Solutions to Chapter 10

Review Questions

1. a. True

3. a. True

5. c. The name of an array can be used with the indirection operator to reference data.

7. b. `int ary[] [SIZE2]`

9. a. Allocated memory can be referred to only through pointers; it does not have its own identifier.

11. e. To ensure that it is released, allocated memory should be freed before the program ends.

Exercises

13.

a. `*(tax + 6)`

b. `*(score + 7)`

c. `*(num + 4)`

d. `*(prices + 9)`

15. If we interpret the sixth element as `ary[5]`, and if `p` is pointing to `ary[3]`, we can access `ary[5]` using `*(p + 2)`, or we can use an index, like so: `p[2]`.

17.

```
6 6
3 4
6 2
4 6
```

19.

```
// Function Declaration
void fun (int** ary);

// Function call
fun (table);
```

21. See Figure 10-1.

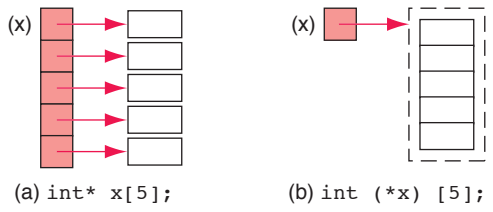

Figure 10-1 Solution for Exercise 21.

a. `x` is an array of pointers where each pointer can point to an integer.

b. `x` is a pointer to an array of integers.

23.

```
4 5 2
7 6 9
```

See Figure 10-2 for Explanation:

First Call: `p` is pointing to the whole first row, so `(*p)` is the first row itself. Then using the first row, `(*p)[0]` refers to the first element (4), `(*p)[1]` refers to the second element (5), and `(*p)[2]` refers to the third element (2).

Second call: `p` is pointing to the whole second row (`x+1`), so `(*p)` is the second row itself. Then using the second row, `(*p)[0]` refers to the first element (7), `(*p)[1]` refers to the second element (6), and `(*p)[2]` refers to the third element (9).

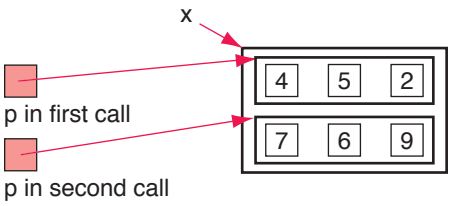

Figure 10-2 Solution for Exercise 23

25.

a. e

b. m

27.

a. 4

b. 4

c. address of `i`

d. 4

e. address of `i`

29.

a. `num[2]`

b. `num[i + j]`

c. `num[num[1]]`

d. `num[j]`

e. `num[1] + num[j]`

31.

a. `&num[0]`

b. `num[0]`

c. `num[0] + 1`

d. `num[1]`

e. `num[j]`

33.

a. not valid: `mushem` needs two addresses.

b. Valid (`&i` and `&j`) are addresses.

c. valid

d. not valid: `i` and `j` are values not addresses

e. valid, but discards return value from `mushem`

35.

```
1. 9 4 17
2. 31 17 10 18 7 19 10
```

Problems

37. See Program 10-1.

Program 10-1 Solution to Problem 37

```
/* ================== checkData ====================
   This function checks if every element of array 1 is
   equal to its corresponding element in array 2.
      Pre   pointer to array 1 and
            pointer to array 2
            integer containing number of array elements
      Post   returns true if array values equal
                     false if not
*/
bool checkData (int* pAry1, int* pAry2, int size)
{
// Local Declarations
   int* p1;
   int* p2;
   int* pLast;
   bool chk;

// Statements
   chk = true;
   pLast = pAry1 + size - 1;
   for (p1 = pAry1, p2 = pAry2;
        p1 <= pLast;
        p1++, p2++)
          {
           if (*p1 != *p2)
                chk = false;
          } // for

   return (chk);
}  // checkData
```

39. See Program 10-2.

Program 10-2 Solution to Problem 39

```
/* ==================== Pascal ====================
   This function creates a 2-dimensional ragged array
   representing the Pascal Triangle.
      Pre   size of triangle as an integer
      Post  returns pointer to ragged array
*/
int** Pascal (int size)
{
// Local Declarations
   int** p;

// Statements
   // allocation for the array for ragged array
   p = (int**) calloc (size + 2, sizeof (int*));
   *(p + size + 1) = NULL;
```

Program 10-2 Solution to Problem 39 (continued)

```
   for (int row = 0; row < size + 1; row++)
        *(p + row) = (int*)calloc (row + 1, sizeof (int));

   // Filling data according to the size of triangle
   for (int row = 0; row < size + 1; row++)
       {
        for (int col = 0; col < row + 1; col++)
            {
             if (row == col || col == 0)
                 *(*(p + row) + col) = 1;
             else
                 *(*(p + row) + col) =
                   *(*(p + row - 1) + (col - 1)) +
                   *(*(p + row - 1) + col);
            } // for col
       } // for row
   return p;
}  // Pascal
```

41. See Program 10-3.

Program 10-3 Solution to Problem 41

```
/* ================ convertArray ==================
   This function copies a 1-dimensional array of n
   elements into a 2-dimensional array of j rows and
   k columns.
      Pre  One-dimensional array
           Number of elements in one-dimensional array
           Number of rows in two-dimensional array
           Number of columns in two-dimensional array
      Post if elements != toRow * toCol, returns null
           else, returns pointer to 2-dimensional array
*/
int** convertArray (int* fromAry, int elements,
                    int  toRow,   int toCol)
{
// Local Declarations
   int** twoDimAry;
   int** lastRow;
   int*  lastCol;
   int   fromIndex = 0;

// Statements
   if (elements != toRow * toCol)
       twoDimAry = NULL;
   else
      {
       twoDimAry = (int**)calloc(toRow, sizeof (int*));
       lastRow = twoDimAry + toRow - 1;
       for (int** row = twoDimAry; row <= lastRow; row++)
           {
            *row = (int*)calloc(toCol, sizeof (int));
            lastCol = *row + toCol - 1;
            for (int* col = *row;
                 col <= lastCol;
                 fromIndex++, col++)
                *col = *(fromAry + fromIndex);
           } // for row
      } // if else

   return twoDimAry;
}  // convertArray
```

43. See Program 10-4.

Program 10-4 Solution to Problem 43

```
// Local Declarations
   char  a[40];
   char* plast;
   char* walker;

// Statements
   plast  = a + 39;
   for (walker = a; walker <= plast; walker++)
      {
       printf ("Please enter character:  ");
       scanf  (" %c", walker);
      } // for
```

45. See Program 10-5.

Program 10-5 Solution to Problem 45

```
// Local Declarations
   char  temp;
   char  a[6] = {'z', 'x', 'm', 's', 'e', 'h'};
   char* walker;
   char* plast;

// Statements
   plast = a + 5;
   temp = *a;
   for (walker = a; walker < plast; walker++)
       *walker = *(walker + 1);
   *walker = temp;
```