

Figure 8.14 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. }

```

(continued)

Copyright ©2004 Pearson Addison-Wesley. All rights reserved. 8-2

Figure 8.14 Functions push and pop (cont'd)

```

12. char
13. pop(char stack[], /* input/output - the stack */
14. int *top) /* input/output - pointer to top of stack */
15. {
16.     char item; /* value popped off the stack */
17.
18.     if (*top >= 0) {
19.         item = stack[*top];
20.         --(*top);
21.     } else {
22.         item = STACK_EMPTY;
23.     }
24.
25.     return (item);
26. }

```

Copyright ©2004 Pearson Addison-Wesley. All rights reserved. 8-3

Figure 8.15 Function That Searches for a Target Value in an Array

```

1. #define NOT_FOUND -1 /* Value returned by search function if target not
2. found */
3.
4. /*
5. * Searches for target item in first n elements of array arr:
6. * Returns index of target or NOT_FOUND
7. * Pre: target and first n elements of array arr are defined and n>=0
8. */
9. int
10. search(const int arr[], /* input - array to search */
11. int n) /* input - number of elements to search */
12. {
13.     int i;
14.     found = 0; /* whether or not target has been found */
15.     where; /* index where target found or NOT_FOUND */
16.
17.     /* Compare each element to target */
18.     while (found && i < n) {
19.         if (arr[i] == target)
20.             found = 1;
21.         ++i;
22.     }
23.
24.     /* Returns index of element matching target or NOT_FOUND */
25.     if (found)
26.         where = i;
27.     else
28.         where = NOT_FOUND;
29.
30.     return (where);
31. }

```

Copyright ©2004 Pearson Addison-Wesley. All rights reserved. 8-4

Figure 8.16 Trace of Selection Sort

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

[0]	[1]	[2]	[3]
74	45	83	16

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

[0]	[1]	[2]	[3]
16	45	83	74

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

[0]	[1]	[2]	[3]
16	45	74	83

Copyright ©2004 Pearson Addison-Wesley. All rights reserved. 8-5

Figure 8.17 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(const int list[], int first, int last);
9.
10. /*
11. * Sorts the data in array list
12. * Pre: first n elements of list are defined and n >= 0
13. */
14. void
15. select_sort(int list[], /* input/output - array being sorted */
16. int n) /* input - number of elements to sort */
17. {
18.     int fill; /* first element in unsorted subarray */
19.     temp; /* temporary storage */
20.     index_of_min; /* subscript of next smallest element */
21.
22.     for (fill = 0; fill < n-1; ++fill) {
23.         /* Find position of smallest element in unsorted subarray */
24.         index_of_min = get_min_range(list, fill, n-1);
25.
26.         /* Exchange elements at fill and index_of_min */
27.         if (fill != index_of_min) {
28.             temp = list[index_of_min];
29.             list[index_of_min] = list[fill];
30.             list[fill] = temp;
31.         }
32.     }
33. }

```

Copyright ©2004 Pearson Addison-Wesley. All rights reserved. 8-6

Figure 8.18 A Tic-tac-toe Board Stored as Array tictac

	Column		
	0	1	2
Row 0	X	O	X
1	O	X	O
2	O	X	X

← tictac[1][2]

Figure 8.19 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 8.20 Three-Dimensional Array enroll

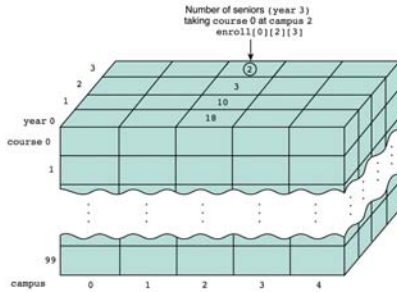


Figure 8.21 Sales Analysis Output

Sales Summary					
Salesperson	Fall	Winter	Spring	Summer	TOTAL
0	2785.00	2282.00	5720.00	6330.00	17117.00
1	4715.00	1676.00	6067.00	929.00	13387.00
2	1253.00	1495.00	2884.00	4173.00	9805.00
3	3946.00	1508.00	2844.00	2969.00	11267.00
4	3558.00	3481.00	2408.00	5590.00	15237.00
QUARTER TOTALS	14257.00	10442.00	20233.00	19991.00	

Figure 8.22 Sales Analysis Main Function

```

1. /*
2.  * Scan sales figures for one year and stores them in a table organized
3.  * by salesperson and quarter. Displays the table and the annual totals
4.  * for each person and the sales totals for each quarter.
5.  */
6.
7. #include <stdio.h>
8.
9. #define SALES_FILE "sales.dat" /* name of sales data file */
10. #define NUM_QUARTERS 4
11. #define NUM_SALES_PERSONS 5
12.
13. typedef enum
14. { Fall, Winter, Spring, Summer }
15. quarter_t;
16.
17. int scan_table(double sales[][NUM_QUARTERS], int num_rows);
18. void sum_row(double row_sum[], double sales[][NUM_QUARTERS], int num_rows);
19. void sum_col(double col_sum[], double sales[][NUM_QUARTERS],
20.             int num_cols);
21. void display_table(double sales[][NUM_QUARTERS], const double
22.                  person_totals[], const double quarter_totals[],
23.                  int num_rows);
24. /* Insert function prototypes for any helper functions. */
25.
26. int
27. main(void)
28. {
29.     double sales[NUM_SALES_PERSONS][NUM_QUARTERS]; /* table of sales */
30.     double person_totals[NUM_SALES_PERSONS]; /* row totals */
31.     double quarter_totals[NUM_QUARTERS]; /* column totals */
32.     int i;
33.
34.     status = scan_table(sales, NUM_SALES_PERSONS);
35.     if (status == 1) {
36.         sum_row(person_totals, sales, NUM_SALES_PERSONS);
37.         sum_col(quarter_totals, sales, NUM_SALES_PERSONS);
38.         display_table(sales, person_totals, quarter_totals,
39.                      NUM_SALES_PERSONS);
40.     }
41.     return 0;
42. }
    
```

Figure 8.23 Function scan_table and Helper Function initialize

```

1. /*
2.  * Scans the sales data from SALES_FILE and computes and stores the sales
3.  * results in the sales table. Flags out-of-range data and data format
4.  * errors.
5.  * Post: Each entry of sales represents the sales total for a particular
6.  * salesperson and quarter.
7.  * Returns 1 for successful table scan, 0 for error in scan.
8.  * Calls: initialize to initialize table to all zeros
9.  */
10. int
11. scan_table(double sales[][NUM_QUARTERS], /* output */
12.            int num_rows) /* input */
13. {
14.     double trans_amt; /* transaction amount */
15.     int trans_person; /* salesperson number */
16.     quarter_t quarter; /* sales quarter */
17.     FILE *sales_file; /* file pointer to sales file */
18.     int valid_table = 1; /* data valid so far */
19.
20.     /* ... */
21. }
    
```

Figure 8.23
Function
scan_table and
Helper Function
initialize (cont'd)

```

19: int status; /* input status */
20: char ch; /* new character in buf line */
21:
22: /* Initialize table to all zeros */
23: initialize(sales, num_rows, 0.0);
24:
25: /* Read and store the valid sales data */
26: sales_file = fopen(SALES_FILE, "r");
27: for (status = fscanf(sales_file, "%d%lf", &strm_persem,
28: &quarter, &strm_amt);
29: status == 1 && !feof(sales_file); strm_persem,
30: status = fscanf(sales_file, "%d%lf", &strm_persem,
31: &quarter, &strm_amt)) {
32: if (fall <= quarter && quarter <= summer &&
33: strm_persem == 0 && strm_persem <= num_rows) {
34: sales[strm_persem][quarter] = strm_amt;
35: } else {
36: printf("Invalid input on quarter = %d\n",
37: quarter);
38: printf(" person is %d, quarter is ", strm_persem);
39: display_quarter(quarter);
40: printf("\n");
41: valid_table = 0;
42: }
43: }
44:
45: if (!valid_table) { /* error already processed */
46: status = 0;
47: } else if (status == EOF) { /* end of data without error */
48: status = 1;
49: } else { /* data format error */
50: printf("Error in sales data format. See data.\n");
51: printf("*****\n");
52: for (status = fscanf(sales_file, "%d", &ch);
53: status == 1 && ch != '\n';
54: status = fscanf(sales_file, "%d", &ch))
55: printf("%c", ch);
56: printf("\n");
57: status = 0;
58: }
59: return (status);
60: }

```

Figure 8.23 Function scan_table and Helper
Function initialize (cont'd)

```

60: /*
61: * Stores value in all elements of sales.
62: * Pre: value is defined and num_rows is the number of rows in
63: * sales.
64: * Post: All elements of sales have the desired value.
65: */
66: void
67: initialize(double sales[][NUM_QUARTERS], /* output */
68: int num_rows, /* input */
69: double value) /* input */
70: {
71: int row;
72: quarter_t quarter;
73:
74: for (row = 0; row < num_rows; ++row)
75: for (quarter = fall; quarter <= summer; ++quarter)
76: sales[row][quarter] = value;
77: }

```

Figure 8.24
Function
display_table
and Helper
Function
display_quarter

```

1: /*
2: * Displays the sales table data in table form along with the row and column
3: * sums.
4: * Pre: sales, person_totals, quarter_totals, and num_rows are defined.
5: * Post: Values stored in the three arrays are displayed.
6: */
7: void
8: display_table(double sales[][NUM_QUARTERS], /* input */
9: const double person_totals[], /* input */
10: const double quarter_totals[], /* input */
11: int num_rows) /* input */
12: {
13: int person;
14: quarter_t quarter;
15:
16: /* Display heading */
17: printf("*****Sales Summary*****\n");
18: printf("Sales", "Salesperson", " ");
19: for (quarter = fall; quarter <= summer; ++quarter)
20: display_quarter(quarter);
21: printf("\n", " ");
22: }
23: printf("*****\n");
24: printf("*****\n");
25:
26: /* Display table */
27: for (person = 0; person < num_rows; ++person) {
28: printf("Sales", person, " ");
29: for (quarter = fall; quarter <= summer; ++quarter)
30: printf("%10.2f", " ", sales[person][quarter]);
31: printf("%10.2f\n", " ", person_totals[person]);
32: }
33: printf("*****\n");
34: printf("*****\n");
35: printf("*****\n");
36: for (quarter = fall; quarter <= summer; ++quarter)
37: printf("%10.2f", " ", quarter_totals[quarter]);
38: printf("\n");
39: }

```

Figure 8.24 Function display_table and
Helper Function display_quarter (cont'd)

```

41: /*
42: * Display an enumeration constant of type quarter_t
43: */
44: void
45: display_quarter(quarter_t quarter)
46: {
47: switch (quarter) {
48: case fall: printf("Fall");
49: break;
50:
51: case winter: printf("Winter");
52: break;
53:
54: case spring: printf("Spring");
55: break;
56:
57: case summer: printf("Summer");
58: break;
59:
60: default: printf("Invalid quarter %d", quarter);
61: }
62: }
63: }

```