Conditional operator

A very compact if-else.

(condition) ? expression2 : expression3

means

if (condition)

expression2

else

expression3

Array Pointers
Chapter 7

Problem Solving & Program Design in C

Eighth Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

- To learn how to declare and use arrays for storing collections of values of the same type
- To understand how to use a subscript to reference the individual values in an array
- To learn how to process the elements of an array in sequential order using loops

Chapter Objectives

- To understand how to pass individual array elements and entire arrays through function arguments
- To learn a method for searching an array
- To learn a method for sorting an array
- To learn how to use multidimensional arrays for storing tables of data
- To understand the concept of parallel arrays
- To learn how to declare and use your own data types

Basic Terminology

- data structure
  - a composite of related data items stored under the same name
- array
  - a collection of data items of the same type

Declaring and Referencing Arrays

- array element
  - a data item that is part of an array
- subscripted variable
  - a variable followed by a subscript in brackets, designating an array element
- array subscript
  - a value or expression enclosed in brackets after the array name, specifying which array element to access

```
double x[8];
```

Array `x`

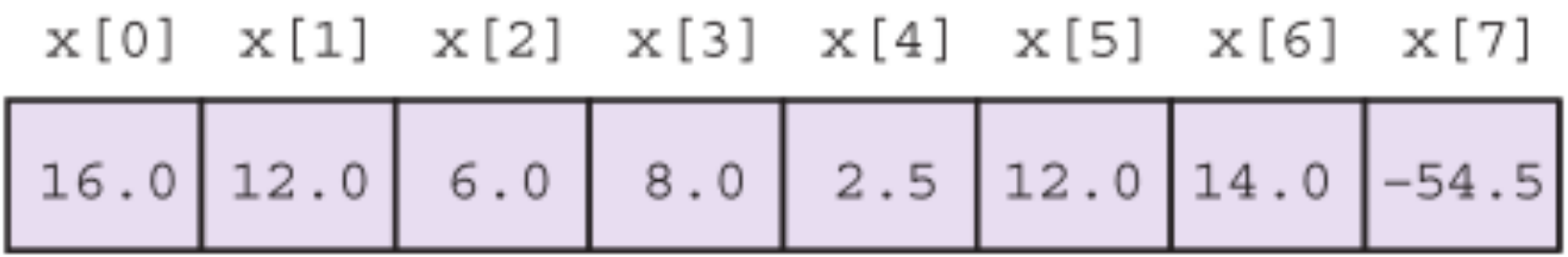


TABLE 7.1 Statements That Manipulate Array x

Statement	Explanation
`printf("%.1f", x[0]);`	Displays the value of `x[0]`, which is `16.0`.
`x[3] = 25.0;`	Stores the value `25.0` in `x[3]`.
`sum = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]`, which is `28.0` in the variable `sum`.
`sum += x[2];`	Adds `x[2]` to `sum`. The new `sum` is `34.0`.
`x[3] += 1.0;`	Adds `1.0` to `x[3]`. The new `x[3]` is `26.0`.
`x[2] = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]` in `x[2]`. The new `x[2]` is `28.0`.

Array `x`

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	28.0	26.0	2.5	12.0	14.0	-54.5

Array Initialization

```
int prime_lt_100[] = {2, 3, 5, 7, 11, 13, 17, 19,
                      23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
                      67, 71, 73, 79, 83, 89, 97}

char vowels[] = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’, ‘y’}
```

Array Subscripts

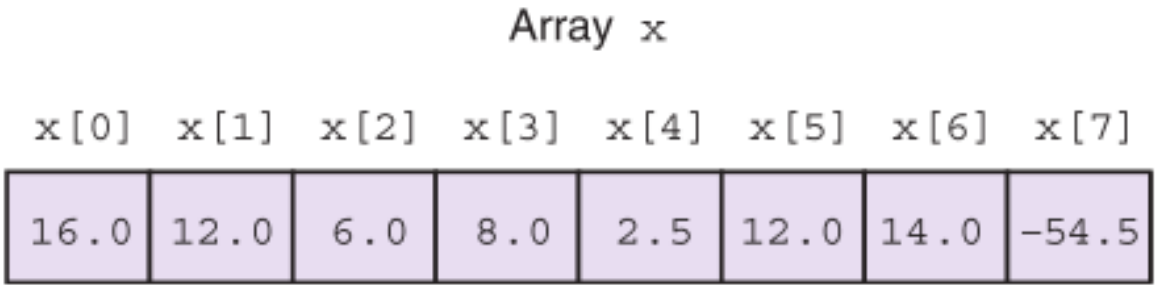
- Syntax:

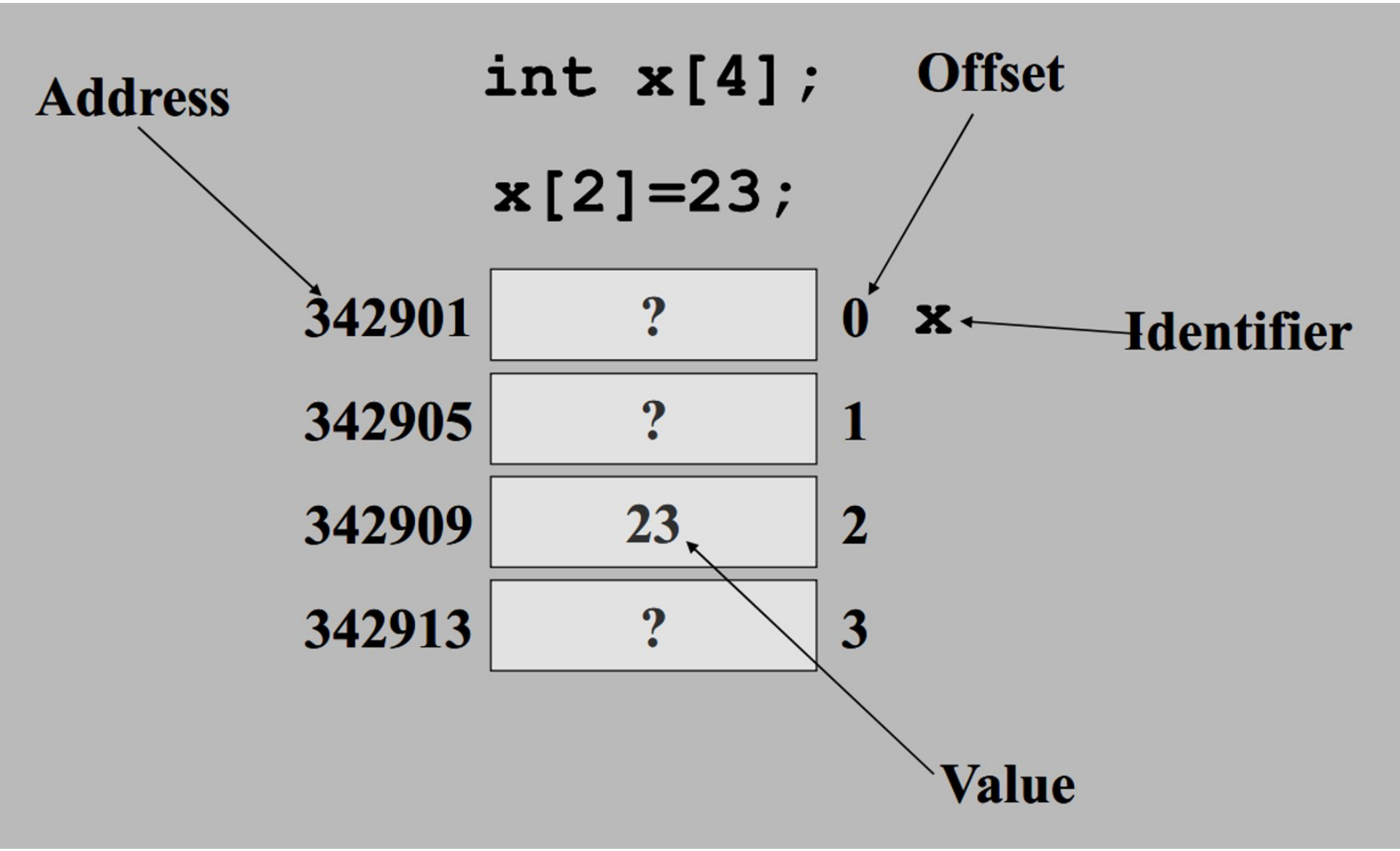
  aname [subscript]

- Examples:

  x[3]

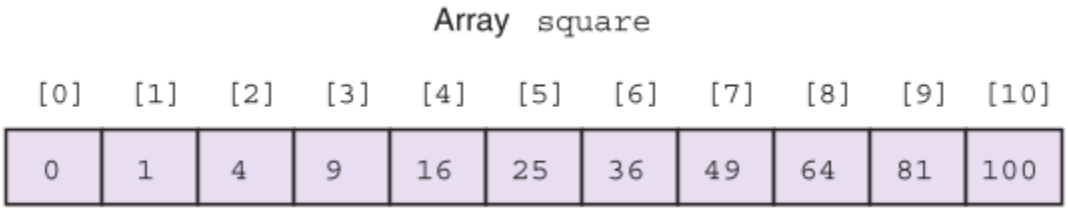
  x[i + 1]



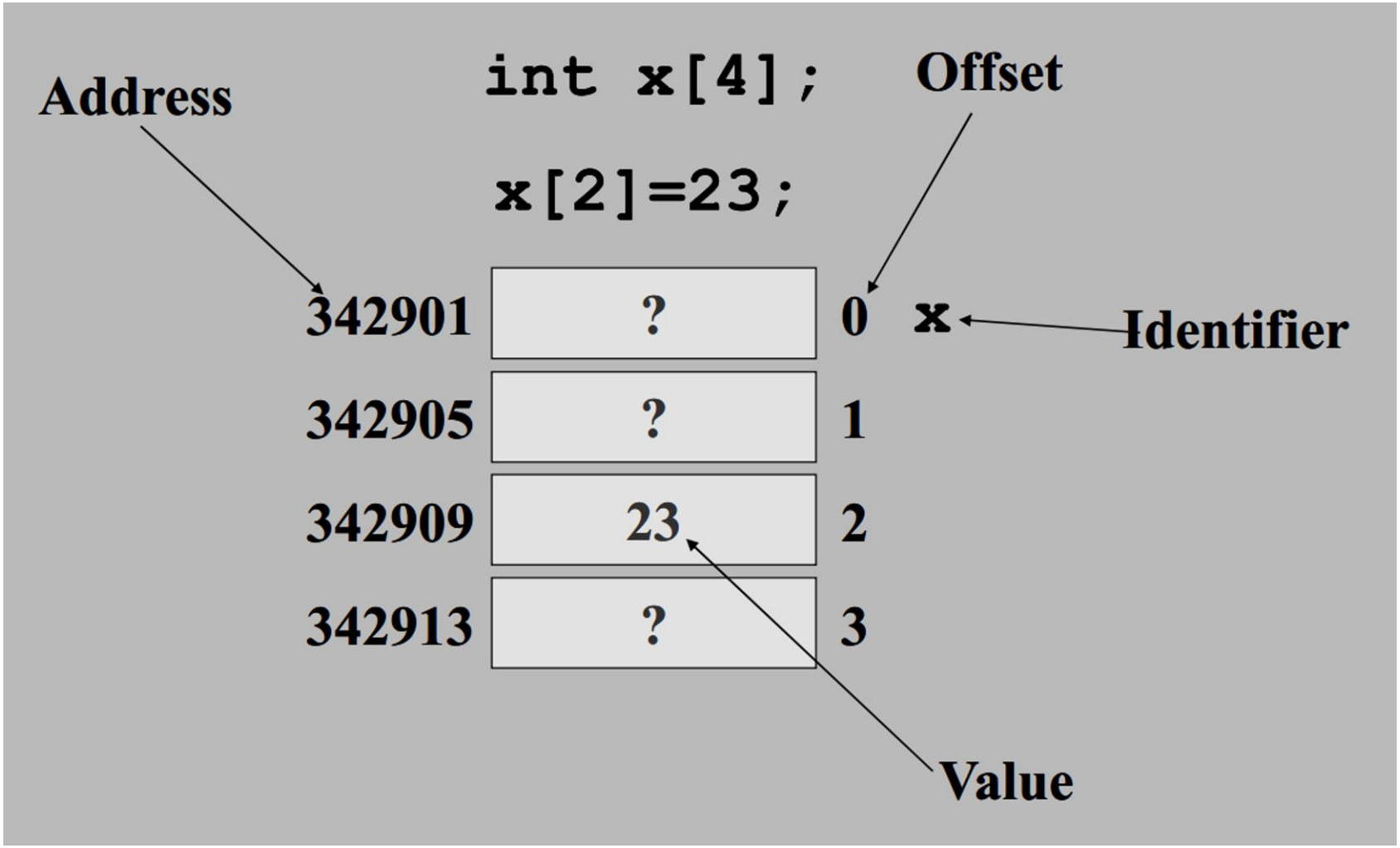


Using for Loops for Sequential Access

```
for (i = 0; i < SIZE; ++i)
    square[i] = i * i;
```



What's at x[5]?



Partially Filled Arrays

- A program may need to process many lists of similar data but the lists may not all be the same length.
- In order to reuse an array for processing more than one data set, you can declare an array large enough to hold the largest data set anticipated.
- Then your program should keep track of how many array elements are actually in use.

Multidimensional Arrays

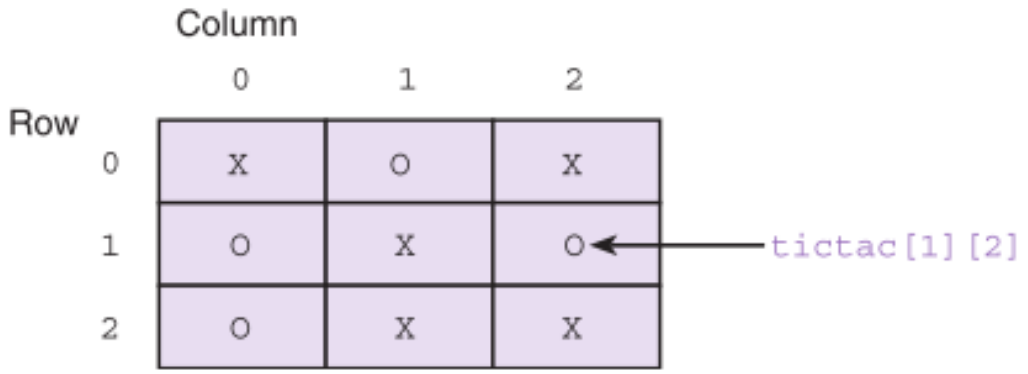
- multidimensional array

type arr_name[dim1val][dim2val]

tictac[3][3]

FIGURE 7.20

A Tic-tac-toe Board Stored as Array tictac



Using Array Elements as Function Arguments

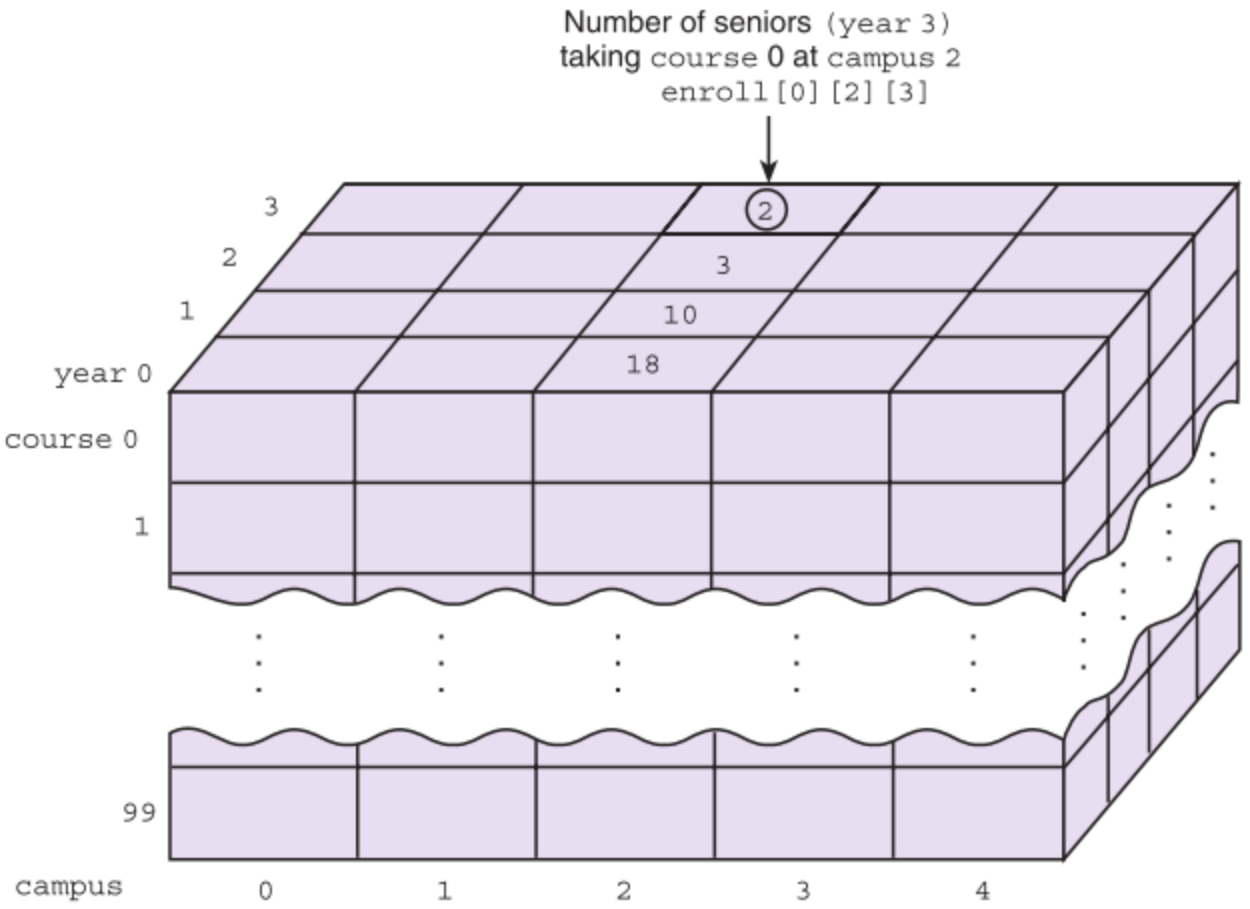
```
scanf("%lf",  &x[i]);
```

FIGURE 7.21 Function to Check Whether Tic-tac-toe Board Is Filled

```
1.  /* Checks whether a tic-tac-toe board is completely filled.                */
2.  int
3.  filled(char ttt_brd[3][3])  /* input - tic-tac-toe board                   */
4.  {
5.        int r, c, /* row and column subscripts   */
6.            ans;  /* whether or not board filled */
7.
8.        /* Assumes board is filled until blank is found                      */
9.        ans = 1;
10.
11.       /* Resets ans to zero if a blank is found                            */
12.       for (r = 0; r < 3; ++r)
13.          for  (c = 0; c < 3; ++c)
14.             if (ttt_brd[r][c] == ' ')
15.                  ans = 0;
16.
17.       return (ans);
18. }
```

FIGURE 7.22

Three-Dimensional Array enroll



Array Arguments

- We can write functions that have arrays as arguments.

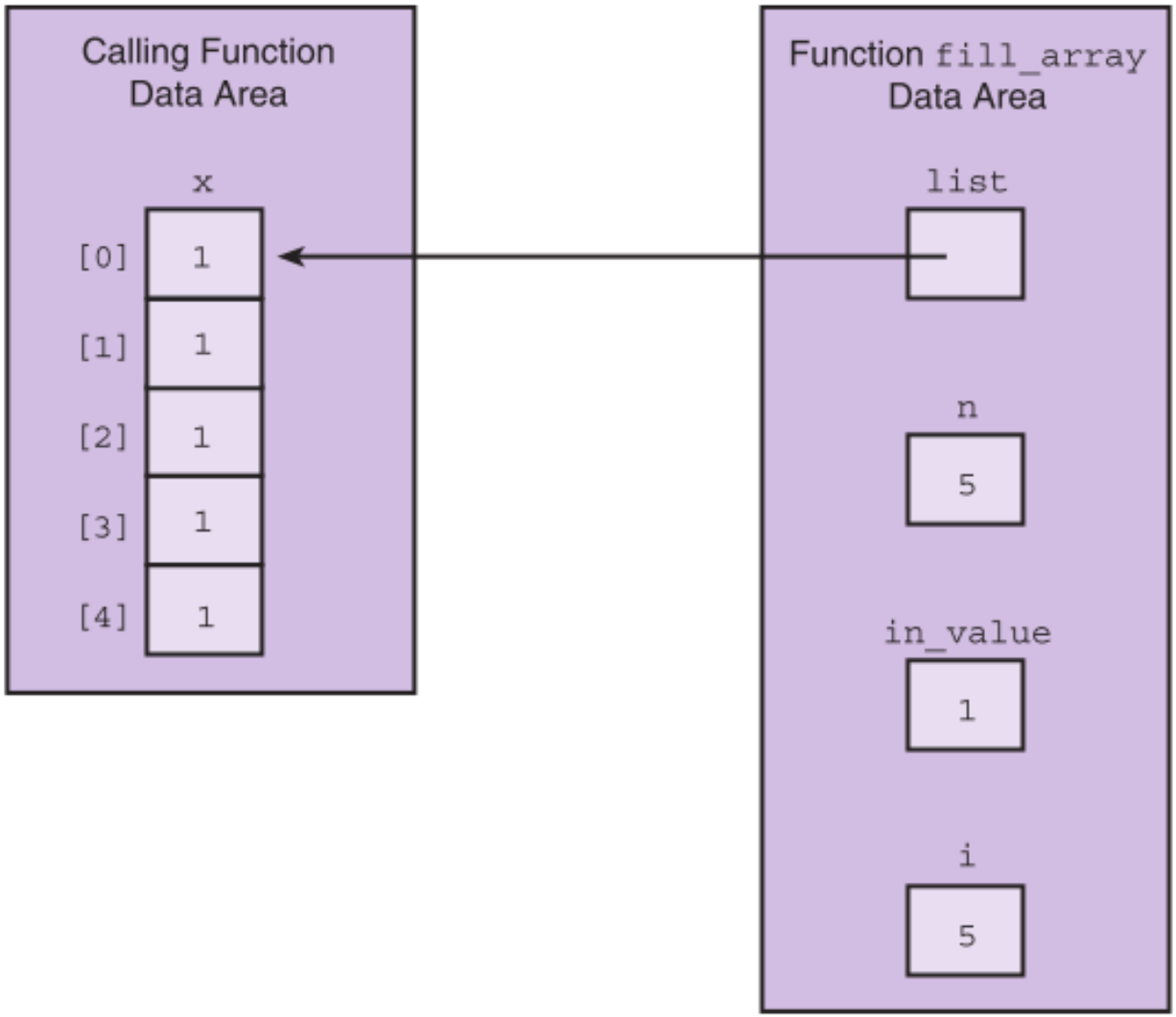
- Such functions can manipulate some, or all, of the elements corresponding to an actual array argument.

FIGURE 7.4 Function fill_array

```
1.  /*
2.   * Sets all elements of its array parameter to in_value.
3.   * Pre: n and in_value are defined.
4.   * Post: list[i] = in_value, for 0 <= i < n.
5.   */
6.  void
7.  fill_array (int list[],    /* output - list of n integers                */
8.              int n,         /* input - number of list elements            */
9.              int in_value)  /* input - initial value                      */
10. {
11.
12.       int i;             /* array subscript and loop control           */
13.
14.       for  (i = 0; i < n; ++i)
15.           list[i] = in_value;
16. }
```

FIGURE 7.5

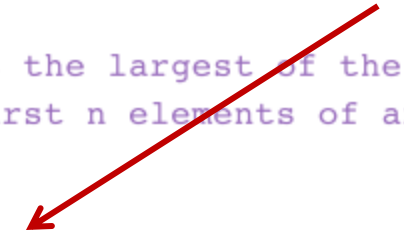
Data Areas Before Return from `fill_array (x, 5, 1);`



Arrays as Input Arguments

- ANSI C provides a qualifier that we can include in the declaration of the array formal parameter in order to notify the C compiler that the array is only an input to the function and the function does not intend to modify the array.
- The qualifier const allows the compiler to mark as an error any attempt to change an array element within the function.

FIGURE 7.6 Function to Find the Largest Element in an Array



```
1.  /*
2.   * Returns the largest of the first n values in array list
3.   * Pre: First n elements of array list are defined and n > 0
4.   */
5.  int
6.  get_max(const int list[], /* input - list of n integers                    */
7.          int       n)     /* input - number of list elements to examine    */
8.  {
9.        int i,
10.           cur_large;     /* largest value so far                          */
11.
12.       /* Initial array element is largest so far.                         */
13.       cur_large = list[0];
14.
15.       /* Compare each remaining list element to the largest so far;
16.          save the larger                                                  */
17.       for  (i = 1; i < n; ++i)
18.           if (list[i] > cur_large)
19.                 cur_large = list[i];
20.
21.       return (cur_large);
22. }
```

Returning an Array Result

- In C, it is not legal for a function's return type to be an array.
- You need to use an output parameter to send your array back to the calling module.

FIGURE 7.7

Diagram of a Function That Computes an Array Result

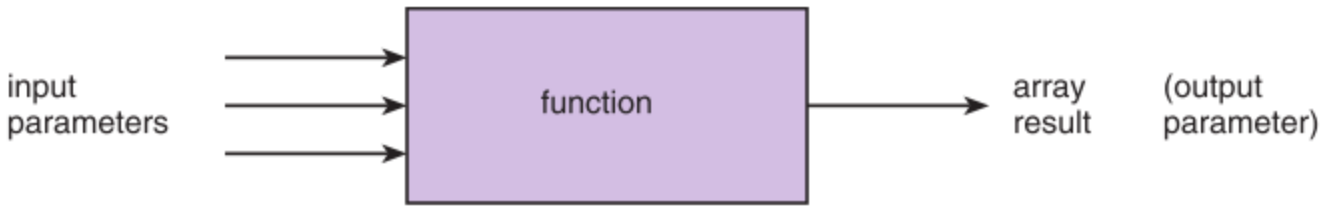
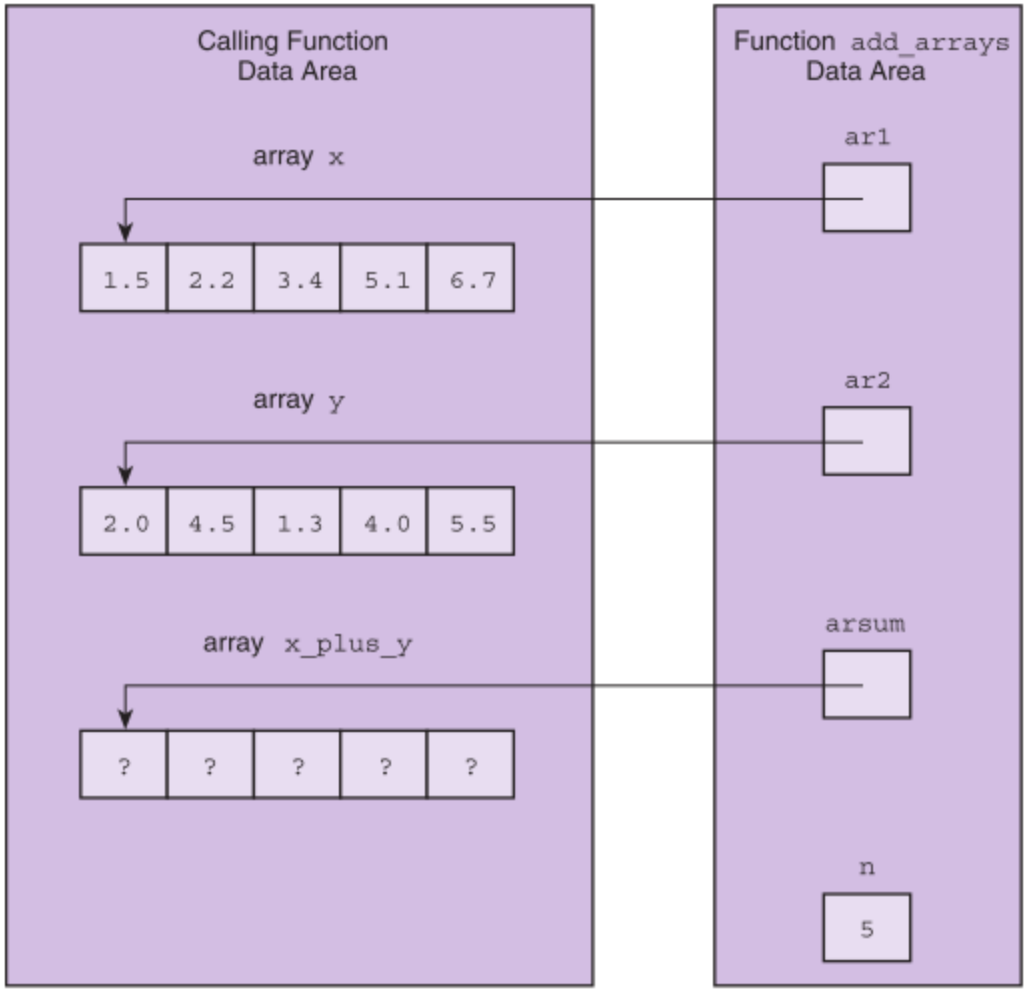


FIGURE 7.8 Function to Add Two Arrays

```
1.  /*
2.   * Adds corresponding elements of arrays ar1 and ar2, storing the result in
3.   * arsum. Processes first n elements only.
4.   * Pre: First n elements of ar1 and ar2 are defined. arsum's corresponding
5.   *      actual argument has a declared size >= n (n >= 0)
6.   */
7.  void
8.  add_arrays(const double ar1[],    /* input -                              */
9.             const double ar2[],    /* arrays being added                   */
10.            double       arsum[],  /* output - sum of corresponding
11.                                       elements of ar1 and ar2             */
12.            int          n)        /* input - number of element
13.                                       pairs summed                        */
14. {
15.       int i;
16.
17.       /* Adds corresponding elements of ar1 and ar2                       */
18.       for  (i = 0; i < n; ++i)
19.           arsum[i] = ar1[i] + ar2[i];
20. }
```

FIGURE 7.9

Function Data Areas for `add_arrays(x, y, x_plus_y, 5);`



Stacks

- A stack is a data structure in which only the top element can be accessed.
- pop
  - remove the top element of a stack
- push
  - insert a new element at the top of the stack

FIGURE 7.13 Functions push and pop

```
1.  void
2.  push(char stack[],     /* input/output - the stack */
3.       char item,        /* input - data being pushed onto the stack */
4.       int  *top,        /* input/output - pointer to top of stack */
5.       int  max_size)    /* input - maximum size of stack */
6.  {
7.       if (*top < max_size-1) {
8.           ++(*top);
9.           stack[*top] = item;
10.      }
11. }
12.
13. char
14. pop(char stack[],     /* input/output - the stack */
15.     int *top)         /* input/output - pointer to top of stack */
16. {
17.      char item;       /* value popped off the stack */
18.
19.      if (*top >= 0) {
20.           item = stack[*top];
21.           --(*top);
22.      } else {
23.           item = STACK_EMPTY;
24.      }
25.
26.      return (item);
27. }
```

Array Search

1. Assume the target has not been found.
2. Start with the initial array element.
3. repeat while the target is not found and there are more array elements
   4. if the current element matches the target
      5. Set a flag to indicate that the target has been found

      else
      6. Advance to the next array element.
7. if the target was found
   8. Return the target index as the search result

   else
   9. Return -1 as the search result.

FIGURE 7.14 Function That Searches for a Target Value in an Array

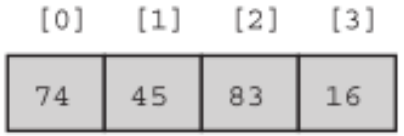
```
#define NOT_FOUND -1     /* Value returned by search function if target not
                            found                                              */

/*
 * Searches for target item in first n elements of array arr
 * Returns index of target or NOT_FOUND
 * Pre:  target and first n elements of array arr are defined and n>=0
 */
int
search(const int arr[],   /* input - array to search                          */
       int       target,  /* input - value searched for                       */
       int       n)       /* input - number of elements to search             */
{
      int i,
          found = 0,      /* whether or not target has been found             */
          where;          /* index where target found or NOT_FOUND            */

      /* Compares each element to target                                      */
      i = 0;
      while (!found && i < n) {
          if (arr[i] == target)
                found = 1;
          else
                ++i;
      }

      /* Returns index of element matching target or NOT_FOUND                */
      if (found)
            where = i;
      else
            where = NOT_FOUND;

      return (where);
}
```

Selection Sort

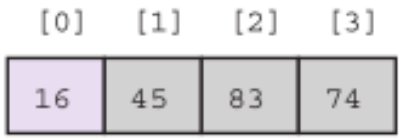
1. for each value of fill from 0 to n-2
    2. Find index_of_min, the index of the smallest element in the unsorted subarray list[fill] through list[n-1]
    3. if fill is not the position of the smallest element (index_of_min)
        4. Exchange the smallest element with the one at position fill.

FIGURE 7.15

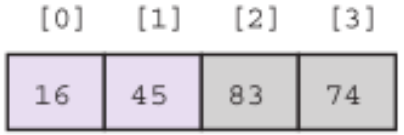
Trace of Selection Sort



`fill` is `0`. Find the smallest element in subarray `list[1]` through `list[3]` and swap it with `list[0]`.



`fill` is `1`. Find the smallest element in subarray `list[1]` through `list[3]`—no exchange needed.



`fill` is `2`. Find the smallest element in subarray `list[2]` through `list[3]` and swap it with `list[2]`.

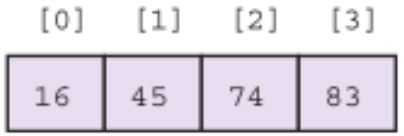


FIGURE 7.16 Function select_sort

```
1.  /*
2.   * Finds the position of the smallest element in the subarray
3.   * list[first] through list[last].
4.   * Pre: first < last and elements 0 through last of array list are defined.
5.   * Post: Returns the subscript k of the smallest element in the subarray;
6.   *       i.e., list[k] <= list[i] for all i in the subarray
7.   */
8.  int get_min_range(int list[], int first, int last);
9.
10.
11. /*
12.  * Sorts the data in array list
13.  * Pre: first n elements of list are defined and n >= 0
14.  */
15. void
16. select_sort(int list[],  /* input/output - array being sorted                */
17.             int n)       /* input - number of elements to sort               */
18. {
19.       int fill,          /* index of first element in unsorted subarray      */
20.           temp,          /* temporary storage                                */
21.           index_of_min;  /* subscript of next smallest element               */
22.
23.       for (fill = 0; fill < n-1; ++fill) {
24.            /* Find position of smallest element in unsorted subarray */
25.            index_of_min = get_min_range(list, fill, n-1);
26.
27.            /* Exchange elements at fill and index_of_min */
28.            if (fill != index_of_min) {
29.                  temp = list[index_of_min];
30.                  list[index_of_min] = list[fill];
31.                  list[fill] = temp;
32.            }
33.       }
34. }
```

Parallel Arrays

- two or more arrays with the same number of elements used for storing related information about a collection of data objects

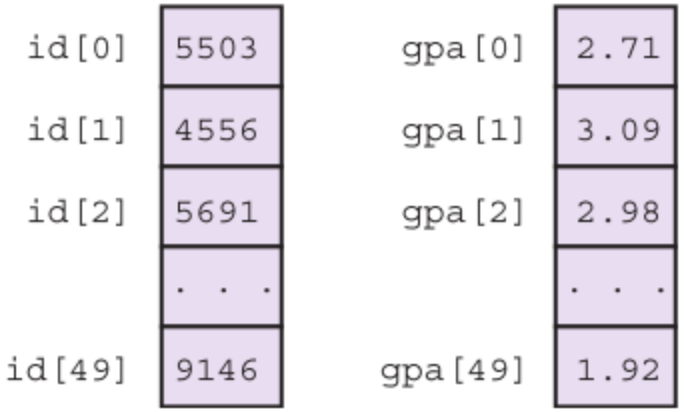


FIGURE 7.17 Student Data in Parallel Arrays

```
1.  /* Read data for parallel arrays and echo stored data.                    */
2.
3.  #include <stdio.h>
4.  #define NUM_STUDENTS 50
5.
6.  int
7.  main(void)
8.  {
9.     int id[NUM_STUDENTS];
10.    double gpa[NUM_STUDENTS];
11.    int i;
12.
13.    for (i = 0; i < NUM_STUDENTS; ++i) {
14.       printf("Enter the id and gpa for student %d: ", i);
15.       scanf("%d%lf", &id[i], &gpa[i]);
16.       printf("%d   %4.2f\n", id[i], gpa[i]);
17.    }
```

(continued)

FIGURE 7.17 (continued)

```
18.
19.     return (0);
20. }

Enter the id and gpa for student 0: 5503 2.71
5503    2.71
Enter the id and gpa for student 1: 4556 3.09
4556    3.09
```

Enumerated Types

- enumerated type
  - a data type whose list of values is specified by the programmer in a type declaration
- enumeration constant
  - an identifier that is one of the values of an enumerated type

```
typedef enum
        {Monday, Tuesday, Wednesday, Thursday,
         Friday, Saturday, Sunday}
day_t;
```

FIGURE 7.18 Enumerated Type for Budget Expenses

```
1.  /* Program demonstrating the use of an enumerated type */
2.
3.  #include <stdio.h>
4.
5.  typedef enum
6.        {entertainment, rent, utilities, food, clothing,
7.         automobile, insurance, miscellaneous}
8.  expense_t;
9.
10. void print_expense(expense_t expense_kind);
11.
12. int
13. main(void)
14. {
15.       expense_t expense_kind;
16.
17.       printf("Enter an expense code between 0 and 7>>");
18.       scanf("%d", &expense_kind);
19.       printf("Expense code represents ");
20.       print_expense(expense_kind);
21.       printf(".\n");
22.
23.       return (0);
24. }
25.
```

(continued)

```
26. /*
27.  * Display string corresponding to a value of type expense_t
28.  */
29. void
30. print_expense(expense_t expense_kind)
31. {
32.       switch (expense_kind) {
33.       case entertainment:
34.             printf("entertainment");
35.             break;
36.
37.       case rent:
38.             printf("rent");
39.             break;
40.
```

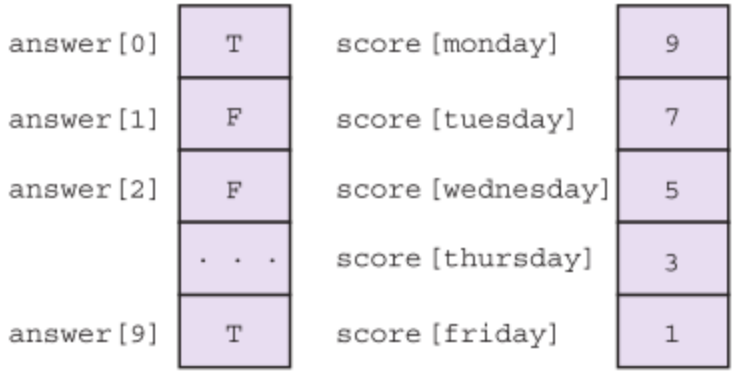
(continued)

FIGURE 7.18 (continued)

```
40.       case utilities:
41.             printf("utilities");
42.             break;
43.
44.       case food:
45.             printf("food");
46.             break;
47.
48.       case clothing:
49.             printf("clothing");
50.             break;
51.
52.       case automobile:
53.             printf("automobile");
54.             break;
55.
56.       case insurance:
57.             printf("insurance");
58.             break;
59.
60.       case miscellaneous:
61.             printf("miscellaneous");
62.             break;
63.
64.       default:
65.             printf("\n*** INVALID CODE ***\n");
66.       }
67. }
```

FIGURE 7.19

Arrays answer and score




```
ascore = 9;
for  (today = monday; today <= friday; ++today) {
    score[today] = ascore;
    ascore -= 2;
}
```

Wrap Up

- A data structure is a grouping of related data items in memory.

- An array is a data structure used to store a collection of data items of the same type.