Computer Programming

Arrays

Declaring Arrays

- Simple memory types: single memory cell
- Group of related data items: adjacent memory cells
  - Array: uses consecutive area in memory
    - Can be referenced as a group
  - Array elements: each data item
    - Can be accessed individually

Ex: double x[8];

- Name of the array is x
- There are eight elements (memory cells)
- Each element is double

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	–54.5

Declaring Arrays

Ex: double x[8];

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	–54.5

- Each element can be accessed individually

Ex: x[0], x[1],…, x[7]

- x[5] is a subscripted variable
- 5 is a array subscript
  - Any integer
  - From 0 to 7 !!!

```
printf(“%.2f”, x[0]);
x[3] = 12.20;
sum = sum +x[5];
x[2] = 13 + x[0];
x[7] = pow(x[1],x[4]);
scanf(“%lf”, &x[0]);
```

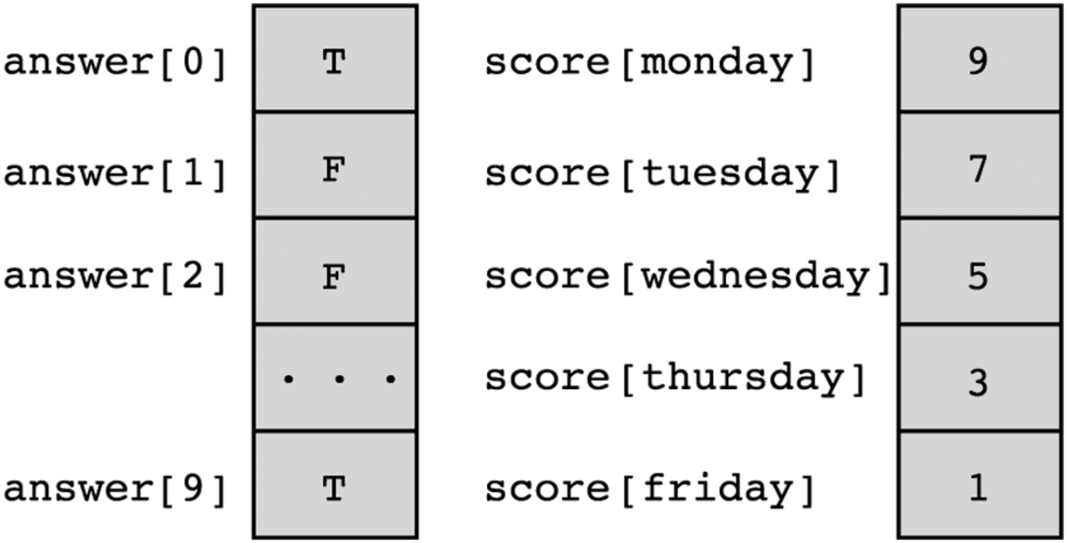
Example: Student Records

```
#define NUM_STUDENTS 50

int id[NUM_STUDENTS];
double gpa[NUM_STUDENTS];
```

- Parallel arrays
  - id[i] and gpa[i] are related
    - First student’s ID is in id[0]
    - First student’s GPA is in gpa[0]

Example: Grading Program

```
#define NUM_QUEST 10
#define NUM_CLASS_DAYS 5

typedef enum
    {monday, tuesday, wednesday, thursday, friday}
class_days_t;

char ansers[NUM_QUEST];
Int score[NUM_CLASS_DAYS];
```



answer[0]	T	score[monday]	9
answer[1]	F	score[tuesday]	7
answer[2]	F	score[wednesday]	5
	. . .	score[thursday]	3
answer[9]	T	score[friday]	1

Declaring Arrays

- More than one array can be declared at once

```
double bolts[20], needle, pins[10];
```

- An array can be initialized in declaration

```
int primes[5] = {2, 3, 5, 7, 11};
int primes[] = {2, 3, 5, 7, 11};
```

Syntax:

```
element_type array_name[size];
element_type array_name[size] = {initialization list};
```

Array Subscripts

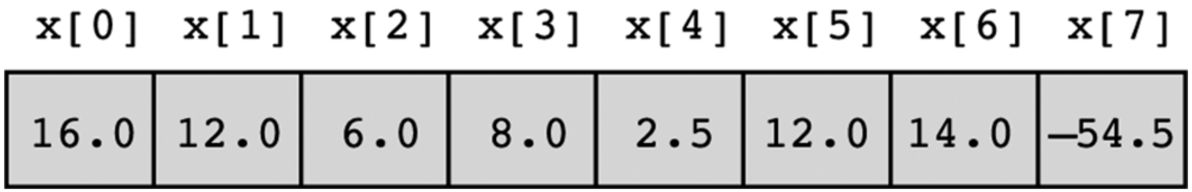
- Subscript specifies array elements
  - Any expression if type int
  - Must be between 0 to size-1

- Syntax

```
array_name[subscript]
```

EX:

```
i = 5;
x[i-2] = x[i]-2;
x[2*i] = x[i--];
i = (int)x[(int)x[3+1]];
```



x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	–54.5

Using for loops to access arrays

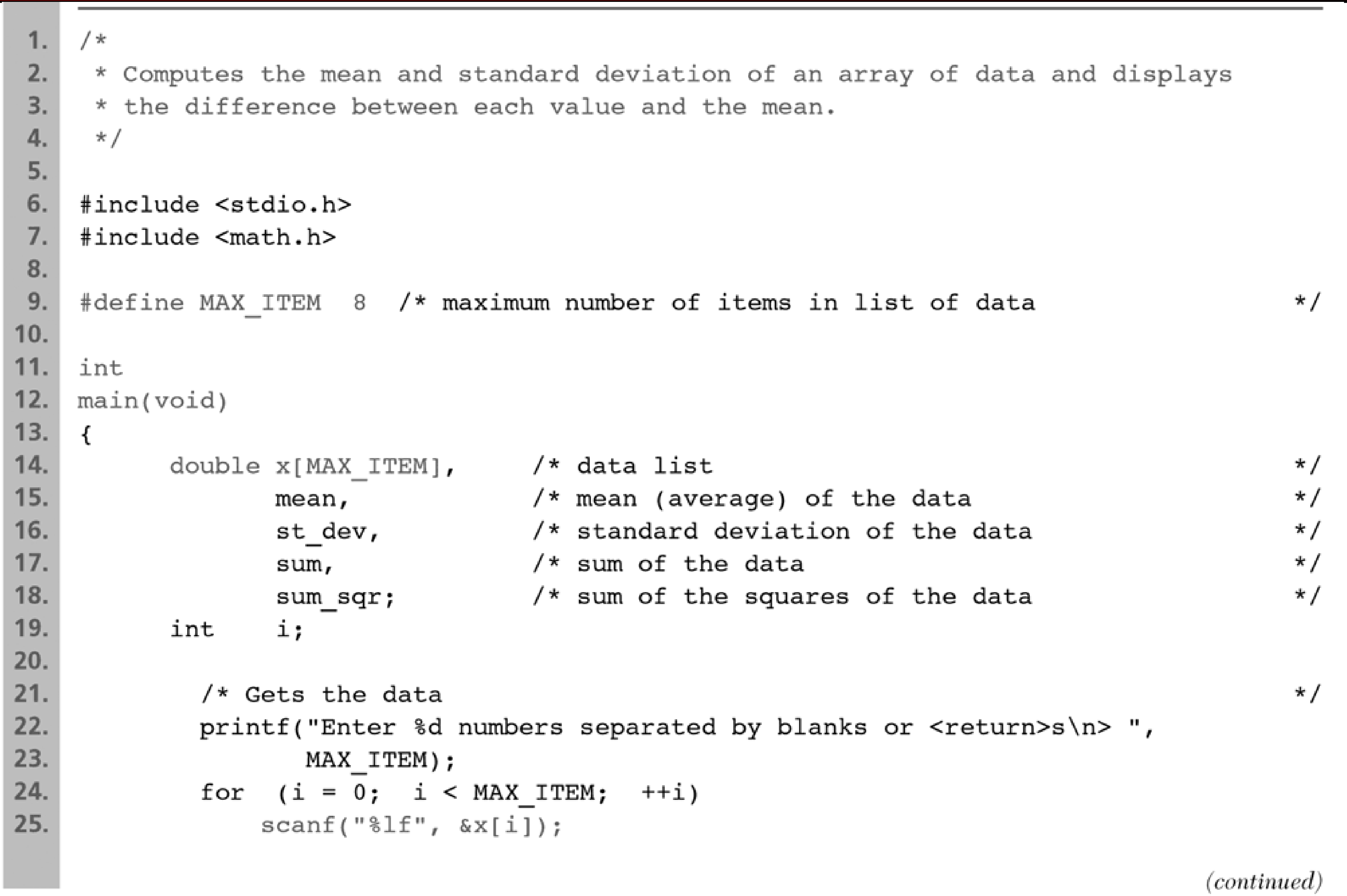
- Processing elements of an array in sequence
- Ex: Array of squares

```
int square[20], i;
for (i = 0; i < 20; i++)
  square[i] = i * i;
```

- Ex: Sum of scores

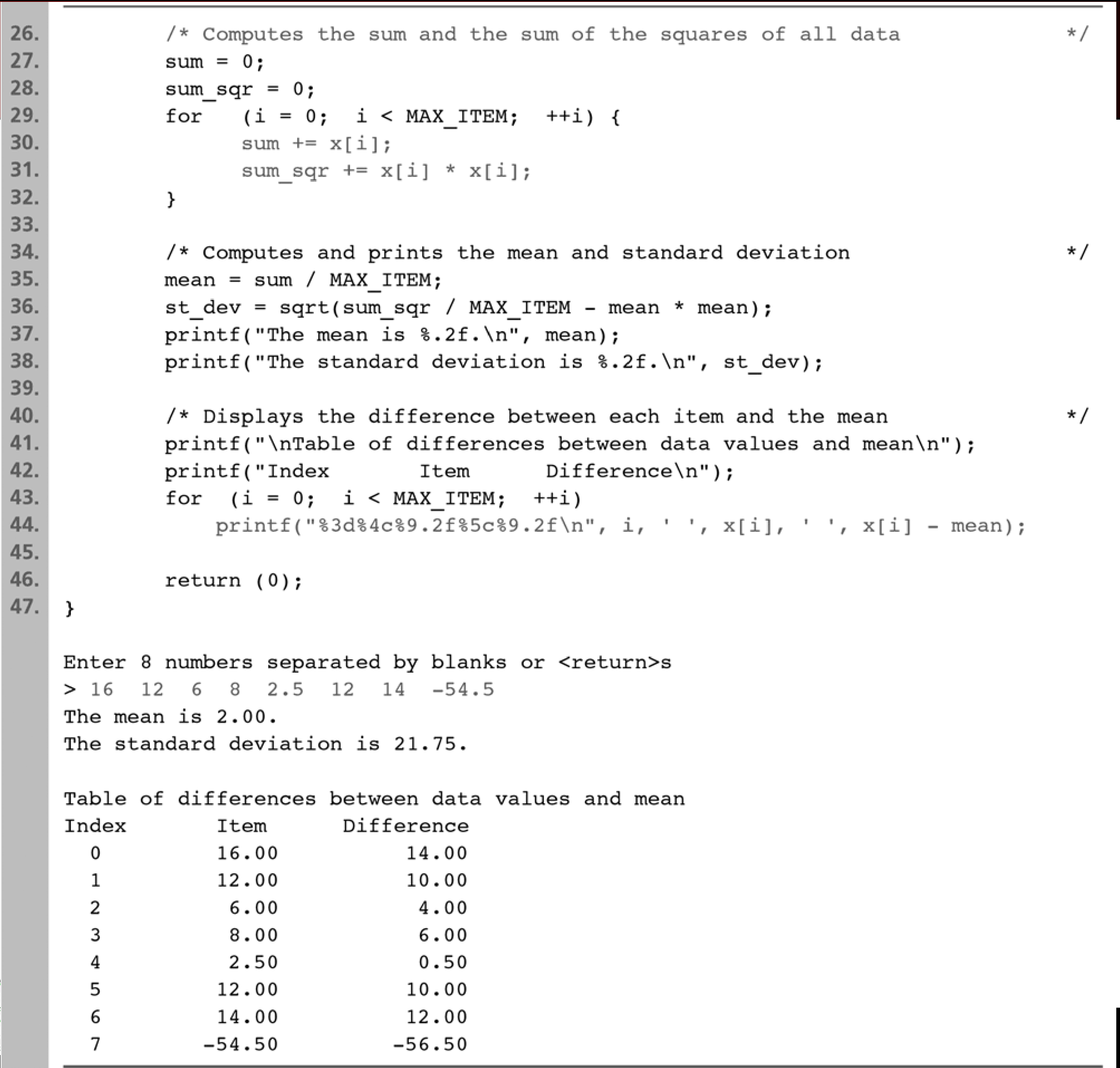
```
sum_score = 0;
for(today = monday; today <= friday; ++today)
  scanf(“%d”, &score[today]);
for(today = monday; today <= friday; ++today)
  sum_score += score[today];
```

Program to Print a Table of Differences



```c
/*
 * Computes the mean and standard deviation of an array of data and displays
 * the difference between each value and the mean.
 */

#include <stdio.h>
#include <math.h>

#define MAX_ITEM  8  /* maximum number of items in list of data                */

int
main(void)
{
      double x[MAX_ITEM],     /* data list                                     */
             mean,            /* mean (average) of the data                    */
             st_dev,          /* standard deviation of the data                */
             sum,             /* sum of the data                               */
             sum_sqr;         /* sum of the squares of the data                */
      int    i;

        /* Gets the data                                                       */
        printf("Enter %d numbers separated by blanks or <return>s\n> ",
               MAX_ITEM);
        for  (i = 0;  i < MAX_ITEM;  ++i)
            scanf("%lf", &x[i]);
```

(continued)



```c
        /* Computes the sum and the sum of the squares of all data             */
        sum = 0;
        sum_sqr = 0;
        for   (i = 0;  i < MAX_ITEM;  ++i) {
              sum += x[i];
              sum_sqr += x[i] * x[i];
        }

        /* Computes and prints the mean and standard deviation                 */
        mean = sum / MAX_ITEM;
        st_dev = sqrt(sum_sqr / MAX_ITEM - mean * mean);
        printf("The mean is %.2f.\n", mean);
        printf("The standard deviation is %.2f.\n", st_dev);

        /* Displays the difference between each item and the mean              */
        printf("\nTable of differences between data values and mean\n");
        printf("Index       Item      Difference\n");
        for  (i = 0;  i < MAX_ITEM;  ++i)
            printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', x[i], ' ', x[i] - mean);

        return (0);
}
```

```
Enter 8 numbers separated by blanks or <return>s
> 16  12  6  8  2.5  12  14  -54.5
The mean is 2.00.
The standard deviation is 21.75.

Table of differences between data values and mean
Index       Item      Difference
  0         16.00          14.00
  1         12.00          10.00
  2          6.00           4.00
  3          8.00           6.00
  4          2.50           0.50
  5         12.00          10.00
  6         14.00          12.00
  7        -54.50         -56.50
```

Array Elements as Function Arguments

- Array elements can be arguments to functions
  - As other variables
  - Input argument

```
                    printf(“%d”, a[1]);
```

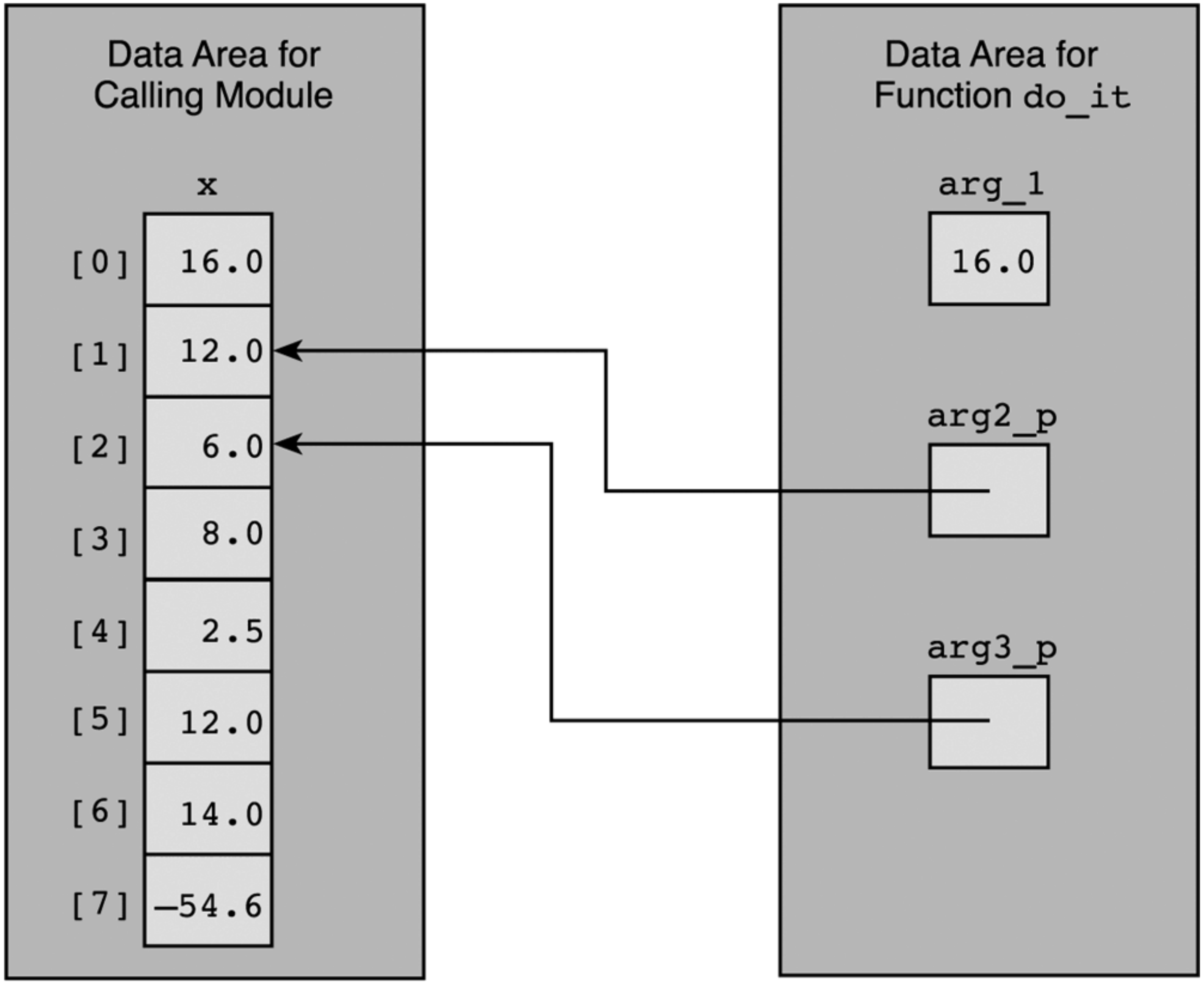
  - Output argument

```
                    scanf(“%d”, &a[1]);
```

  - Input/output argument

```
void do_it(double arg1, double *arg2_p , double *arg3_p);
do_it(p, &r, &s);
do_it(x[0], &x[1], &x[2]);
```

Data Area for Calling Module and do_it



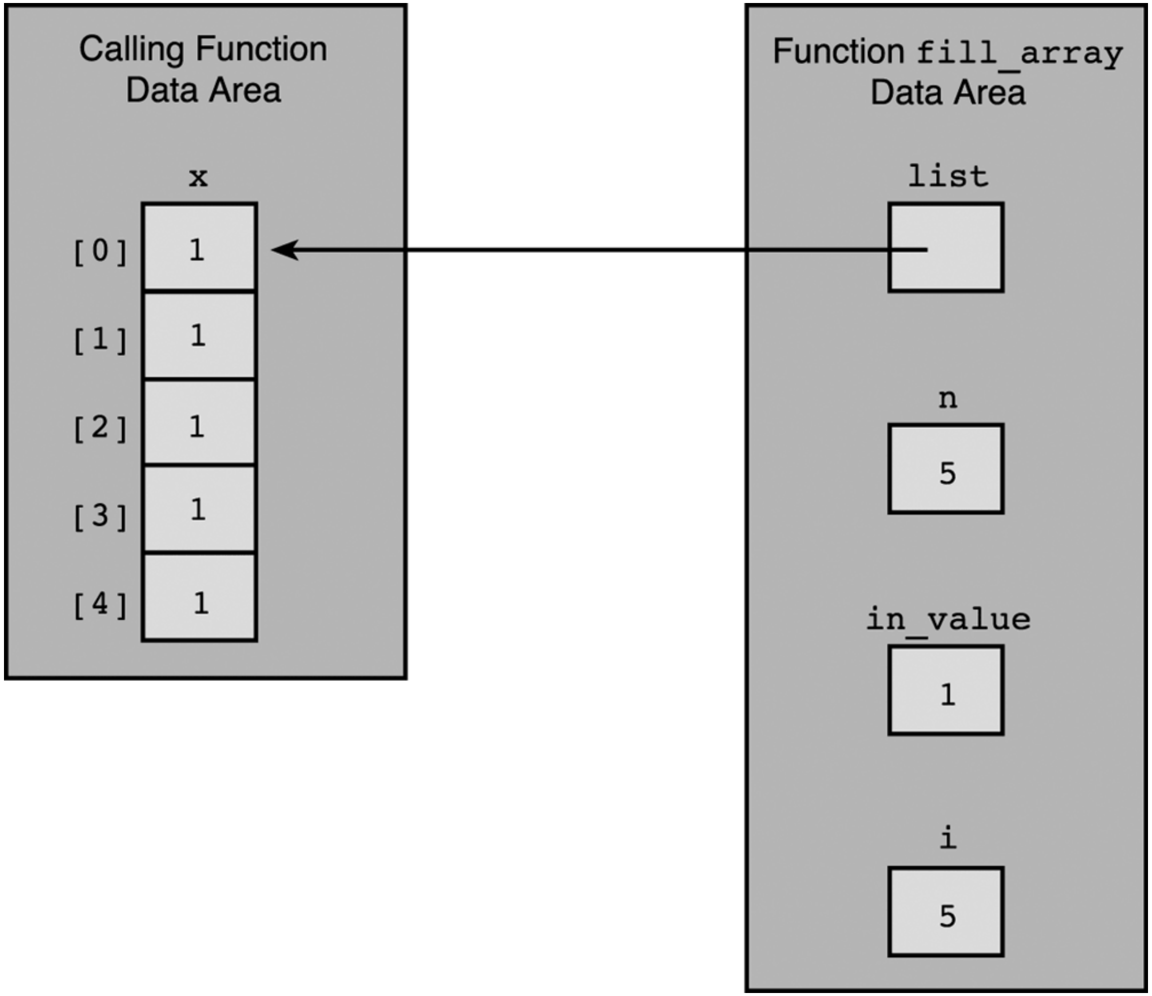
Array Arguments

- Passing whole arrays to functions
  - Array as a actual parameter
    - array name without subscript in the argument list
  - Formal parameter is the address of the first array element
    - Use subscript to access array’s elements
    - Work on the original array not on a copy!...

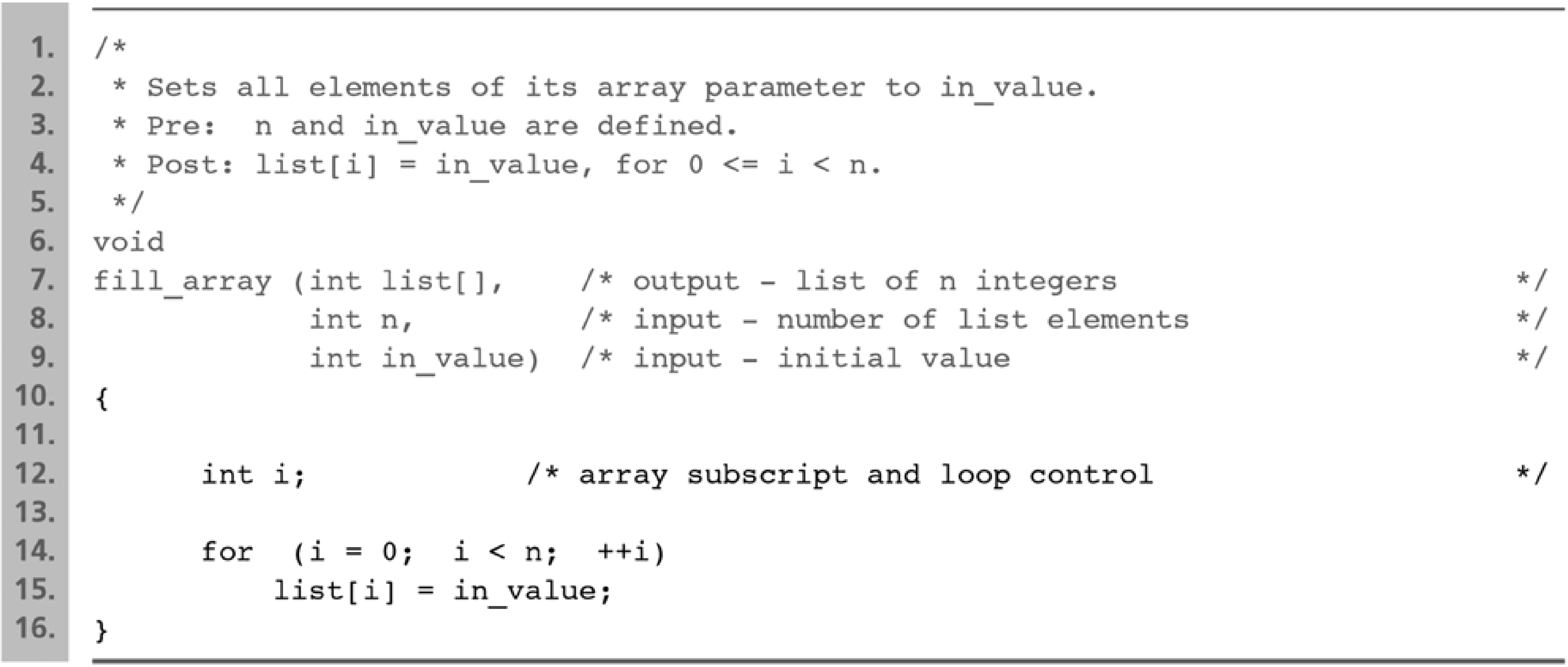
Ex: Fill an array with the same value

```
void fill_array(int list[], int n, int in_value);

fill_array(x, 5, 1)
```

Data Areas for fill_array (x, 5, 1);



Function fill_array



```
/*
 * Sets all elements of its array parameter to in_value.
 * Pre:  n and in_value are defined.
 * Post: list[i] = in_value, for 0 <= i < n.
 */
void
fill_array (int list[],    /* output - list of n integers                   */
            int n,         /* input - number of list elements               */
            int in_value)  /* input - initial value                         */
{

      int i;            /* array subscript and loop control                    */

      for  (i = 0;  i < n;  ++i)
          list[i] = in_value;
}
```

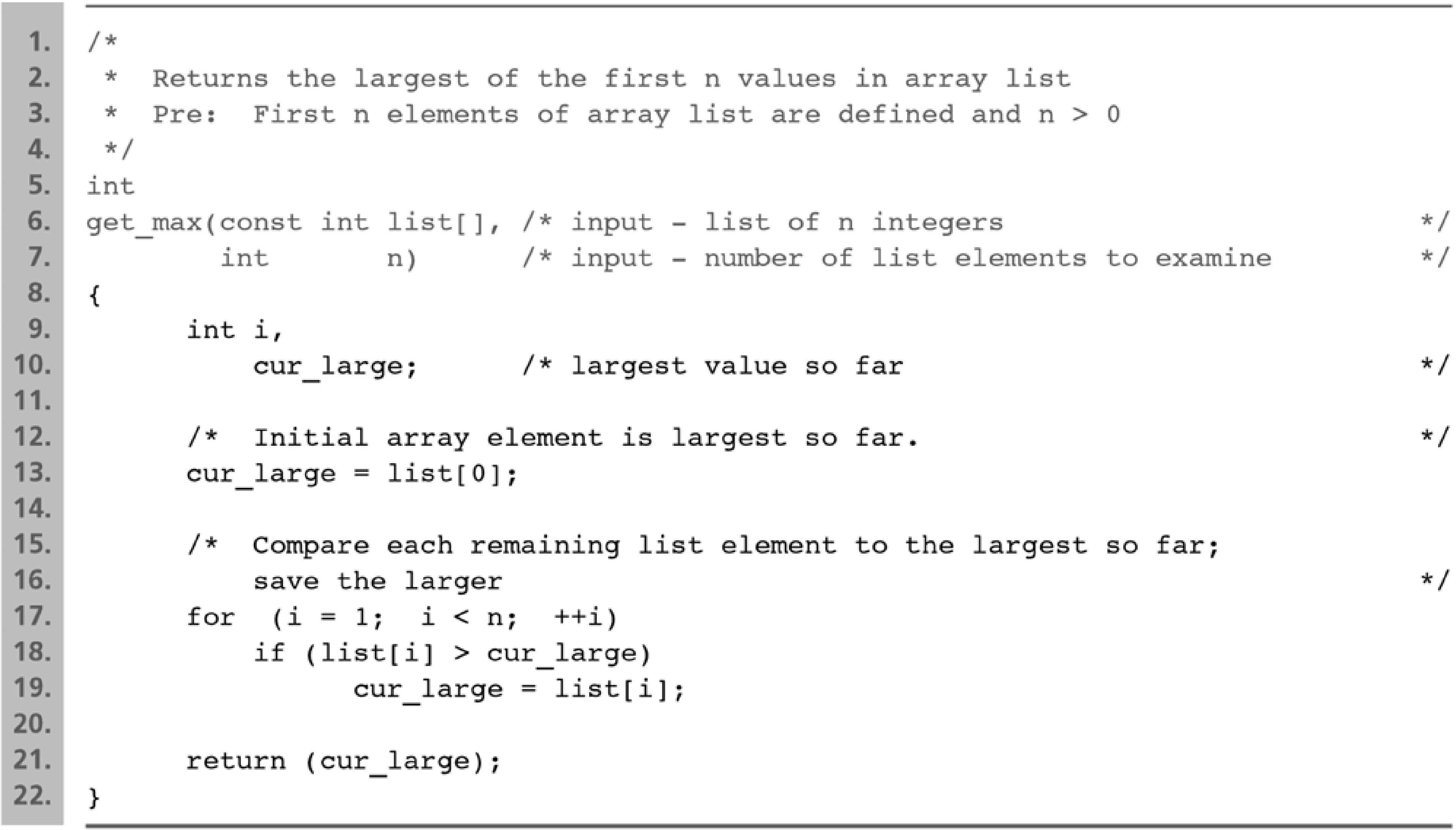
Array Arguments

- You can use *list instead of list[] in a formal parameter list
  - Pass an array as a argument
  - int list[]; means parameter is an array
  - int *list;   is correct as well
    - Array argument: passing the address of the first element
    - But, it does not shot that the argument is an array!
    - You should remember that it is array not output parameter
  - What if the array is only input parameter
    - Use the const qualifier
    - You can not modify const parameters

Ex: Finding max element in an array

- You do not need to modify array elements
- It is safer to use const qualifier

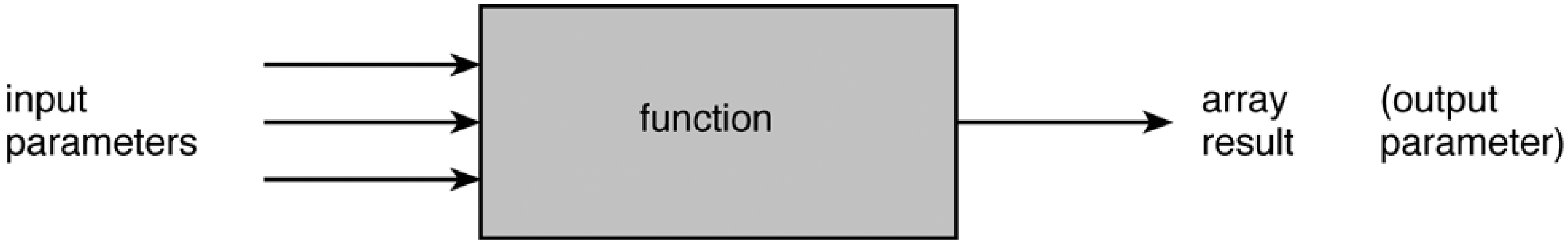
Find the Largest Element



```c
/*
 *  Returns the largest of the first n values in array list
 *  Pre:  First n elements of array list are defined and n > 0
 */
int
get_max(const int list[], /* input - list of n integers                         */
        int       n)      /* input - number of list elements to examine          */
{
      int i,
          cur_large;      /* largest value so far                                */

      /*  Initial array element is largest so far.                               */
      cur_large = list[0];

      /*  Compare each remaining list element to the largest so far;
          save the larger                                                        */
      for  (i = 1;  i < n;  ++i)
          if (list[i] > cur_large)
                cur_large = list[i];

      return (cur_large);
}
```

Returning Array Result

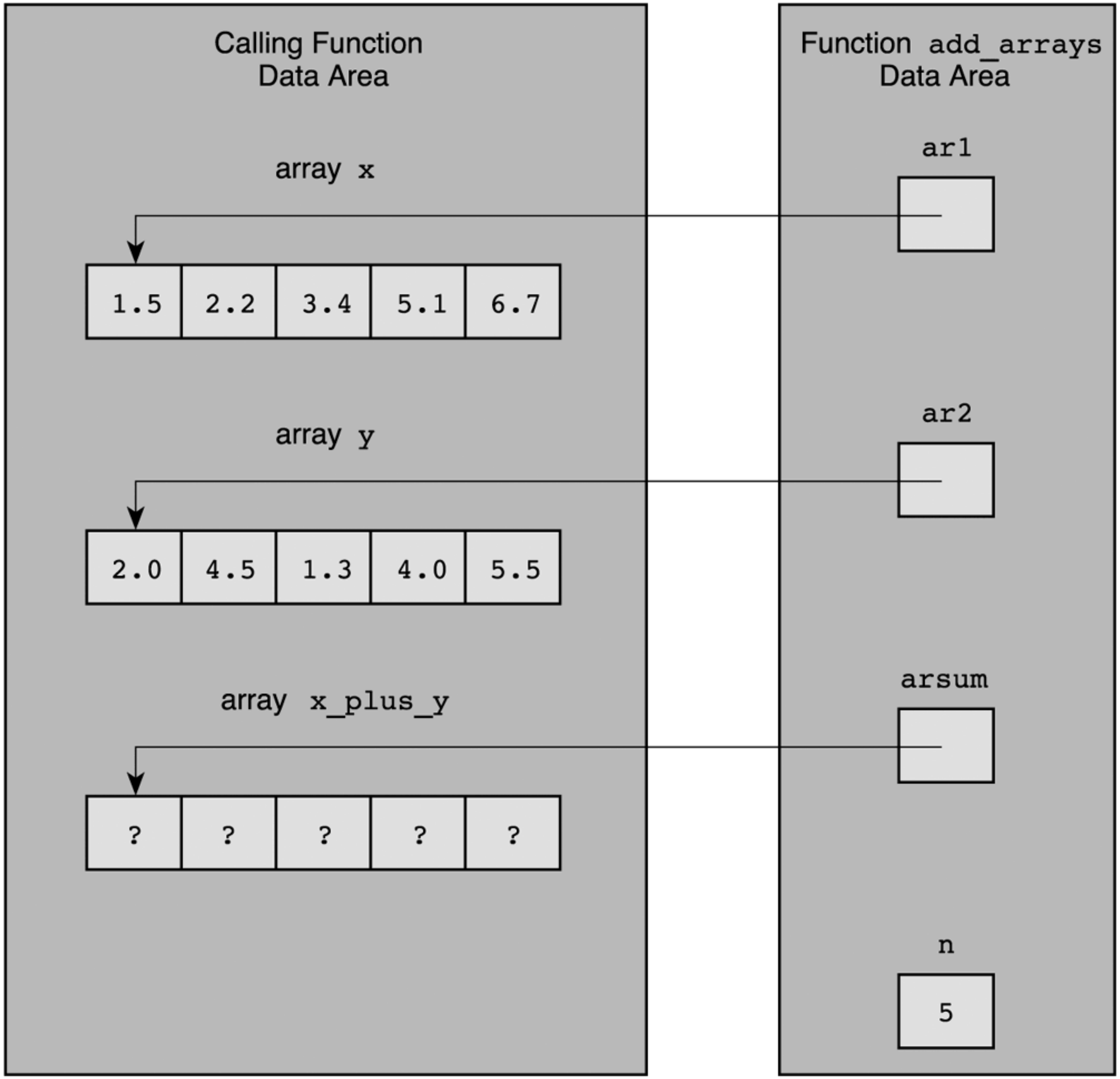
- You can not return an array as a functions return value
- You should define it as a output parameter



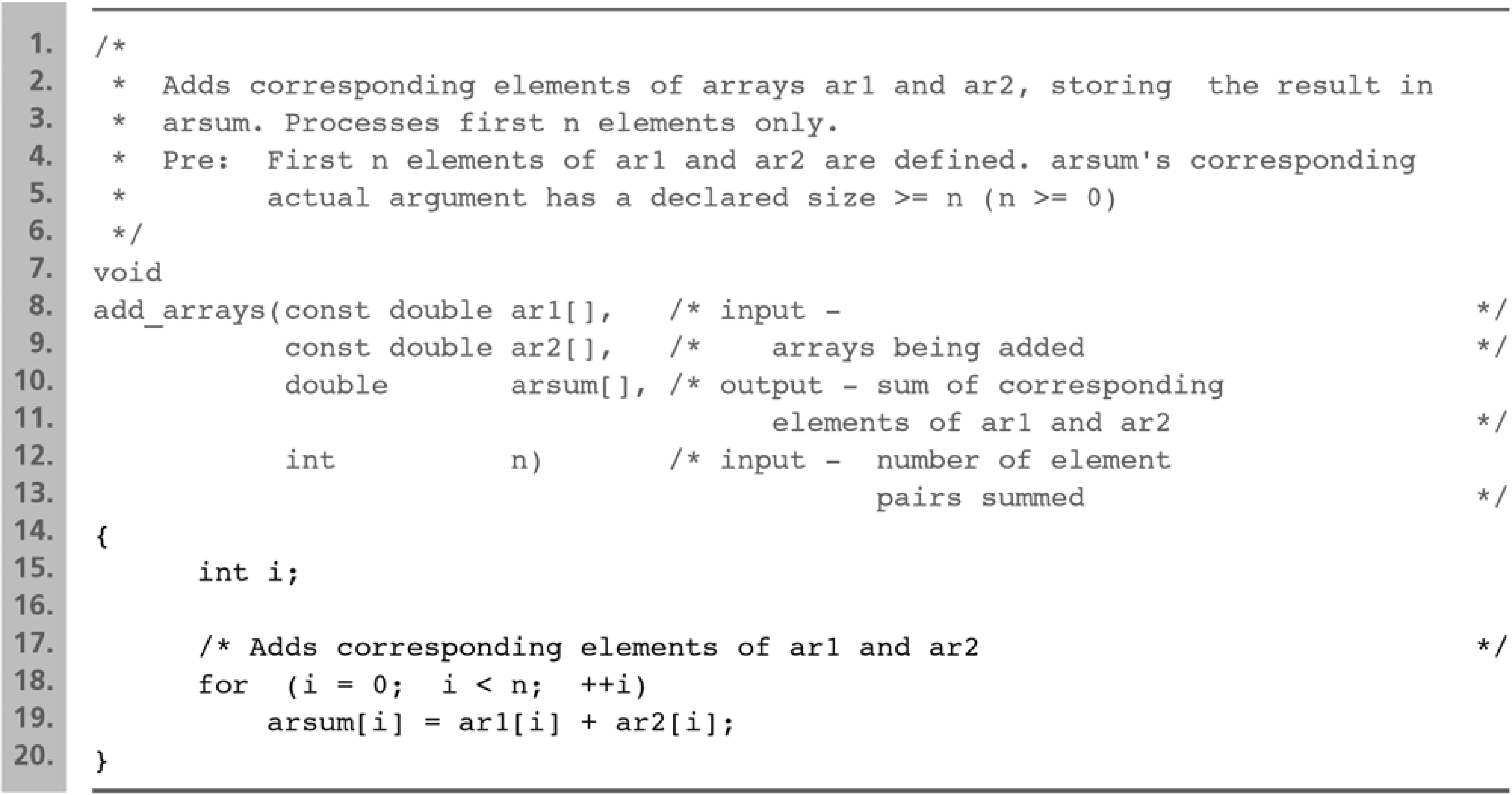
Ex: Adding two arrays

```
void add_arrays(const double ar1[], const double ar2,
                               doube arrsum[], int n);


add_arrays(x, y, x_plus_y, 5);
```

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



Function to Add Two Arrays

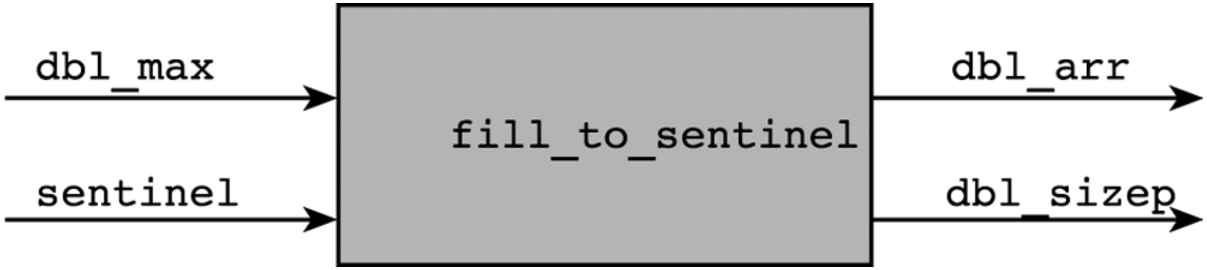


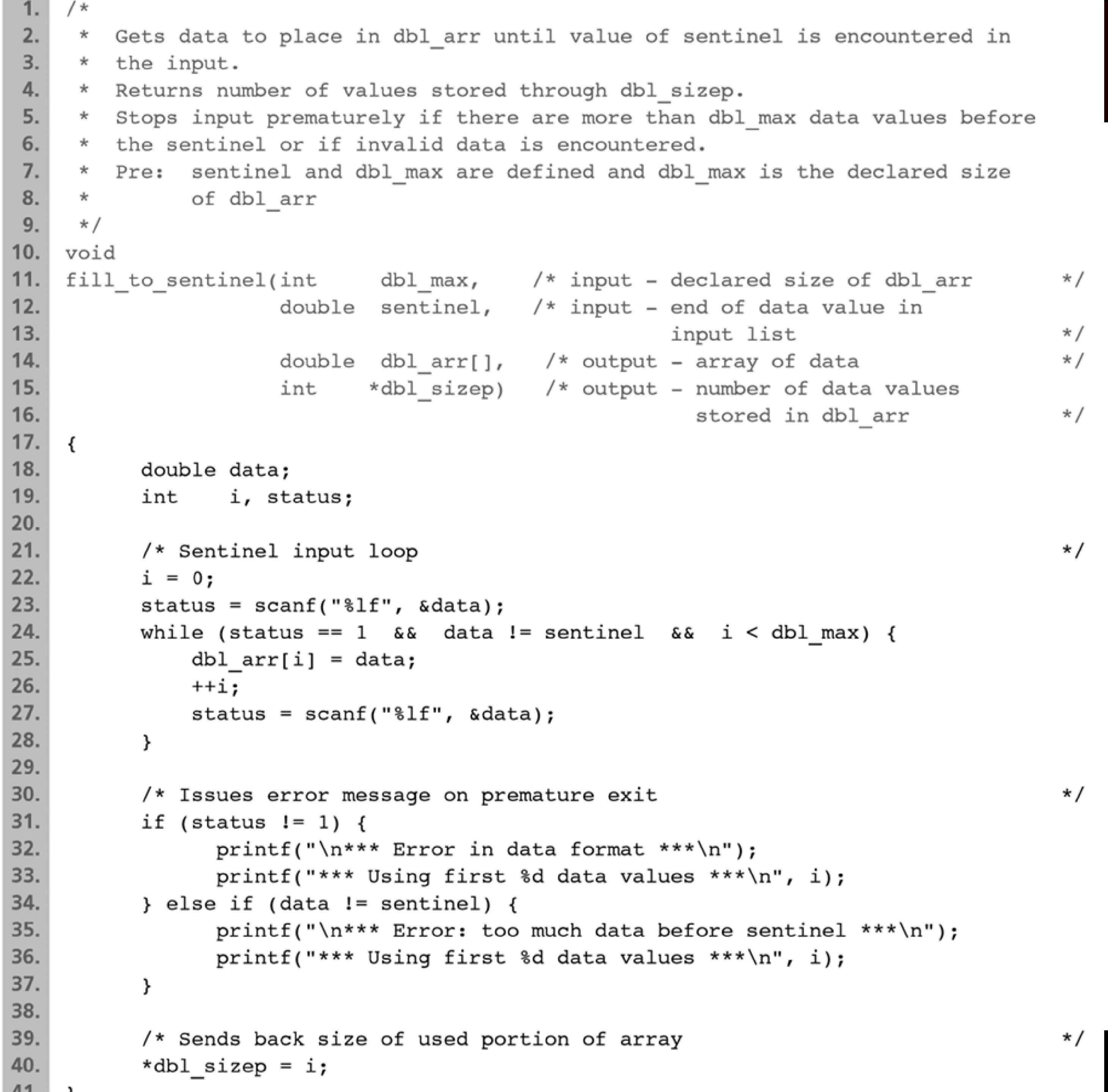
```
/*
 *  Adds corresponding elements of arrays ar1 and ar2, storing  the result in
 *  arsum. Processes first n elements only.
 *  Pre:  First n elements of ar1 and ar2 are defined. arsum's corresponding
 *        actual argument has a declared size >= n (n >= 0)
 */
void
add_arrays(const double ar1[],   /* input -                                    */
           const double ar2[],   /*    arrays being added                      */
           double       arsum[], /* output - sum of corresponding
                                       elements of ar1 and ar2                 */
           int          n)       /* input -  number of element
                                             pairs summed                      */
{
      int i;

      /* Adds corresponding elements of ar1 and ar2                            */
      for  (i = 0;  i < n;  ++i)
          arsum[i] = ar1[i] + ar2[i];
}
```

Partially Filled Arrays

- Array is not completely used
  - Some part is reserved for later use
  - Need to reuse the same array for other purpose later
- Need to remember the actual number of elements in the array
  - Declared size should be larger than actual size!..

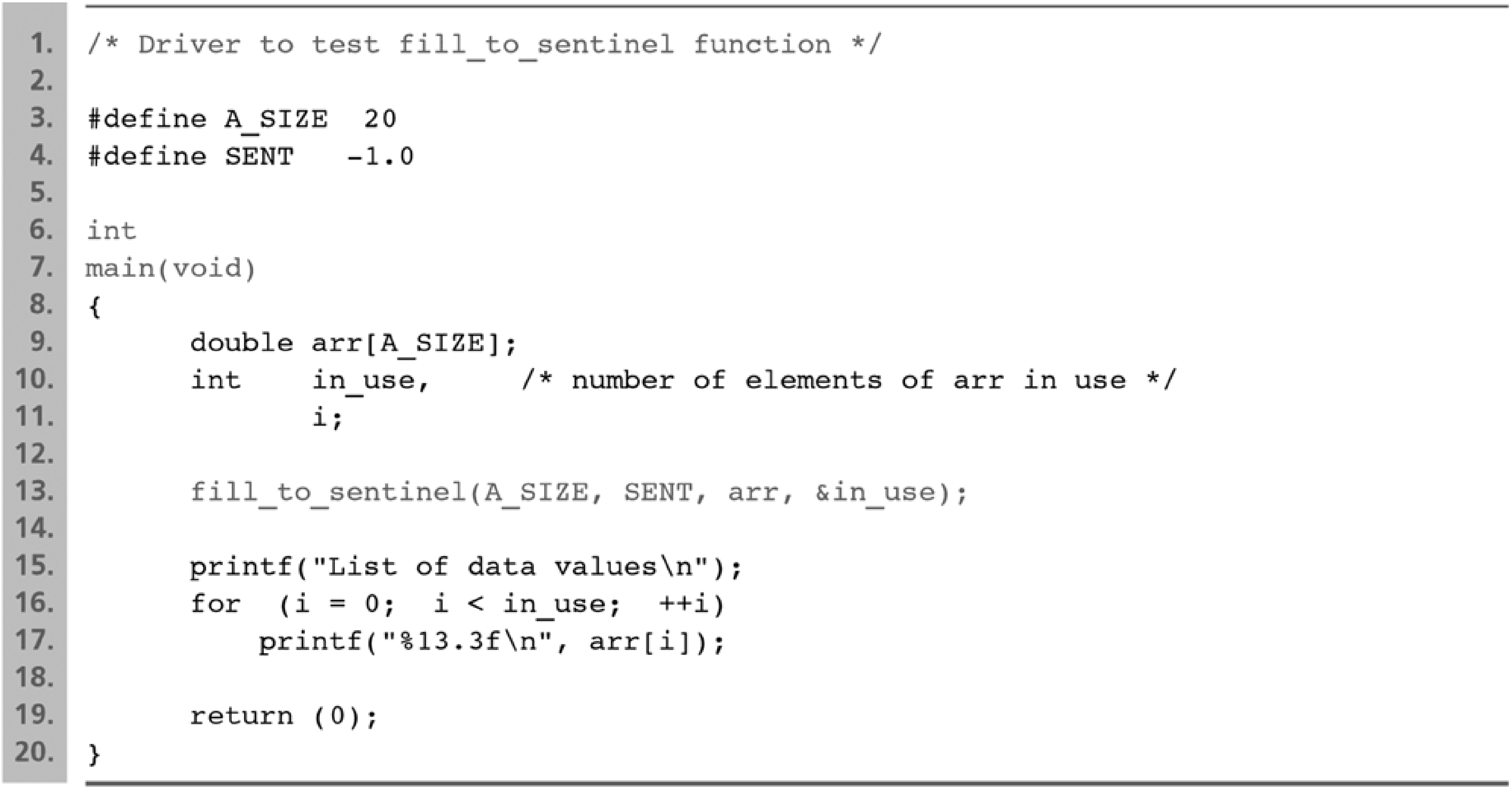
Ex: Fill an array until a sentinel value is entered





```
/*
 *  Gets data to place in dbl_arr until value of sentinel is encountered in
 *  the input.
 *  Returns number of values stored through dbl_sizep.
 *  Stops input prematurely if there are more than dbl_max data values before
 *  the sentinel or if invalid data is encountered.
 *  Pre:  sentinel and dbl_max are defined and dbl_max is the declared size
 *        of dbl_arr
 */
void
fill_to_sentinel(int     dbl_max,    /* input - declared size of dbl_arr        */
                 double  sentinel,   /* input - end of data value in
                                                input list                      */
                 double  dbl_arr[],   /* output - array of data                 */
                 int    *dbl_sizep)   /* output - number of data values
                                                  stored in dbl_arr             */
{
      double data;
      int    i, status;

      /* Sentinel input loop                                                    */
      i = 0;
      status = scanf("%lf", &data);
      while (status == 1  &&  data != sentinel  &&  i < dbl_max) {
          dbl_arr[i] = data;
          ++i;
          status = scanf("%lf", &data);
      }

      /* Issues error message on premature exit                                 */
      if (status != 1) {
            printf("\n*** Error in data format ***\n");
            printf("*** Using first %d data values ***\n", i);
      } else if (data != sentinel) {
            printf("\n*** Error: too much data before sentinel ***\n");
            printf("*** Using first %d data values ***\n", i);
      }

      /* Sends back size of used portion of array                               */
      *dbl_sizep = i;
}
```

Driver for Testing fill_to_sentinel



```c
/* Driver to test fill_to_sentinel function */

#define A_SIZE  20
#define SENT   -1.0

int
main(void)
{
      double arr[A_SIZE];
      int    in_use,     /* number of elements of arr in use */
             i;

      fill_to_sentinel(A_SIZE, SENT, arr, &in_use);

      printf("List of data values\n");
      for  (i = 0;  i < in_use;  ++i)
          printf("%13.3f\n", arr[i]);

      return (0);
}
```

Stacks

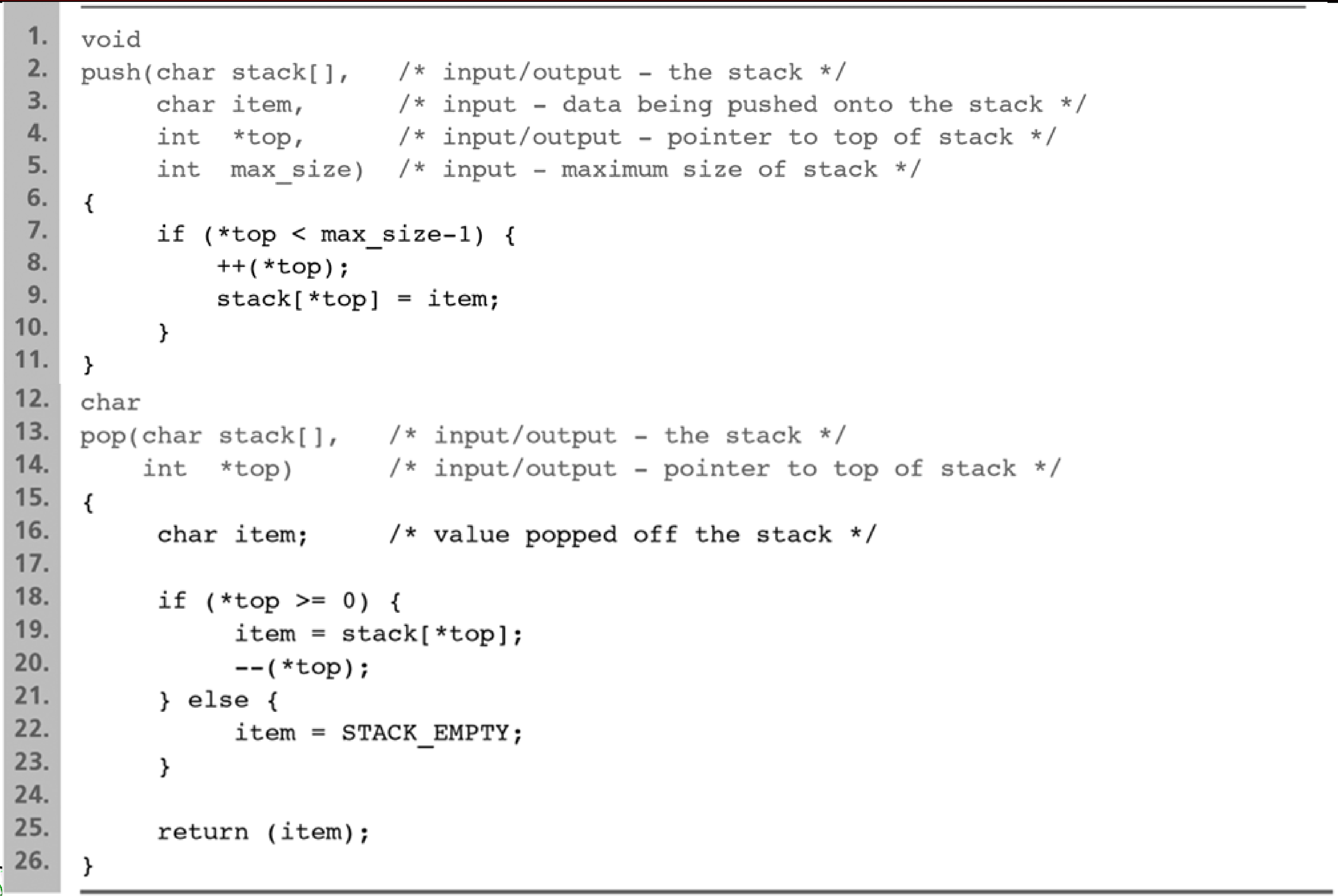
- Remember stack?..
  - Only top element can be accessed
  - Operations
    - Push
    - Pop
  - Array as a stack
  - What should be parameters to push and pop

Stacks

- Remember stack?..
  - Only top element can be accessed
  - Operations
    - Push
    - Pop
  - Array as a stack
  - What should be parameters to push and pop

```
void push(char stack[], char item, int *top, int max_size);
char pop(char stack[], int *top);

push(s, ‘2’, &s_top, STACK_SIZE);
c = pop(s, &s_top);
```

Functions push and pop



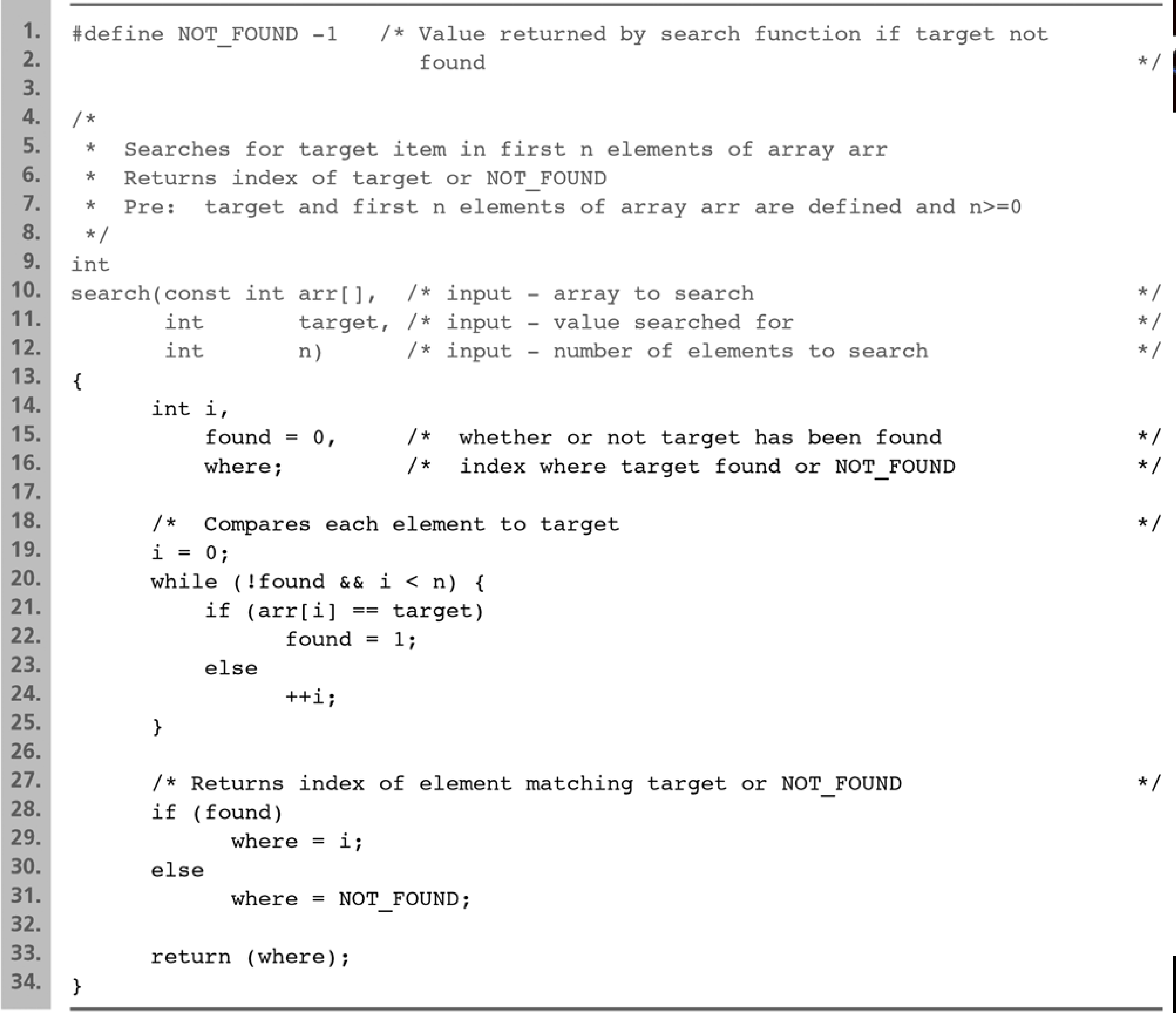
```
void
push(char stack[],    /* input/output - the stack */
     char item,       /* input - data being pushed onto the stack */
     int  *top,       /* input/output - pointer to top of stack */
     int  max_size)   /* input - maximum size of stack */
{
     if (*top < max_size-1) {
         ++(*top);
         stack[*top] = item;
     }
}
char
pop(char stack[],     /* input/output - the stack */
    int  *top)        /* input/output - pointer to top of stack */
{
     char item;       /* value popped off the stack */

     if (*top >= 0) {
          item = stack[*top];
          --(*top);
     } else {
          item = STACK_EMPTY;
     }

     return (item);
}
```

Searching an Array

- Two important problems in processing arrays
  - Searching: Locating a particular value
  - Sorting: Ordering the elements
- Searching: Linear search
  - Test each elements in the array one by one
  - Until the array is exhausted or the target is found

Linear Search Algorithm

```
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 target 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
        return -1 as the search result
```



```c
#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);
}
```

Sorting an Array

- Sorting is quite useful
  - Many operations implemented more efficiently if the data is sorted
  - Output is more understandable if the information is sorted
- Selection sort:
  - Not very efficient but simple
  - Locate the smallest element and move it to location 0
  - Locate the smallest element in the remaining array starting with location 1 and move it to location 1
  - Locate the smallest element in the remaining array starting with location 2 and move it to location 2
  - Continue like this until location n-2

Selection Sort Algorithm

```
1. for each value of fill from 0 to n-2
        2. find 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
                4. exchange the smallest element with the one at the position fill
```

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].

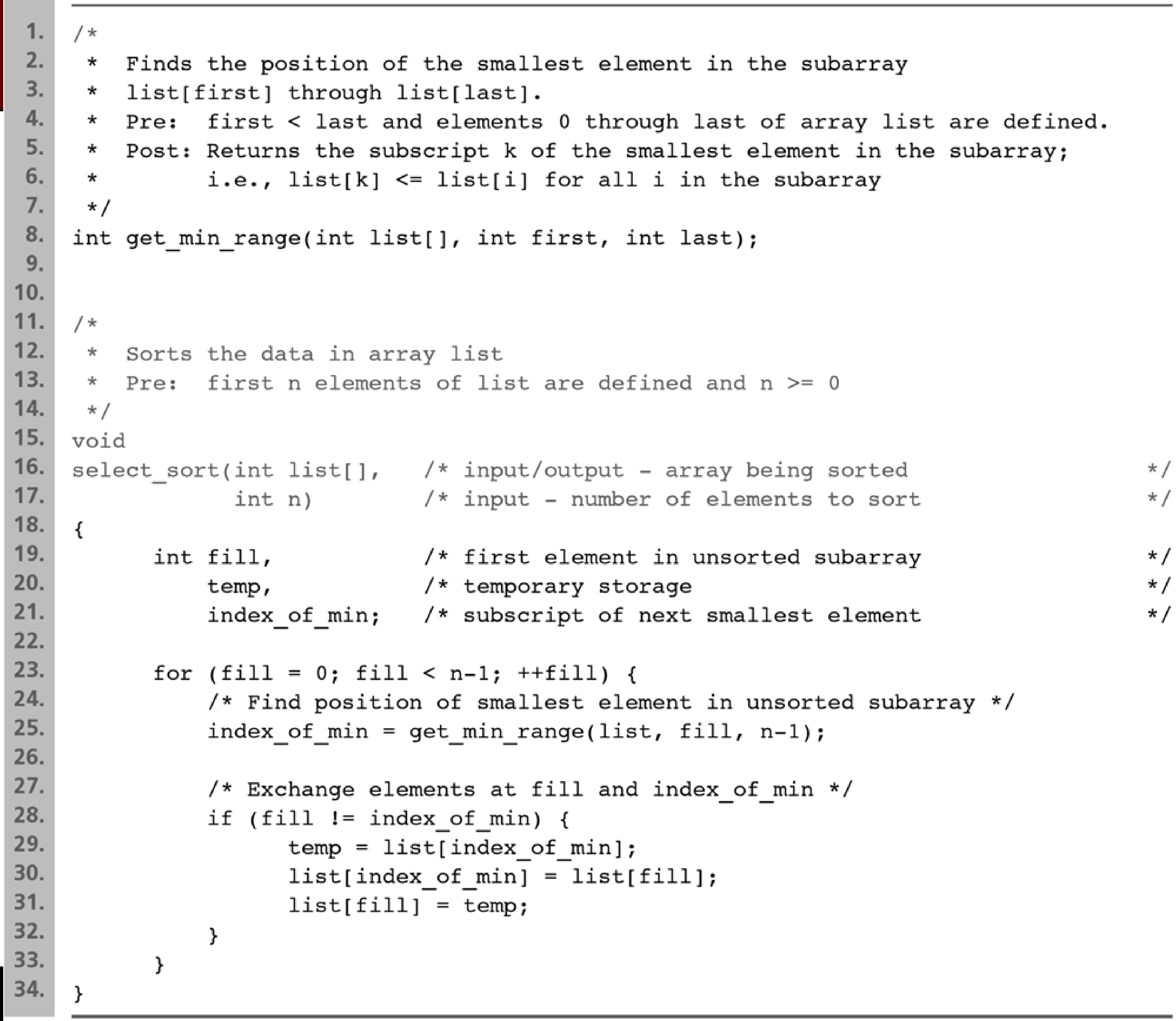
[0]	[1]	[2]	[3]
16	45	83	74

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

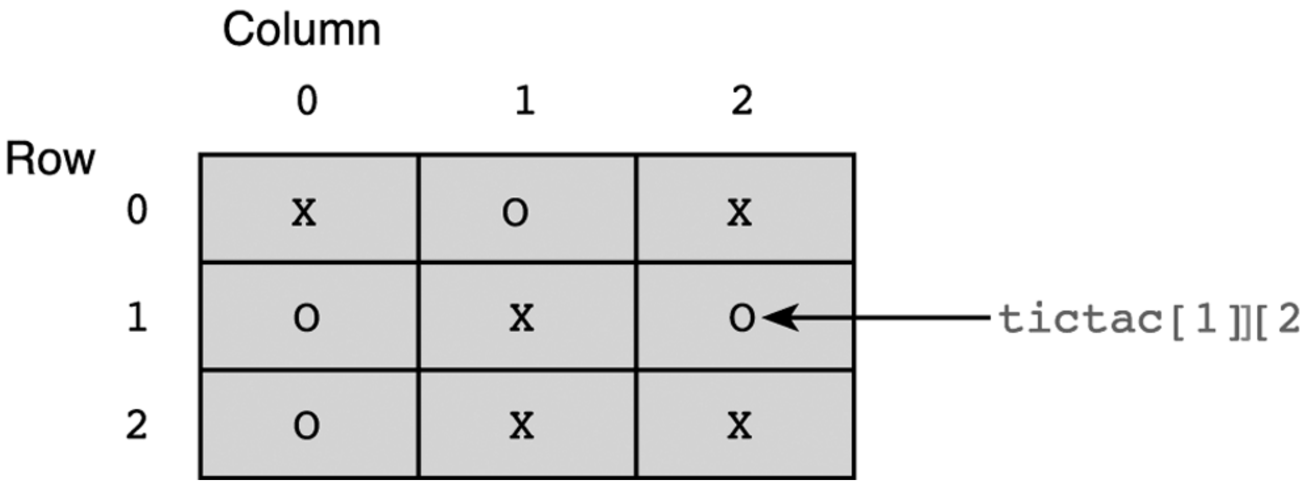


```c
/*
 *  Finds the position of the smallest element in the subarray
 *  list[first] through list[last].
 *  Pre:  first < last and elements 0 through last of array list are defined.
 *  Post: Returns the subscript k of the smallest element in the subarray;
 *        i.e., list[k] <= list[i] for all i in the subarray
 */
int get_min_range(int list[], int first, int last);


/*
 *  Sorts the data in array list
 *  Pre:  first n elements of list are defined and n >= 0
 */
void
select_sort(int list[],   /* input/output - array being sorted            */
            int n)        /* input - number of elements to sort           */
{
      int fill,           /* first element in unsorted subarray           */
          temp,           /* temporary storage                            */
          index_of_min;   /* subscript of next smallest element           */

      for (fill = 0; fill < n-1; ++fill) {
          /* Find position of smallest element in unsorted subarray */
          index_of_min = get_min_range(list, fill, n-1);

          /* Exchange elements at fill and index_of_min */
          if (fill != index_of_min) {
                temp = list[index_of_min];
                list[index_of_min] = list[fill];
                list[fill] = temp;
          }
      }
}
```

Multidimensional Arrays

- Array with two or more dimensions
  - – Tables of data
  - – Matrices
  - – Tic-tac-toe board

    char tictac[3][3];



Column

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

Multidimensional Arrays

- Syntax:

```
element-type aname[size1][size2]…[sizen];
```

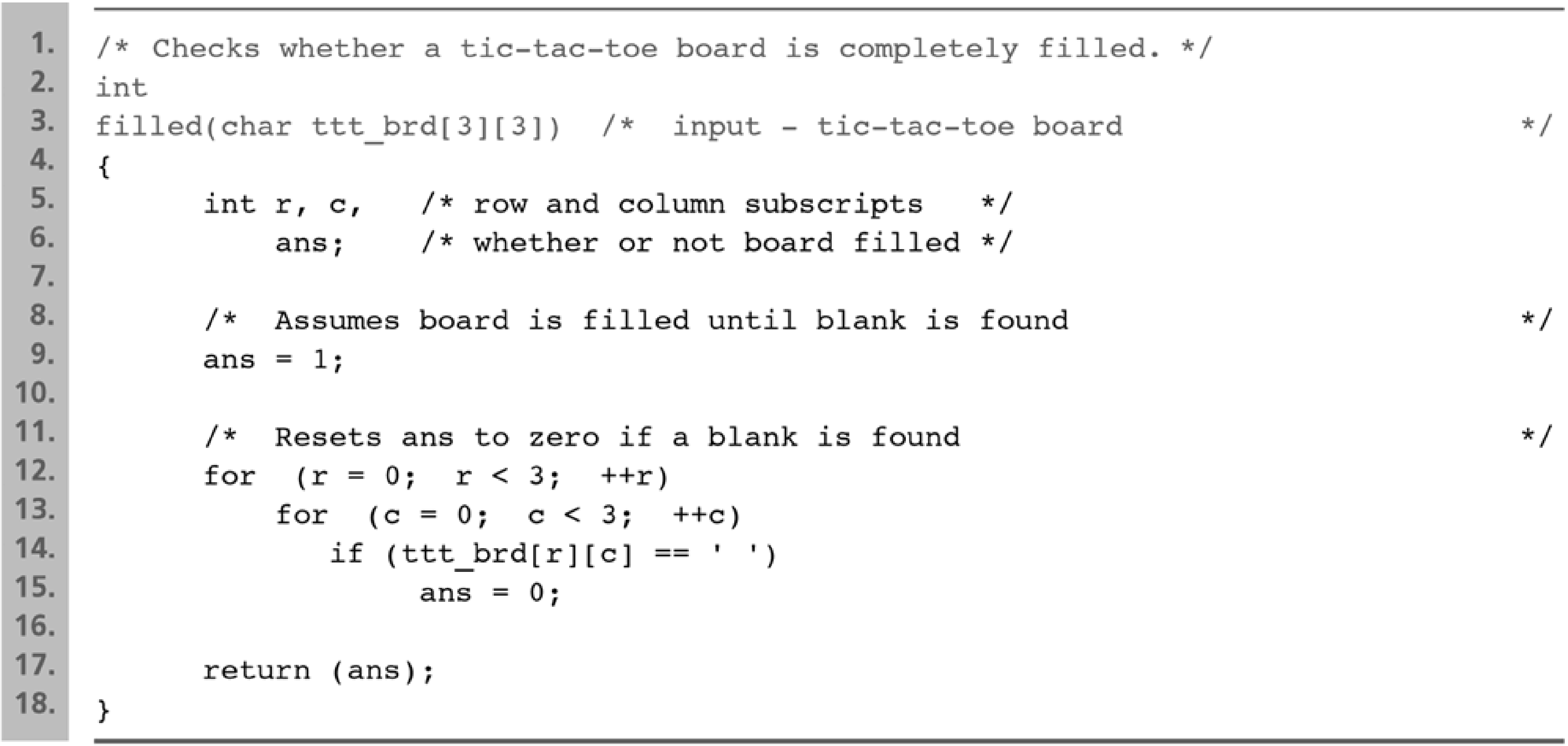
Parameter to a function

```
element-type aname[][size2]…[sizen]
```

Ex:

```
double table[NROWS][NCOLS];
int tt[7][5][6];

void process_matix(double table[][NCOLS], int nrows);
void process_t(int tt[][5][6], int nrows);
```

Check Whether Tic-tac-toe Board Is Filled



```c
/* 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);
}
```

Initialization of Multidimensional Arrays

- Initialize like one dimensional arrays
  - Use group of values as rows

EX:

```
char tictac[3][3] = {{‘ ’,’ ’,’ ’},{‘ ’,’ ’,’ ’},{‘ ’,’ ’,’ ’}};
```

Arrays with Several Dimentions

- Three dimensional array for enrollment data

  int enroll[MAXCRS][5][4];

courses:

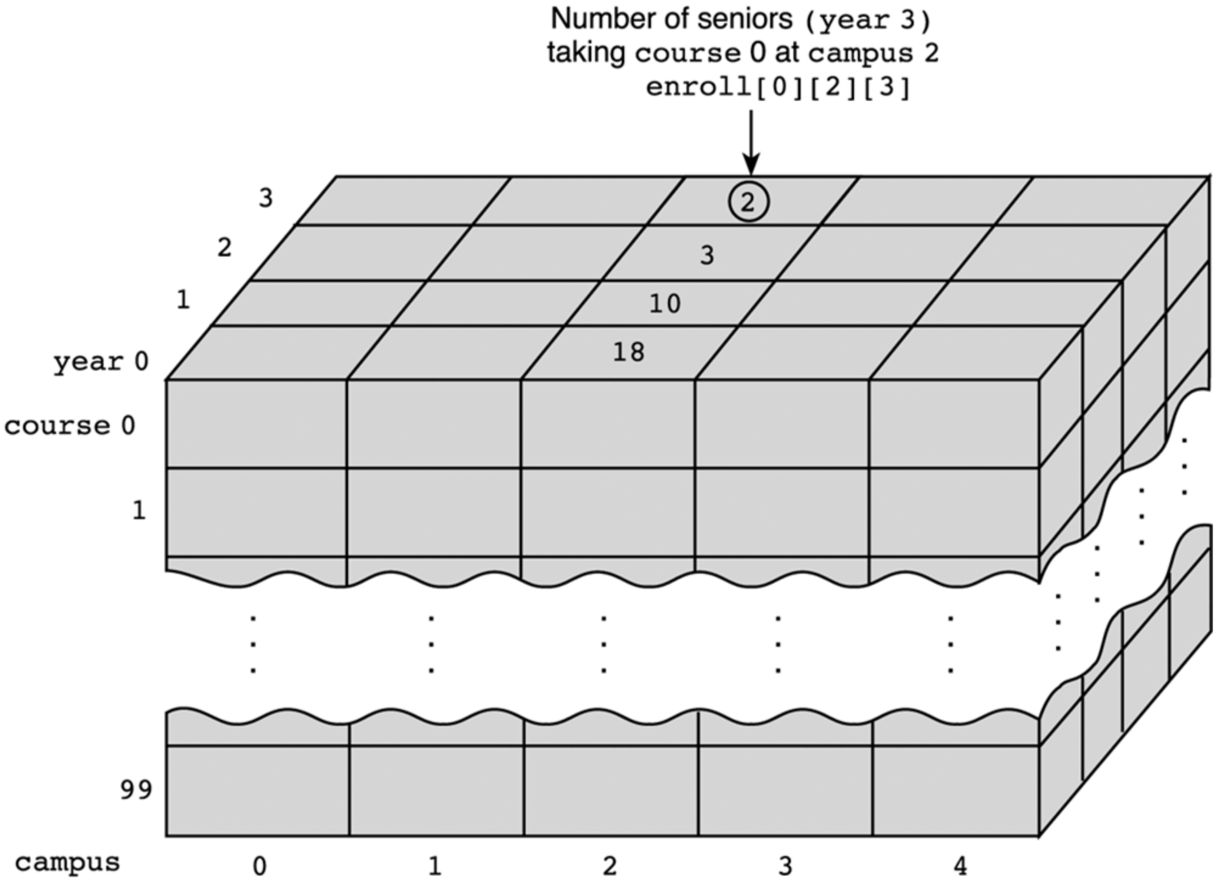
0 to MAXCRS-1

campuses:

0 to 4

years:

0 to 3

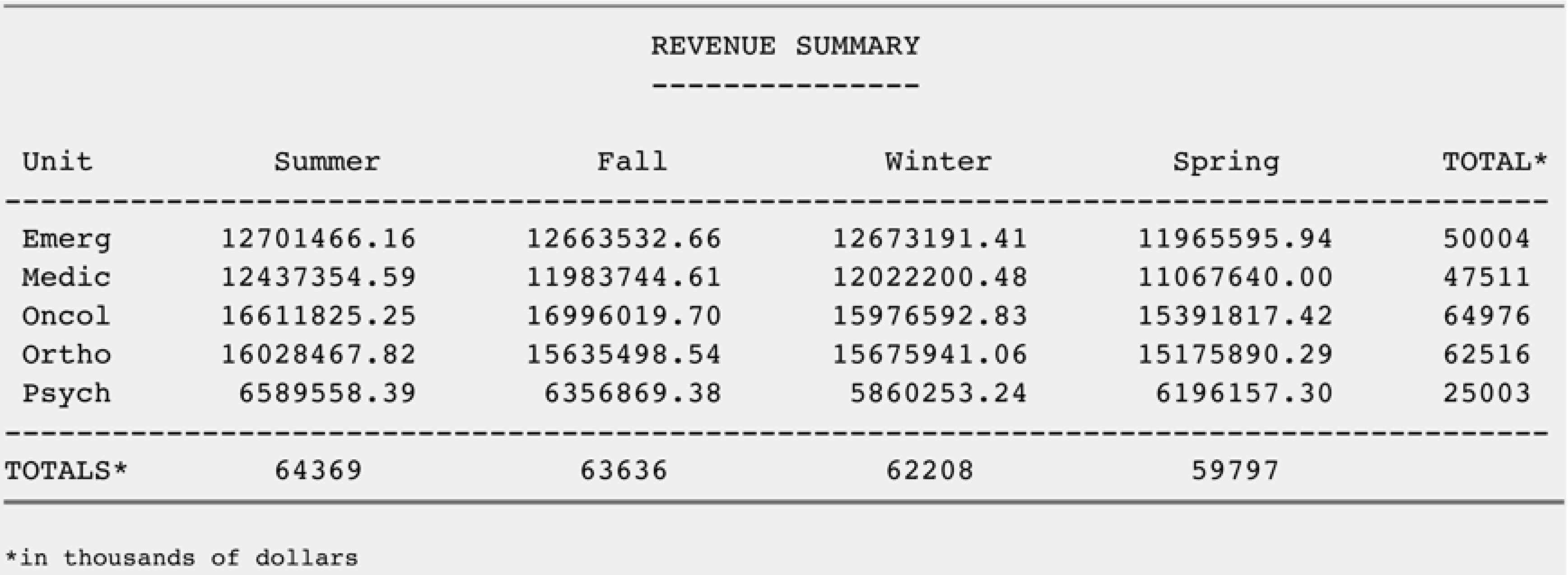


Three-Dimensional Array enroll

- Find and display the total number of students in each course
- Find and display the number of students at each campus

Case Study: Hospital Revenue

- Track revenue by unit and by quarter
  - Input: revenue transactions (in a file)
    - Unit number, quarter, revenue amount
  - Output: a table as following



```
                                REVENUE SUMMARY
                                ---------------

 Unit          Summer            Fall             Winter           Spring         TOTAL*
-------------------------------------------------------------------------------------------
 Emerg      12701466.16      12663532.66      12673191.41      11965595.94        50004
 Medic      12437354.59      11983744.61      12022200.48      11067640.00        47511
 Oncol      16611825.25      16996019.70      15976592.83      15391817.42        64976
 Ortho      16028467.82      15635498.54      15675941.06      15175890.29        62516
 Psych       6589558.39       6356869.38       5860253.24       6196157.30        25003
-------------------------------------------------------------------------------------------
TOTALS*        64369            63636            62208            59797

*in thousands of dollars
```

Case Study: Hospital Revenue

- New types

quarter_t	{fall, winter, spring, summer}
unit_t	{emerg, medic, oncol, ortho, psych}

- Problem constants

NUM_UNITS	5
NUM_QUARTERS	4

- Problem inputs

  Transaction file

  double revenue[NUM_UNITS][NUM_QUARTERS]

- Problem outputs

  double unit_totals[NUM_UNITS]

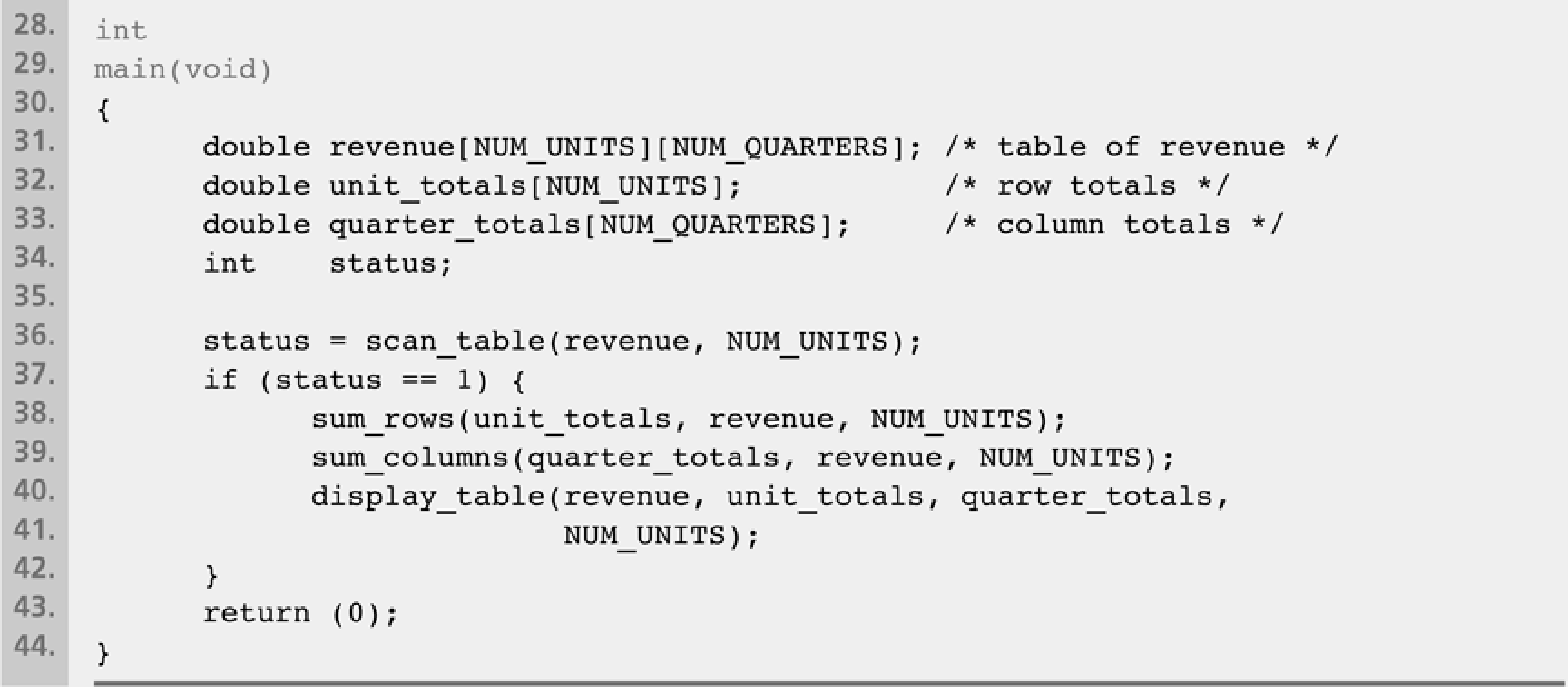
  double quarter_totals[NUM_QUARTERS]

Case Study: Hospital Revenue

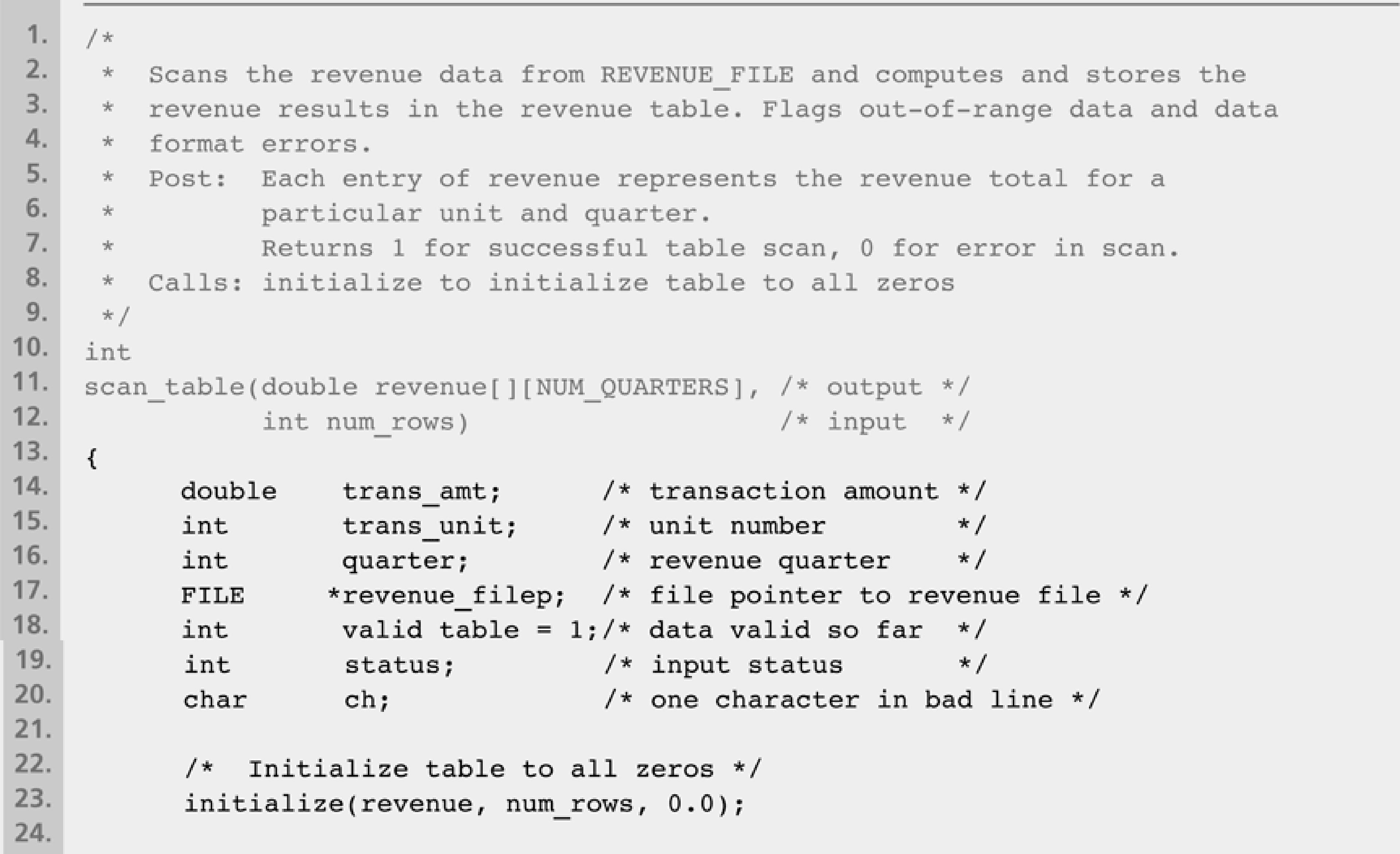
Algorithm:

1. Scan revenue data, posting by unit and quarter, returning a value to show successor failure of the data scan
2. It the data scan proceeded without error
    3. compute unit totals
    4. compute quarterly totals
    5. Display revenue table and row and column sums

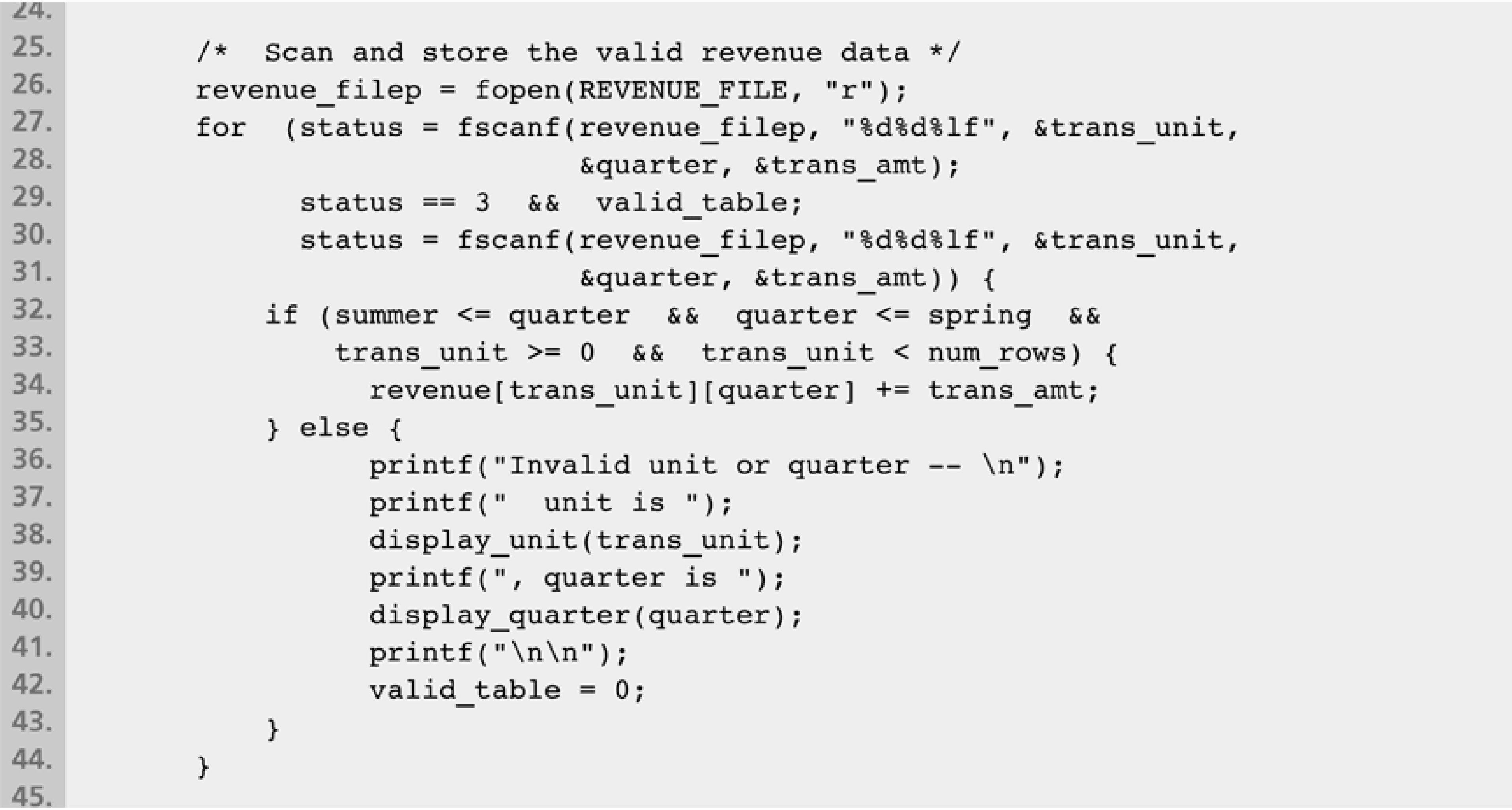
```c
/*
 *  Scans revenue figures for one year and stores them in a table organized
 *  by unit and quarter. Displays the table and the annual totals for each
 *  unit and the revenue totals for each quarter
 */

#include <stdio.h>

#define REVENUE_FILE "revenue.txt"   /* name of revenue data file    */
#define NUM_UNITS    5
#define NUM_QUARTERS 4

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

typedef enum
      {emerg, medic, oncol, ortho, psych}
unit_t;

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

```



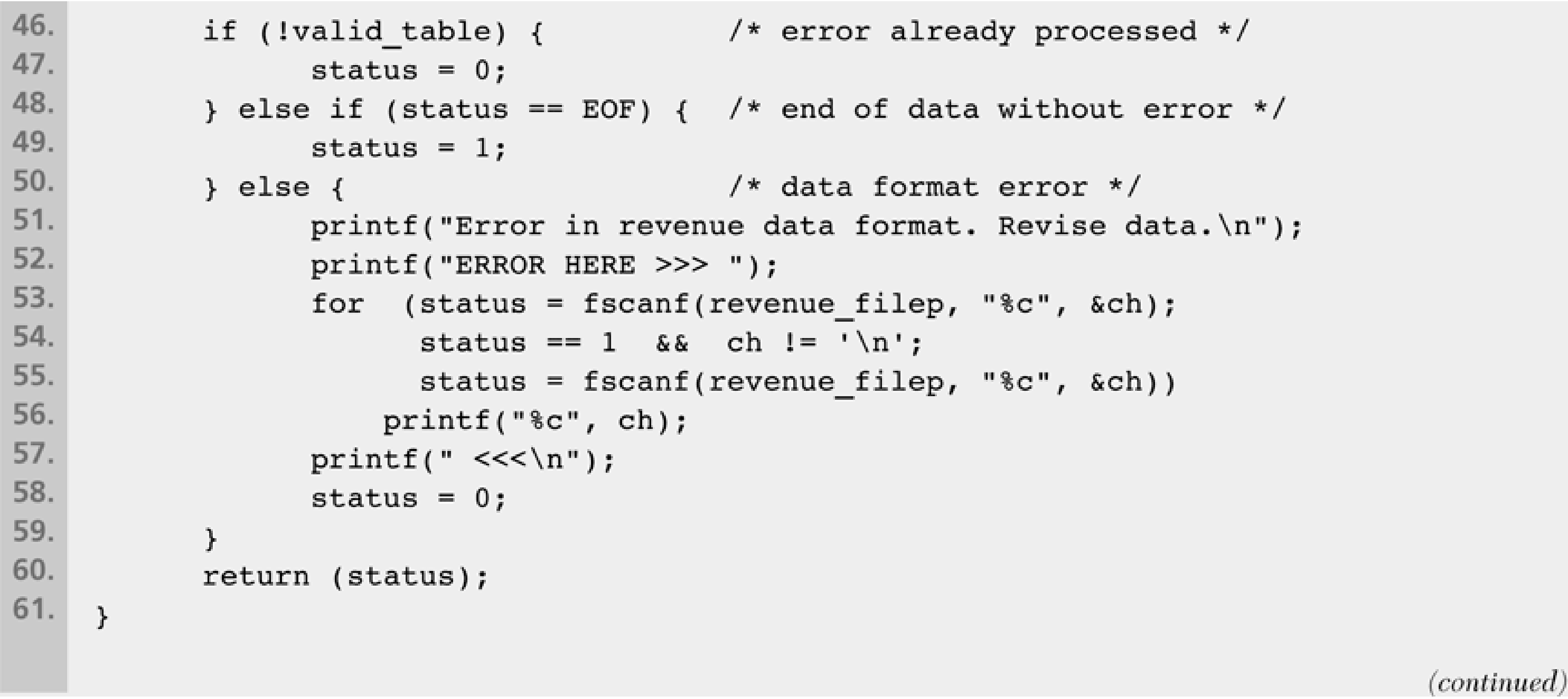
```
int
main(void)
{
      double revenue[NUM_UNITS][NUM_QUARTERS]; /* table of revenue */
      double unit_totals[NUM_UNITS];           /* row totals */
      double quarter_totals[NUM_QUARTERS];     /* column totals */
      int    status;

      status = scan_table(revenue, NUM_UNITS);
      if (status == 1) {
            sum_rows(unit_totals, revenue, NUM_UNITS);
            sum_columns(quarter_totals, revenue, NUM_UNITS);
            display_table(revenue, unit_totals, quarter_totals,
                          NUM_UNITS);
      }
      return (0);
}
```



```
/*
 *  Scans the revenue data from REVENUE_FILE and computes and stores the
 *  revenue results in the revenue table. Flags out-of-range data and data
 *  format errors.
 *  Post:  Each entry of revenue represents the revenue total for a
 *         particular unit 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 revenue[][NUM_QUARTERS], /* output */
           int num_rows)                   /* input  */
{
      double    trans_amt;       /* transaction amount */
      int       trans_unit;      /* unit number        */
      int       quarter;         /* revenue quarter    */
      FILE     *revenue_filep;   /* file pointer to revenue file */
      int       valid table = 1;/* data valid so far  */
      int       status;          /* input status       */
      char      ch;              /* one character in bad line */

      /*  Initialize table to all zeros */
      initialize(revenue, num_rows, 0.0);
```

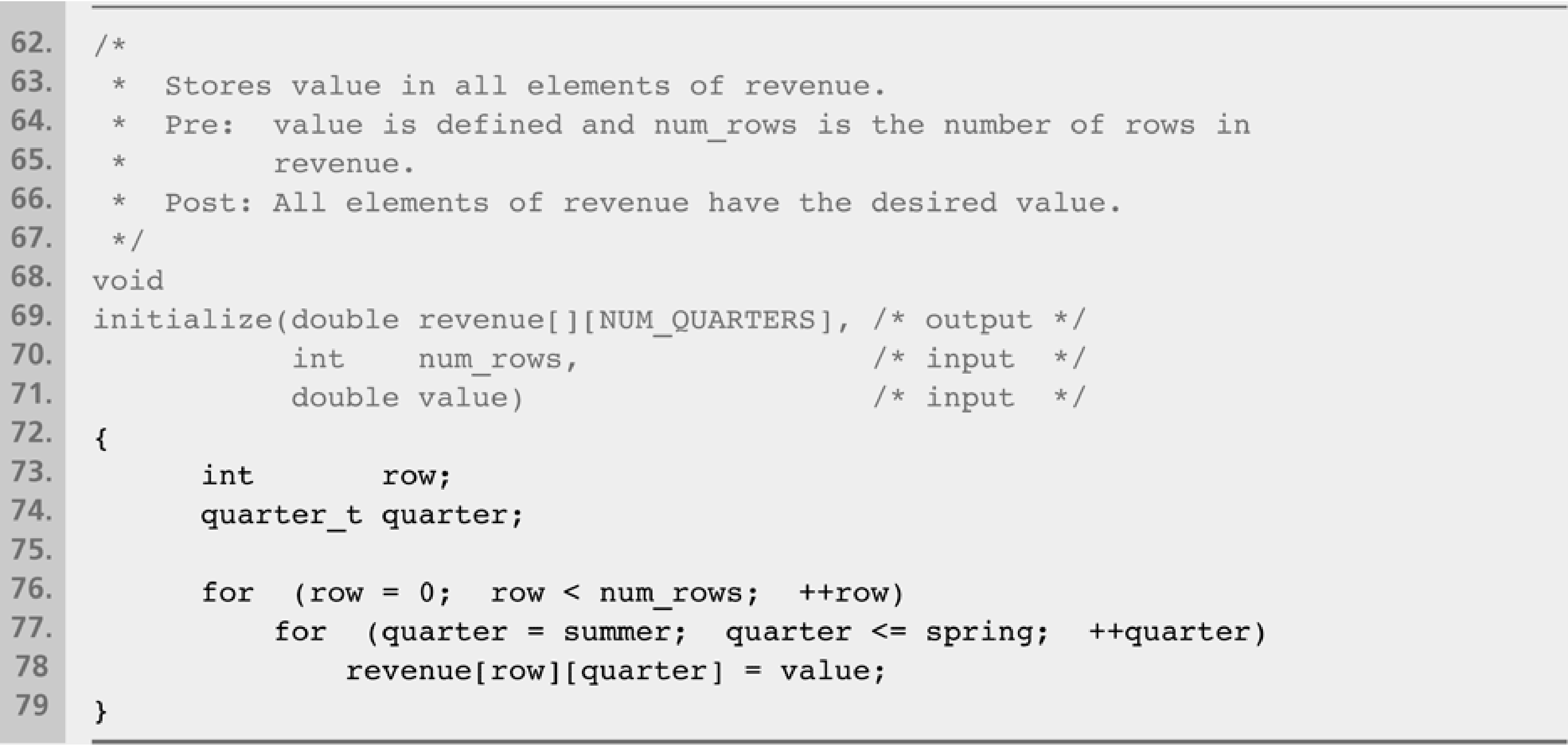


```
/*  Scan and store the valid revenue data */
revenue_filep = fopen(REVENUE_FILE, "r");
for  (status = fscanf(revenue_filep, "%d%d%lf", &trans_unit,
                      &quarter, &trans_amt);
      status == 3  &&  valid_table;
      status = fscanf(revenue_filep, "%d%d%lf", &trans_unit,
                      &quarter, &trans_amt)) {
    if (summer <= quarter  &&  quarter <= spring  &&
        trans_unit >= 0  &&  trans_unit < num_rows) {
          revenue[trans_unit][quarter] += trans_amt;
    } else {
          printf("Invalid unit or quarter -- \n");
          printf("  unit is ");
          display_unit(trans_unit);
          printf(", quarter is ");
          display_quarter(quarter);
          printf("\n\n");
          valid_table = 0;
    }
}
```

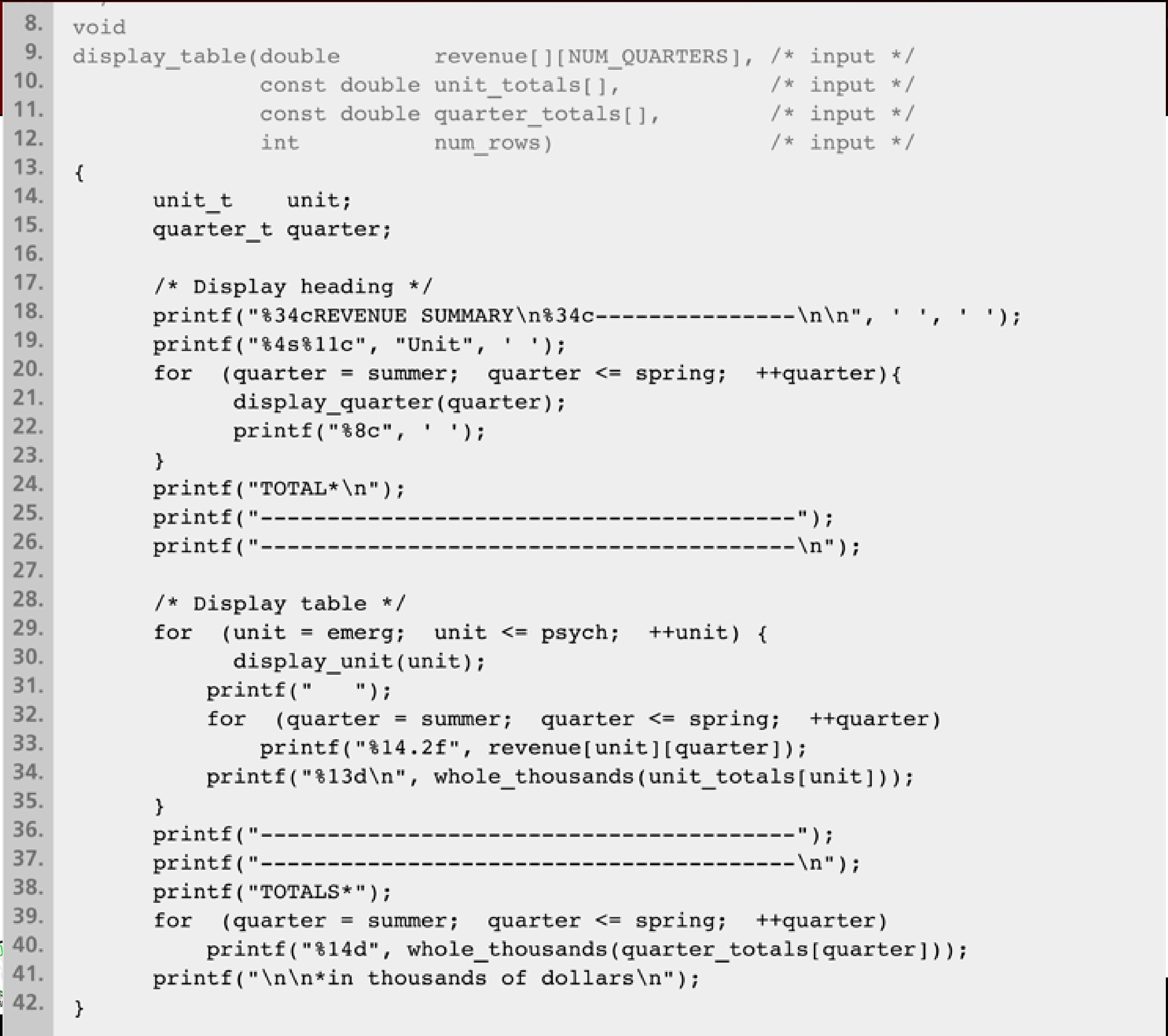


```
      if (!valid_table) {          /* error already processed */
            status = 0;
      } else if (status == EOF) {  /* end of data without error */
            status = 1;
      } else {                     /* data format error */
            printf("Error in revenue data format. Revise data.\n");
            printf("ERROR HERE >>> ");
            for  (status = fscanf(revenue_filep, "%c", &ch);
                  status == 1  &&  ch != '\n';
                  status = fscanf(revenue_filep, "%c", &ch))
                printf("%c", ch);
            printf(" <<<\n");
            status = 0;
      }
      return (status);
}
```

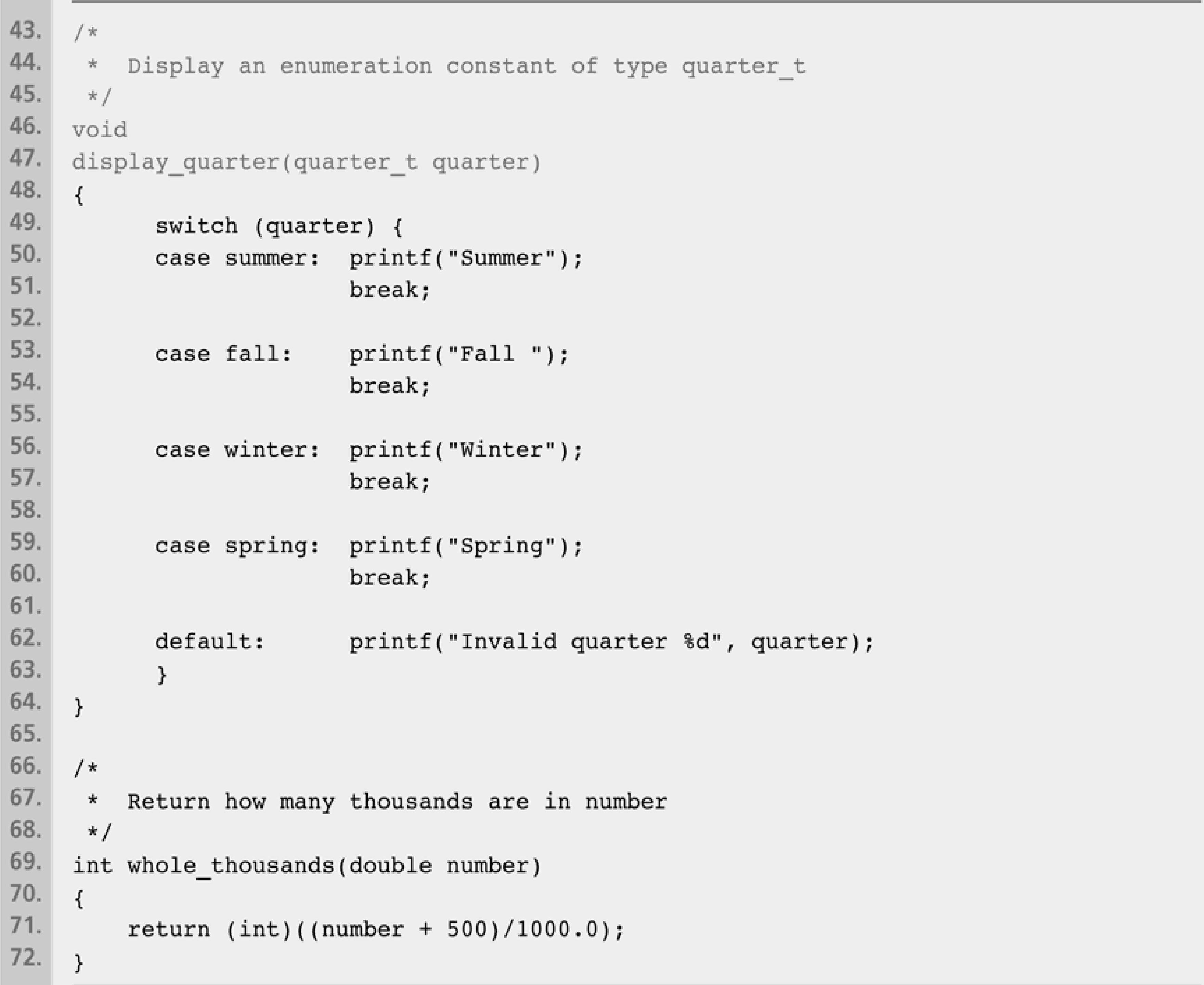
(continued)



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

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



```c
void
display_table(double       revenue[][NUM_QUARTERS], /* input */
              const double unit_totals[],           /* input */
              const double quarter_totals[],        /* input */
              int          num_rows)                /* input */
{
      unit_t    unit;
      quarter_t quarter;

      /* Display heading */
      printf("%34cREVENUE SUMMARY\n%34c---------------\n\n", ' ', ' ');
      printf("%4s%11c", "Unit", ' ');
      for  (quarter = summer;  quarter <= spring;  ++quarter){
            display_quarter(quarter);
            printf("%8c", ' ');
      }
      printf("TOTAL*\n");
      printf("----------------------------------------");
      printf("----------------------------------------\n");

      /* Display table */
      for  (unit = emerg;  unit <= psych;  ++unit) {
            display_unit(unit);
          printf("   ");
          for  (quarter = summer;  quarter <= spring;  ++quarter)
              printf("%14.2f", revenue[unit][quarter]);
          printf("%13d\n", whole_thousands(unit_totals[unit]));
      }
      printf("----------------------------------------");
      printf("----------------------------------------\n");
      printf("TOTALS*");
      for  (quarter = summer;  quarter <= spring;  ++quarter)
          printf("%14d", whole_thousands(quarter_totals[quarter]));
      printf("\n\n*in thousands of dollars\n");
}
```



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

      case fall:    printf("Fall ");
                    break;

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

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

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

/*
 *  Return how many thousands are in number
 */
int whole_thousands(double number)
{
    return (int)((number + 500)/1000.0);
}
```