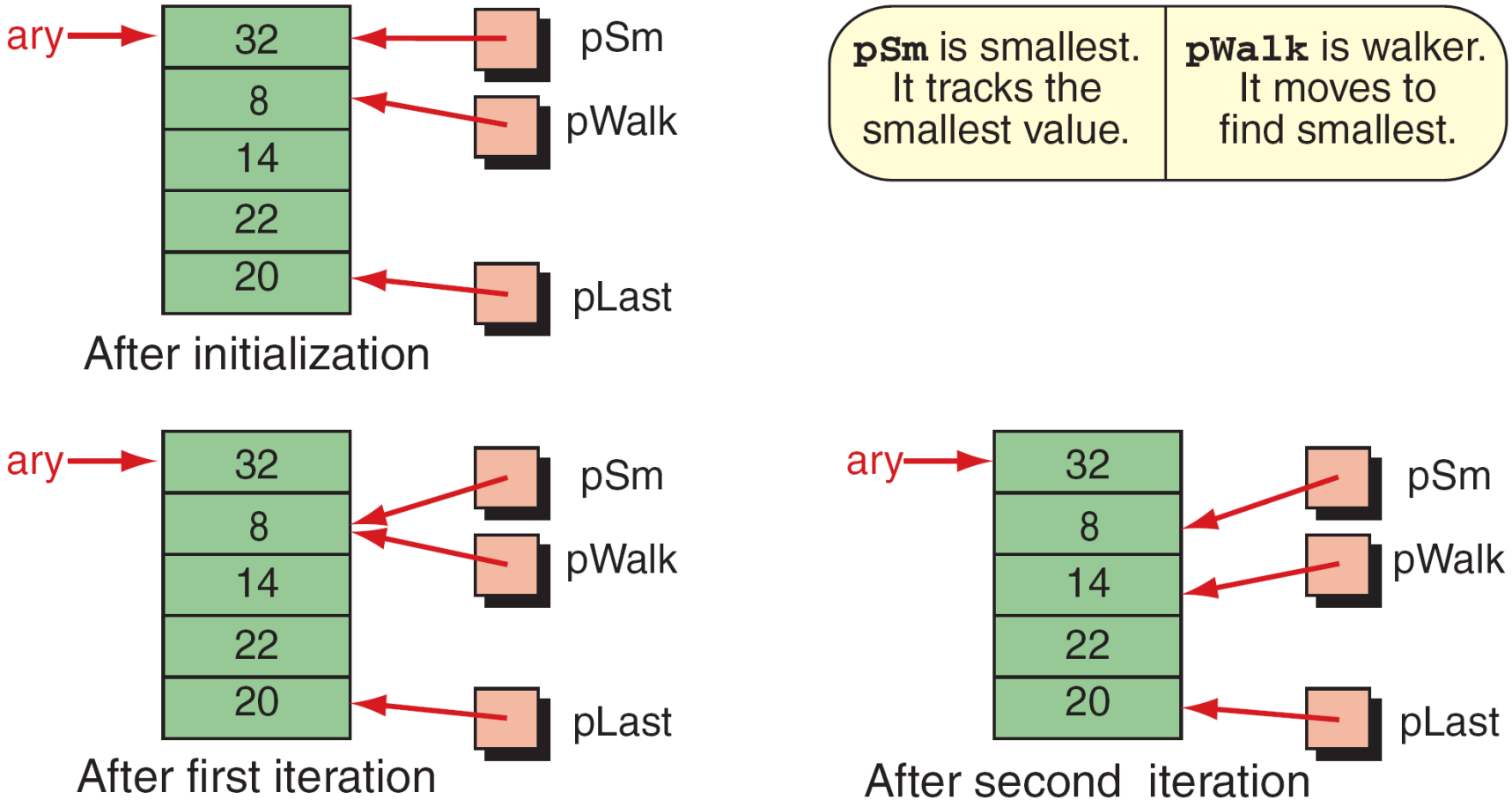
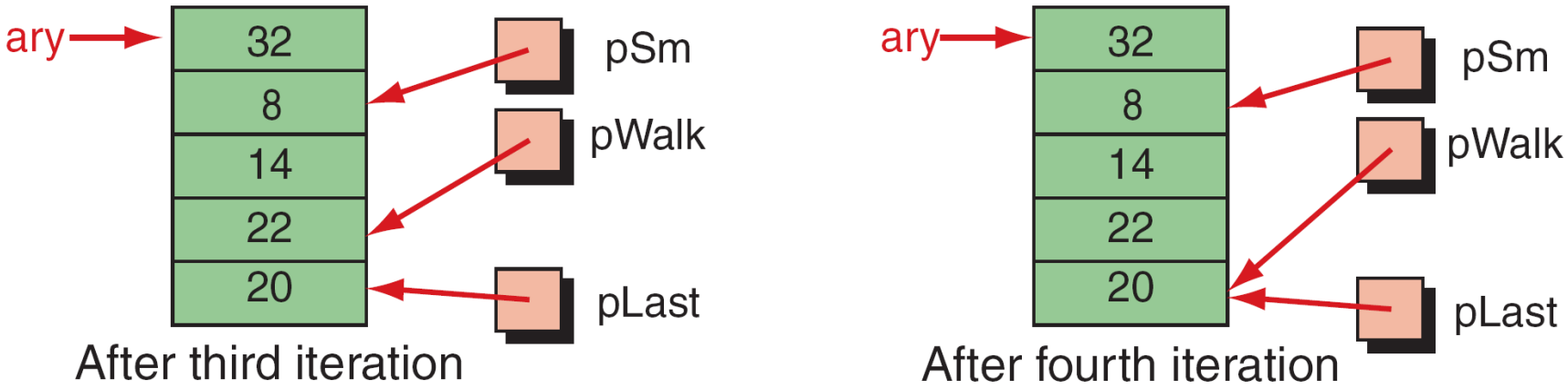


FIGURE 10-8 (Part I) Find Smallest




```
pLast = ary + arySize - 1;
for (pSm = ary, pWalk = ary + 1;
     pWalk <= pLast;
     pWalk++)
   if (*pWalk < *pSm)
        pSm = pWalk;
```

FIGURE 10-8 (Part II) Find Smallest

PROGRAM 10-1 Print Array with Pointers

```
1  /* Print an array forward by adding 1 to a pointer. Then
2     print it backward by subtracting one.
3        Written by:
4        Date:
5  */
6  #include <stdio.h>
7
8  #define MAX_SIZE 10
9
10 int main (void)
11 {
12 // Local Declarations
13    int  ary[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
14    int* pWalk;
15    int* pEnd;
16
17 // Statements
18    // Print array forward
```

PROGRAM 10-1 Print Array with Pointers

```
19      printf("Array forward : ");
20      for (pWalk = ary, pEnd = ary + MAX_SIZE;
21           pWalk <  pEnd;
22           pWalk++)
23         printf ("%3d", *pWalk);
24      printf ("\n");
25
26      // Print array backward
27      printf ("Array backward: ");
28      for (pWalk = pEnd - 1; pWalk >= ary; pWalk--)
29         printf ("%3d", *pWalk);
30      printf ("\n");
31
32      return 0;
33  }   // main
```

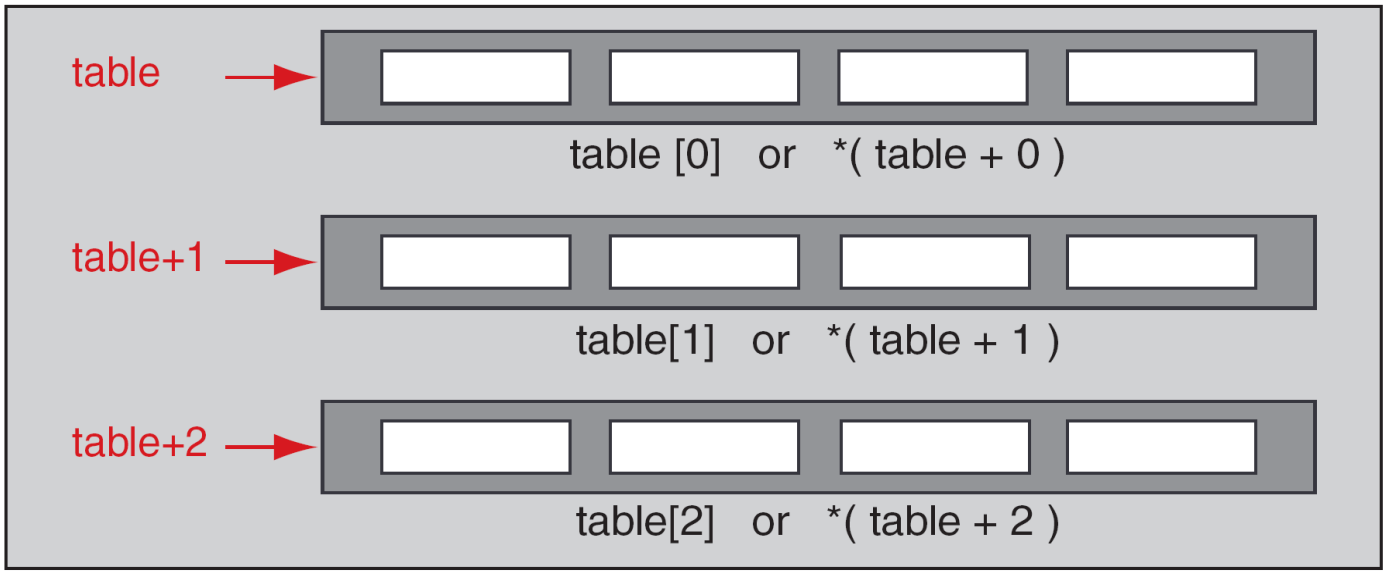
```
Results:
Array forward :   1   2   3   4   5   6   7   8   9  10
Array backward:  10   9   8   7   6   5   4   3   2   1
```

PROGRAM 10-2 Pointers and the Binary Search

```
1  /* ====================binary Search====================
2     Search an ordered list using Binary Search
3        Pre   list must contain at least one element
4              endPtr is pointer to largest element in list
5              target is value of element being sought
6        Post  FOUND: locnPtr pointer to target element
7                     return 1 (found)
8           !FOUND: locnPtr = element below or above target
9                     return 0 (not found)
10 */
11 int binarySearch (int list[], int*  endPtr,
12                    int target, int** locnPtr)
13 {
14 // Local Declarations
15    int* firstPtr;
16    int* midPtr;
17    int* lastPtr;
18
```

PROGRAM 10-2 Pointers and the Binary Search

```
19  // Statements
20     firstPtr = list;
21     lastPtr  = endPtr;
22     while (firstPtr <= lastPtr)
23        {
24         midPtr = firstPtr + (lastPtr - firstPtr) / 2;
25         if (target > *midPtr)
26            // look in upper half
27            firstPtr = midPtr + 1;
28         else if (target < *midPtr)
29            // look in lower half
30            lastPtr = midPtr - 1;
31         else
32            // found equal: force exit
33            firstPtr = lastPtr + 1;
34        } // end while
35     *locnPtr = midPtr;
36     return (target == *midPtr);
37  }  // binarySearch
```



int table[3][4] ;

```
for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)
      printf("%6d", *(*(table + i) + j));
   printf( "\n" );
  } // for i
```

Print Table

FIGURE 10-9 Pointers to Two-dimensional Arrays

Note

We recommend index notation for two-dimensional arrays.

10-3 Passing an Array to a Function

Now that we have discovered that the name of an array is actually a pointer to the first element, we can send the array name to a function for processing. When we pass the array, we do not use the address operator. Remember, the array name is a pointer constant, so the name is already the address of the first element in the array.

Array as a function argument

- doIt (aryName);
- int doIt (int ary[]);
- int doIt (int* ary);
- float doThat (int bigAry[ ][12][5]);

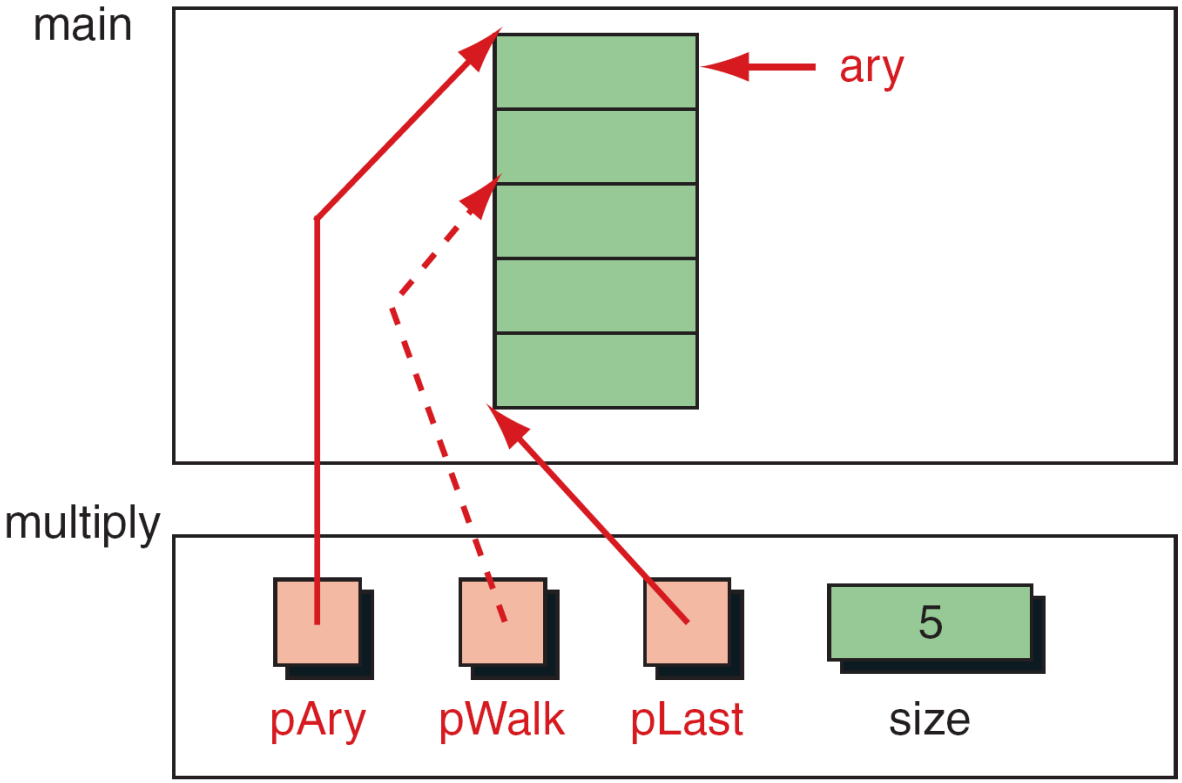


FIGURE 10-10 Variables for Multiply Array Elements By 2

PROGRAM 10-3 Multiply Array Elements by 2

```
 1  /* Read from keyboard & print integers multiplied by 2.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #define SIZE  5
 7
 8  //  Function Declarations
 9  void multiply (int* pAry, int size);
10
11  int main (void)
12  {
13  // Local Declarations
14     int  ary [SIZE];
15     int* pLast;
16     int* pWalk;
17
18  // Statements
```

PROGRAM 10-3 Multiply Array Elements by 2

```
19      pLast = ary + SIZE - 1;
20      for (pWalk = ary; pWalk <= pLast; pWalk++)
21          {
22           printf("Please enter an integer: ");
23           scanf ("%d", pWalk);
24          } // for
25
26      multiply (ary, SIZE);
27
28      printf ("Doubled value is: \n");
29      for (pWalk = ary; pWalk <= pLast; pWalk++)
30           printf (" %3d", *pWalk);
31
32      return 0;
33  }   // main
34
```

PROGRAM 10-3 Multiply Array Elements by 2

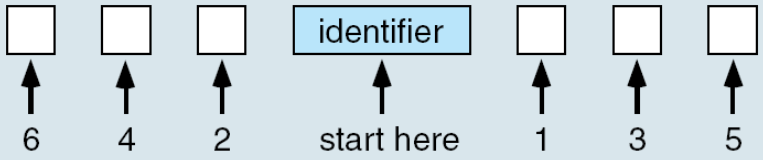
```
35 /* ==================== multiply ====================
36    Multiply elements in an array by 2
37       Pre   array has been filled
38             size indicates number of elements in array
39       Post  values in array doubled
40 */
41 void multiply (int* pAry, int size)
42 {
43 // Local Declarations
44    int* pWalk;
45    int* pLast;
46
47 // Statements
48    pLast = pAry + size - 1;
49    for (pWalk = pAry; pWalk <= pLast; pWalk++)
50        *pWalk = *pWalk * 2;
51    return;
52 }  // multiply
53 // ================= End of Program =================
```

PROGRAM 10-3 Multiply Array Elements by 2

```
Results:
Please enter an integer: 1
Please enter an integer: 2
Please enter an integer: 3
Please enter an integer: 4
Please enter an integer: 5
Doubled value is:
   2   4   6   8  10
```

Understanding Complicated Declarations

To help you read and understand complicated declarations, we have developed the **right-left rule**. Using this rule to interpret a declaration, you start with the identifier in the center of a declaration and "read" the declaration by alternatively going right and then left until all entities have been read. The basic concept is shown below.



Consider the simple declaration

```
int x;
```

This is read as "x is ❒ an integer."[a]

```
int     x     ❒;
 ↑      ↑     ↑
 2      0     1
```

Since there is nothing on the right, we simply go left.

Now consider the example of a pointer declaration. This example is read as "p is ❒ a pointer ❒ to integer."

```
int     *     p     ❒     ❒;
 ↑      ↑     ↑     ↑     ↑
 4      2     0     1     3
```

Note that we keep going right even when there is nothing there until all the entities on the left have been exhausted. For a more complete discussion of complex declarations, see Appendix L.

---

a The box (❒) is just a place holder to show that there is no entry to be considered. Simply ignore it when you read each declaration.

10-4 Memory Allocation Functions

C gives us two choices when we want to reserve memory locations for an object: static allocation and dynamic allocation.

Topics discussed in this section:

Memory Usage
Static Memory Allocation
Dynamic Memory Allocation
Memory Allocation Functions
Releasing Memory (free)

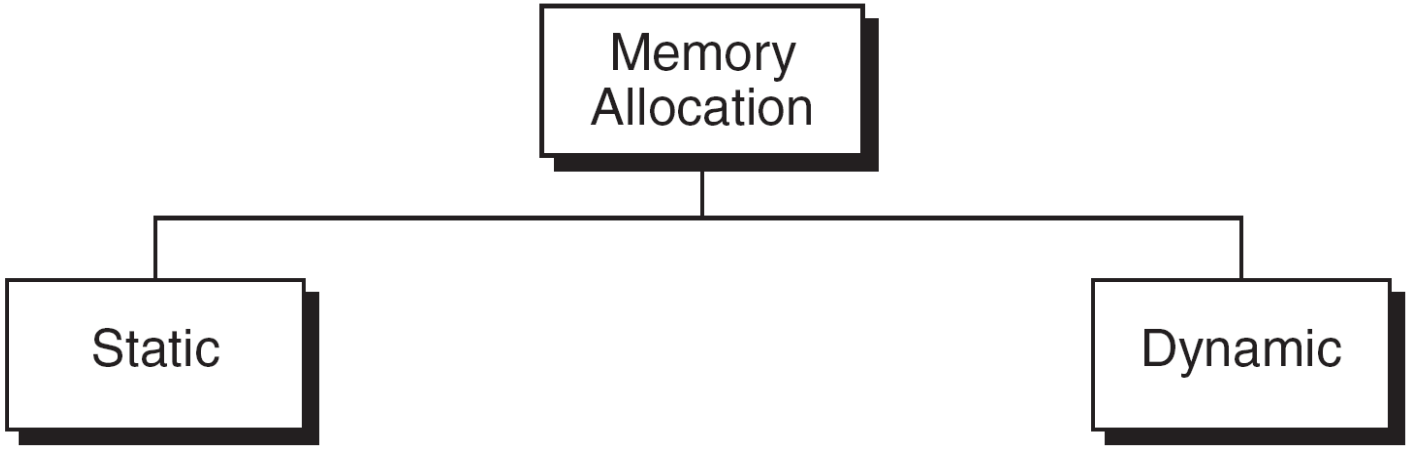


FIGURE 10-11 Memory Allocation

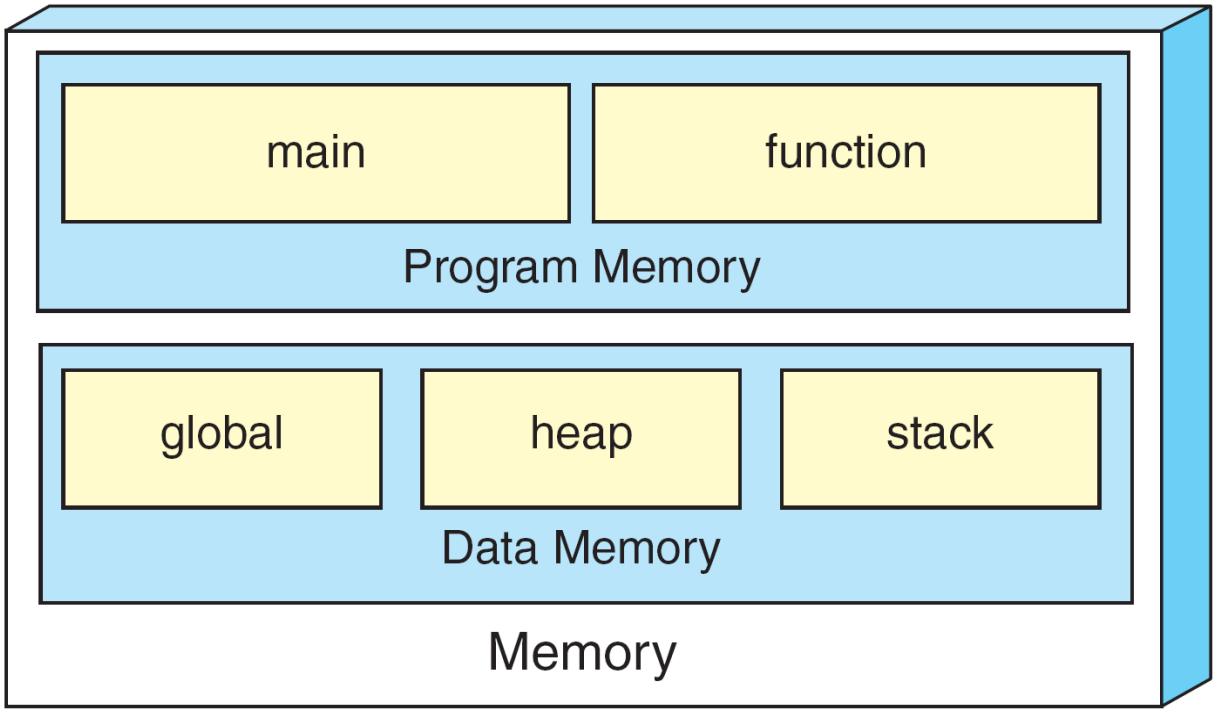


FIGURE 10-12 A Conceptual View of Memory

Note

We can refer to memory allocated in the heap only through a pointer.

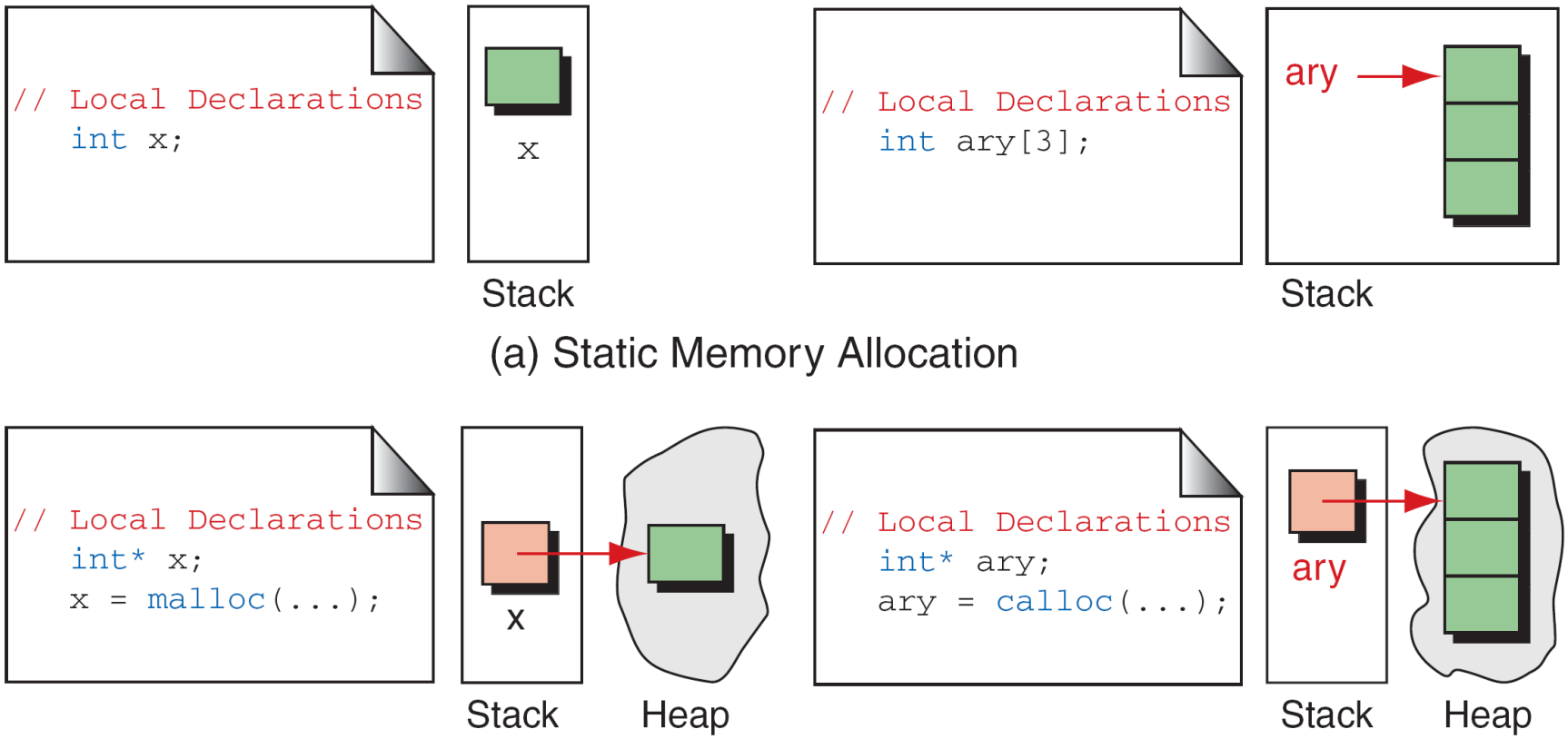


FIGURE 10-13 Accessing Dynamic Memory

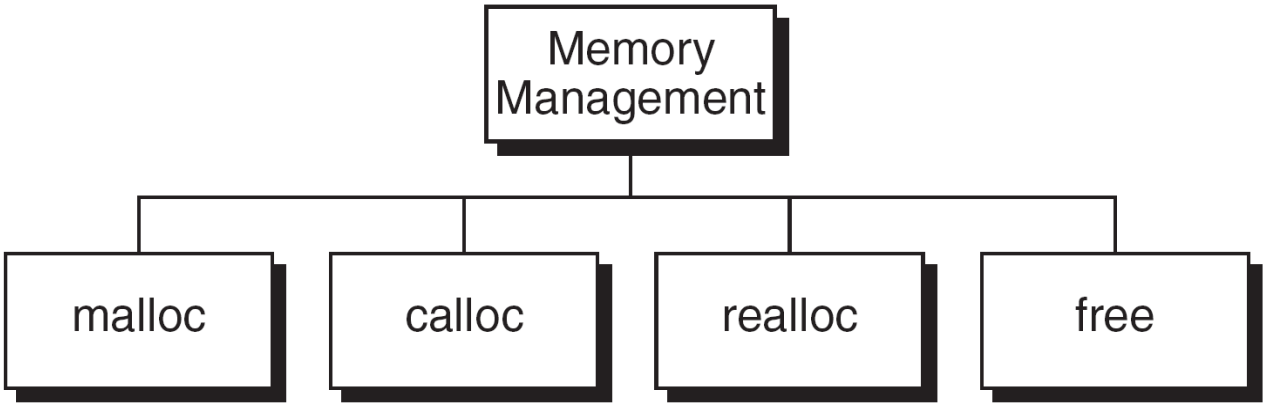


FIGURE 10-14 Memory Management Functions

Dynamic Memory Allocation

- void *malloc (size_t size);
- void *calloc (size_t element-count,
  size_t element-size);
- void *realloc (void* ptr, size_t newSize);

- void free (void* ptr);

Note

Memory Allocation Casting

Prior to C99, it was necessary to cast the pointer returned from a memory allocation function. While it is no longer necessary, it does no harm as long as the cast is correct. If you should be working with an earlier standard, the casting format is: pointer = (type*) malloc(size)

```
if (!(pInt = malloc(sizeof(int))))
    // No memory available
    exit (100) ;
// Memory available
…
```

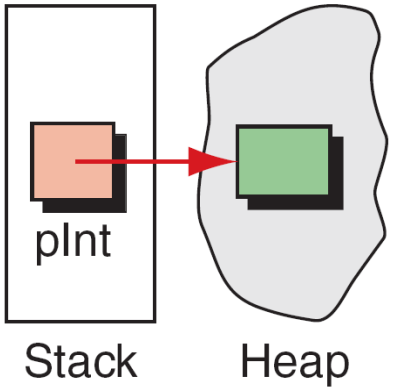
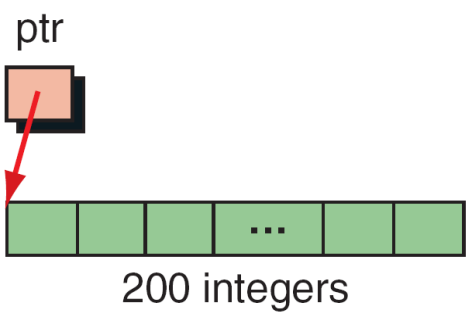


FIGURE 10-15 ***malloc***




```
if (!(ptr = (int*)calloc (200, sizeof(int))))
    // No memory available
    exit (100) ;

// Memory available
…
```

FIGURE 10-16 ***calloc***

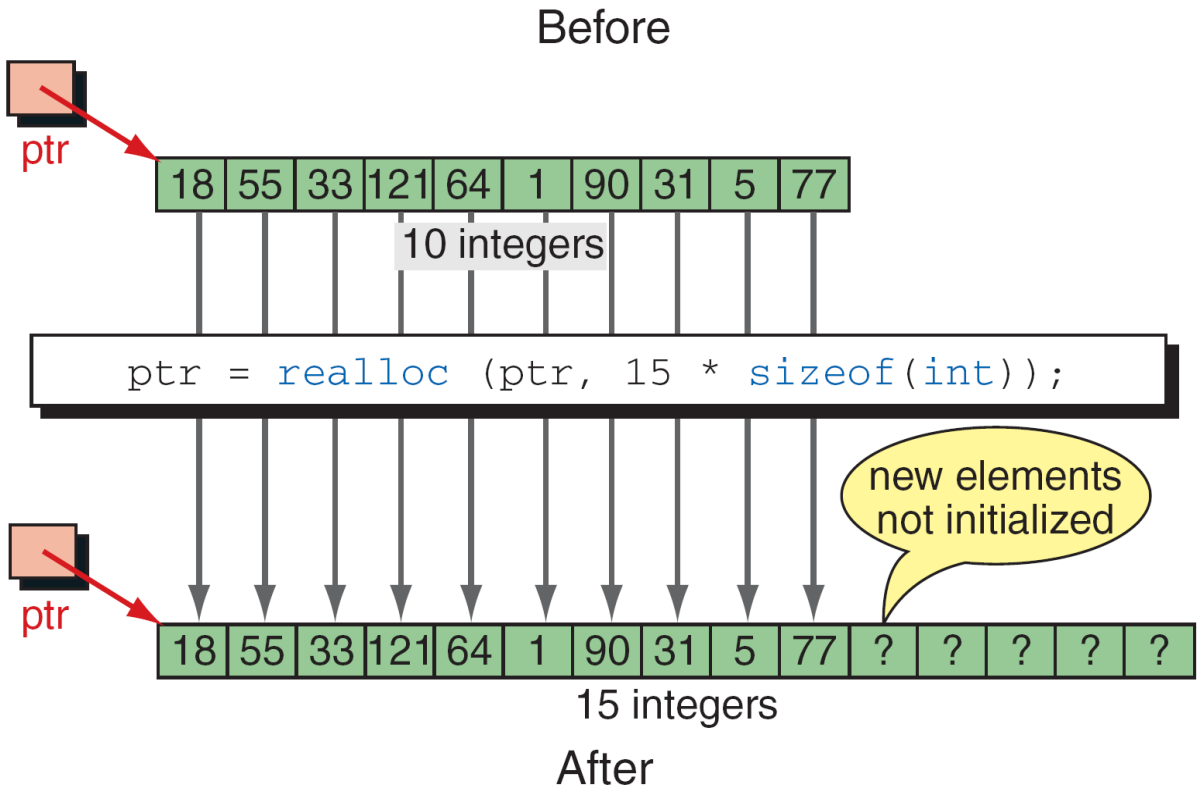


FIGURE 10-17 ***realloc***

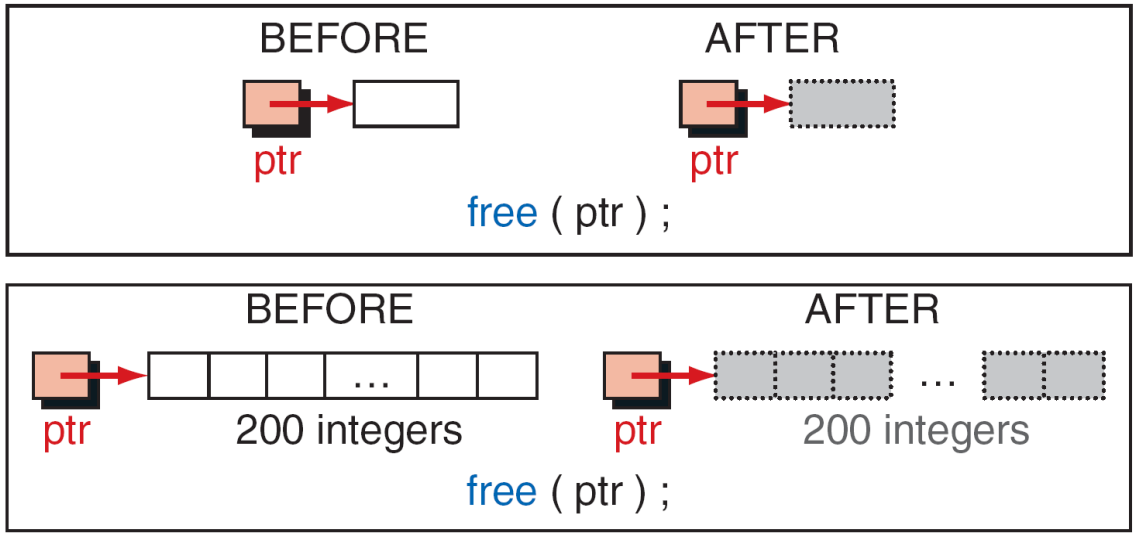


FIGURE 10-18 Freeing Memory

Note

Using a pointer after its memory has been released is a common programming error. Guard against it by clearing the pointer.

Note

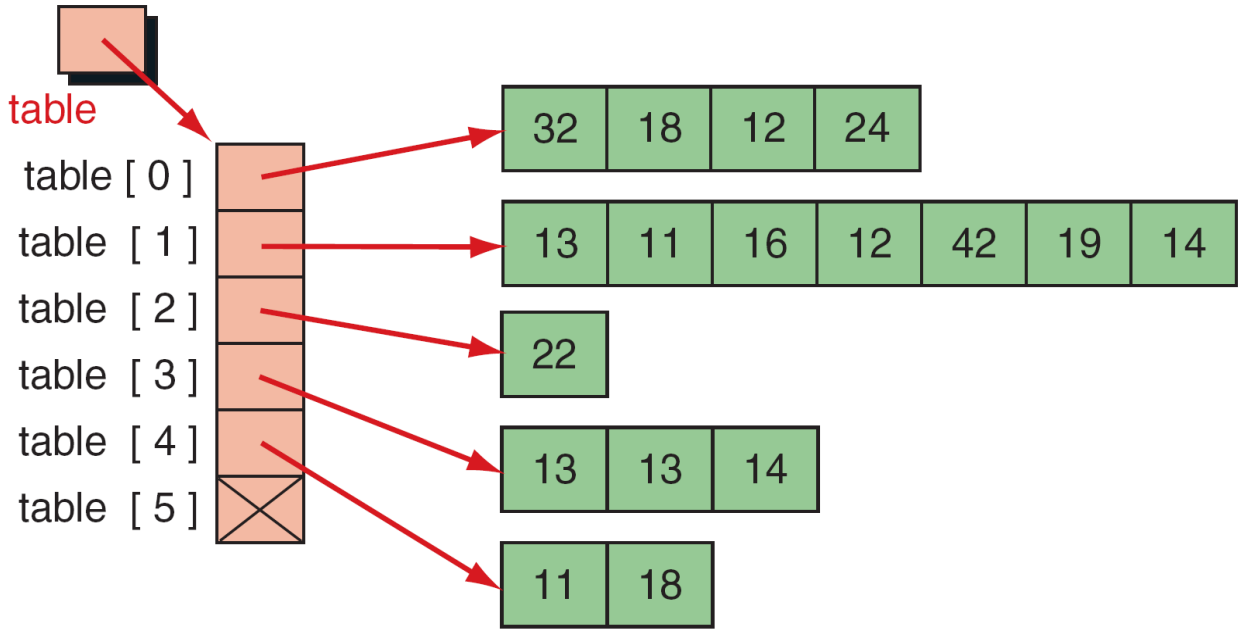
The pointer used to free memory must be of the same type as the pointer used to allocate the memory.

10-5 Array of Pointers

Another useful structure that uses arrays and pointers is an array of pointers. This structure is especially helpful when the number of elements in the array is variable.

32	18	12	24			
13	11	16	12	42	19	14
22						
13	13	14				
11	18					

Table 10-2 A Ragged Table




```
table = (int**)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

FIGURE 10-19 A Ragged Array

10-6 Programming Applications

This section contains two applications. The first is a rewrite of the selection sort, this time using pointers. The second uses dynamic arrays.

Topics discussed in this section:

Selection Sort Revisited

Dynamic Array

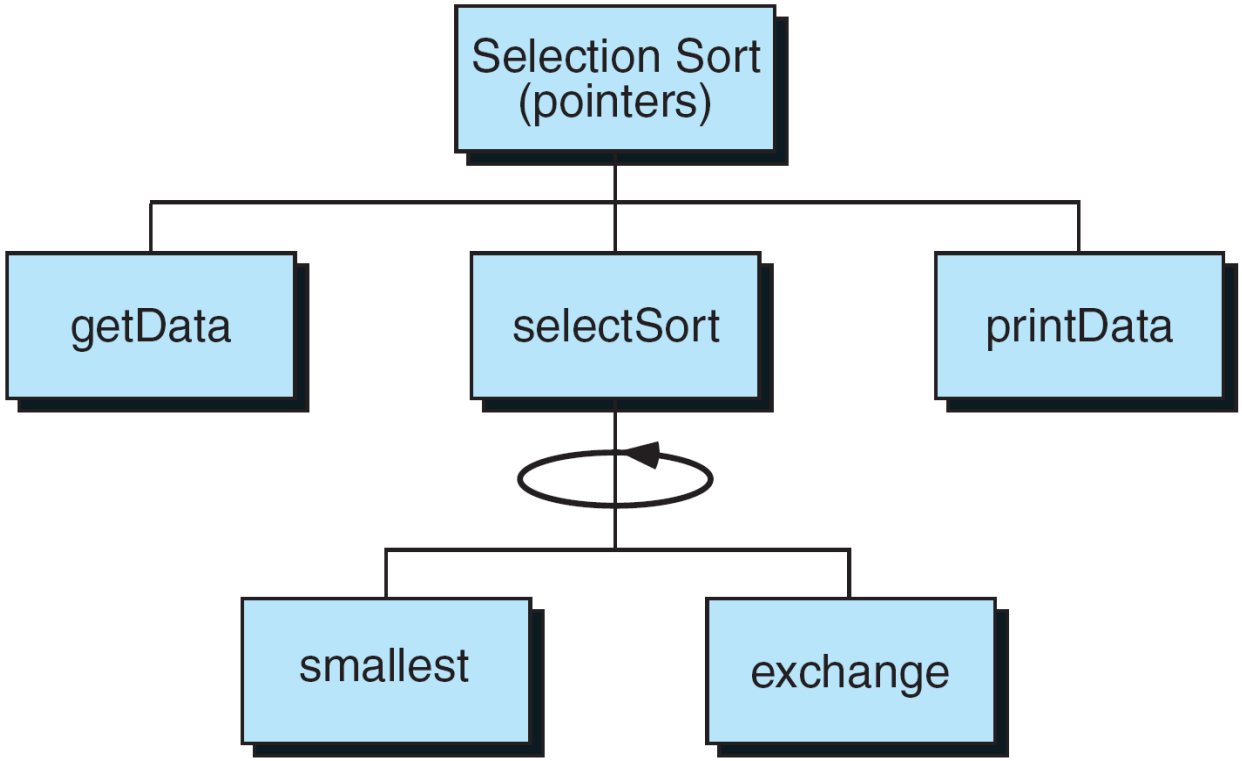


FIGURE 10-20 Selection Sort with Pointers—Structure Chart

PROGRAM 10-4 Selection Sort Revisited

```
 1  /* Demonstrate pointers with Selection Sort
 2        Written by:
 3        Date written:
 4  */
 5  #include <stdio.h>
 6  #define SIZE 25
 7
 8  // Function Declarations
 9  int*  getData    (int* pAry,     int  arySize);
10  void  selectSort (int* pAry,     int* last);
11  void  printData  (int* pAry,     int* last);
12  int*  smallest   (int* pAry,     int* pLast);
13  void  exchange   (int* current, int* smallest);
14
15  int main (void)
16  {
17  // Local Declarations
18     int  ary[SIZE];
19     int* pLast;
20
```

PROGRAM 10-4 Selection Sort Revisited

```
21  // Statements
22     pLast = getData (ary, SIZE);
23     selectSort (ary, pLast);
24     printData  (ary, pLast);
25     return 0;
26  }  // main
27
28  /* ==================== getData ====================
29     Reads data from keyboard into array for sorting.
30        Pre   pAry is pointer to array to be filled
31              arySize is integer with maximum array size
32        Post  array filled. Returns address of last element
33  */
34  int* getData (int* pAry, int arySize)
35  {
36  // Local Declarations
37     int  ioResult;
38     int  readCnt = 0;
39     int* pFill   = pAry;
```

PROGRAM 10-4 Selection Sort Revisited

```
40
41  // Statements
42      do
43          {
44           printf("Please enter number or <EOF>: ");
45           ioResult = scanf("%d", pFill);
46           if (ioResult == 1)
47              {
48               pFill++;
49               readCnt++;
50              } // if
51          } while (ioResult == 1 && readCnt < arySize);
52
53      printf("\n\n%d numbers read.", readCnt);
54      return (--pFill);
55  }  // getData
56
```

PROGRAM 10-4 Selection Sort Revisited

```
57  /* ===================== selectSort ====================
58     Sorts by selecting smallest element in unsorted
59     portion of the array and exchanging it with element
60     at the beginning of the unsorted list.
61        Pre   array must contain at least one item
62              pLast is pointer to last element in array
63        Post  array rearranged smallest to largest
64  */
65  void selectSort (int* pAry,  int* pLast)
66  {
67  // Local Declarations
68     int* pWalker;
69     int* pSmallest;
70
71  // Statements
72     for (pWalker = pAry; pWalker < pLast; pWalker++)
73          {
74           pSmallest = smallest (pWalker, pLast);
75           exchange (pWalker, pSmallest);
76          } // for
77     return;
78  }  // selectSort
```

PROGRAM 10-4 Selection Sort Revisited

```
79
80  /* ==================== smallest ====================
81     Find smallest element starting at current pointer.
82        Pre   pAry points to first unsorted element
83        Post  smallest element identified and returned
84  */
85  int* smallest (int* pAry, int* pLast)
86  {
87  // Local Declarations
88     int* pLooker;
89     int* pSmallest;
90
91  // Statements
92     for (pSmallest = pAry, pLooker = pAry + 1;
93          pLooker <= pLast;
94          pLooker++)
95        if (*pLooker < *pSmallest)
96             pSmallest = pLooker;
97     return pSmallest;
98  }  // smallest
99
```

PROGRAM 10-4 Selection Sort Revisited

```
100 /* ======================== exchange ======================
101    Given pointers to two array elements, exchange them
102       Pre   p1 & p2 are pointers to exchange values
103       Post  exchange is completed
104 */
105 void exchange (int* p1, int* p2)
106 {
107 // Local Declarations
108    int temp;
109
110 // Statements
111    temp  =  *p1;
112    *p1   =  *p2;
113    *p2   =  temp;
114    return;
115 }  // exchange
116
```

PROGRAM 10-4 Selection Sort Revisited

```
117 /* ==================== printData ====================
118    Given a pointer to an array, print the data.
119       Pre   pAry points to array to be filled
120             pLast identifies last element in the array
121       Post  data have been printed
122 */
123 void  printData (int* pAry, int* pLast)
124 {
125 // Local Declarations
126    int  nmbrPrt;
127    int* pPrint;
128
129 // Statements
130    printf("\n\nYour data sorted are: \n");
131    for (pPrint = pAry, nmbrPrt = 0;
132         pPrint <= pLast;
133         nmbrPrt++, pPrint++)
134       printf ("\n#%02d %4d", nmbrPrt, *pPrint);
135    printf("\n\nEnd of List ");
136    return;
137 }  // PrintData
138 // ================ End of Program ================
```

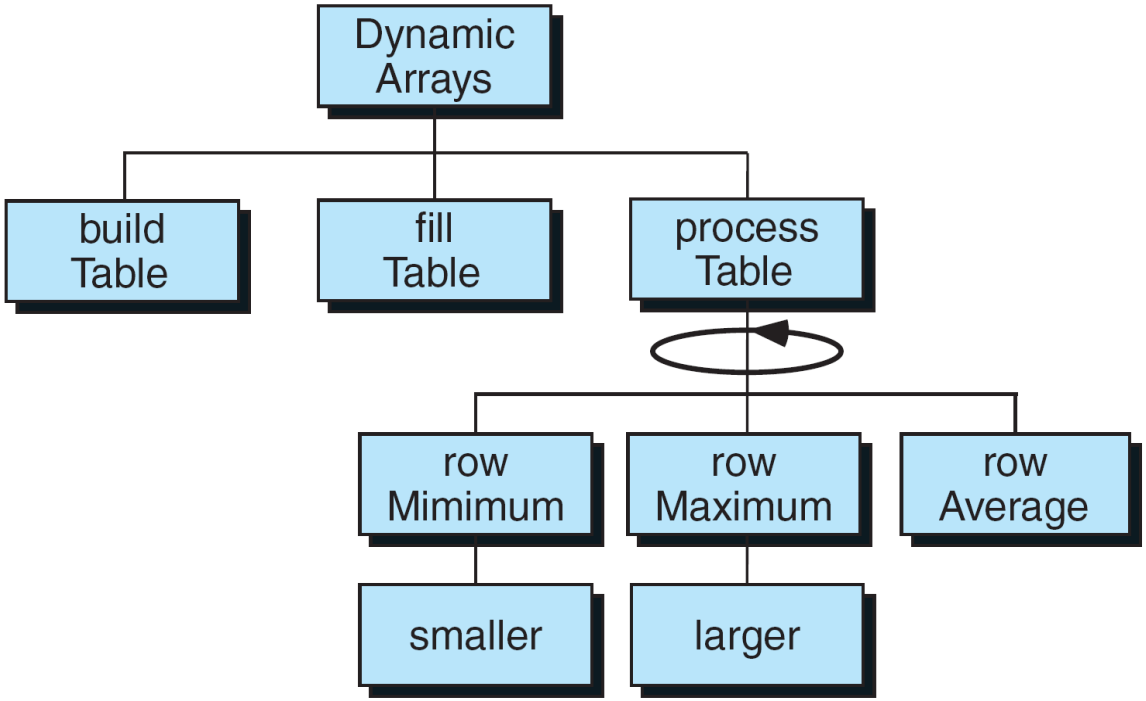


FIGURE 10-21 Dynamic Array Structure Chart

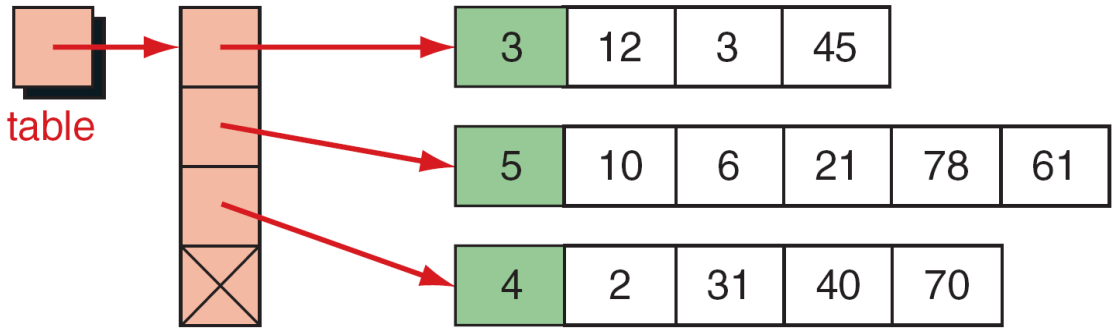


FIGURE 10-22 Ragged Array Structure

PROGRAM 10-5 Dynamic Arrays: main

```
 1  /* Demonstrate storing arrays in the heap. This program
 2     builds and manipulates a variable number of ragged
 3     arrays. It then calculates the minimum, maximum, and
 4     average of the numbers in the arrays.
 5        Written by:
 6        Date:
 7  */
 8  #include <stdio.h>
 9  #include <stdlib.h>
10  #include <limits.h>
11
12  // Function Declarations
13  int** buildTable   (void);
14  void  fillTable    (int** table);
15  void  processTable (int** table);
16  int   smaller      (int   first, int second);
17  int   larger       (int   first, int second);
18  int   rowMinimum   (int*  rowPtr);
19  int   rowMaximum   (int*  rowPtr);
20  float rowAverage   (int*  rowPtr);
21
```

PROGRAM 10-5 Dynamic Arrays: main

```
22  int main (void)
23  {
24  // Local Declarations
25     int** table;
26
27  // Statements
28     table = buildTable();
29     fillTable    (table);
30     processTable (table);
31     return 0;
32  }  // main
```

PROGRAM 10-6 Dynamic Arrays: buildTable

```
1  /* ==================== buildTable ====================
2     Create backbone of the table by creating an array of
3     pointers, each pointing to an array of integers.
4        Pre   nothing
5        Post  returns pointer to the table
6  */
7  int** buildTable (void)
8  {
9  // Local Declarations
10    int   rowNum;
11    int   colNum;
12    int** table;
13    int   row;
14
15 // Statements
```

PROGRAM 10-5 Dynamic Arrays: buildTable

```
16      printf("\nEnter the number of rows in the table: ");
17      scanf ("%d", &rowNum);
18      table = (int**) calloc(rowNum + 1, sizeof(int*));
19      for (row = 0; row < rowNum; row++)
20          {
21           printf("Enter number of integers in row %d: ",
22                   row + 1);
23           scanf ("%d", &colNum);
24           table[row] = (int*)calloc(colNum + 1,
25                                     sizeof(int));
26           table[row] [0] = colNum;
27          } // for
28      table[row] = NULL;
29      return table;
30  }   // buildTable
```

PROGRAM 10-7 Dynamic Arrays: fillTable

```
1  /* ==================== fillTable ====================
2     This function fills the array rows with data.
3        Pre   array of pointers
4        Post  array filled
5  */
6  void fillTable (int** table)
7  {
8  // Local Declarations
9     int row = 0;
10
11 // Statements
12    printf("\n ================================");
13    printf("\n Now we fill the table.\n");
14    printf("\n For each row enter the data");
15    printf("\n and press return: ");
16    printf("\n ================================\n");
17
```

PROGRAM 10-7 Dynamic Arrays: fillTable

```
18      while (table[row] != NULL)
19          {
20           printf("\n row %d (%d integers) =====> ",
21                   row + 1, table[row][0]);
22           for (int column = 1;
23                    column <= *table[row];
24                    column++)
25                scanf("%d", table[row] + column);
26           row++;
27          } // while
28      return;
29  }  // fillTable
```

PROGRAM 10-8 Dynamic Arrays: Process Table

```
1  /* ================== processTable ====================
2     Process the table to create the statistics.
3        Pre   table
4        Post  row statistics (min, max, and average)
5  */
6  void processTable (int** table)
7  {
8  // Local Declarations
9  int   row = 0;
10 int   rowMin;
11 int   rowMax;
12 float rowAve;
13
14 // Statements
15    while (table[row] != NULL)
16       {
```

PROGRAM 10-8 Dynamic Arrays: Process Table

```
17          rowMin = rowMinimum (table[row]);
18          rowMax = rowMaximum (table[row]);
19          rowAve = rowAverage (table[row]);
20          printf("\nThe statistics for row %d ", row + 1);
21          printf("\nThe minimum: %5d",    rowMin);
22          printf("\nThe maximum: %5d",    rowMax);
23          printf("\nThe average: %8.2f ", rowAve);
24          row++;
25        } // while
26      return;
27  }  // processTable
```

PROGRAM 10-9 Dynamic Arrays: Find Row Minimum

```
1  /* ==================== rowMinimum ====================
2     Determines the minimum of the data in a row.
3        Pre   given pointer to the row
4        Post  returns the minimum for that row
5  */
6  int rowMinimum (int* rowPtr)
7  {
8
9  // Local Declarations
10    int rowMin = INT_MAX;
11
12 // Statements
13    for (int column = 1; column <= *rowPtr; column++)
14          rowMin = smaller (rowMin, *(rowPtr + column));
15    return rowMin;
16 }  // rowMinimum
```

PROGRAM 10-9 Dynamic Arrays: Find Row Minimum

```
1  /* ==================== rowMaximum ====================
2     Calculates the maximum of the data in a row.
3        Pre   given pointer to the row
4        Post  returns the maximum for that row
5  */
6  int rowMaximum (int* rowPtr)
7  {
8  // Local Declarations
9     int rowMax = INT_MIN;
10
11 // Statements
12    for (int column = 1; column <= *rowPtr; column++)
13          rowMax = larger  (rowMax, *(rowPtr + column));
14    return rowMax;
15 }  // rowMaximum
```

PROGRAM 10-10 Dynamic Arrays: Find Row Maximum

```
1  /* ==================== rowAverage ====================
2     Calculates the average of data in a row.
3        Pre   pointer to the row
4        Post  returns the average for that row
5  */
6  float  rowAverage (int* rowPtr)
7  {
8  // Local Declarations
9     float  total = 0;
10    float  rowAve;
11
12 // Statements
13    for (int column = 1; column <= *rowPtr; column++)
14         total += (float)*(rowPtr + column);
15    rowAve = total / *rowPtr;
16    return rowAve;
17 }  // rowAverage
```

PROGRAM 10-11 Dynamic Arrays: Find Row Average

```
1  /* ==================== rowAverage ====================
2     Calculates the average of data in a row.
3        Pre   pointer to the row
4        Post  returns the average for that row
5  */
6  float  rowAverage (int* rowPtr)
7  {
8  // Local Declarations
9     float  total = 0;
10    float  rowAve;
11
12 // Statements
13    for (int column = 1; column <= *rowPtr; column++)
14          total += (float)*(rowPtr + column);
15    rowAve = total / *rowPtr;
16    return rowAve;
17 }  // rowAverage
```

PROGRAM 10-12 Dynamic Arrays: Find Smaller

```
1  /* ==================== smaller ====================
2     This function returns the smaller of two numbers.
3        Pre   two numbers
4        Post  returns the smaller
5  */
6  int  smaller (int first, int second)
7  {
8  // Statements
9     return (first < second ? first : second);
10 }  // smaller
```

PROGRAM 10-13 Dynamic Arrays: Find Larger

```
1  /*  ====================== larger ======================
2     This function returns the larger of two numbers.
3        Pre   two numbers
4        Post  returns the larger
5  */
6  int larger (int first, int second)
7  {
8  // Statements
9     return (first > second ? first : second);
10 }  // larger
```

10-7 Software Engineering

Pointer applications need careful design to ensure that they work correctly and efficiently. The programmer not only must take great care in the program design but also must carefully consider the data structures that are inherent with pointer applications.

Topics discussed in this section:

Pointers and Function Calls
Pointers and Arrays
Array Index Commutativity
Dynamic Memory: Theory versus Practice

Note

Whenever possible, use value parameters.

PROGRAM 10-14 Testing Memory Reuse

```
1  /* This program tests the reusability of dynamic memory.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main (void)
9  {
10 // Local Declarations
11    int  looper;
12    int* ptr;
13
14 // Statements
15    for (looper = 0; looper < 5; looper++)
16       {
17        ptr = malloc(16);
18        printf("Memory allocated at: %p\n", ptr);
19
```

PROGRAM 10-14 Testing Memory Reuse

```
20          free (ptr);
21        } // for
22     return 0;
23  }  // main
```

```
Results in Personal Computer:
Memory allocated at: 0x00e7fc32
Memory allocated at: 0x00e8024a
Memory allocated at: 0x00e8025c
Memory allocated at: 0x00e8026e
Memory allocated at: 0x00380280

Results in UNIX system:
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
Memory allocated at: 0x00300f70
```