Chapter 8

Arrays

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)

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. }
```


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      target,  /* input - value searched for                  */
12.        int      n)       /* input - number of elements to search        */
13. {
14.     int i,
15.         found = 0,       /*  whether or not target has been found       */
16.         where;           /*  index where target found or NOT_FOUND      */
17.
18.     /*  Compares each element to target                                 */
19.     i = 0;
20.     while ((found && i < n) {
21.         if (arr[i] == target)
22.             found = 1;
23.         else
24.             ++i;
25.     }
26.
27.     /* Returns index of element matching target or NOT_FOUND           */
28.     if (found)
29.         where = i;
30.     else
31.         where = NOT_FOUND;
32.
33.     return (where);
34. }
```


Figure 8.16 Trace of Selection Sort

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

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

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

`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

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(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,            /* 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. }
```


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

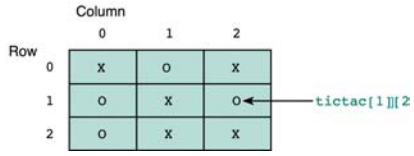

Figure 8.19 Function to Check Whether Tic-tac-toe Board Is Filled

```
/* Checks whether a tic-tac-toe board is completely filled. */
int
filled(char ttt_brd[3][3])  /*  input - tic-tac-toe board                  */
{
    int r, c,   /* row and column subscripts   */
        ans;    /* whether or not board filled */

    /*  Assumes board is filled until blank is found                       */
    ans = 1;

    /*  Resets ans to zero if a blank is found                             */
    for  (r = 0;  r < 3;  ++r)
        for  (c = 0;  c < 3;  ++c)
            if (ttt_brd[r][c] == ' ')
                ans = 0;

    return (ans);
}
```


Figure 8.20 Three-Dimensional Array enroll

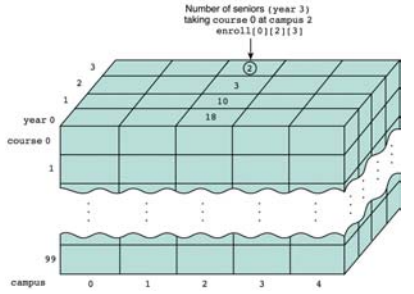

Figure 8.21 Sales Analysis Output

```
                                Sales Summary
                                ----- -------

Salesperson      Fall       Winter      Spring      Summer       TOTAL
-----------------------------------------------------------------------
          0   2785.00     2282.00     5720.00     6330.00    17117.00
          1   4715.00     1676.00     4067.00      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     2608.00     5590.00    15237.00
-----------------------------------------------------------------------
QUARTER TOTALS  16257.00    10442.00    20123.00    19991.00
```


Figure 8.22 Sales Analysis Main Function

```
/*
 *  Scans sales figures for one year and stores them in a table organized
 *  by salesperson and quarter.  Displays the table and the annual totals
 *  for each person and the sales totals for each quarter
 */

#include <stdio.h>

#define SALES_FILE      "sales.dat"   /* name of sales data file */
#define NUM_QUARTERS    4
#define NUM_SALES_PEOPLE 5

typedef enum
        {fall, winter, spring, summer}
quarter_t;

int  scan_table(double sales[][NUM_QUARTERS], int num_rows);
void sum_rows(double row_sum[], double sales[][NUM_QUARTERS], int num_rows);
void sum_columns(double col_sum[], double sales[][NUM_QUARTERS],
                 int num_rows);
void display_table(double sales[][NUM_QUARTERS], const double
                   person_totals[], const double quarter_totals[],
                   int num_rows);
/*  Insert function prototypes for any helper functions. */

int
main(void)
{
    double sales[NUM_SALES_PEOPLE][NUM_QUARTERS]; /* table of sales */
    double person_totals[NUM_SALES_PEOPLE];       /* row totals */
    double quarter_totals[NUM_QUARTERS];          /* column totals */
    int    status;

    status = scan_table(sales, NUM_SALES_PEOPLE);
    if (status == 1) {
        sum_rows(person_totals, sales, NUM_SALES_PEOPLE);
        sum_columns(quarter_totals, sales, NUM_SALES_PEOPLE);
        display_table(sales, person_totals, quarter_totals,
                      NUM_SALES_PEOPLE);
    }
    return (0);
}
```


Figure 8.23 Function scan_table and Helper Function initialize

```
/*
 *  Scans the sales data from SALES_FILE and computes and stores the sales
 *  results in the sales table.  Flags out-of-range data and data format
 *  errors.
 *  Post:  Each entry of sales represents the sales total for a particular
 *         salesperson and quarter.
 *         Returns 1 for successful table scan, 0 for error in scan.
 *  Calls: initialize to initialize table to all zeros
 */
int
scan_table(double sales[][NUM_QUARTERS], /* output */
           int num_rows)                 /* input  */
{
    double    trans_amt;        /* transaction amount */
    int       trans_person;     /* salesperson number */
    quarter_t quarter;          /* sales quarter      */
    FILE      *sales_filep;     /* file pointer to sales file */
    int       valid_table = 1;  /* data valid so far  */

                                                    (continued)
```

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

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

```
/*
 *  Stores value in all elements of sales.
 *  Pre:  value is defined and num_rows is the number of rows in
 *        sales.
 *  Post: All elements of sales have the desired value.
 */
void
initialize(double sales[][NUM_QUARTERS],  /* output */
           int    num_rows,               /* input  */
           double value)                  /* input  */
{
    int       row;
    quarter_t quarter;

    for  (row = 0;  row < num_rows;  ++row)
        for  (quarter = fall;  quarter <= summer;  ++quarter)
            sales[row][quarter] = value;
}
```


Figure 8.24 Function display_table and Helper Function display_quarter

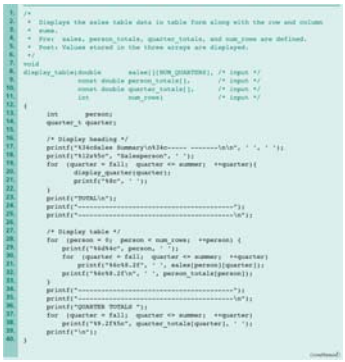

Figure 8.24 Function display_table and Helper Function display_quarter (cont'd)

```
/*
 *  Display an enumeration constant of type quarter_t
 */
void
display_quarter(quarter_t quarter)
{
    switch (quarter) {
    case fall:   printf("Fall");
                 break;

    case winter: printf("Winter");
                 break;

    case spring: printf("Spring");
                 break;

    case summer: printf("Summer");
                 break;

    default:     printf("Invalid quarter %d", quarter);
    }
}
```