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)

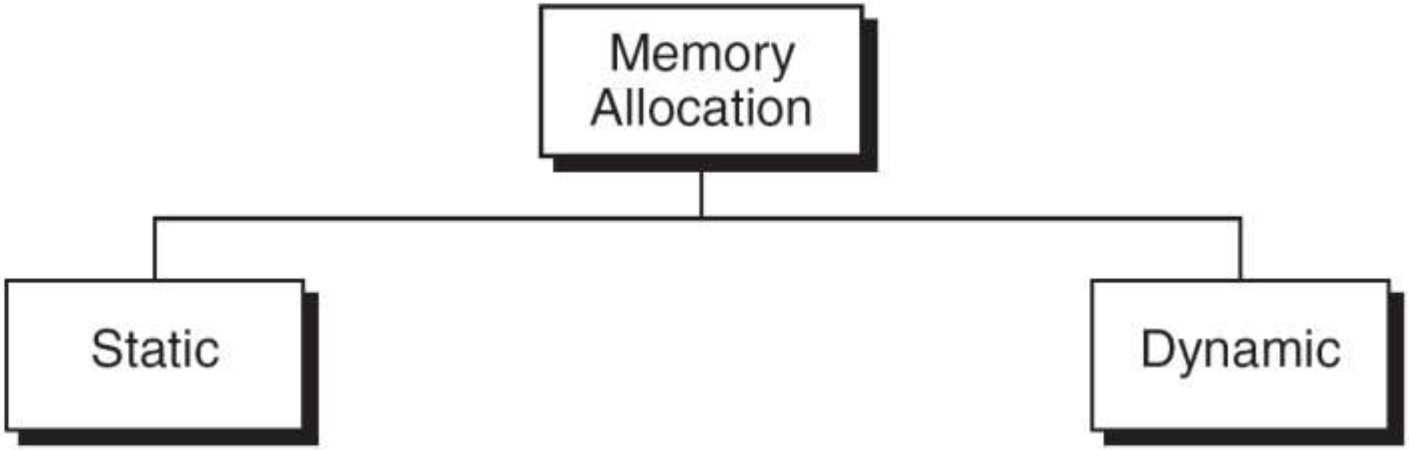


FIGURE 10-11 Memory Allocation

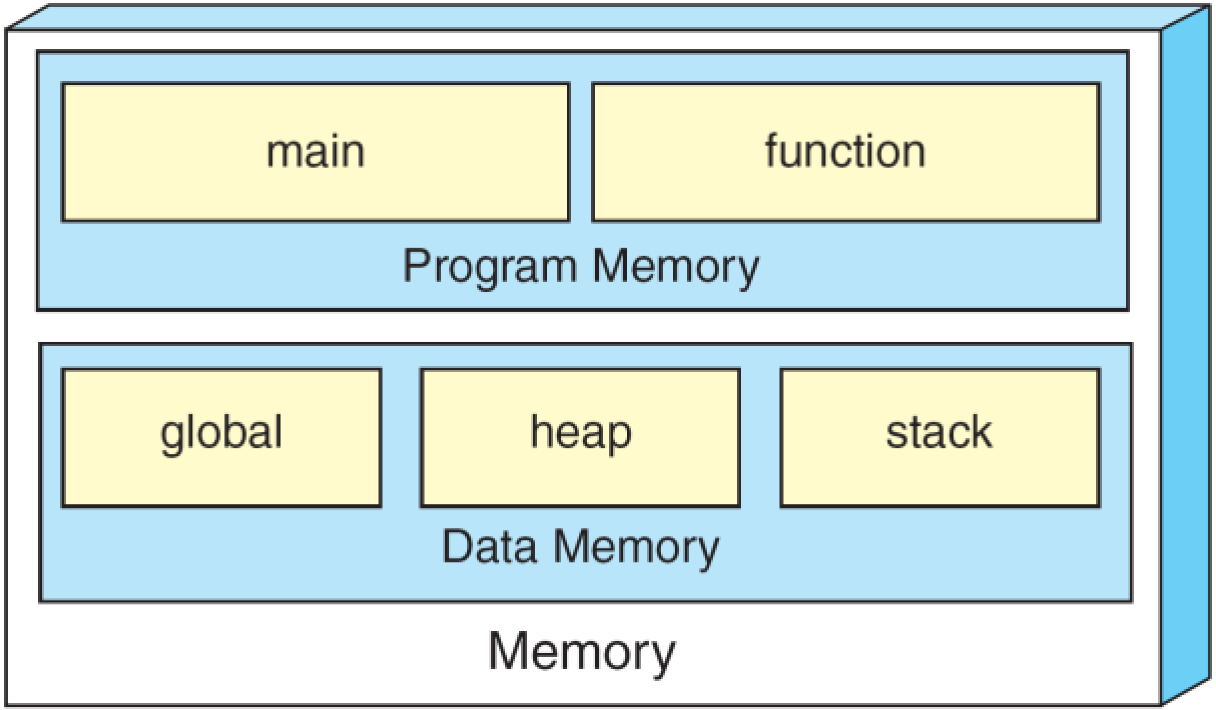
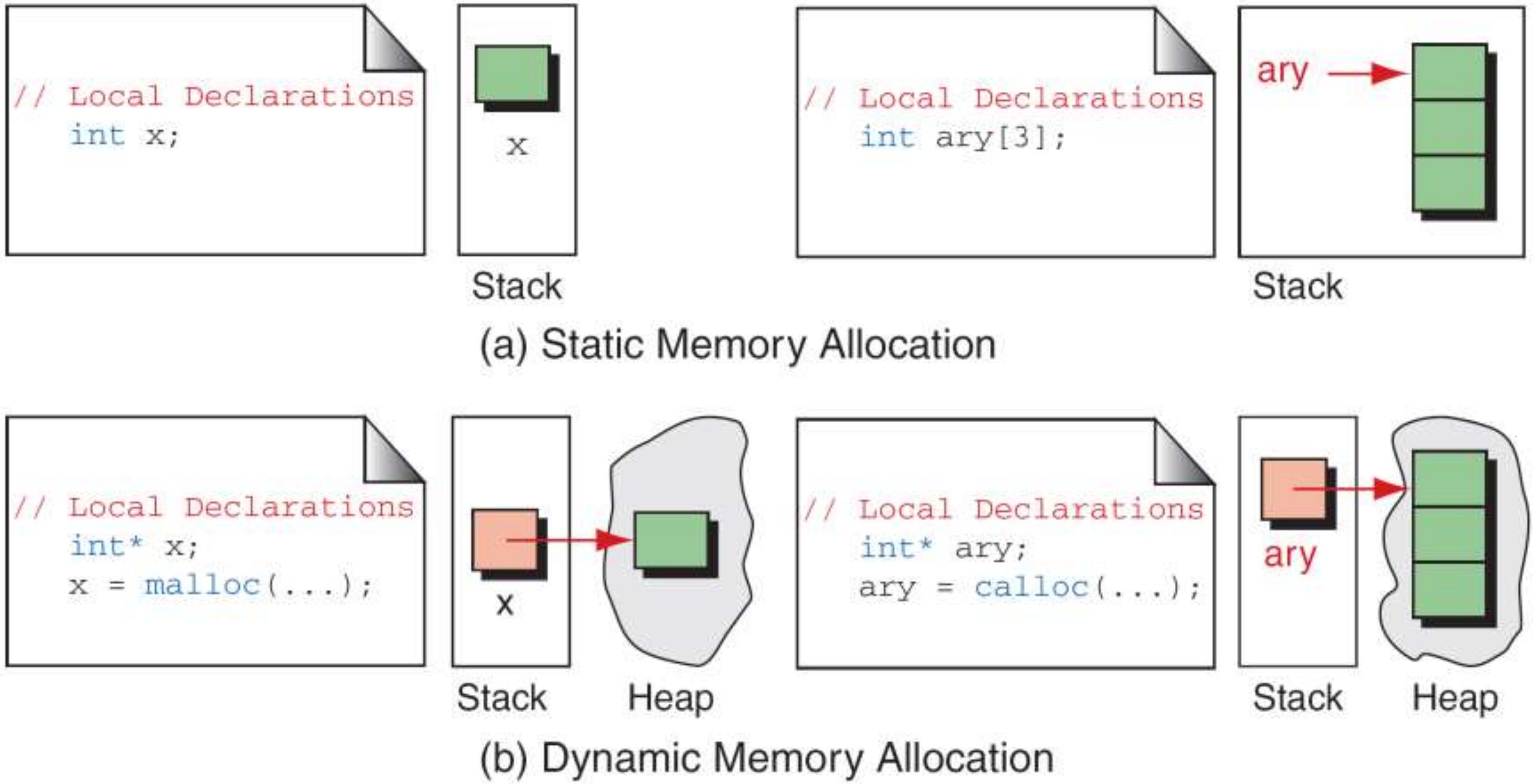


FIGURE 10-12 **A Conceptual View of Memory**

Note

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



```
// Local Declarations
   int x;
```

```
// Local Declarations
   int ary[3];
```

(a) Static Memory Allocation

```
// Local Declarations
   int* x;
   x = malloc(...);
```

```
// Local Declarations
   int* ary;
   ary = calloc(...);
```

(b) Dynamic Memory Allocation

FIGURE 10-13 Accessing Dynamic Memory

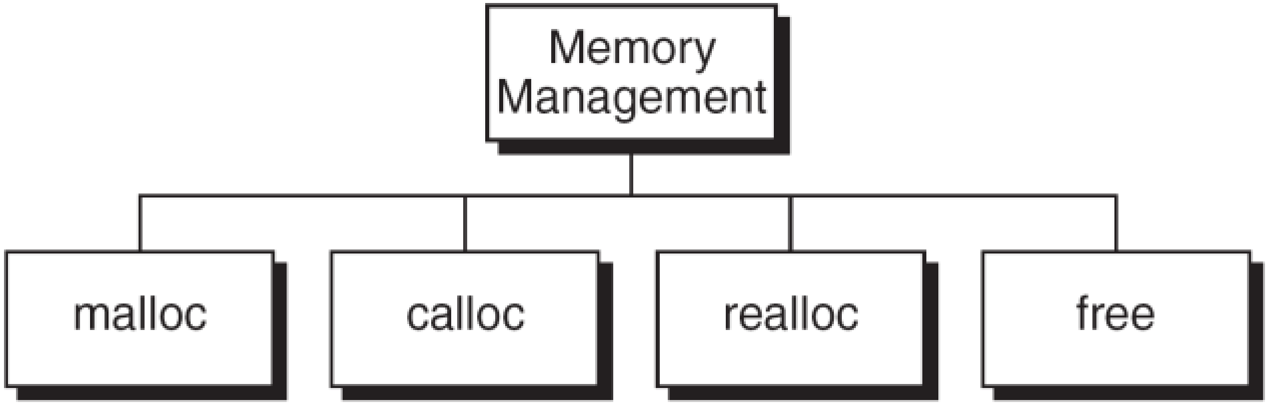
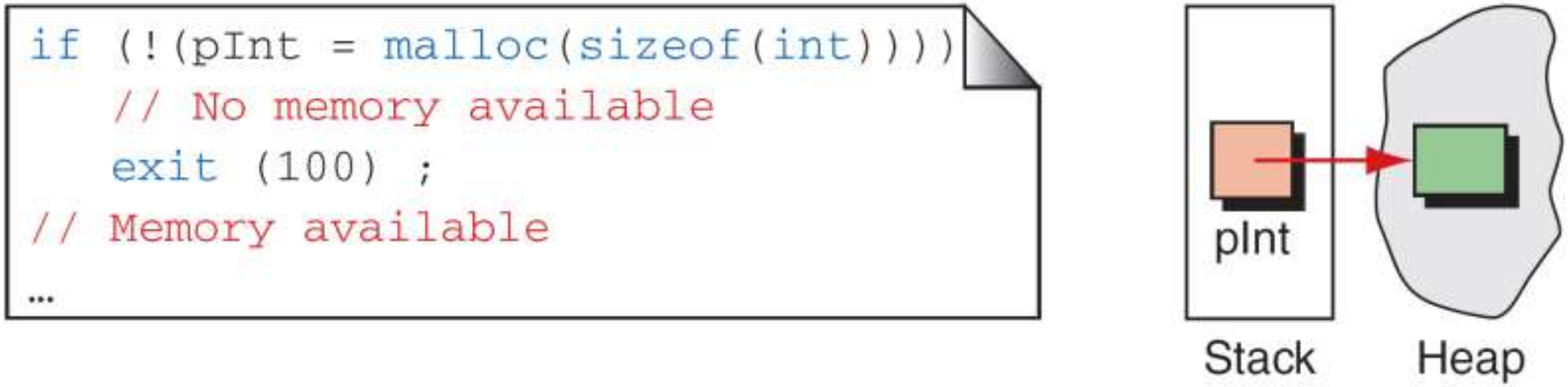


FIGURE 10-14 **Memory Management Functions**

Note

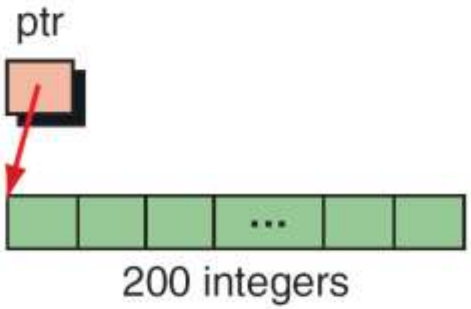
Memory Allocation Casting

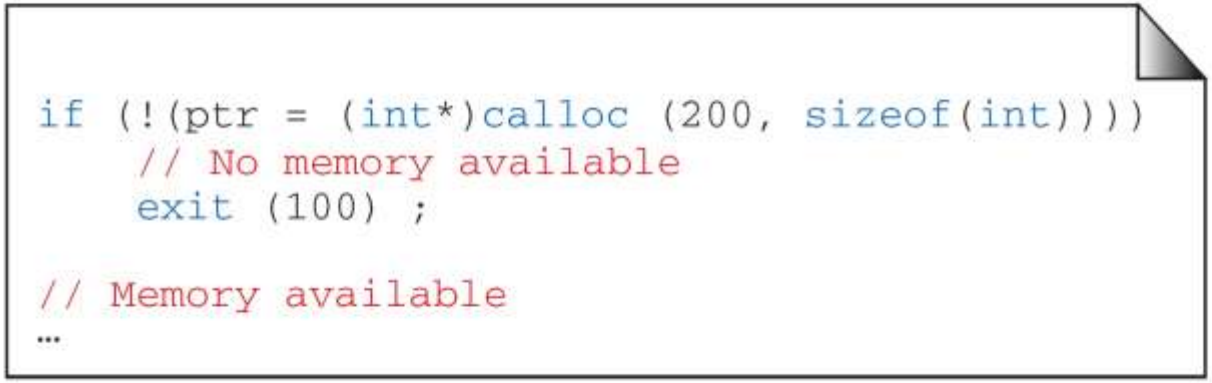
Prior to C99, it was necessary to cast the pointer returned from a memory allocation function. While it is no longer necessary, it does no harm as long as the cast is correct. If you should be working with an earlier standard, the casting format is: pointer = (type*) malloc(size)



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

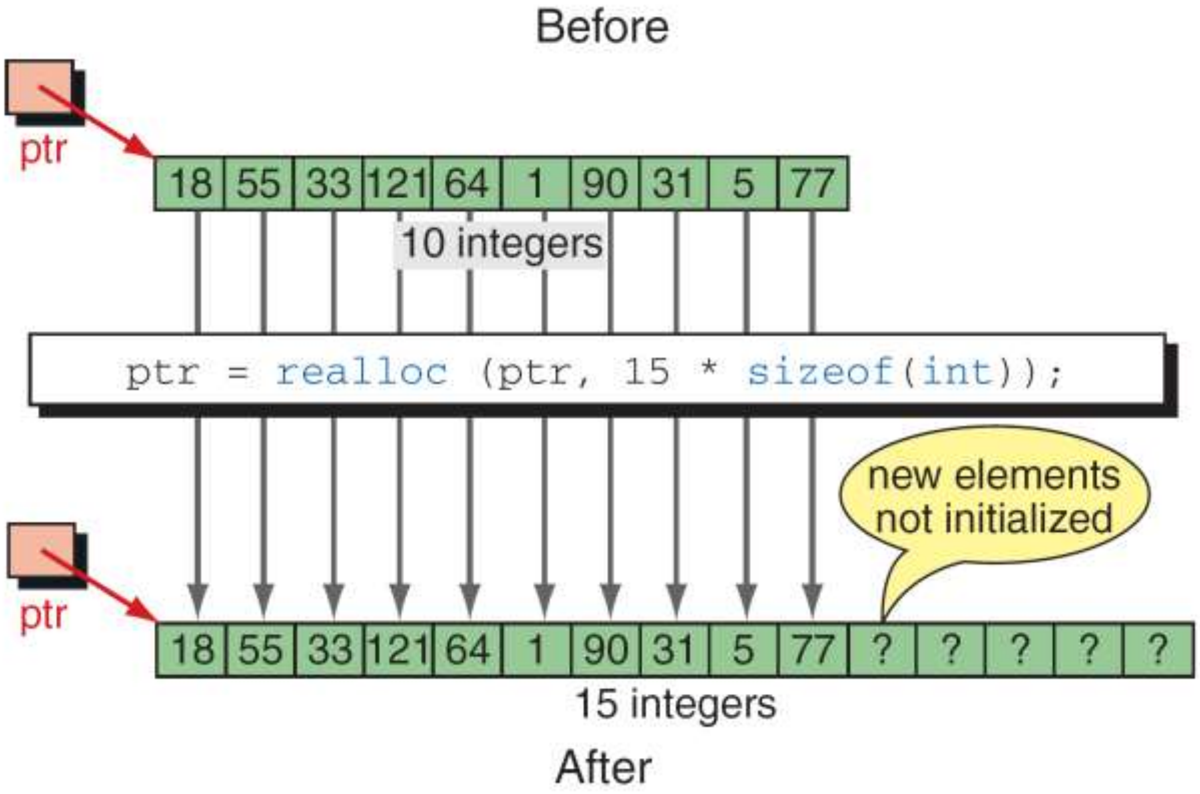
FIGURE 10-15 *malloc*





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

// Memory available
...
```

FIGURE 10-16 *calloc*



```
ptr = realloc (ptr, 15 * sizeof(int));
```

FIGURE 10-17 *realloc*

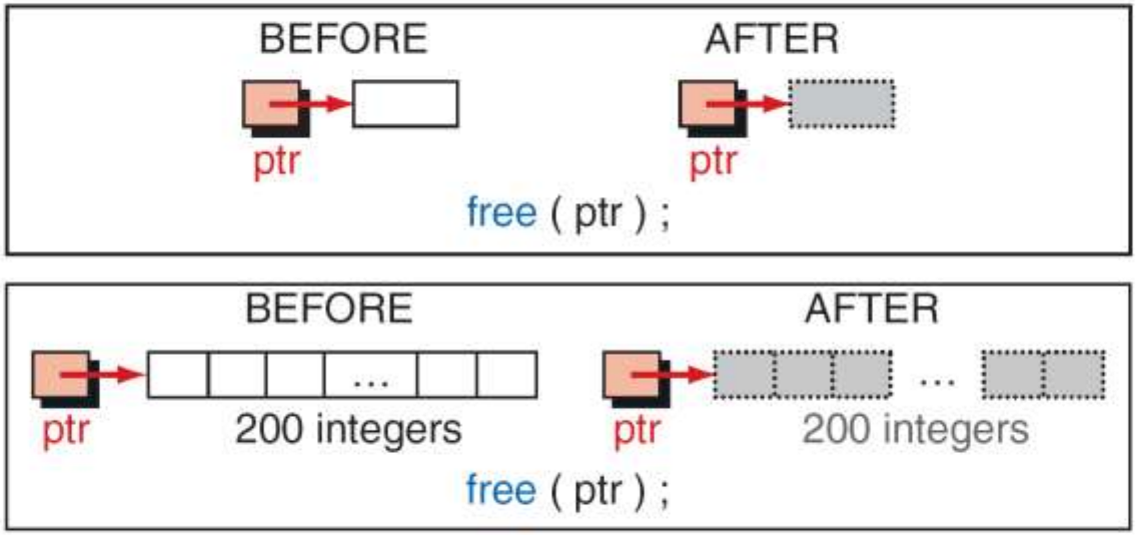


FIGURE 10-18 **Freeing Memory**

Note

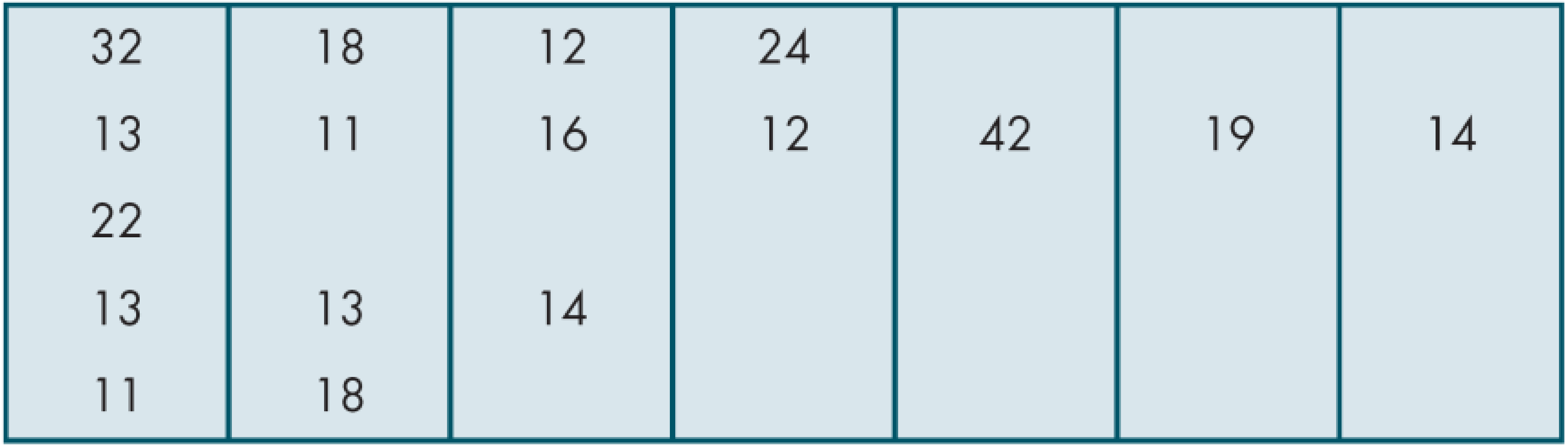
Using a pointer after its memory has been released is a common programming error. Guard against it by clearing the pointer.

Note

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

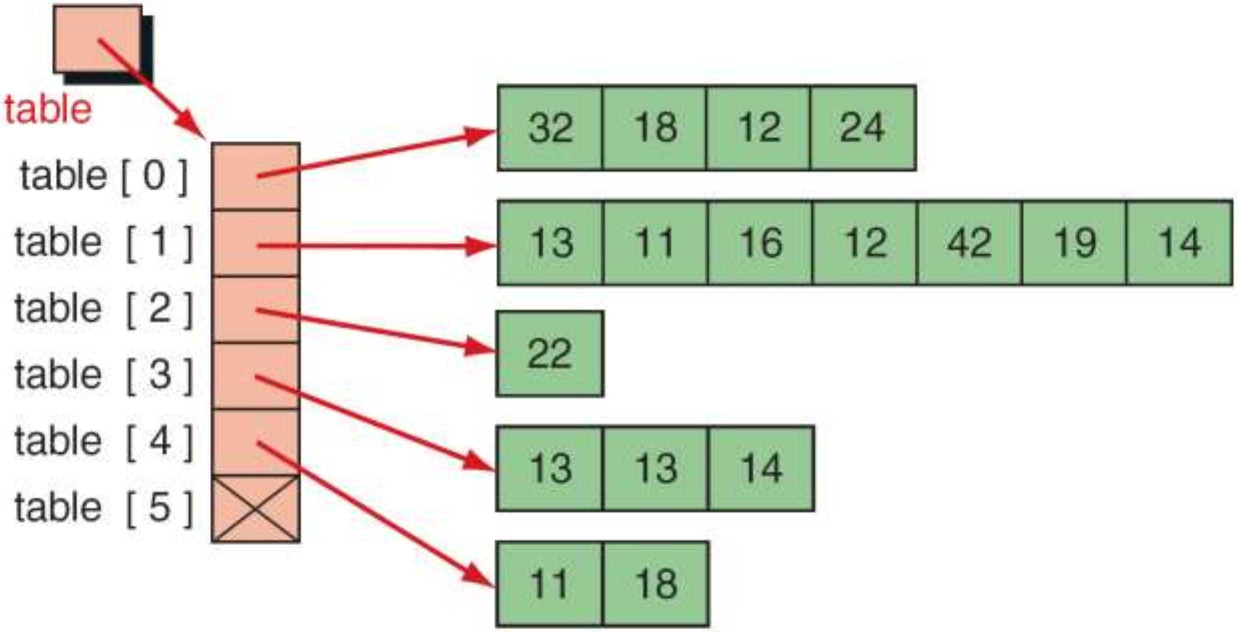
10-5 Array of Pointers

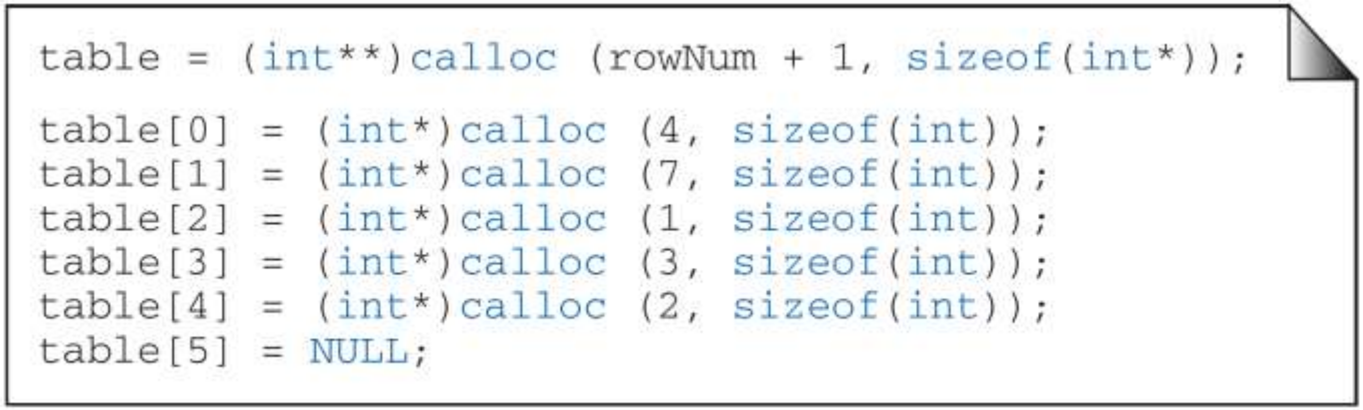
Another useful structure that uses arrays and pointers is an array of pointers. This structure is especially helpful when the number of elements in the array is variable.



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

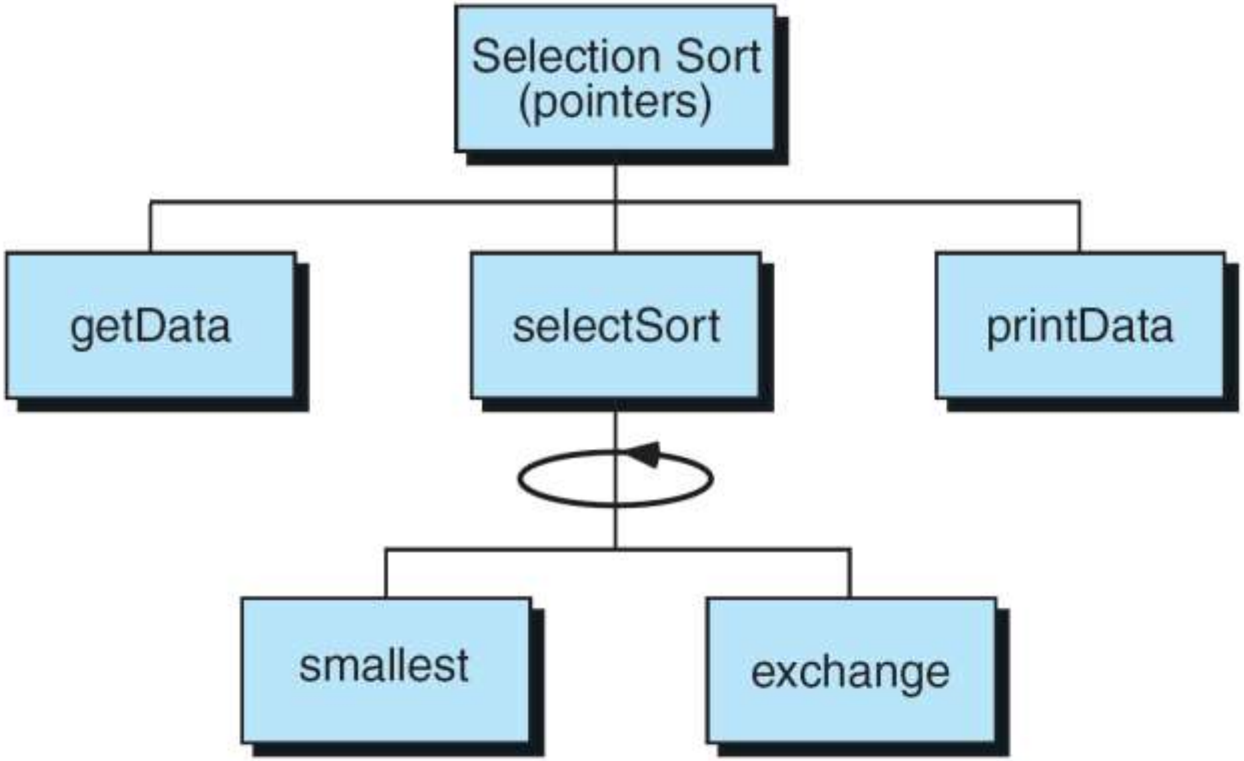
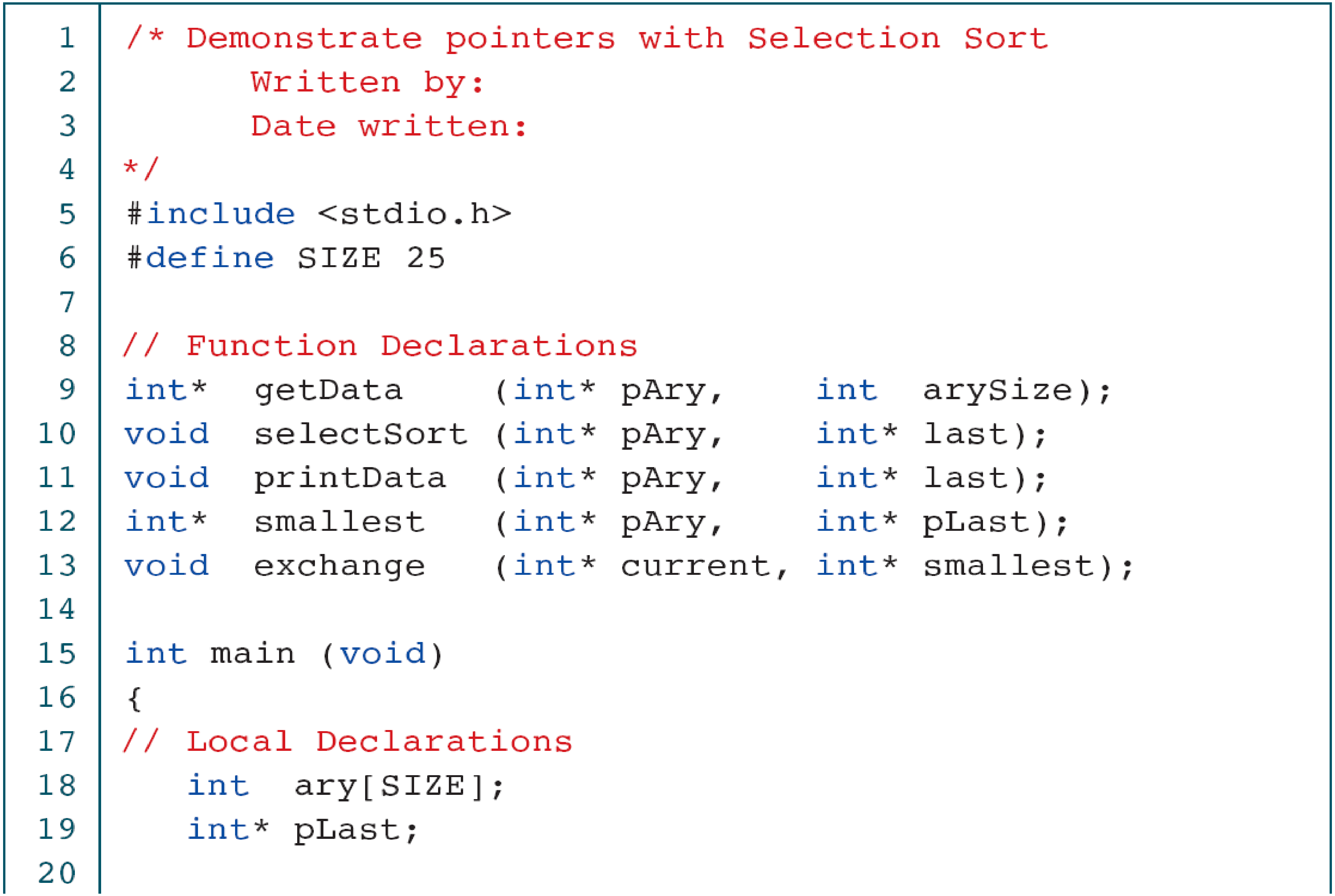


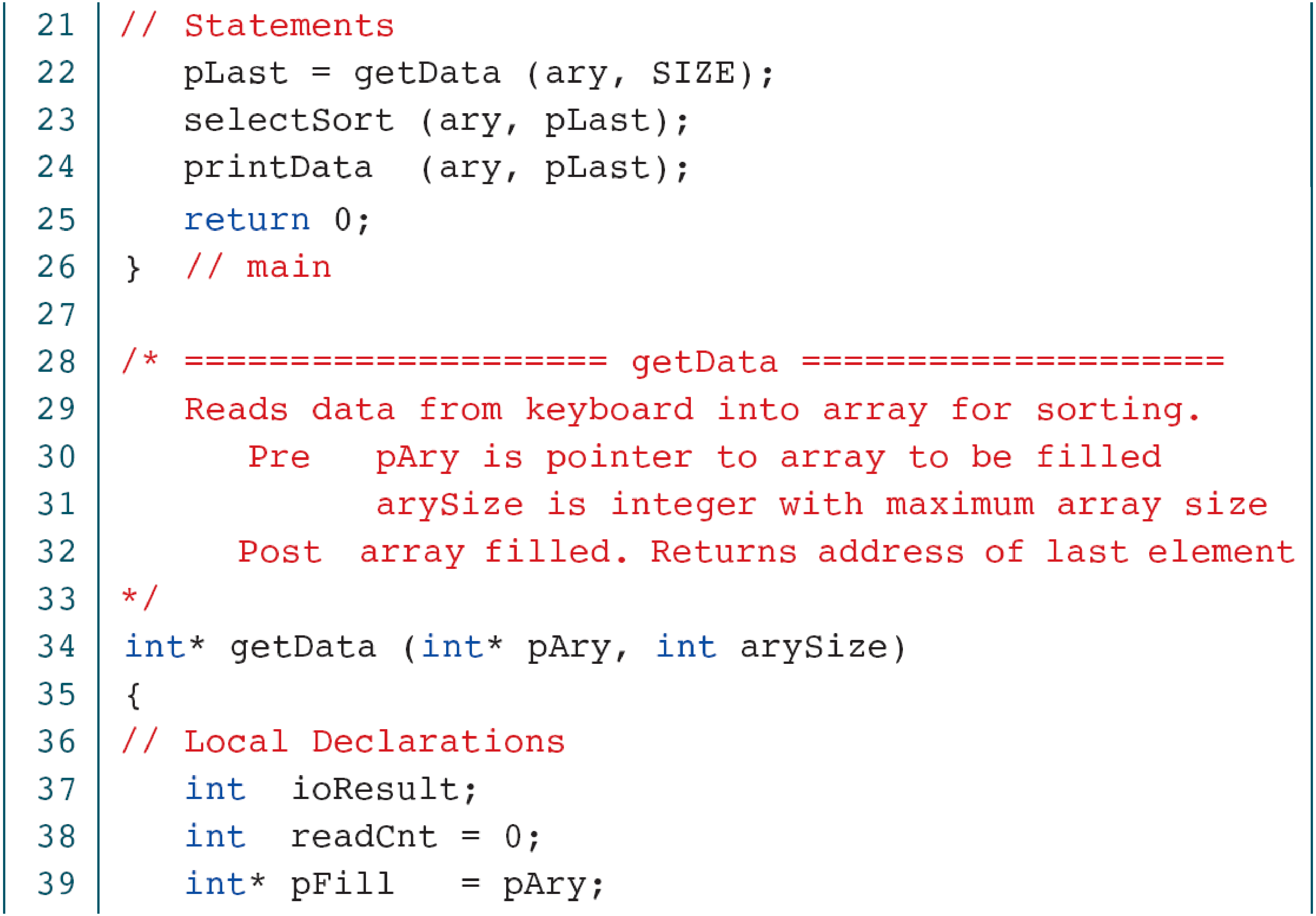
FIGURE 10-20 Selection Sort with Pointers—Structure Chart

PROGRAM 10-4 Selection Sort Revisited



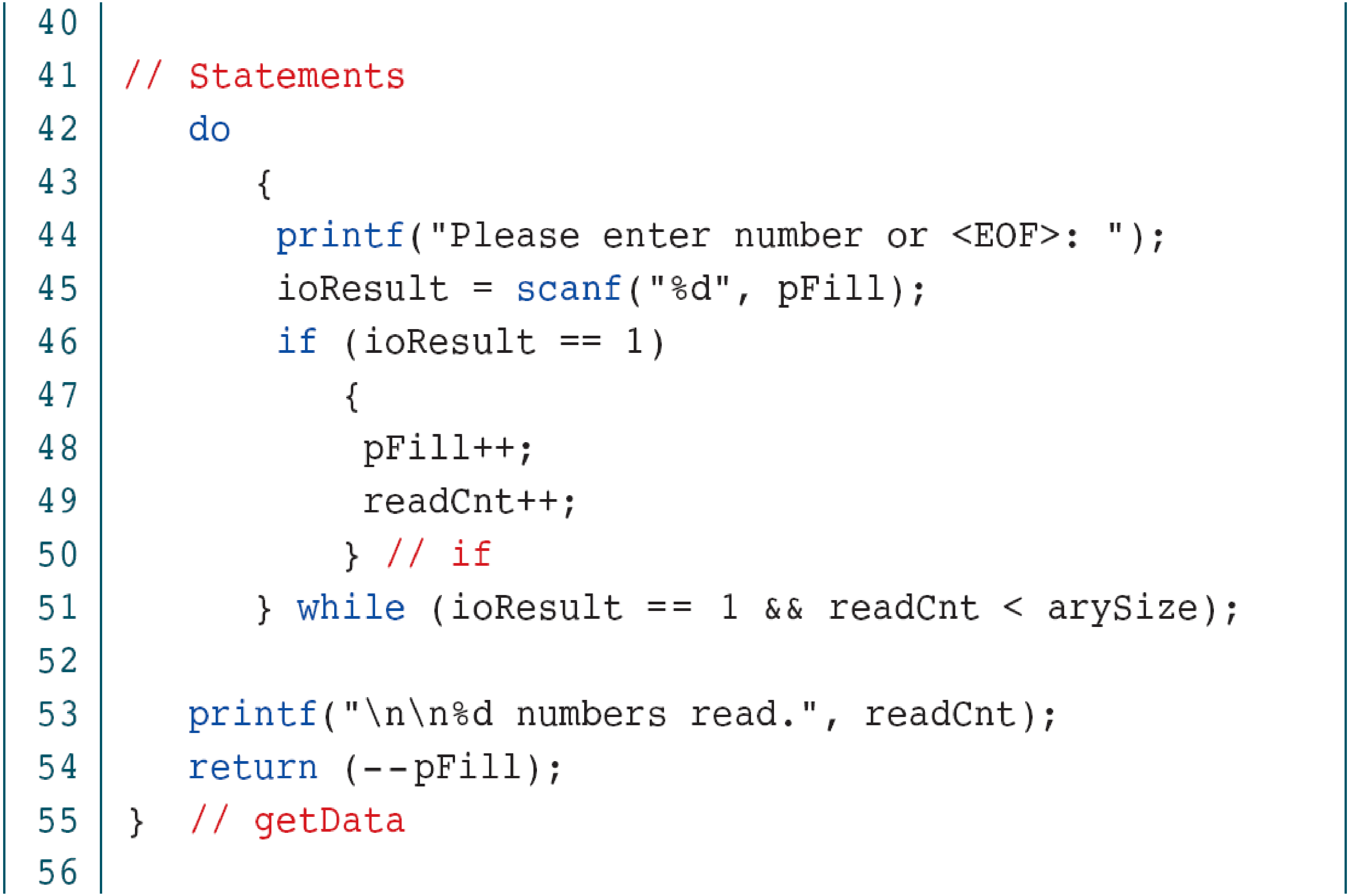
```c
/* Demonstrate pointers with Selection Sort
      Written by:
      Date written:
*/
#include <stdio.h>
#define SIZE 25

// Function Declarations
int*  getData     (int* pAry,     int  arySize);
void  selectSort (int* pAry,     int* last);
void  printData  (int* pAry,     int* last);
int*  smallest   (int* pAry,     int* pLast);
void  exchange   (int* current, int* smallest);

int main (void)
{
// Local Declarations
   int  ary[SIZE];
   int* pLast;

```

PROGRAM 10-4 Selection Sort Revisited



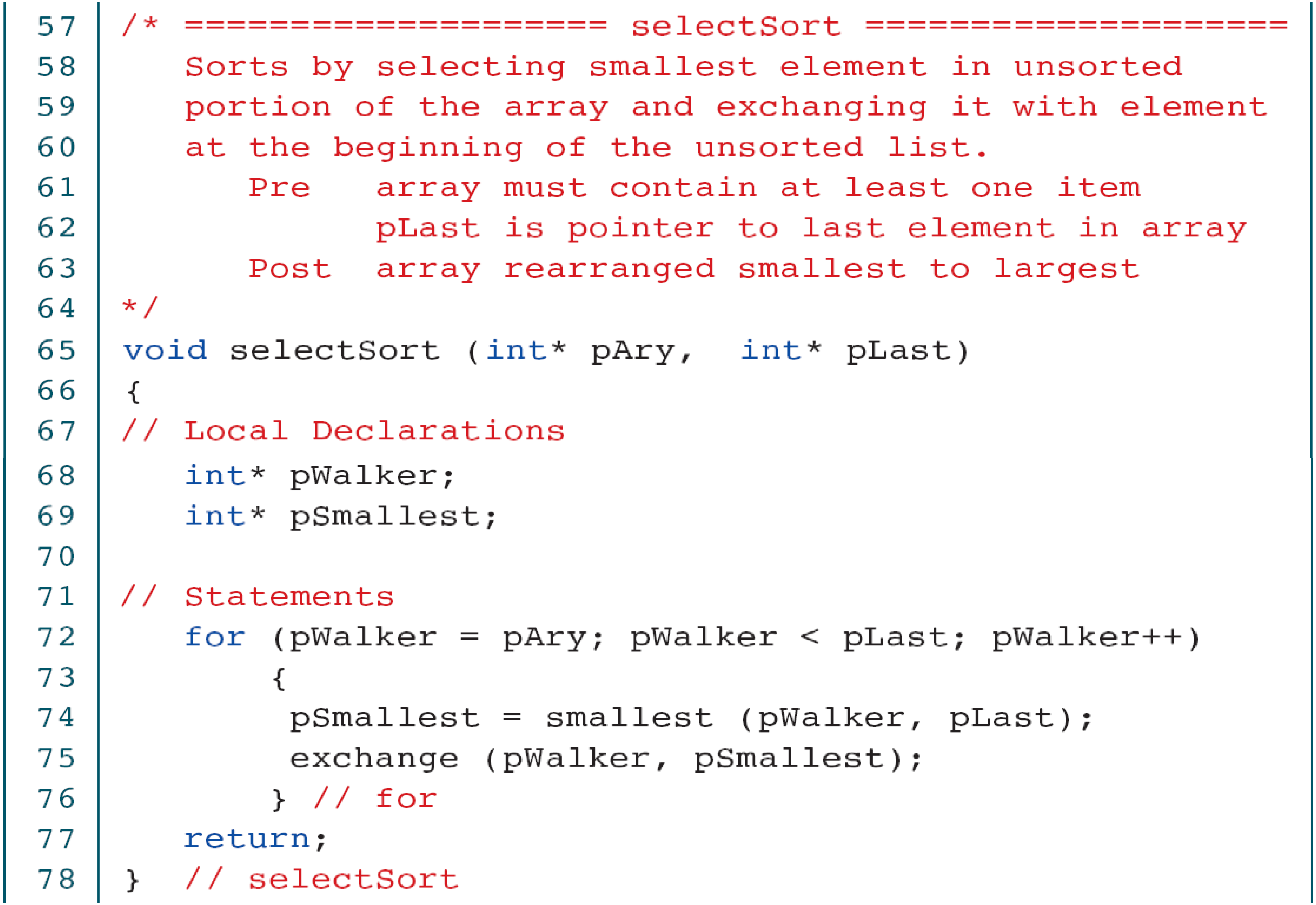
```
// Statements
   pLast = getData (ary, SIZE);
   selectSort (ary, pLast);
   printData  (ary, pLast);
   return 0;
}  // main

/* ==================== getData ====================
   Reads data from keyboard into array for sorting.
      Pre   pAry is pointer to array to be filled
            arySize is integer with maximum array size
      Post  array filled. Returns address of last element
*/
int* getData (int* pAry, int arySize)
{
// Local Declarations
   int  ioResult;
   int  readCnt = 0;
   int* pFill   = pAry;
```

PROGRAM 10-4 Selection Sort Revisited



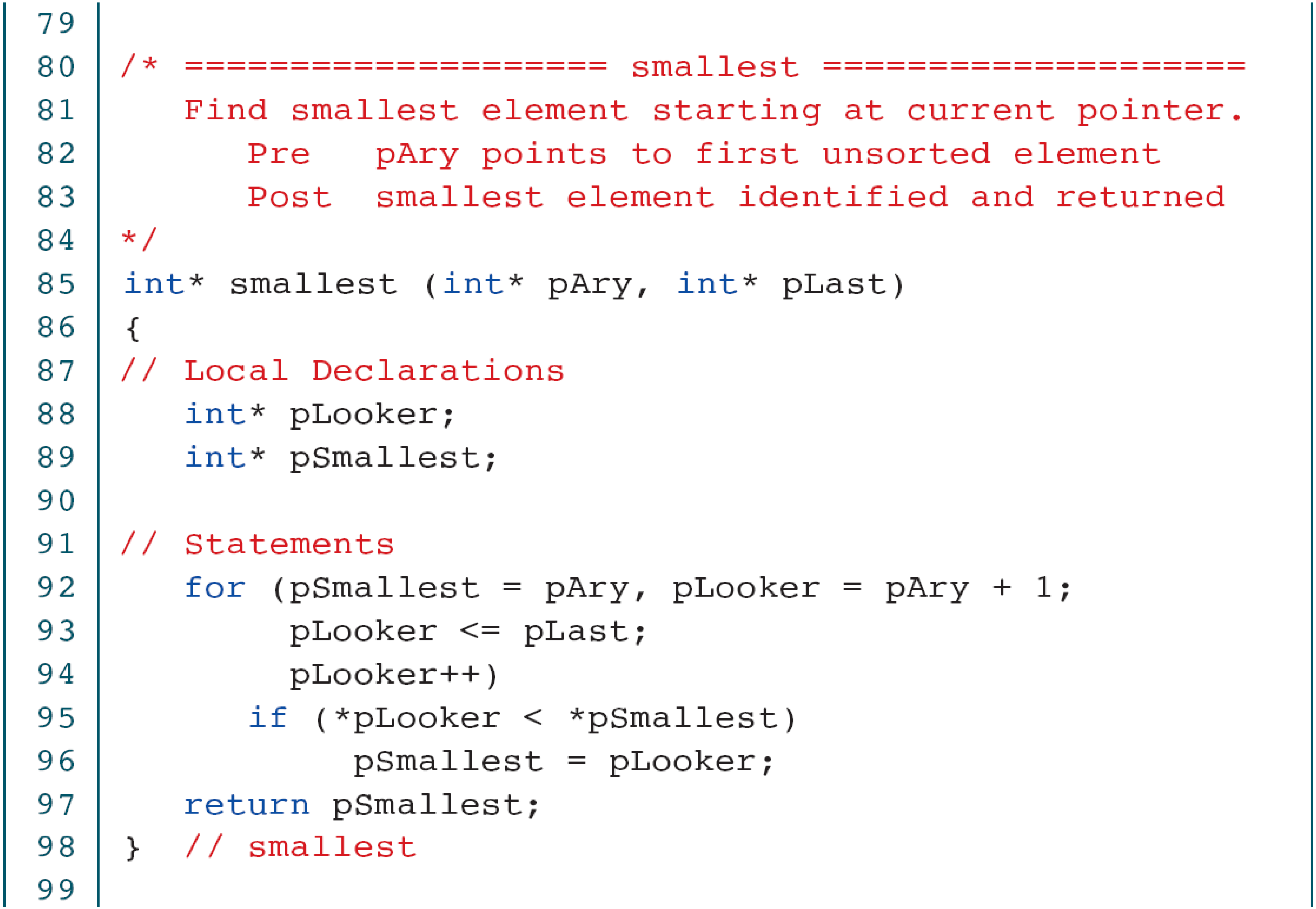
```
// Statements
   do
      {
       printf("Please enter number or <EOF>: ");
       ioResult = scanf("%d", pFill);
       if (ioResult == 1)
          {
           pFill++;
           readCnt++;
          } // if
      } while (ioResult == 1 && readCnt < arySize);

   printf("\n\n%d numbers read.", readCnt);
   return (--pFill);
}  // getData
```

PROGRAM 10-4 Selection Sort Revisited



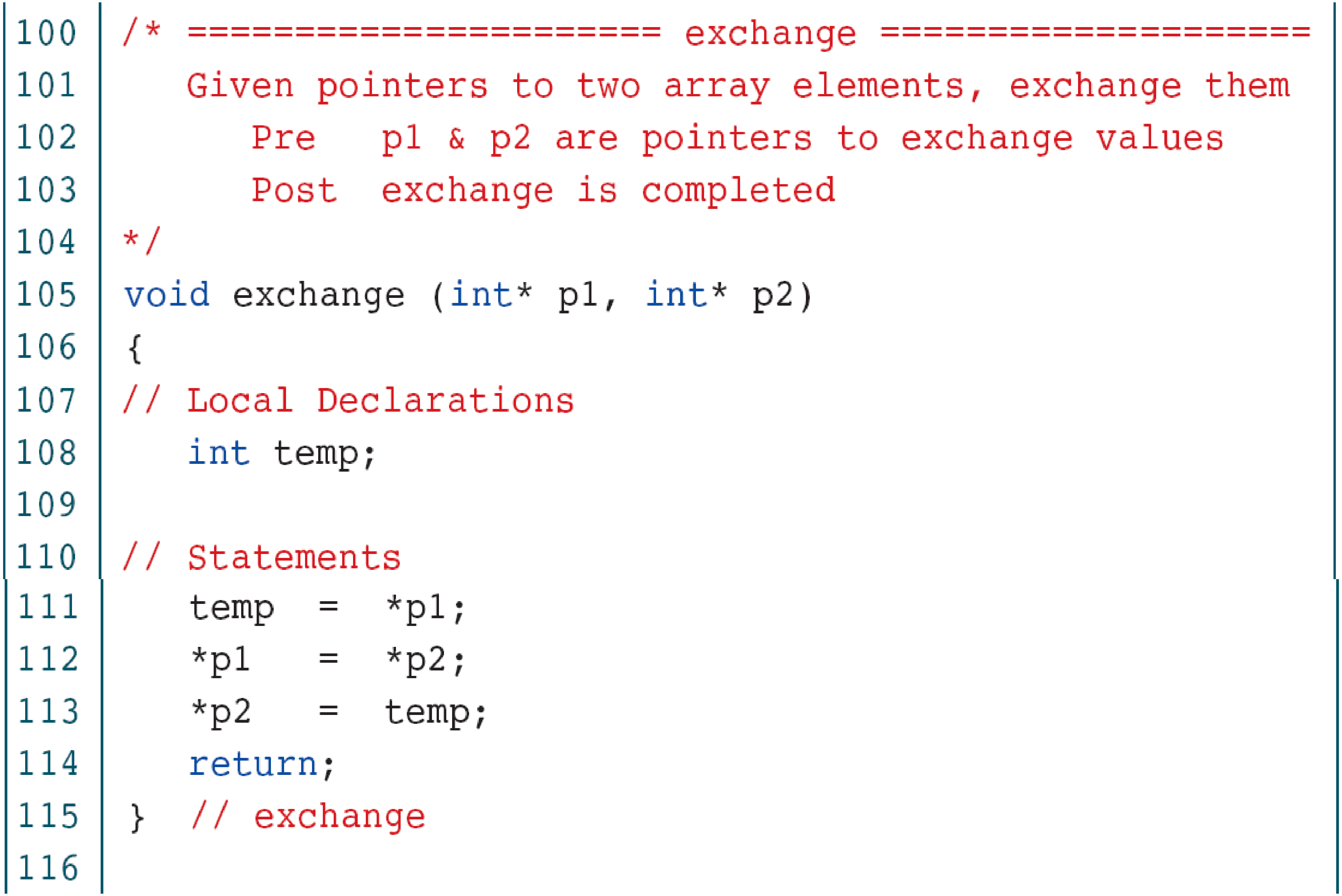
```c
/* ==================== selectSort ====================
   Sorts by selecting smallest element in unsorted
   portion of the array and exchanging it with element
   at the beginning of the unsorted list.
      Pre    array must contain at least one item
             pLast is pointer to last element in array
      Post   array rearranged smallest to largest
*/
void selectSort (int* pAry,  int* pLast)
{
// Local Declarations
   int* pWalker;
   int* pSmallest;

// Statements
   for (pWalker = pAry; pWalker < pLast; pWalker++)
       {
        pSmallest = smallest (pWalker, pLast);
        exchange (pWalker, pSmallest);
       } // for
   return;
}  // selectSort
```

PROGRAM 10-4 Selection Sort Revisited



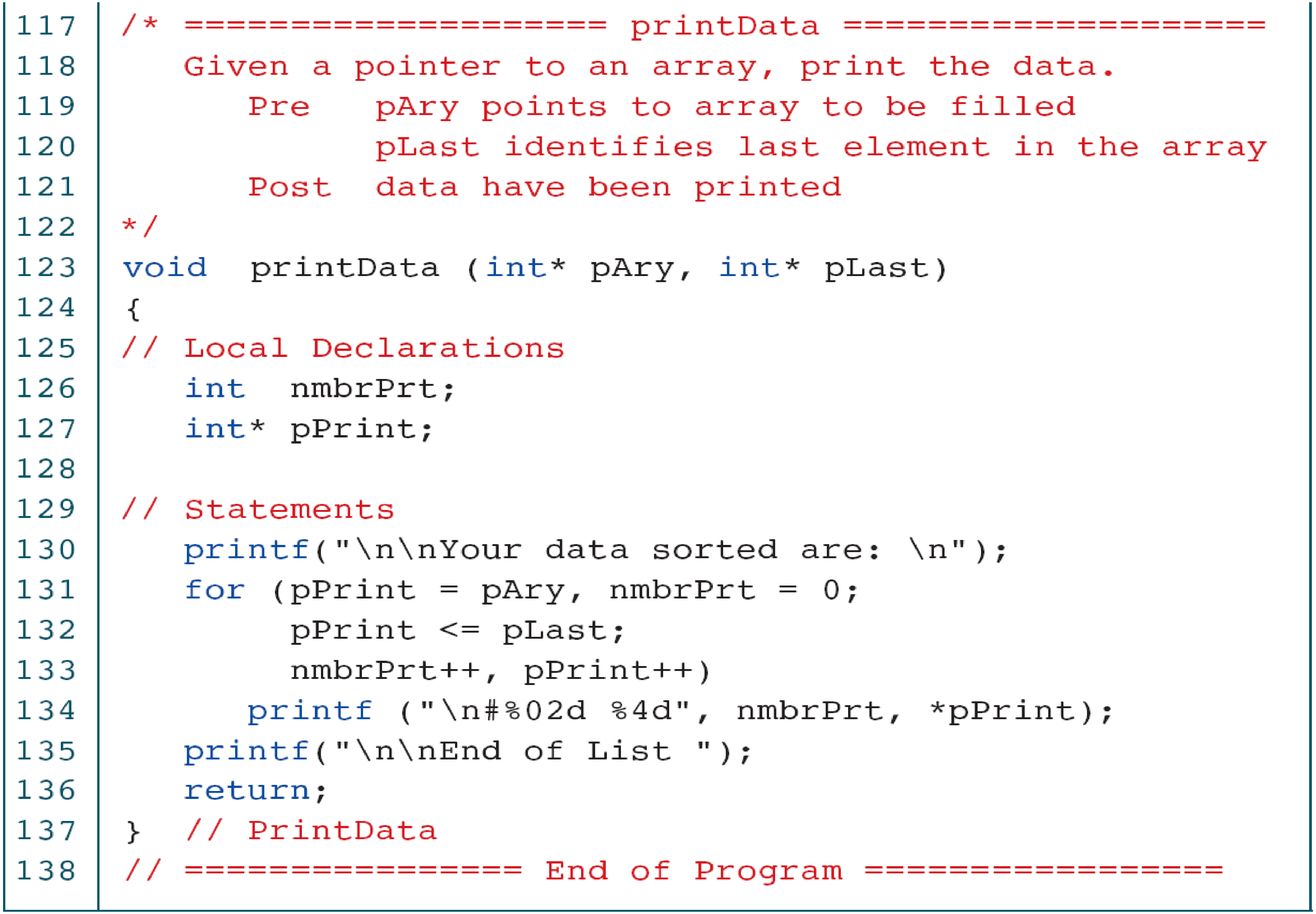
```c

/* ==================== smallest ====================
   Find smallest element starting at current pointer.
      Pre    pAry points to first unsorted element
      Post   smallest element identified and returned
*/
int* smallest (int* pAry, int* pLast)
{
// Local Declarations
   int* pLooker;
   int* pSmallest;

// Statements
   for (pSmallest = pAry, pLooker = pAry + 1;
        pLooker <= pLast;
        pLooker++)
      if (*pLooker < *pSmallest)
           pSmallest = pLooker;
   return pSmallest;
}  // smallest

```

PROGRAM 10-4 Selection Sort Revisited



```c
/* ====================== exchange ====================
   Given pointers to two array elements, exchange them
      Pre   p1 & p2 are pointers to exchange values
      Post  exchange is completed
*/
void exchange (int* p1, int* p2)
{
// Local Declarations
   int temp;

// Statements
   temp  =  *p1;
   *p1   =  *p2;
   *p2   =  temp;
   return;
}  // exchange

```

PROGRAM 10-4 Selection Sort Revisited



```
/* ==================== printData ====================
   Given a pointer to an array, print the data.
      Pre    pAry points to array to be filled
             pLast identifies last element in the array
      Post   data have been printed
*/
void  printData (int* pAry, int* pLast)
{
// Local Declarations
   int  nmbrPrt;
   int* pPrint;

// Statements
   printf("\n\nYour data sorted are: \n");
   for (pPrint = pAry, nmbrPrt = 0;
        pPrint <= pLast;
        nmbrPrt++, pPrint++)
      printf ("\n#%02d %4d", nmbrPrt, *pPrint);
   printf("\n\nEnd of List ");
   return;
}  // PrintData
// ================ End of Program =================
```

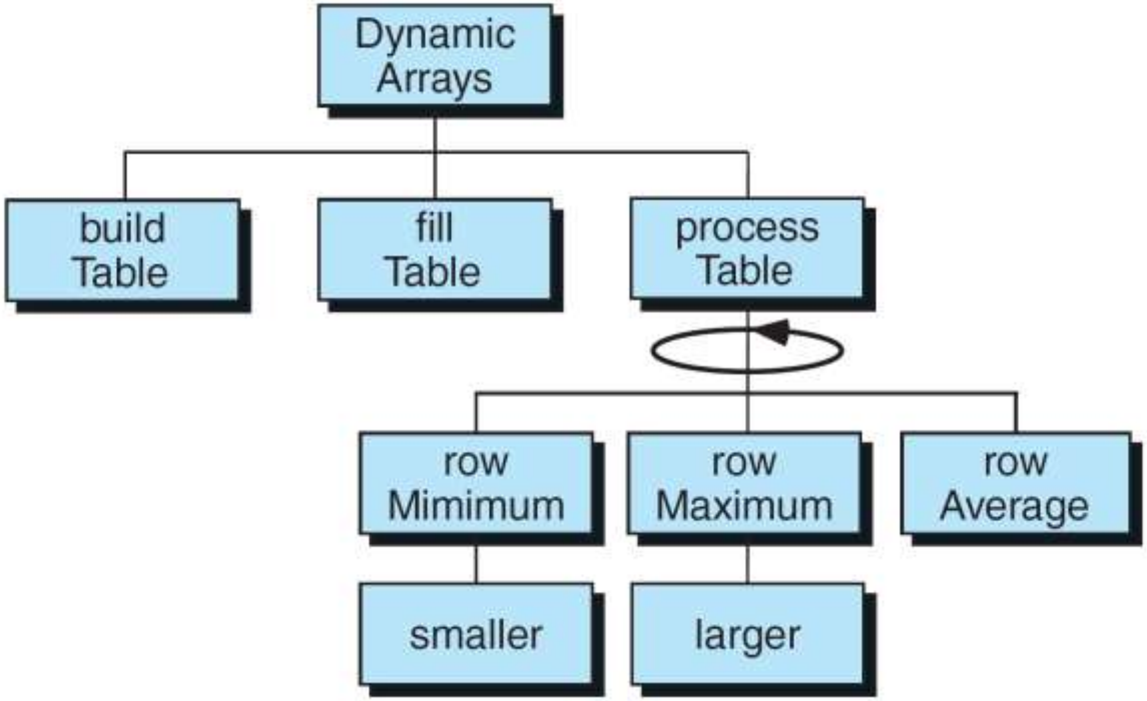


FIGURE 10-21 Dynamic Array Structure Chart

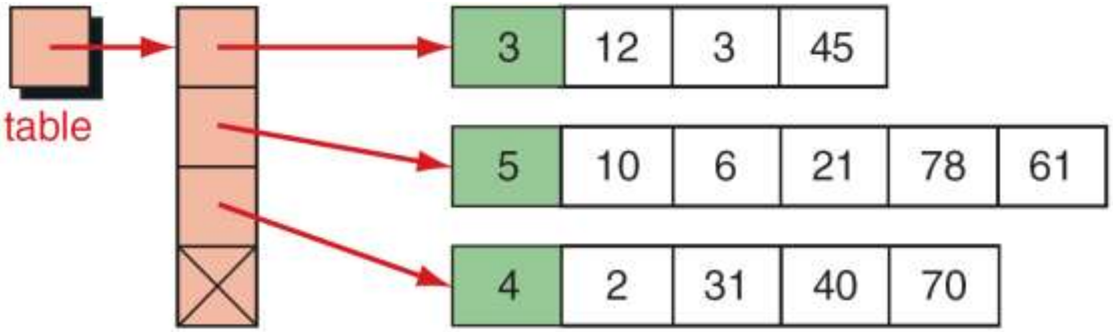
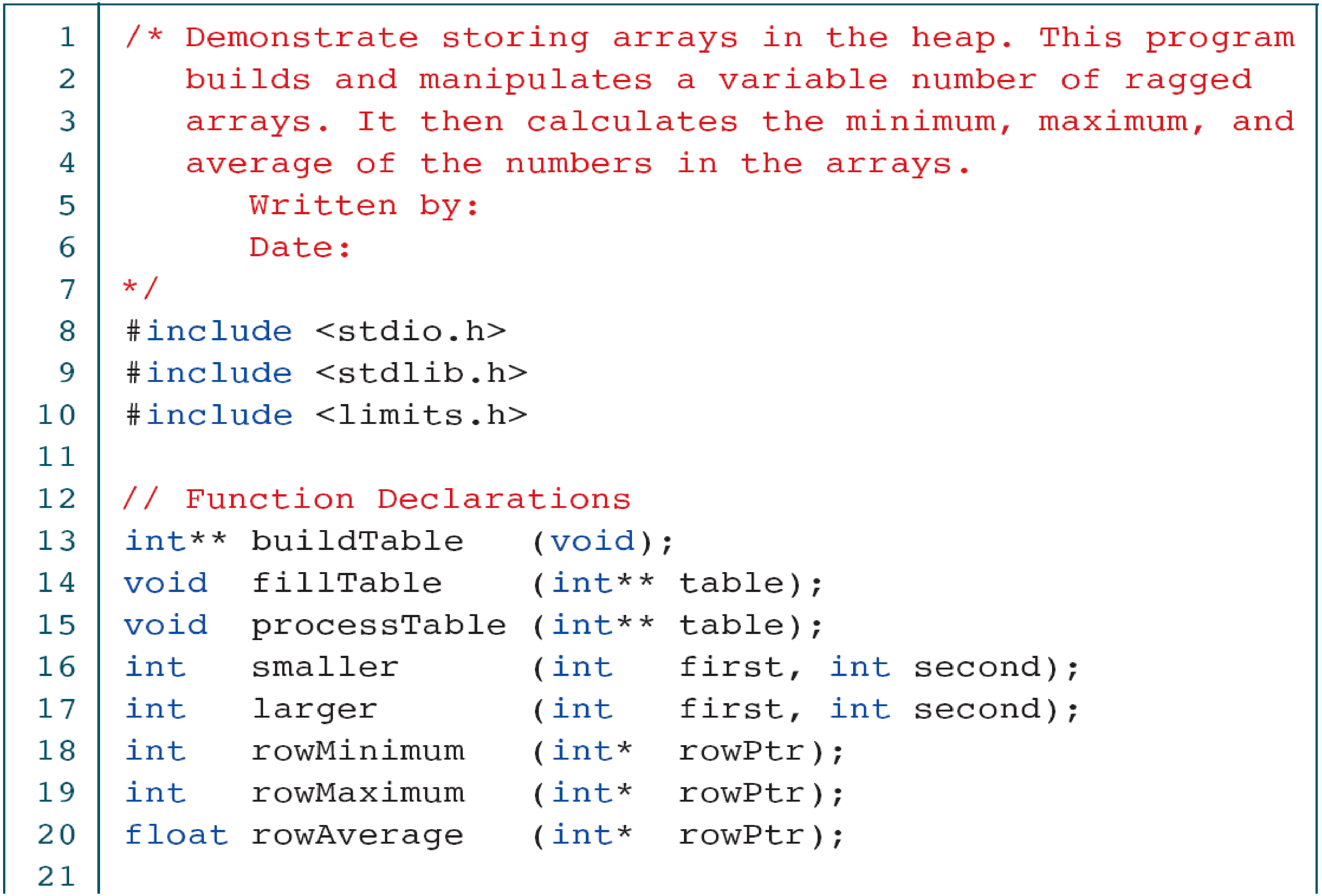


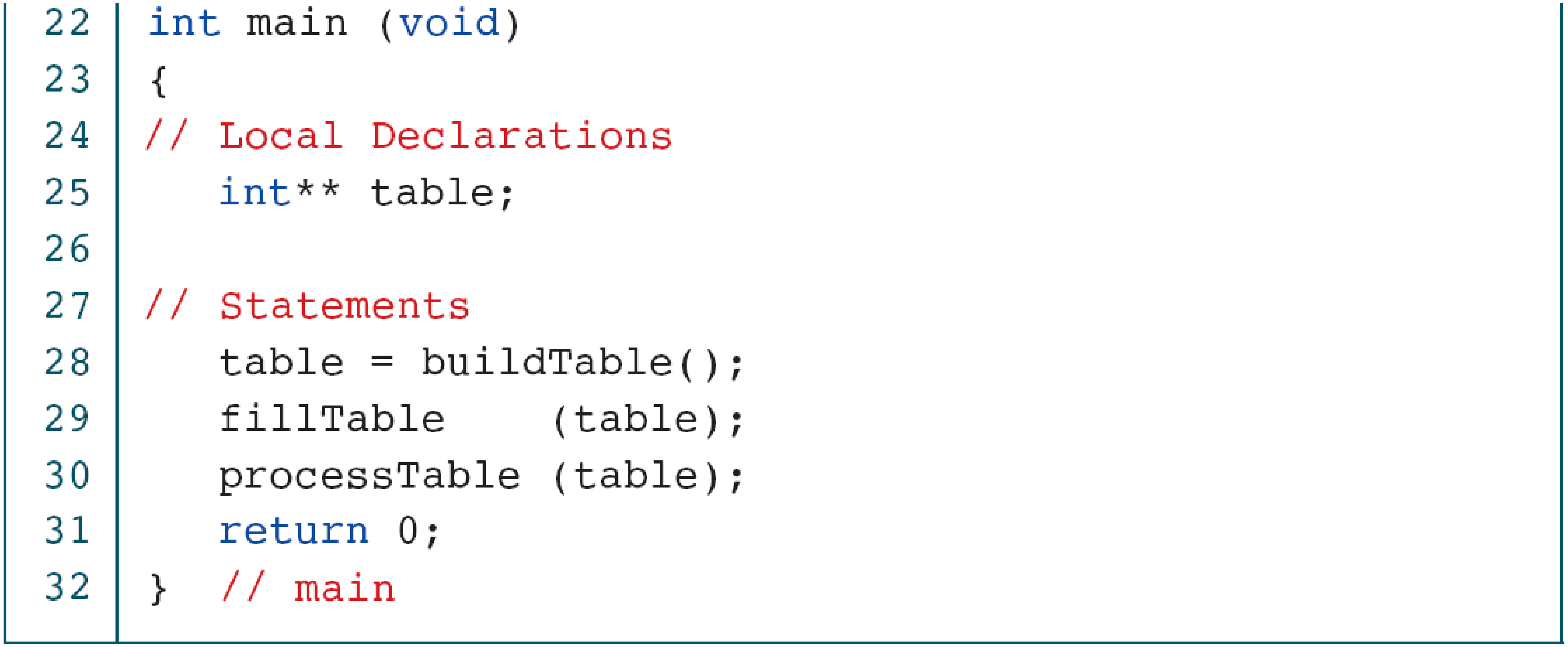
FIGURE 10-22 Ragged Array Structure

PROGRAM 10-5 Dynamic Arrays: main



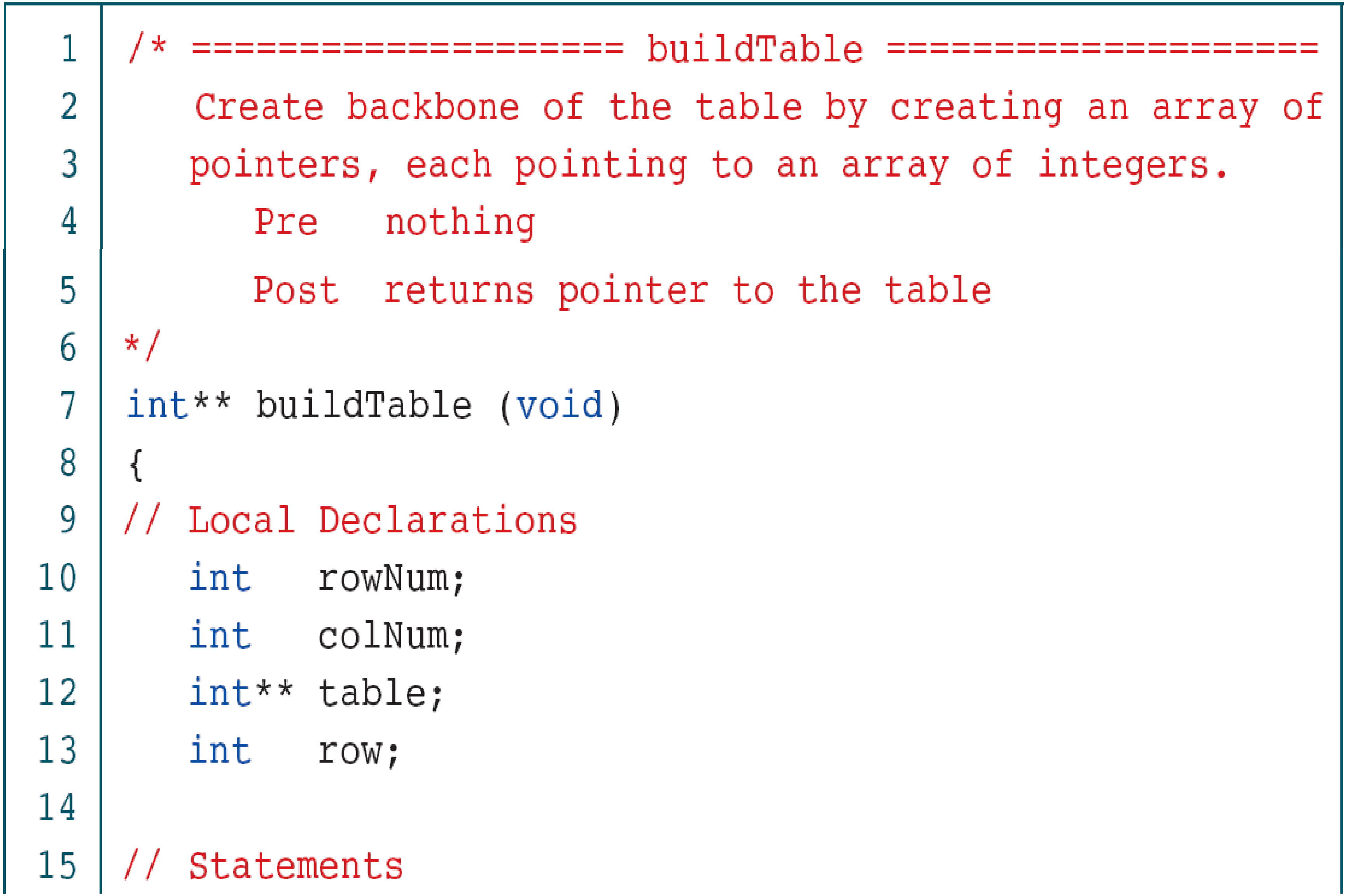
```c
/* Demonstrate storing arrays in the heap. This program
   builds and manipulates a variable number of ragged
   arrays. It then calculates the minimum, maximum, and
   average of the numbers in the arrays.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// Function Declarations
int** buildTable   (void);
void  fillTable    (int** table);
void  processTable (int** table);
int   smaller      (int   first, int second);
int   larger       (int   first, int second);
int   rowMinimum   (int*  rowPtr);
int   rowMaximum   (int*  rowPtr);
float rowAverage   (int*  rowPtr);

```

PROGRAM 10-5 Dynamic Arrays: main



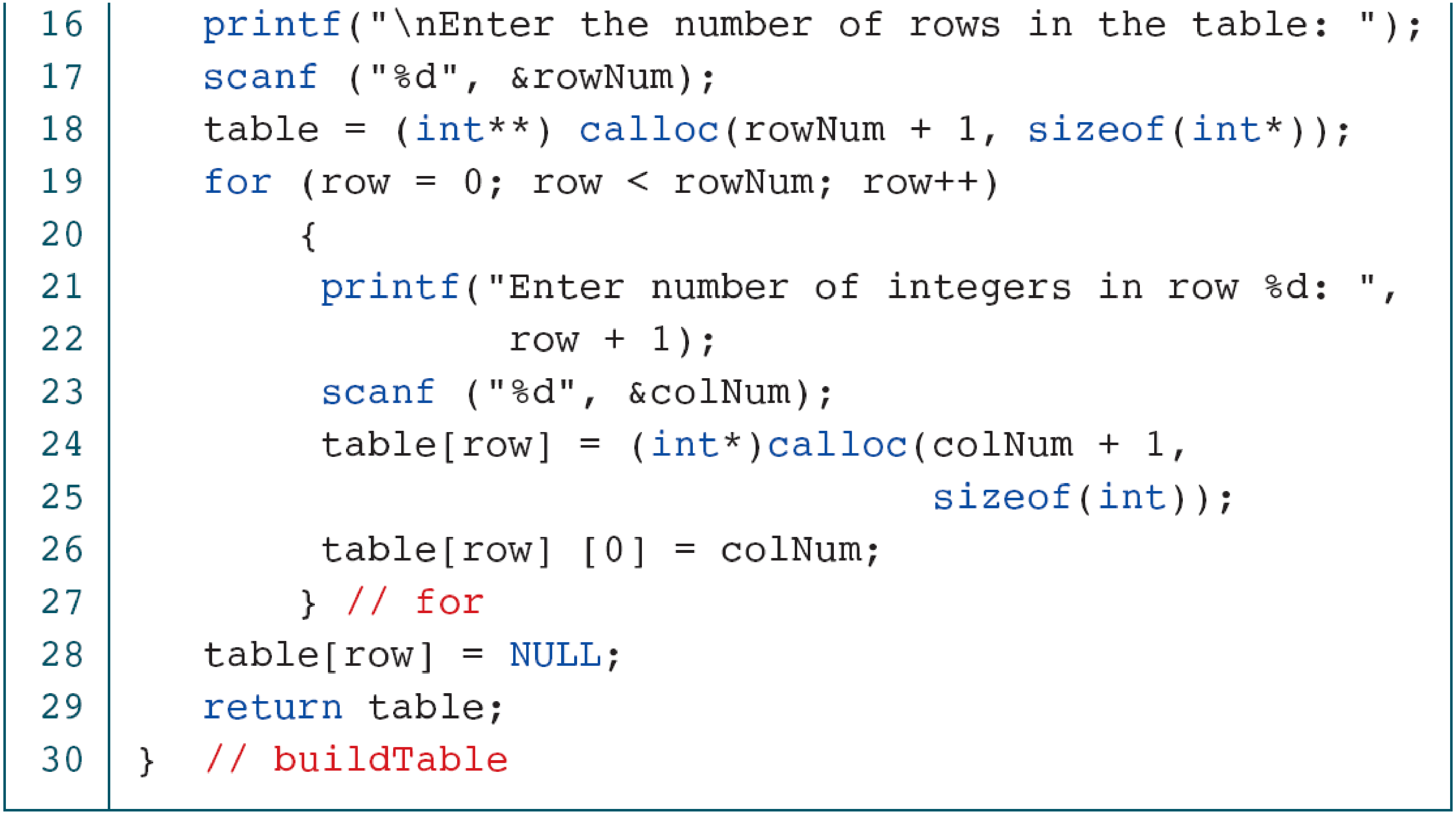
```c
int main (void)
{
//  Local Declarations
    int** table;

//  Statements
    table = buildTable();
    fillTable    (table);
    processTable (table);
    return 0;
}  // main
```

PROGRAM 10-6 Dynamic Arrays: buildTable



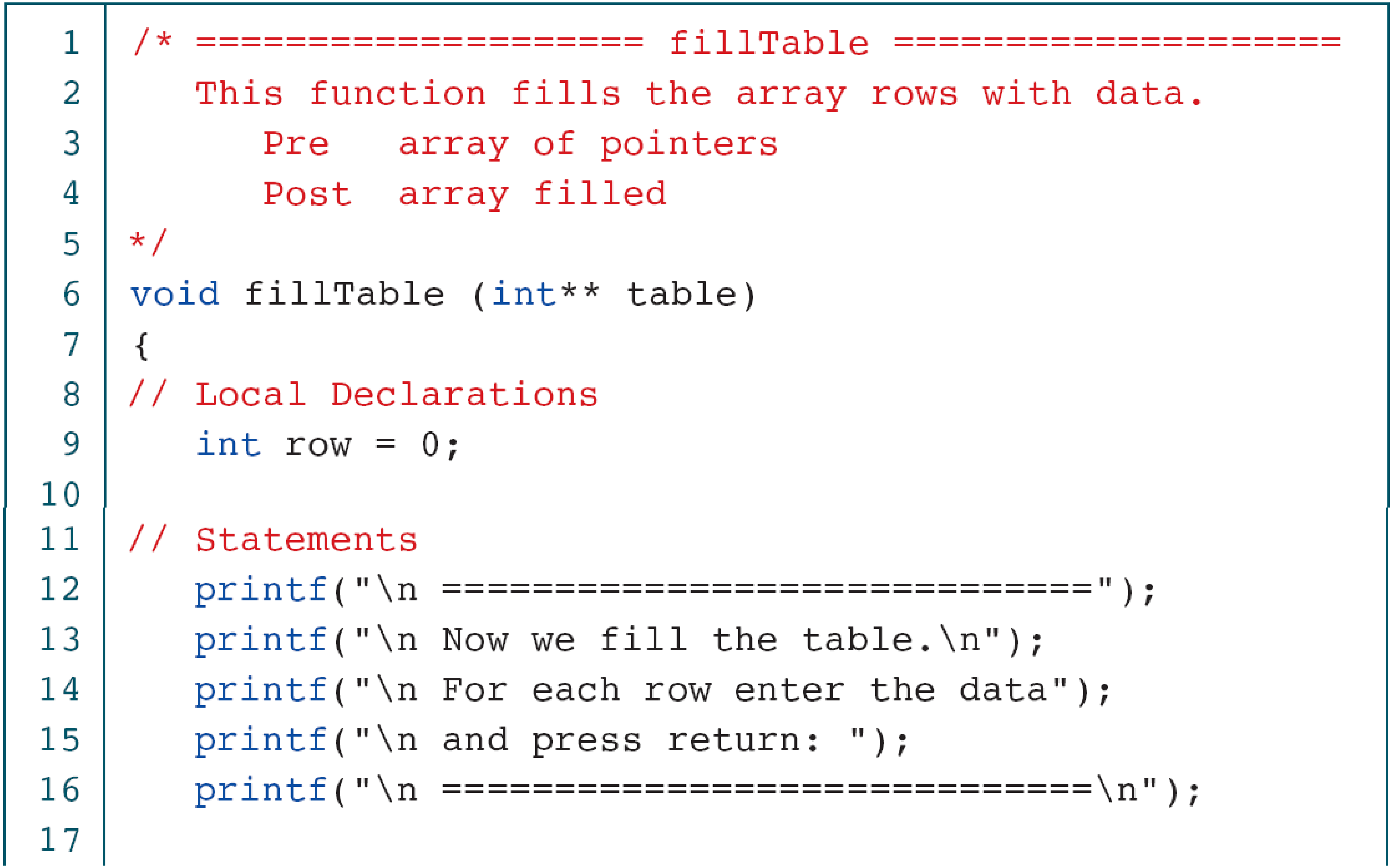
```
/* ==================== buildTable =====================
   Create backbone of the table by creating an array of
   pointers, each pointing to an array of integers.
      Pre   nothing
      Post  returns pointer to the table
*/
int** buildTable (void)
{
// Local Declarations
   int   rowNum;
   int   colNum;
   int** table;
   int   row;

// Statements
```

PROGRAM 10-5 Dynamic Arrays: buildTable



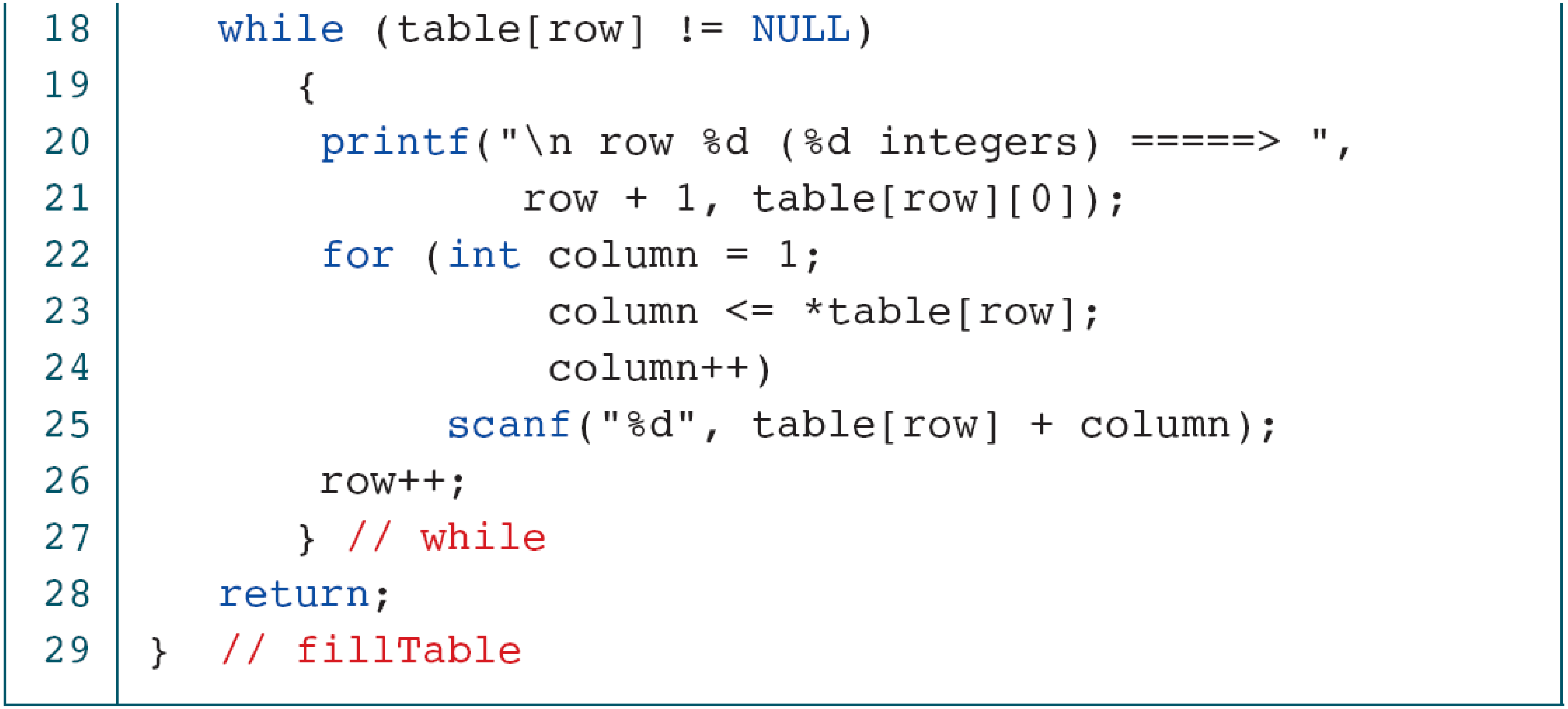
```
   printf("\nEnter the number of rows in the table: ");
   scanf ("%d", &rowNum);
   table = (int**) calloc(rowNum + 1, sizeof(int*));
   for (row = 0; row < rowNum; row++)
       {
        printf("Enter number of integers in row %d: ",
                row + 1);
        scanf ("%d", &colNum);
        table[row] = (int*)calloc(colNum + 1,
                                  sizeof(int));
        table[row] [0] = colNum;
       } // for
   table[row] = NULL;
   return table;
}  // buildTable
```

PROGRAM 10-7 Dynamic Arrays: fillTable



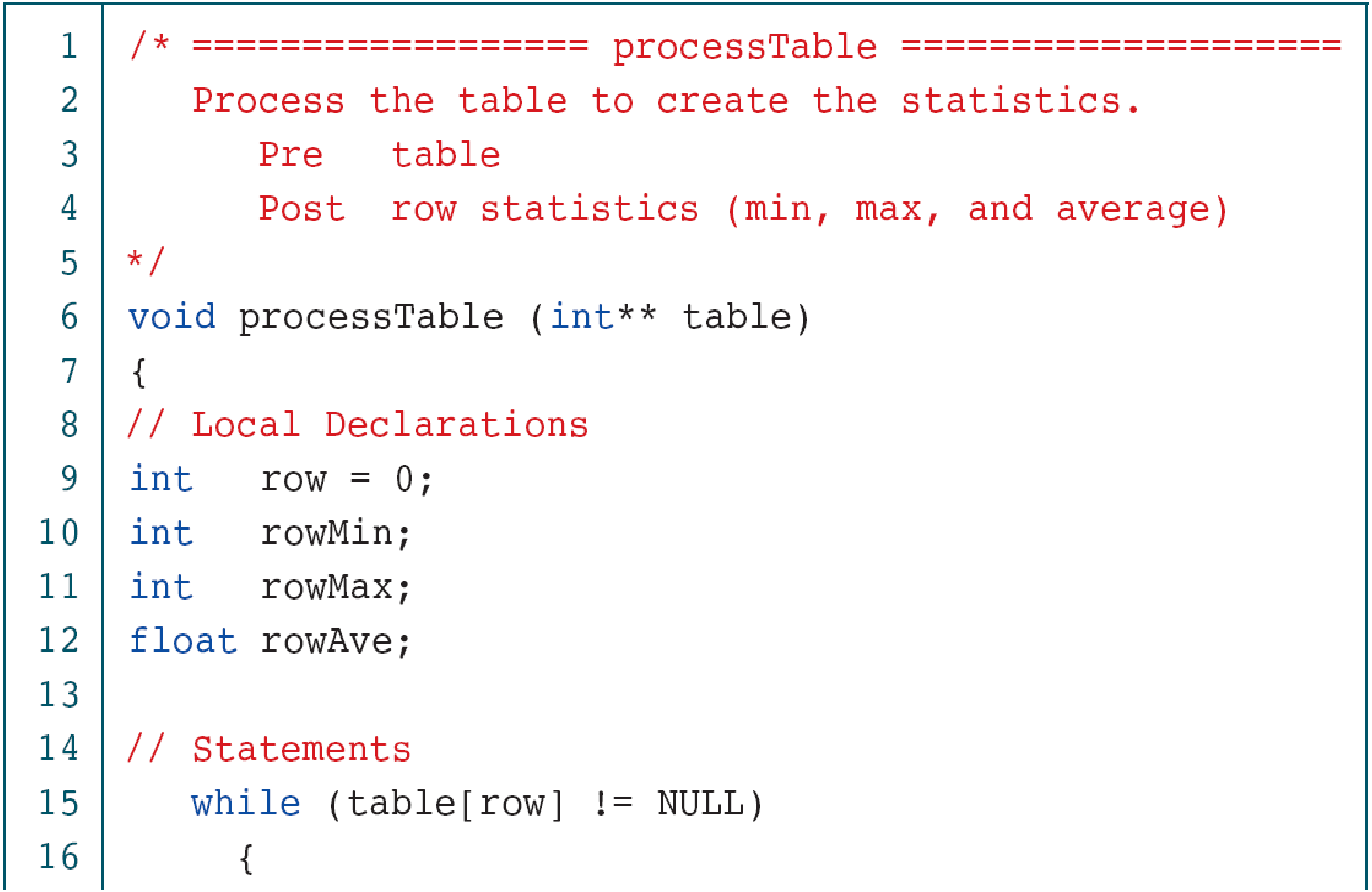
```c
/* ==================== fillTable ====================
   This function fills the array rows with data.
      Pre   array of pointers
      Post  array filled
*/
void fillTable (int** table)
{
// Local Declarations
   int row = 0;

// Statements
   printf("\n =============================");
   printf("\n Now we fill the table.\n");
   printf("\n For each row enter the data");
   printf("\n and press return: ");
   printf("\n =============================\n");

```

PROGRAM 10-7 Dynamic Arrays: fillTable



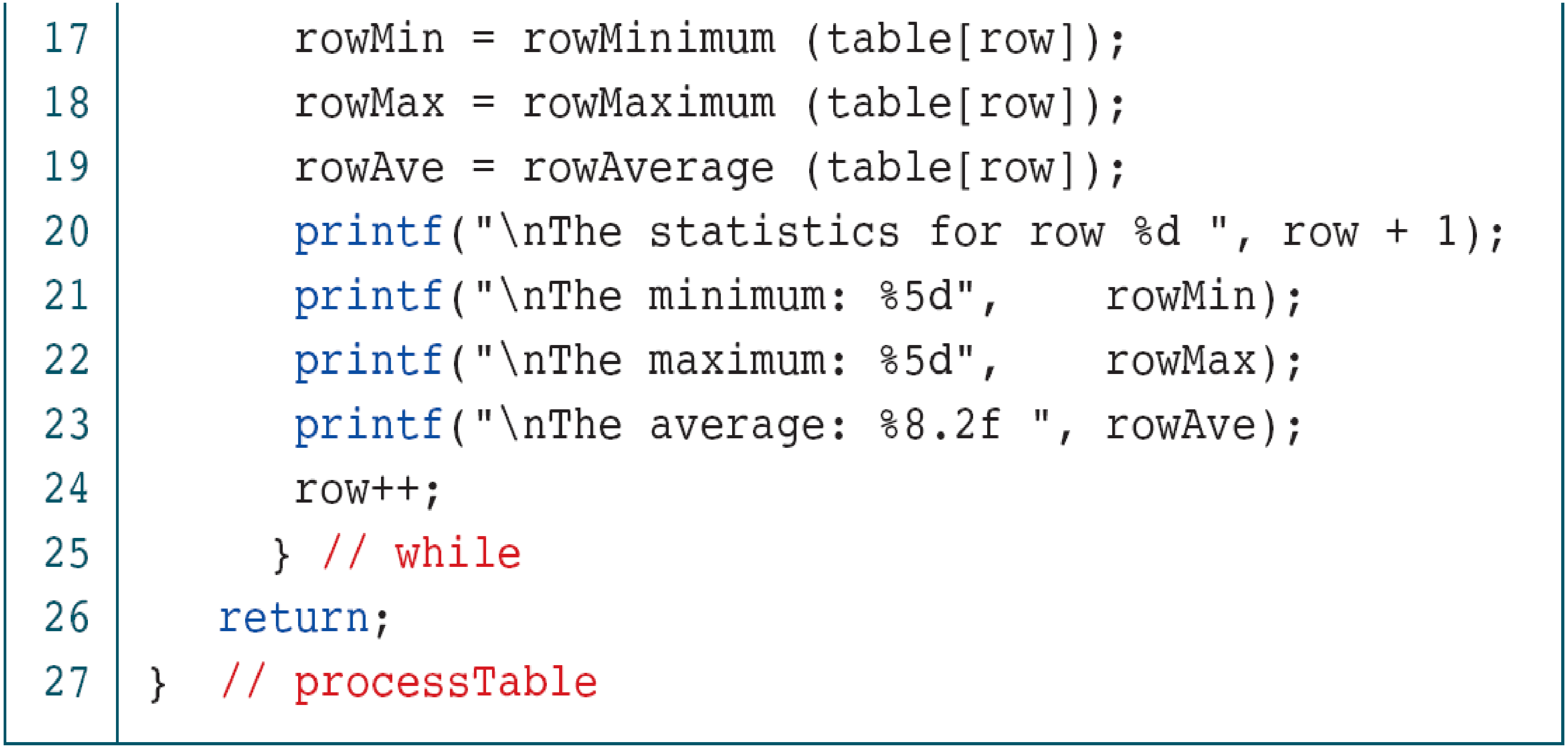
```c
   while (table[row] != NULL)
      {
       printf("\n row %d (%d integers) =====> ",
               row + 1, table[row][0]);
       for (int column = 1;
                column <= *table[row];
                column++)
            scanf("%d", table[row] + column);
       row++;
      } // while
   return;
}  // fillTable
```

PROGRAM 10-8 Dynamic Arrays: Process Table



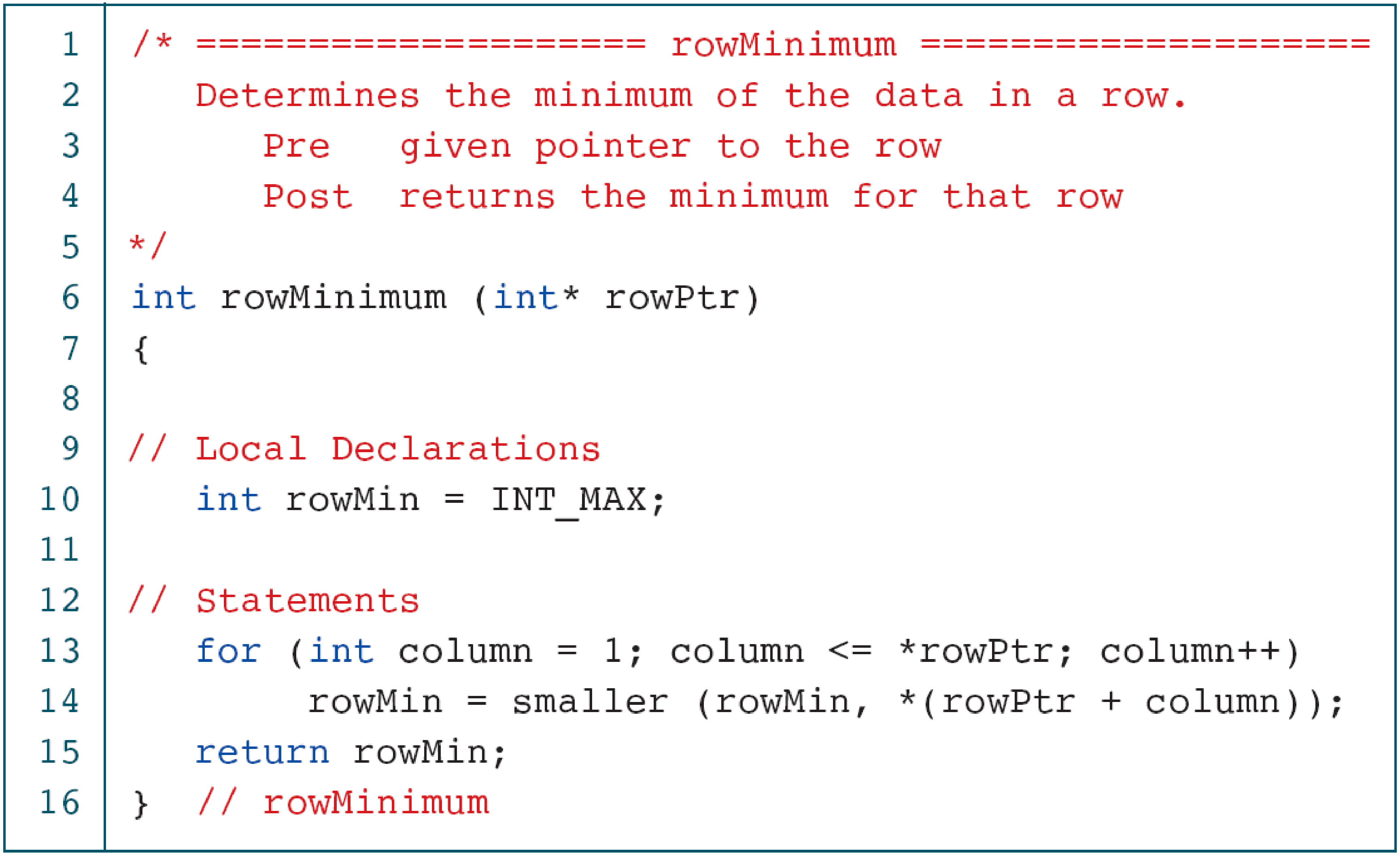
```c
/* ================== processTable ====================
   Process the table to create the statistics.
      Pre   table
      Post  row statistics (min, max, and average)
*/
void processTable (int** table)
{
// Local Declarations
int   row = 0;
int   rowMin;
int   rowMax;
float rowAve;

// Statements
   while (table[row] != NULL)
     {
```

PROGRAM 10-8 Dynamic Arrays: Process Table



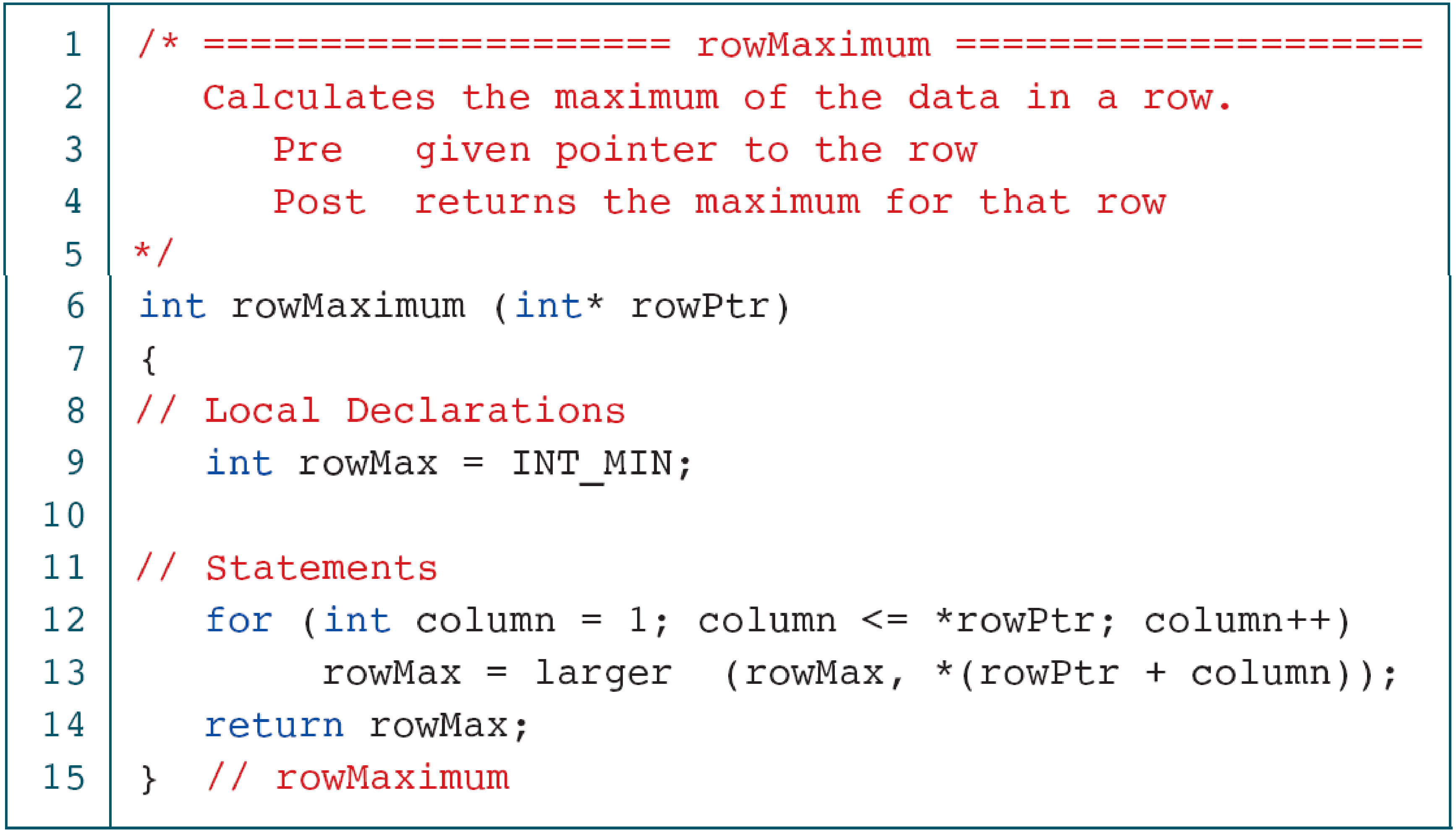
```
      rowMin = rowMinimum (table[row]);
      rowMax = rowMaximum (table[row]);
      rowAve = rowAverage (table[row]);
      printf("\nThe statistics for row %d ", row + 1);
      printf("\nThe minimum: %5d",    rowMin);
      printf("\nThe maximum: %5d",    rowMax);
      printf("\nThe average: %8.2f ", rowAve);
      row++;
     } // while
   return;
}  // processTable
```

PROGRAM 10-9 Dynamic Arrays: Find Row Minimum



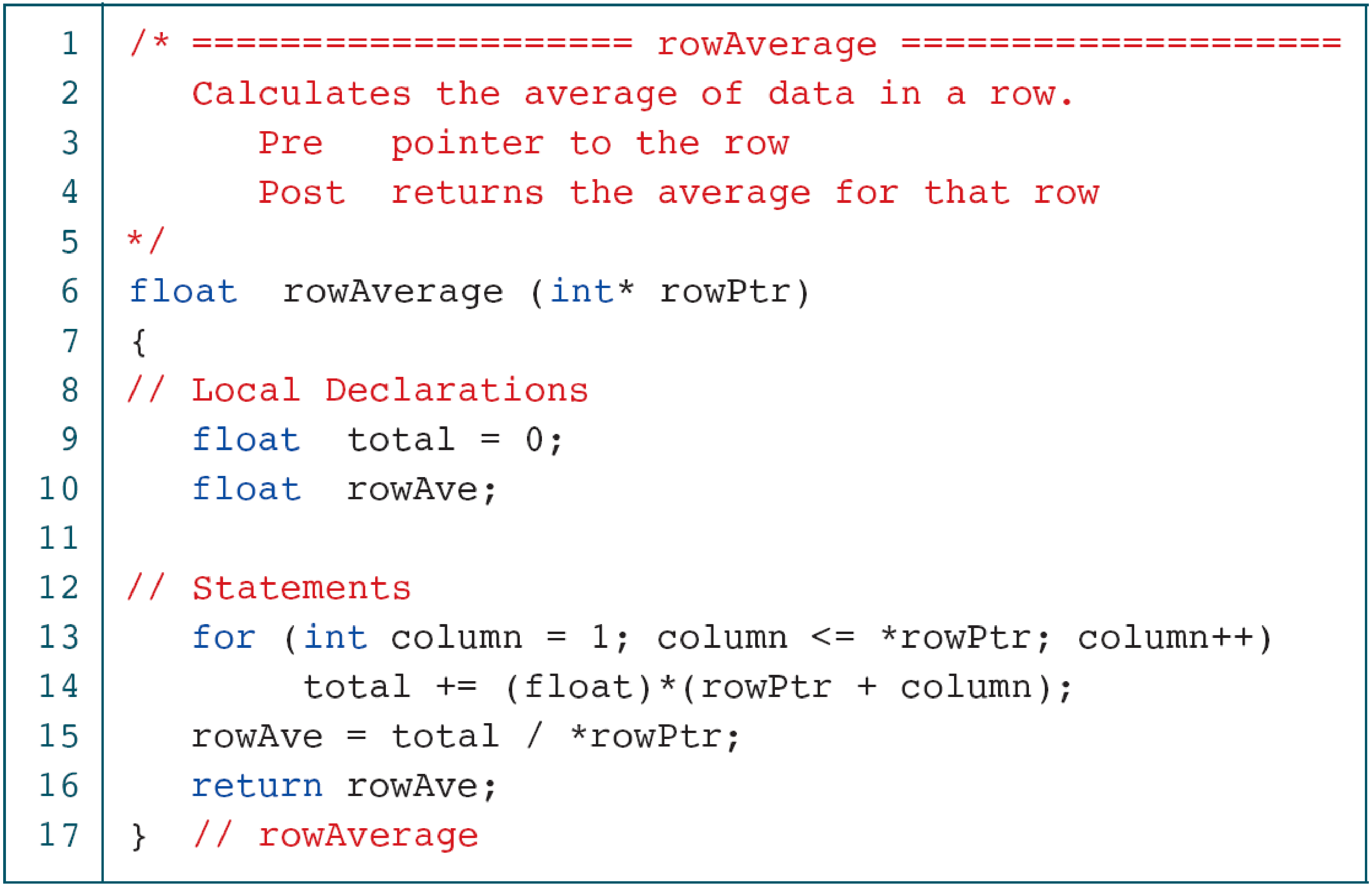
```c
/* ==================== rowMinimum ====================
   Determines the minimum of the data in a row.
      Pre   given pointer to the row
      Post  returns the minimum for that row
*/
int rowMinimum (int* rowPtr)
{

// Local Declarations
   int rowMin = INT_MAX;

// Statements
   for (int column = 1; column <= *rowPtr; column++)
        rowMin = smaller (rowMin, *(rowPtr + column));
   return rowMin;
}  // rowMinimum
```

PROGRAM 10-9 Dynamic Arrays: Find Row Minimum



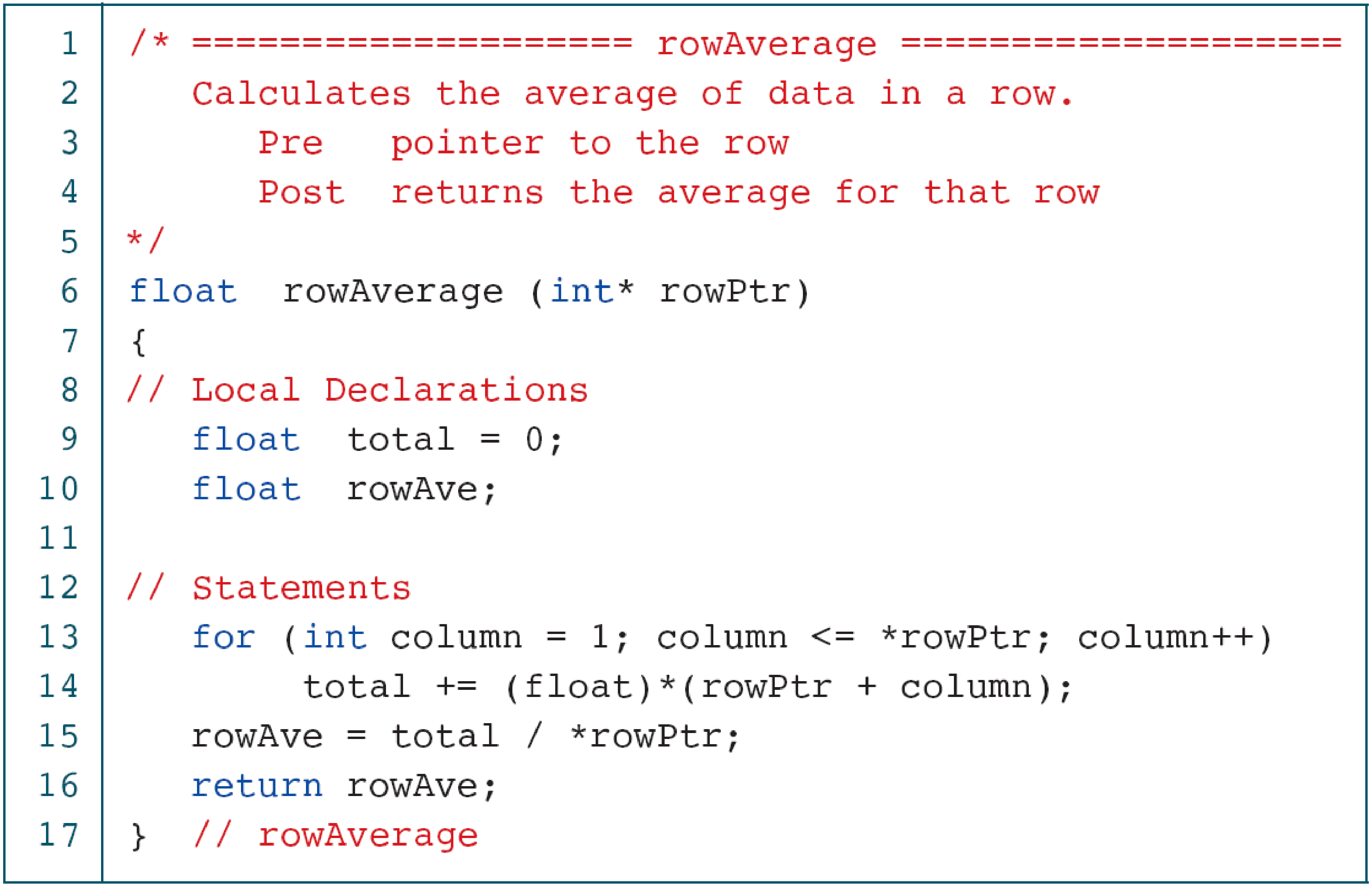
```c
/* ==================== rowMaximum ====================
   Calculates the maximum of the data in a row.
      Pre   given pointer to the row
      Post  returns the maximum for that row
*/
int rowMaximum (int* rowPtr)
{
// Local Declarations
   int rowMax = INT_MIN;

// Statements
   for (int column = 1; column <= *rowPtr; column++)
        rowMax = larger  (rowMax, *(rowPtr + column));
   return rowMax;
}  // rowMaximum
```

PROGRAM 10-10 Dynamic Arrays: Find Row Maximum



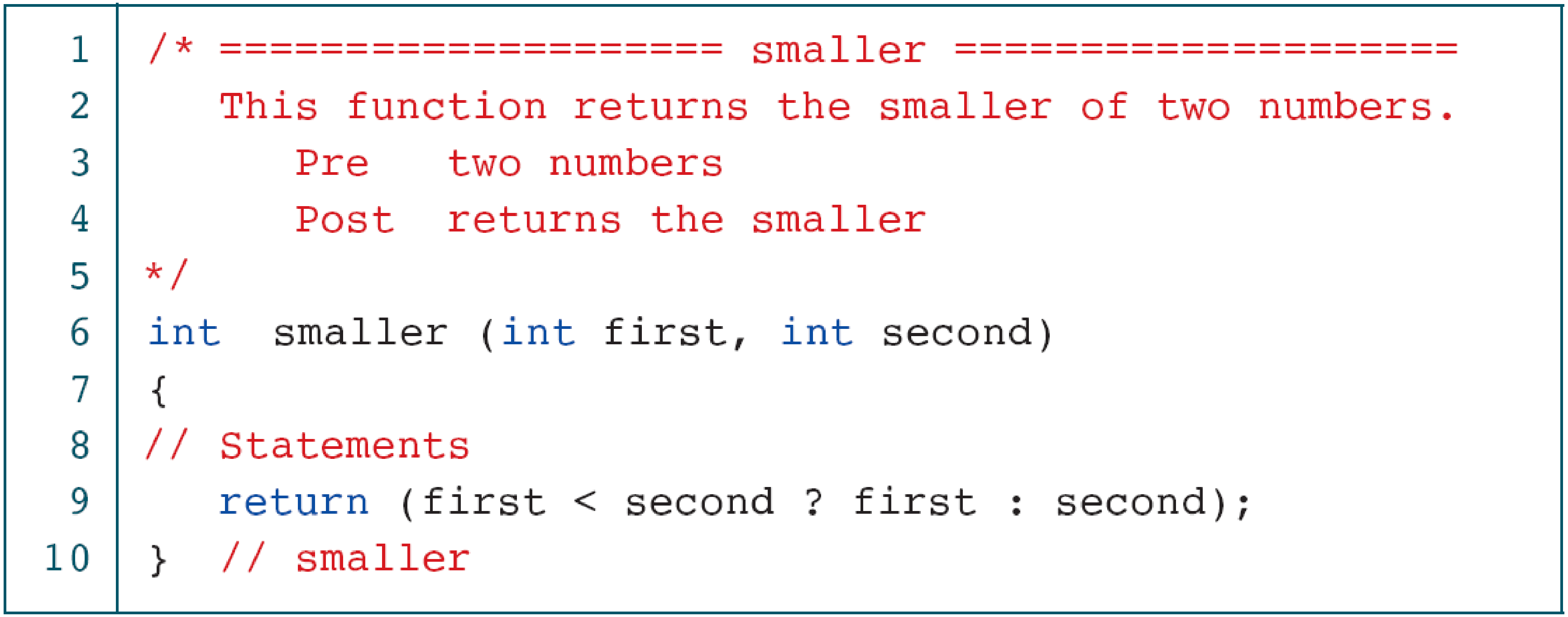
```
/* ==================== rowAverage ====================
   Calculates the average of data in a row.
      Pre   pointer to the row
      Post  returns the average for that row
*/
float  rowAverage (int* rowPtr)
{
// Local Declarations
   float  total = 0;
   float  rowAve;

// Statements
   for (int column = 1; column <= *rowPtr; column++)
        total += (float)*(rowPtr + column);
   rowAve = total / *rowPtr;
   return rowAve;
}  // rowAverage
```

PROGRAM 10-11 Dynamic Arrays: Find Row Average



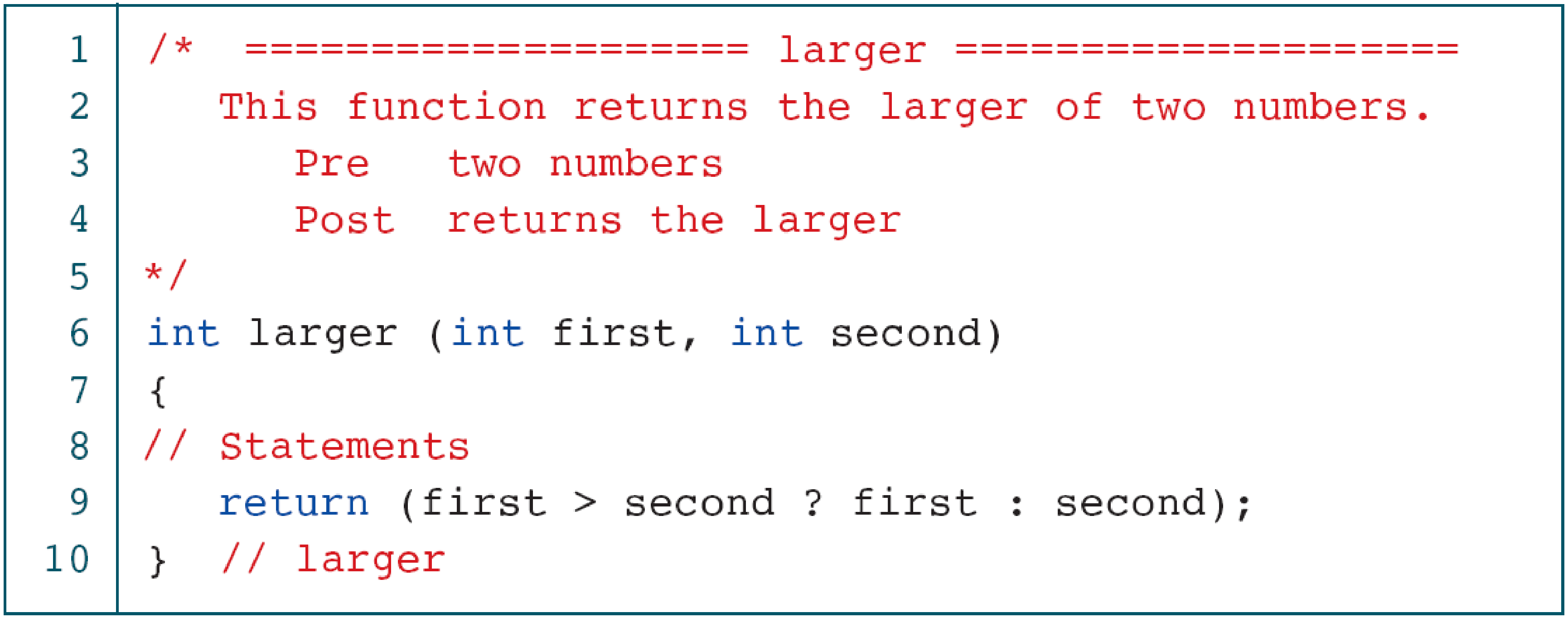
```
/* ==================== rowAverage ====================
   Calculates the average of data in a row.
      Pre   pointer to the row
      Post  returns the average for that row
*/
float  rowAverage (int* rowPtr)
{
// Local Declarations
   float  total = 0;
   float  rowAve;

// Statements
   for (int column = 1; column <= *rowPtr; column++)
        total += (float)*(rowPtr + column);
   rowAve = total / *rowPtr;
   return rowAve;
}  // rowAverage
```

PROGRAM 10-12 Dynamic Arrays: Find Smaller



```c
/* ==================== smaller ====================
   This function returns the smaller of two numbers.
      Pre   two numbers
      Post  returns the smaller
*/
int  smaller (int first, int second)
{
// Statements
   return (first < second ? first : second);
}  // smaller
```

PROGRAM 10-13 Dynamic Arrays: Find Larger



```c
/*  ==================== larger ====================
   This function returns the larger of two numbers.
      Pre   two numbers
      Post  returns the larger
*/
int larger (int first, int second)
{
// Statements
   return (first > second ? first : second);
}  // 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.

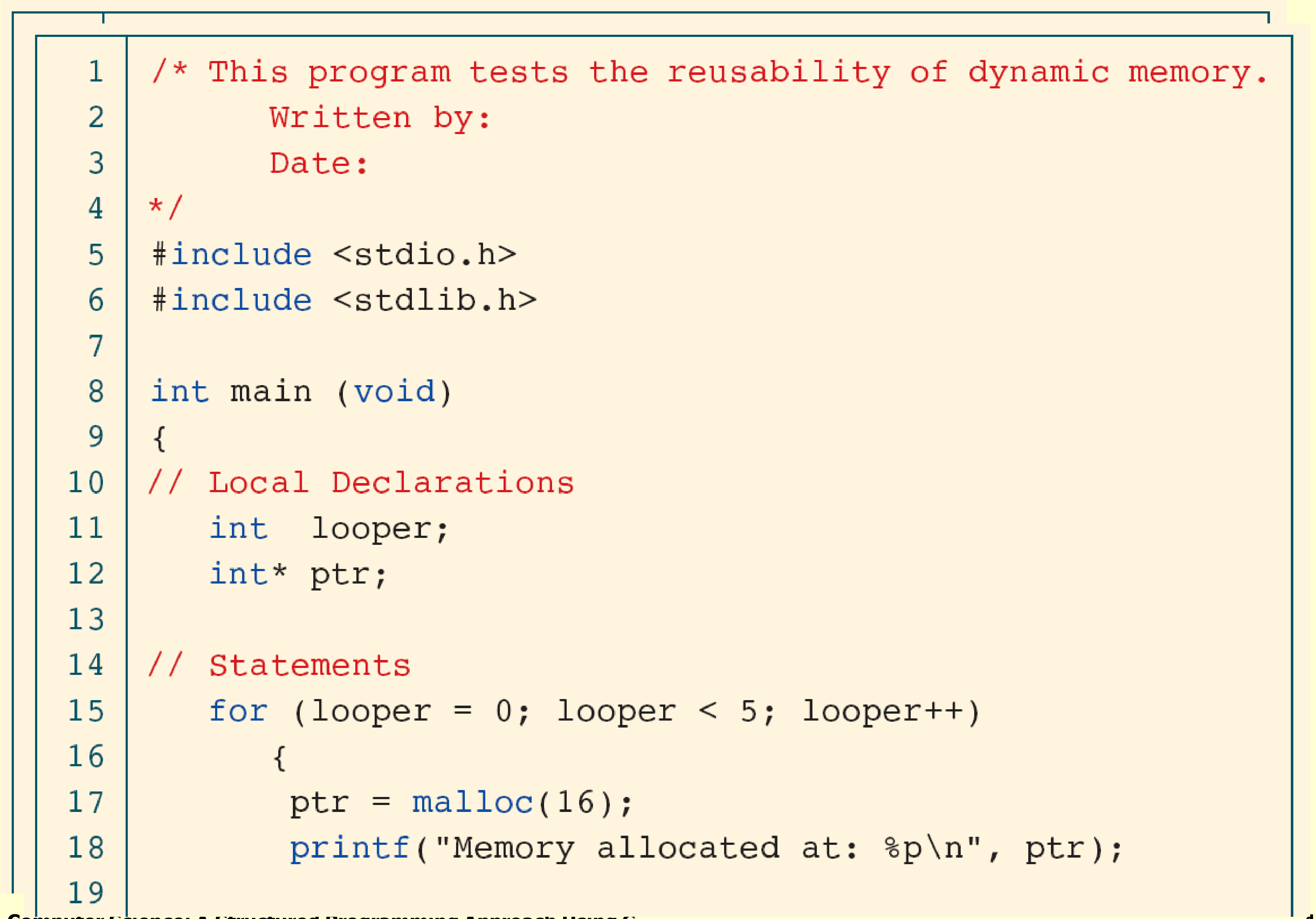
Topics discussed in this section:

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

Note

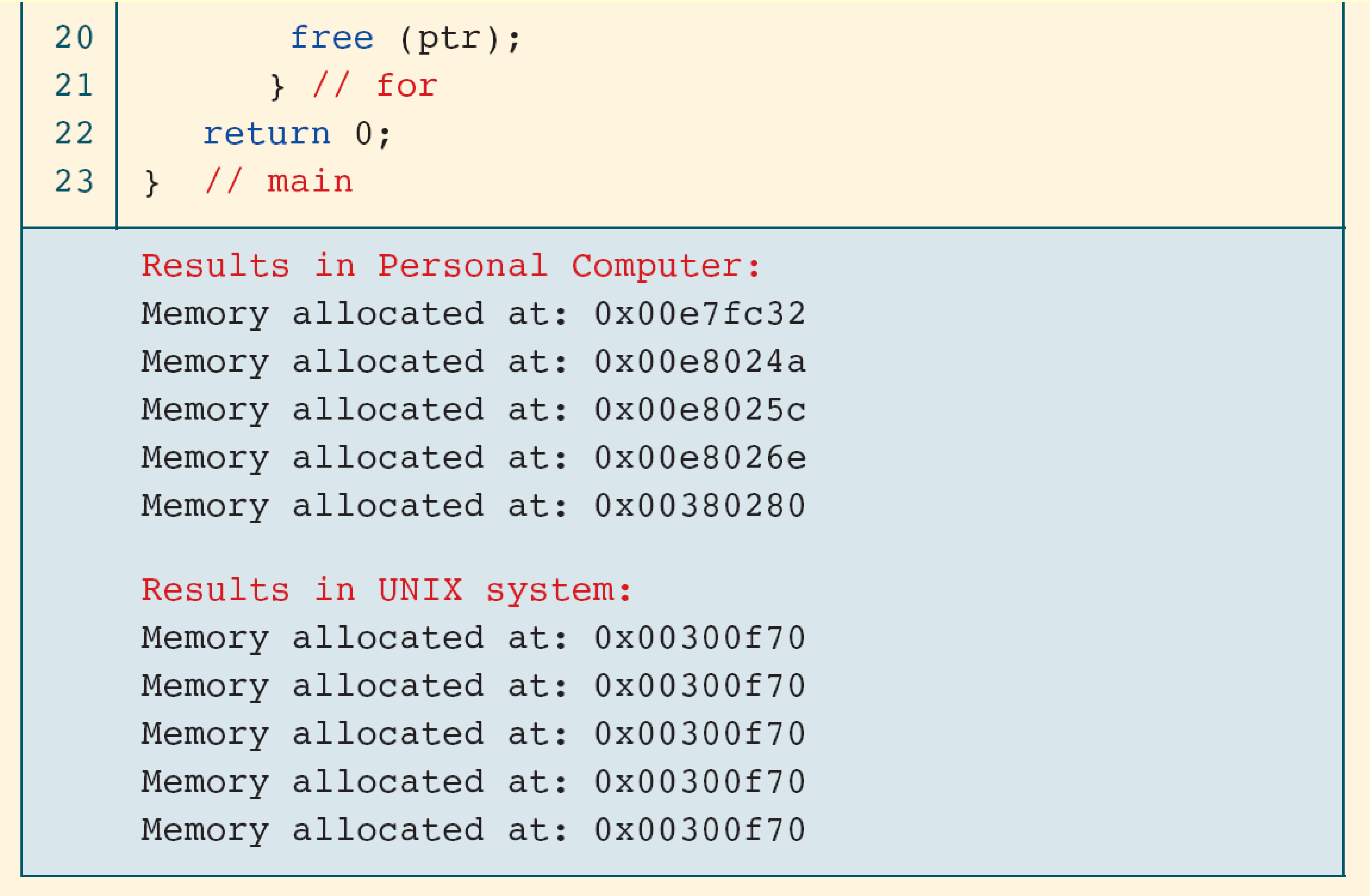
Whenever possible, use value parameters.

PROGRAM 10-14 Testing Memory Reuse



```c
/* This program tests the reusability of dynamic memory.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
// Local Declarations
   int  looper;
   int* ptr;

// Statements
   for (looper = 0; looper < 5; looper++)
      {
       ptr = malloc(16);
       printf("Memory allocated at: %p\n", ptr);

```

PROGRAM 10-14 Testing Memory Reuse



```c
       free (ptr);
      } // for
   return 0;
}  // 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
```