Chapter 9

Two Dimensional Arrays

In Chapter 7 we have seen that C provides a compound data structure for storing a list of related data. For some applications, however, such a structure may not be sufficient to capture the organization of the data. For example, in our payroll task, we have several pieces of information (hours worked, rate of pay, and regular and over time pay) for a list of employees. We have organized such data as individual arrays, one for each "column", to form the payroll data base; but conceptually, this information is all related. In this chapter we will introduce a data structure which allows us to group together all such information into a single structure — a two dimensional array. For a data base application, we can think of this 2D organization as an array of arrays. As another example of where such a structure is convenient, consider an array of names. We have seen that we can store a name in a *string*, which is an array of characters. Then an array of strings is also an array of arrays, or a two dimensional array.

In this chapter we will discuss how we can declare two dimensional arrays, and use them in applications. We will see how we can access the data in such a structure using indices and pointers, and see how this concept can be extended to multi-dimensional arrays. We will present examples of 2 dimensional arrays for data base applications, string sorting and searching, and solutions to systems of simultaneous linear algebraic equations, useful in scientific, engineering, and other applications, e.g. electronic circuit analysis, economic analysis, structural analysis, etc. The one restriction in the use of this data type is that all of the data stored in the structure must be of the same type. (We will see how we can remove this restriction in the next chapter).

9.1 Two Dimensional Arrays

Our first task is to consider a number of exams for a class of students. The score for each exam is to be weighted differently to compute the final score and grade. For example, the first exam may contribute 30% of the final score, the second may contribute 30%, and the third contribute 40%. We must compute a weighted average of the scores for each student. The sum of the weights for all the exams must add up to 1, i.e. 100%. Here is our task:

WTDAVG: Read the exam scores from a file for several exams for a class of students. Read the percent weight for each of the exams. Compute the weighted average score for each student. Also, compute the averages of the scores for each exam and for the weighted average scores.

We can think of the exam scores and the weighted average score for a single student as a data record and and represent it as a row of information. The data records for a number of students,

then, is a table of such rows. Here is our conceptual view of this collection of data:

	exam1	exam2	exam3	weighted avg
$student_1$	50	70	75	??
$student_2$	90	95	100	??
$student_3$	89	87	92	??
⋮				
$student_n$	90	90	91	??

Let us assume that all scores will be stored as integers; even the weighted averages, which will be computed as float, will be rounded off and stored as integers. To store this information in a data structure, we can store each student's data record, a row containing three exam scores and the weighted average score, in a one dimensional array of integers. The entire table, then, is an array of these one dimensional arrays — i.e. a two dimensional array. With this data structure, we can access a record for an individual student by accessing the corresponding row. We can also access the score for one of the exams or for the weighted average for all students by accessing each column. The only restriction to using this data structure is that all items in an array must be of the same data type. If the student id is an integer, we can even include a column for the id numbers.

Suppose we need to represent id numbers, scores in 3 exams, and weighted average of scores for 10 students; we need an array of ten data records, one for each student. Each data record must be an array of five elements, one for each exam score, one for the weighted average score, and one for the student id number. Then, we need an array, `scores[10]` that has ten elements; each element of this array is, itself, an array of 5 integer elements. Here is the declaration of an array of integer arrays:

```
int scores[10][5];
```

The first range says the array has ten elements: `scores[0]`, `scores[1]`, ... `scores[9]`. The second range says that each of these ten arrays is an array of five elements. For example, `scores[0]` has five elements: `scores[0][0]`, `scores[0][1]`, ... `scores[0][4]`. Similarly, any other element may be referenced by specifying two appropriate indices, `scores[i][j]`. The first array index references the i^{th} one dimensional array, `scores[i]`; the second array index references the j^{th} element in the one dimensional array, `scores[i][j]`.

A two dimensional array lends itself to a visual display in rows and columns. The first index represents a row, and the second index represents a column. A visual display of the array, `scores[10][5]`, is shown in Figure 9.1. There are ten rows, (0-9), and five columns (0-4). An element is accessed by row and column index. For example, `scores[2][3]` references an integer element at row index 2 and column index 3.

We will see in the next section that, as with one dimensional arrays, elements of a two dimensional array may be accessed indirectly using pointers. There, we will see the connection between two dimensional arrays and pointers. For now, we will use array indexing as described above and remember that arrays are always accessed indirectly. Also, just as with one dimensional arrays, a 2D array name can be used in function calls, and the called function accesses the array indirectly.

We can now easily set down the algorithm for our task:

	col. 0	col. 1	col. 2	col. 3	col. 4
row 0	scores[0][0]	scores[0][1]	scores[0][2]	scores[0][3]	scores[0][4]
row 1	scores[1][0]	scores[1][1]	scores[1][2]	scores[1][3]	scores[1][4]
row 2	scores[2][0]	scores[2][1]	scores[2][2]	scores[2][3]	scores[2][4]
row 3	scores[3][0]	scores[3][1]	scores[3][2]	scores[3][3]	scores[3][4]
row 4	scores[4][0]	scores[4][1]	scores[4][2]	scores[4][3]	scores[4][4]
row 5	scores[5][0]	scores[5][1]	scores[5][2]	scores[5][3]	scores[5][4]
row 6	scores[6][0]	scores[6][1]	scores[6][2]	scores[6][3]	scores[6][4]
row 7	scores[7][0]	scores[7][1]	scores[7][2]	scores[7][3]	scores[7][4]
row 8	scores[8][0]	scores[8][1]	scores[8][2]	scores[8][3]	scores[8][4]
row 9	scores[9][0]	scores[9][1]	scores[9][2]	scores[9][3]	scores[9][4]

Figure 9.1: Rows and Columns in A Two Dimensional Array

```
read the number of exams into no_of_exams
get weights for each of the exams

read exam scores and id number for each student
     into a two dimensional array

for each student, compute weighted average of scores in the exams
compute average score for each of the exams and
     for the weighted average
print results
```

We can easily write the top level program driver using functions to do the work of reading scores, getting the weights, computing the weighted averages, printing scores, averaging each set of scores, and printing the averages. The driver is shown in Figure 9.2.

We have declared an array, `scores[][]`, with `MAX` rows and `COLS` columns, where these macro values are large enough to accommodate the expected data. We have used several functions, which we will soon write and include in the same program file. Their prototypes as well as those of other functions are declared at the head of the file. In the driver, `getwts()` reads the weights for the exams into an array, `wts[]`, returning the number of exams. The function, `read_scores()`, reads the data records into the two dimensional array, `scores[][]`, and returns the number of data records. The function, `wtd_avg()`, computes the weighted averages of all exam scores, and `avg_scores()` computes an average of each exam score column as well as that of the weighted average column. Finally, `print_scores()` and `print_avgs()` print the results including the input data, the weighted averages, and the averages of the exams.

Let us first write `getwts()`. It merely reads the weight for each of the exams as shown in Figure 9.3. The function prompts the user for the number of exam scores, and reads the corresponding number of float values into the `wts[]` array. Notice that the loop index, `i` begins with the value 1. This is because the element `wts[0]`, corresponding to the student id column, does not have a weight and should be ignored. After the weights have been read, we flush the keyboard buffer of any remaining white space so that any kind of data (including character data) can be read from

```
/*   File: wtdavg.c
     Other Source Files: avg.c
     Header Files: avg.h
     This program computes weighted averages for a set of exam scores for
     several individuals. The program reads scores from a file, computes
     weighted averages for each individual, prints out a table of scores,
     and prints averages for each of the exams and for the weighted average.
*/

#include <stdio.h>

#define MAX 20
#define COLS 5
int getwts(float wts[]);
FILE *openinfile(void);
int read_scores(int ex[][COLS], int lim, int nexs);
void wtd_avg(int ex[][COLS], int lim, int nexs, float wts[]);
void avg_scores(int ex[][COLS], int avg[], int lim, int nexs);
void print_scores(int ex[][COLS], int lim, int nexs);
void print_avgs(int avg[], int nexs);

main()
{    int no_of_stds, no_of_exams;
     int avg[COLS];
     int scores[MAX][COLS];
     float wts[COLS];

     printf("***Weighted Average of Scores***\n\n");
     no_of_exams = getwts(wts);
     no_of_stds = read_scores(scores, MAX, no_of_exams);

     wtd_avg(scores, no_of_stds, no_of_exams, wts);
     print_scores(scores, no_of_stds, no_of_exams);
     avg_scores(scores, avg, no_of_stds, no_of_exams);
     print_avgs(avg, no_of_exams);
}
```

Figure 9.2: Driver for Student Scores Program

```
/* File: wtdavg.c - continued */
/* Gets the number of exams and weights for the exams; flushes
   the input buffer and returns the number of exams.
*/
int getwts(float wts[])
{    int i, n;

     printf("Number of exams: ");
     scanf("%d", &n);

     for (i = 1; i <= n; i++) {
     printf("Percent Weight for Exam %d: ", i);
     scanf("%f", &wts[i]);
     }

     while (getchar() != '\n')
          ;
     return n;
}
```

Figure 9.3: Code for `getwts()`

the input. The function returns the number of exams, `n`.

We will assume that the data for the student scores is stored in a file in the format of one line per student, with each line containing the student id followed by the exam scores. To read this data into a two dimensional array, we must first open the input file. This is done by the function `openfile()` shown in Figure 9.4, which prompts for the file name and tries to open the file. If the file opens successfully, the file pointer is returned. Otherwise, the function prints a message and asks the user to retype the file name. The user may quit at any time by typing a newline or end of file. If an end of file is typed or the typed string is empty, the program is terminated. Once the input file is opened, we read data items into the array, filling in the elements one row (student) at a time. We use two index variables, `row` and `col`, varying the `row` to access each row in turn; and, within each row, we vary `col` to access elements of each column in turn. We will need a doubly nested loop to read the data in this manner. The function is given the number of students, the variable `stds`, and the number of exams, `nexs`. We will use column 0 to store the student id numbers and the next `nexs` columns to store the scores. Thus, in each row, we read `nexs+1` data values into the array. This is done by the function, `read_scores()`, also shown in Figure 9.4. The input file is first opened using `openfile()`, and the data records are read into the array called `ex[][]` within the function. The function returns the number of records read either when `EOF` is reached or when the array is filled. Each integer data item is read from a file, `fp`, into a temporary variable, `n`. This value is then assigned to the appropriate element, `ex[row][col]`. When all data has been read, the input file is closed and the number of records read is returned.

Notice in `main()` in Figure 9.2, we pass the 2D array to `read_scores()` just as we did for one dimensional arrays, passing the array name. As we shall see in the next section, the array

```
/* File: wtdavg.c - continued */
/* Opens the input file and returns the file pointer. */
FILE *openinfile(void)
{    FILE *fp;
     char infile[25];

     printf("Input File, RETURN to quit: ");
     while (gets(infile)) {
          if (!*infile) exit(0);   /* empty string, exit */

          fp = fopen(infile, "r");

          if (!fp) {          /* no such file, continue */
               printf("Unable to open input file - retype\n");
               continue;
          }

          else return fp;     /* file opened, return fp */
     }
     exit(0);                 /* end of file typed, exit */
}

/* Opens the input file and reads scores for nexs exams; returns
   the number of individual student records.
*/
int read_scores(int ex[][COLS], int stds, int nexs)
{    int row, col, n, x;
     FILE * fp;

     fp = openinfile();
     for (row = 0; row < stds; row++)
          for (col = 0; col <= nexs; col++) {
               x = fscanf(fp, "%d", &n);

               if (x == EOF) {
                    fclose(fp);
                    return row;
               }

               ex[row][col] = n;
          }
     fclose(fp);
     return row;
}
```

Figure 9.4: Code for `openfile()` and `read_scores()`

name is a pointer that allows indirect access to the array elements. The two dimensional array as as argument must be declared in the function definition as a formal parameter. In Figure 9.4, we have declared it as `ex[][COL]` with two sets of square brackets to indicate that it points to a two dimensional array. In our declaration, we **must** include the number of columns in the array because this specifies the size of each row. Recall, the two dimensional array is an array of rows. Once the compiler knows the size of a row in the array, it is able to correctly determine the beginning of each row.

The next function called in `main()` computes the weighted average for each row. The weighted average for one record is just the sum of each of the exam score times the actual weight of that exam. If the scores are in the array, `ex[][]`, then the following code will compute a weighted average for a single row, `row`:

```
wtdavg = 0.0;
for (col = 1; col <= nexs; col++)
     wtdavg += ex[row][col] * wts[col] / 100.0;
```

We convert the percent weight to the actual weight multiply by the score, and accumulate it in the sum, `wtdavg` yielding a float value. The `wtdavg` will be stored in the integer array, `ex[][]`, after rounding to a nearest integer. If we simply cast `wtdavg` to an integer, it will be truncated. To round to the nearest integer, we add 0.5 to `wtdavg` and then cast it to integer:

```
ex[row][nexs + 1] = (int) (0.5 + wtdavg);
```

The weighted average is stored into the column of the array after the last exam score. The entire function is shown in Figure 9.5

Computing average of each of the exams and the weighted average is simple. We just sum each of the columns and divide by the number of items in the column, and is also shown in Figure 9.5. For each exam and for the weighted average column, the scores are added and divided by `lim`, the number of rows in the array, using floating point computation. The result is rounded to the nearest integer and stored in the array, `avg[]`. Figure 9.6 shows the final two functions for printing the results.

Running the program with data file,`wtdin.dat` as follows:

```
3    70   76
52   92   80
53   95   56
54   48   52
55   98   95
57   100  95
61   100  65
62   95   76
63   86   65
70   100  90
71   73   73
75   94   79
```

produces the following sample session:

```
***Weighted Average of Scores***
```

```
/* File: wtdavg.c - continued */
/* Computes the weighted average of the exam scores in ex[][] for
   lim individuals, nexs number of exams, and weights given by wts[].
*/
void wtd_avg(int ex[][COLS], int lim, int nexs, float wts[])
{    int i, j;
     float wtdavg;

     for (i = 0; i < lim; i++) {
          wtdavg = 0.0;

          for (j = 1; j <= nexs; j++)
               wtdavg += ex[i][j] * wts[j] / 100.0;

          ex[i][nexs + 1] = (int) (wtdavg + 0.5);
     }
}

/* Averages exam and weighted average scores. */
void avg_scores(int ex[][COLS], int avg[], int lim, int nexs)
{    int i, j;

     for (j = 1; j <= nexs + 1; j++) {
          avg[j] = 0;

          for (i = 0; i < lim; i++)
               avg[j] += ex[i][j];

          avg[j] = (int) (0.5 + (float) avg[j] / (float) lim);
     }
}
```

Figure 9.5: Code for `wtd_avg()` and `avg_scores()`

```
/* File: wtdavg.c - continued */
/* Prints the scores for exams and the weighted average. */
void print_scores(int ex[][COLS], int lim, int nexs)
{    int i, j;

     printf("ID #\t");
     for (j = 1; j <= nexs; j++)
          printf("Ex%d\t", j);          /* print the headings */

     printf("WtdAvg\n");
     for (i = 0; i < lim; i++) {        /* print the scores and wtd avg */

          for (j = 0; j <= nexs + 1; j++)
               printf("%d\t", ex[i][j]);

          printf("\n");
     }
}

/* Prints the averages of exams and the average of the weighted average
   of exams.
*/
void print_avgs(int avg[], int nexs)
{    int i;

     for (i = 1; i <= nexs; i++)
          printf("Average for Exam %d = %d\n", i, avg[i]);
     printf("Average of the weighted average = %d\n", avg[nexs + 1]);
}
```

Figure 9.6: Code for `print_scores()` and `print_avgs()`

```
Number of exams:  2
Percent Weight for Exam 1:  50
Percent Weight for Exam 2:  50
Input File, RETURN to quit:  wtdin.dat
 ID #  Ex1  Ex2  WtdAvg
 3     70   76   73
 52    92   80   86
 53    95   56   76
 54    48   52   50
 55    98   95   97
 57    100  95   98
 61    100  65   83
 62    95   76   86
 63    86   65   76
 70    100  90   95
 71    73   73   73
 75    94   79   87
Average for Exam 1 = 88
Average for Exam 2 = 75
Average of the weighted average = 82
```

In this program, we have assumed that the input file contains only the data to be read, i.e. the student id numbers and exam scores. Our `read_scores()` function is written with this assumption. However, the input file might also contain some heading information such as the course name and column headings in the first few lines of the file. We can easily modify `read_scores()` to discard the first few lines of headings.

As a second example of application of two dimensional arrays, consider our previous payroll example. In this case, the data items in a pay data record are not all of the same data type. The id numbers are integers, whereas all the other items are float. Therefore, we must use an array of integers to store the id numbers, and a two dimensional float array to store the rest of the data record. The algorithm is no different from the program we developed in Chapter 7 that computed pay. The difference is that now we use a two dimensional array for all float payroll data instead of several one dimensional arrays. The id numbers are still stored in a separate one dimensional array. Since the data structures are now different, we must recode the functions to perform the tasks of getting data, calculating pay, and printing results, but still using the same algorithms.

The program driver and the header files are shown in Figure 9.7. The program declares an integer array for id numbers and a two dimensional float array for the rest of the data record. The successive columns in the two dimensional array store the hours worked, rate of pay, regular pay, overtime pay, and total pay, respectively. We have defined macros for symbolic names for these index values. As in the previous version, the program gets data, calculates pay, and prints data. The difference is in the data structures used. Functions to perform the actual tasks are shown in Figure 9.8 and 9.9 and included in the same program source file. Each function uses a two dimensional array, `payrec[][]`. The row index specifies the data record for a single id, and the column index specifies a data item in the record. The data record also contains the total pay. A sample interaction with the program, `pay2rec.c`, is shown below.

```
/*   File: pay2rec.c
     Program calculates and stores payroll data for a number of id's.
     The program uses a one dimensional array for id's, and a two
     dimensional array for the rest of the pay record. The first column
     is hours, the second is rate, the third is regular pay, the fourth
     is overtime pay, and the fifth is total pay.
*/

#include <stdio.h>
#define MAX 10
#define REG_LIMIT 40
#define OT_FACTOR 1.5
#define HRS 0
#define RATE 1
#define REG 2
#define OVER 3
#define TOT 4

int get2data(int id[], float payrec[][TOT + 1], int lim);
void calc2pay(float payrec[][TOT + 1], int n);
void print2data(int id[], float payrec[][TOT + 1], int n);

main()
{    int n = 0, id[MAX];
     float payrec[MAX][TOT + 1];

     printf("***Payroll Program - Records in 2 D arrays***\n\n");
     n = get2data(id, payrec, MAX);
     calc2pay(payrec, n);
     print2data(id, payrec, n);
}
```

Figure 9.7: Driver for Payroll Program Using 2D Arrays

```
/*   File: pay2rec.c - continued */
/*   Gets id's in one array, and the rest of input data records
     in a two dimensional array.
*/
int get2data(int id[], float payrec[][TOT + 1], int lim)
{    int n = 0;
     float x;

     while (n < lim) {
          printf("ID <zero to quit>: ");
          scanf("%d", &id[n]);

          if (id[n] <= 0)
               return n;

          printf("Hours Worked: ");
          scanf("%f", &x);
          payrec[n][HRS] = x;

          printf("Rate of Pay: ");
          scanf("%f", &x);
          payrec[n][RATE] = x;
          n++;
     }
     if (n == lim) {
          printf("Table full, processing data\n");
          return n;
     }
}
```

Figure 9.8: Code for Payroll Program Functions — `get2data()`

```
/* Calculates pay for each id record in a two dimensional array. */
void calc2pay(float payrec[][TOT + 1], int n)
{    int i;

     for (i = 0; i < n; i++) {
          if (payrec[i][HRS] <= REG_LIMIT) {
               payrec[i][REG] = payrec[i][HRS] * payrec[i][RATE];
               payrec[i][OVER] = 0;
          }

          else {
               payrec[i][REG] = REG_LIMIT * payrec[i][RATE];
               payrec[i][OVER] = (payrec[i][HRS] - REG_LIMIT) *
                                  OT_FACTOR * payrec[i][RATE];
          }

          payrec[i][TOT] = payrec[i][REG] + payrec[i][OVER];
     }
}

/*   Prints a table of payroll data for all id's. Id's in one array,
     and the rest of the records in a two dim. array.
*/
void print2data(int id[], float payrec[][TOT + 1], int n)
{    int i, j;

     printf("***PAYROLL: FINAL REPORT***\n\n");
     printf("%10s%10s%10s%10s%10s%10s\n", "ID", "HRS",
               "RATE", "REG", "OVER", "TOT");

     for (i = 0; i < n; i++) {
          printf("%10d", id[i]);

          for (j = 0; j <= TOT; j++)
               printf("%10.2f", payrec[i][j]);

          printf("\n");
     }
}
```

Figure 9.9: Code for Payroll Program Functions — `calc2pay()` and `print2data()`

Sample Session:

```
***Payroll Program - Records in 2 D arrays***

ID <zero to quit>:  5
Hours Worked:  30
Rate of Pay:  10
ID <zero to quit>:  10
Hours Worked:  50
Rate of Pay:  12
ID <zero to quit>:  0
***PAYROLL: FINAL REPORT***
 ID    HRS   RATE     REG    OVER     TOT
  5  30.00  10.00  300.00    0.00  300.00
 10  50.00  12.00  480.00  180.00  660.00
```

9.2 Implementing Multi-Dimensional Arrays

In the last section we saw how we can use two dimensional arrays — how to declare them, pass them to functions, and access the date elements they contain using array indexing notation. As with one dimensional arrays, we can access elements in a 2D array using pointers as well. In order to understand how this is done, in this section we look at how multi dimensional arrays are implemented in C.

As we saw in Chapter 7, a one dimensional array is stored in a set of contiguous memory cells and the *name* of the array is associated with a pointer cell to this block of memory. In C, a two dimensional array is considered to be a one dimensional array of rows, which are, themselves, one dimensional arrays. Therefore, a two dimensional array of integers, `AA[][]`, is stored as a contiguous sequence of elements, each of which is a one dimensional array. The rows are stored in sequence, starting with row 0 and continuing until the last row is stored, i.e. `AA[0]` is stored first, then `AA[1]`, then `AA[2]`, and so on to `AA[MAX-1]`. Each of these "elements" is an array, so is stored as a contiguous block of integer cells as seen in Figure 9.10. This storage organization for two dimensional arrays is called `row major order`. The same is true for higher dimensional arrays. An n dimensional array is considered to be a one dimensional array whose elements are, themselves, arrays of dimension $n-1$. As such, in C, an array of any dimension is stored in row major order in memory.

With this storage organization in mind, let us look at what implications this has to referencing the array with pointers. Recall that an array name (without an index) represents a pointer to the first object of the array. So the name, `AA`, is a pointer to the element `AA[0]`. iBut, `AA[0]` is a one dimensional array; so, `AA[0]` points to the first object in row 0, i.e. `AA[0]` points to `AA[0][0]`. Similarly, for any k `AA[k]` points to the beginning of the kth row, i.e. `AA[k]` is the address of `AA[k][0]`. Since `AA[k]` points to `AA[k][0]`, `*AA[k]` accesses `AA[k][0]`, an object of type integer. If we add 1 to the pointer `AA[k]`, the resulting pointer will point to the next integer type element, i.e. the value of `AA[k][1]`. In general, `AA[k] + j` points to `AA[k][j]`, and `*(AA[k] + j)` accesses the value of `AA[k][j]`. This is shown in Tables 9.1 and 9.2. Each `AA[k]` points to an integer type object. When an integer is added to the pointer `AA[k]`, the resulting pointer points to the next object of the integer type.

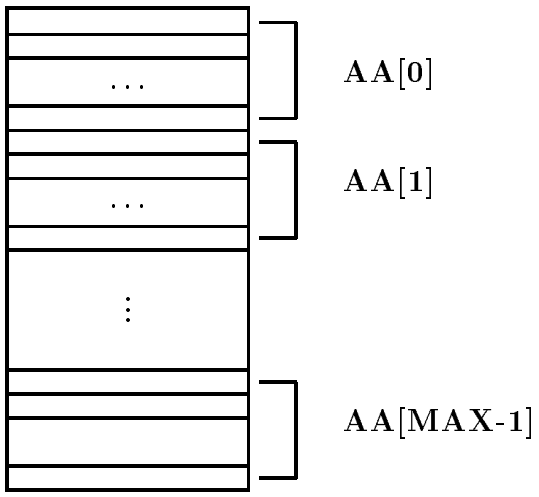


Figure 9.10: A Two Dimensional Array in Row Major Order

AA[0]	&AA[0][0]
AA[1]	&AA[1][0]
AA[2]	&AA[2][0]
AA[k]	&AA[k][0]
AA[0] + 1	&AA[0][1]
AA[0] + j	&AA[0][j]
AA[k] + j	&AA[k][j]

Table 9.1: Array Pointers and Sub-Arrays

* AA[0]	AA[0][0]
AA[k]	AA[k][0]
(AA[0] + 1)	AA[0][1]
(AA[0] + j)	AA[0][j]
(AA[k] + j)	AA[k][j]

Table 9.2: Dereferencing Array Pointers

* AA	AA[0]	&AA[0][0]
AA + 1	AA[0] + 1	&AA[0][1]
AA + j	AA[0] + j	&AA[0][j]
(AA + 1)	AA[1]	&AA[1][0]
(AA + k)	AA[k]	&AA[k][0]
(* AA)	*AA[0]	AA[0][0]
(* (AA + 1))	*AA[1]	AA[1][0]
(* (AA + k) + j)	*(AA[k] + j)	AA[k][j]

Table 9.3: Pointer Equivalence for Two Dimensional Arrays

The name, `AA`, is the name of the entire array, whose elements are themselves arrays of integers. Therefore, `AA` points to the first object in this array of arrays, i.e. `AA` points to the array `AA[0]`. The addresses represented by `AA` and `AA[0]` are the same; however, they point to objects of different types. `AA[0]` points to `AA[0][0]`, so it is an integer pointer. `AA` points to `AA[0]`, so it is a pointer to an integer pointer. If we add 1 to `AA`, the resulting pointer, `AA + 1`, points to the array `AA[1]`, and `AA + k` points to the array `AA[k]`. When we add to a pointer to some type, we point to the next object of that type. Therefore, adding to `AA` and `AA[0]` result in pointers to different objects. Adding 1 to `AA` results in a pointer that points to the next array or row, i.e. `AA[1]`; whereas adding 1 to `AA[0]` results in a pointer that points to `AA[0][1]`. Dereferencing such a pointer, `*(AA + k)`, accesses `AA[k]`, which as we saw, was `&AA[k][0]`. It follows that `*(*(AA + k) + j)` accesses the integer, `AA[k][j]`. This pointer equivalence for two dimensional arrays is shown in Table 9.3.

The C compiler converts array indexing to indirect access by dereferenced pointers as shown in the table; thus, all array access is indirect access When we pass a 2D array to a function, we pass its name (`AA`). The function can access elements of the array argument either by indexing or by pointers. We generally think of a two dimensional array as a table consisting of rows and columns as seen in Figure 9.11. As such, it is usually easiest to access the elements by indexing; however, the pointer references are equally valid as seen in the figure.

The relationships between different pointers for a two dimensional array is further illustrated with the program shown in Figure 9.12. The two-dimensional array, `a`, is not an integer pointer, it points to the array of integers, `a[0]`. However, `*a` is an integer pointer; it points to an integer object, `a[0][0]`. To emphasize this point, we initialize an integer pointer, `intptr` to `*a`, i.e. `a[0]`. The initial value of `intptr` is the address of `a[0][0]`. We next print the values of `a` and `*a`, which are the same address even though they point to different types of objects. In the `for` loop, we print the value of `a + i`, which is the same as that of `a[i]` even though they point to different types of objects. In the inner for loop, we print the address of the i^{th} row and the j^{th} column element of the row major array using pointers:

```
*a + COL * i + j
```

The same value is printed using the *address of* operator, `&a[i][j]`. Finally, the value of `a[i][j]` is printed using array indices as well as by dereferencing pointers, i.e. `*(*(a + i) + j)`.

The value of `intptr`, initialized to `*a`, is incremented after each element value is printed;

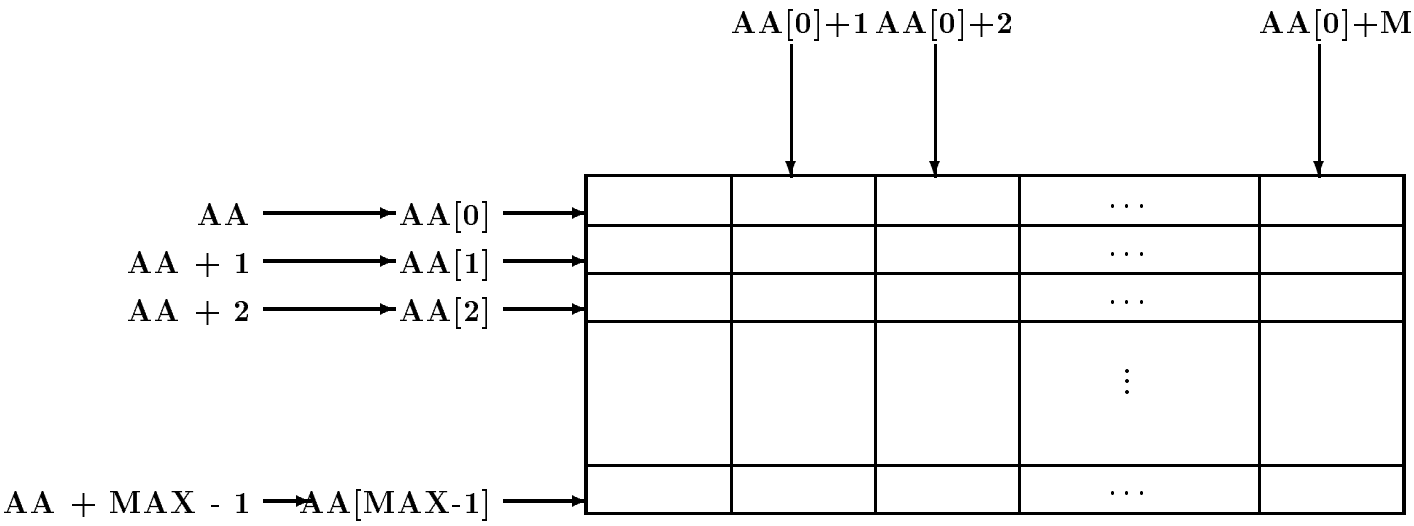


Figure 9.11: Pointers and Two Dimensional Arrays

making it point to the next element. The value of `intptr` is printed as it is incremented. Observe that it prints the address of each element of the array in one row, and proceeds to the next row in sequence. This shows that arrays are stored in row major form.

Finally, the function, `print2aray()` is used to print the two dimensional array in rows and columns. The output of a sample run is shown below.

```
***2D Arrays, Pointers ***

array (row) pointer a = 65474, *a = 65474
a + 0 = 65474

*a + COL * 0 + 0 = 65474; intptr = 65474
&a[0][0] = 65474
a[0][0] = 12; *(*(a + 0) + 0) = 12

*a + COL * 0 + 1 = 65476; intptr = 65476
&a[0][1] = 65476
a[0][1] = 24; *(*(a + 0) + 1) = 24

*a + COL * 0 + 2 = 65478; intptr = 65478
&a[0][2] = 65478
a[0][2] = 29; *(*(a + 0) + 2) = 29

a + 1 = 65480
*a + COL * 1 + 0 = 65480; intptr = 65480
&a[1][0] = 65480
a[1][0] = 23; *(*(a + 1) + 0) = 23
```

```
/*   File: ar2ptr.c
     Other Source Files: ar2util.c
     Header Files: ar2def.h, ar2util.h
     Program shows relations between arrays and pointers for 2 dimensional
     arrays.
*/

#include <stdio.h>
#define ROW 2
#define COL 3
print2aray(int x[][COL], int r, int c);

main()
{    int i, j, *intptr, a[ROW][COL] =
                        { {12, 24, 29}, {23, 57, 19} };

     printf("***2D Arrays, Pointers ***\n\n");
     intptr = *a;
     printf("array (row) pointer a = %u,  *a = %u\n", a, *a);
     for (i = 0; i < ROW; i++) {
          printf("a + %d = %u\n", i, a + i);
          for (j = 0; j < COL; j++) {
               printf("*a + COL * %d + %d = %u; intptr = %u\n",
                         i, j, *a + COL * i + j, intptr);
               printf("&a[%d][%d] = %u\n", i, j, &a[i][j]);

               printf("a[%d][%d] = %d; *(*(a + %d) + %d) = %d\n",
                         i, j, a[i][j],
                         i, j, *(*(a + i) + j));
               intptr++;
          }
     }
     print2aray(a, ROW, COL);
}
     /*   This Function prints a two dimensional integer array. */
     print2aray(int x[][COL], int r, int c)
     { int i, j;

          printf("\nThe two dimensional array is:\n\n");
          for (i = 0; i < r; i++) {
               for (j = 0; j < c; j++)
                    printf("%d  ", x[i][j]);
               printf("\n");
          }
     }
```

Figure 9.12: Program Illustrating 2D Array Pointers

```
*a + COL * 1 + 1 = 65482; intptr = 65482
&a[1][1] = 65482
a[1][1] = 57; *(*(a + 1) + 1) = 57

*a + COL * 1 + 2 = 65484; intptr = 65484
&a[1][2] = 65484
a[1][2] = 19; *(*(a + 1) + 2) = 19

The two dimensional array is:
 12  24  29
 23  57  19
```

As we mentioned in the last section, when a two dimensional array is passed to a function, the parameter declaration in the function **must** include the number of columns. We can now see why this is so. The number of columns in a row specifies the size of each row in the array of rows. Since the passed parameter is a pointer to a row object, it can be incremented and dereferenced, as shown in Table 9.3, to access the elements of the two dimensional array. The compiler must know the size of the row in order to be able to increment the pointer to the next row.

As we stated earlier, multi-dimensional arrays are arrays of arrays just like two dimensional arrays. An n dimensional array is an array of $n-1$ dimensional arrays. The same general approach applies as for two dimensional arrays. When passing an n dimensional array, the declaration of the formal parameter must specify all index ranges except for the first index.

As was seen in the program in Figure 9.12, multi-dimensional arrays may also be initialized in declarations by specifying constant initializers within braces. Each initializer must be appropriate for the corresponding lower dimensional array. For example, a two dimensional array may be initialized as follows:

```
int x[2][3] = { {10, 23}, {0, 12, 17} };
```

The array has two elements, each of which is an array of three elements. The first initializer initializes the first row of `x`. Since only the first two elements of the row are specified, the third element is zero. The second element initializes the second row. Thus, `x` is initialized to the array:

```
10   23   0
0    12   17
```

9.3 Arrays of Strings

Besides data base applications, another common application of two dimensional arrays is to store an array of strings. In this section we see how an array of strings can be declared and operations such as reading, printing and sorting can be performed on them.

A string is an array of characters; so, an array of strings is an array of arrays of characters. Of course, the maximum size is the same for all the strings stored in a two dimensional array. We can declare a two dimensional character array of `MAX` strings of size `SIZE` as follows:

```
char names[MAX][SIZE];
```

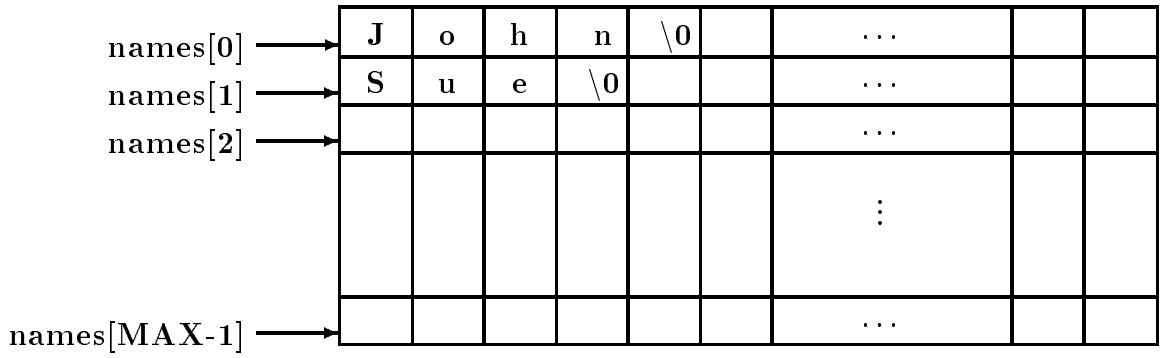


Figure 9.13: An Array of Strings

Since `names` is an array of character arrays, `names[i]` is the i^{th} character array, i.e. it points to the i^{th} character array or string, and may be used as a string of maximum size `SIZE - 1`. As usual with strings, a `NULL` character must terminate each character string in the array. We can think of an array of strings as a table of strings, where each row of the table is a string as seen in Figure 9.13.

We will need an array of strings in our next task to read strings, store them in an array, and print them.

NAMES: Read and store a set of strings. Print the strings.

We can store a string into `names[i]` by reading a string using `gets()` or by copying one into it using `strcpy()`. Since our task is to read strings, we will use `gets()`. The algorithm is simple:

```
while array not exhausted and not end of file,
     read a string into an array element
print out the strings in the array of strings
```

We will organize the program in several source files since we will be using some of the functions in several example programs. The program driver and the header file are shown in Figure 9.14.

The program reads character strings into an array, in this case, names. The program can, of course, serve to read in any strings. The `for` loop in `main()` reads strings into an array using `gets()` to read a string into `names[n]`, the n^{th} row of the array. That is, the string is stored where `names[n]` points to. The variable `n` keeps track of the number of names read. The loop is terminated either if the number of names equals `MAX`, or when `gets()` returns `NULL` indicating end of file has been reached. Next, the program calls on `printstrtab()` to print the names stored in the two dimensional array, `names`. The arguments passed are the array of strings and the number of strings, `n`.

The function, `printstrtab()` is included in the file `strtab.c` and its prototype is included in the file `strtab.h`. Remember, the second range of the two dimensional array of strings must be specified in the formal parameter definition, otherwise the number of columns in a row are unknown and the function cannot access successive rows correctly. A sample interaction for the compiled and linked program is shown below:

Sample Session:

```
***Table of Strings - Names***
```

```
/*   File: strtab.h */
#define SIZE 31     /* maximum size of a name plus a NULL */
void printstrtab(char strtab[][SIZE], int n);

     /*   File: names.c
          Other Source Files: strtab.c
          Header Files: strtab.h
          This program reads a set of names or strings into a two
          dimensional array. It then prints out the names.
     */

     #include <stdio.h>
     #define MAX 10
     #include "strtab.h"

     main()
     {    int n;                    /* number of names */
          char names[MAX][SIZE];    /* 2-d array of names */

          printf("***Table of Strings - Names***\n\n");
          printf("Enter one name per line, EOF to terminate\n");

          for (n = 0; (n < MAX) && gets(names[n]); n++)
               ;

          if (n == MAX)
               printf("\n**Table full - input terminated\n");
          printstrtab(names, n);
     }

     /*   File: strtab.c */
     #include <stdio.h>
     #include "strtab.h"

     /*   Prints n strings in the array strtab[][]. */
     void printstrtab(char strtab[][SIZE], int n)
     {    int k;

          printf("Names are:\n");
          for (k = 0; k < n; k++)
               puts(strtab[k]);
     }
```

Figure 9.14: Code for String Table Driver

```
Enter one name per line, EOF to terminate
John Smith
David Jones
Helen Peterson
Maria Schell
^D
Names are:
John Smith
David Jones
Helen Peterson
Maria Schell
```

9.3.1 String Sorting and Searching

Our next couple of tasks are simple and build on the last one. In one task we search (sequentially) for a string and in another we sort a set of strings.

SRCHSTR: Search for a key string in a set of strings.

We will use a function, `srchstr()`, to search for a string in an array of a specified size. The function returns either a valid index where the string is found or it returns -1 to indicate failure. The algorithm is simple enough and the implementation of a test program driver is shown in Figure 9.15.

The file `strtab.h` includes the prototypes for functions in file `strtab.c`. Observe the initialization of the two dimensional array `names[][]` using constant initializers written in braces separated by commas. Each initializer initializes a one dimensional string array written as a string constant. The program calls `srchstrtab()` searching for the string `"John Smith"`, and prints the returned index value.

As we have seen in Chapter 8, the library function `strcmp()` is used to compare two strings. The function returns zero if the argument strings are equal. A sequential search process for strings is easily developed by modifying the sequential search function of Chapter 10 replacing the equality operator with the function `strcmp()` to compare two strings.

In the function, `srchstrtab()`, we compare each string in the array with the desired string until we either find a match or the array is exhausted. The function call requires the name of the array of strings, the number of valid elements in the array, and the item to be searched for. For example, suppose we wish to search for a string, `key`, in the array, `names` with `n` string elements, then, the function call would be:

```
k = srchstrtab(names, n, key);
```

The value returned is assigned to an integer variable, `k`. If successful, `srchstrtab()` returns the index where the string was found; otherwise, it returns -1. The function is shown in Figure 9.16. In the for loop, the string that `strtab[i]` points to is compared with the string that `key` points to. If they are equal, `strcmp()` returns zero and the value of `i` is returned by the function. Otherwise, `i` is incremented, and the process is repeated. The loop continues until the valid array is exhausted, in which case -1 is returned. Again, the formal parameter definition for the two dimensional array, `x`, requires the size of the second dimension, `SIZE`. A sample run of the program is shown below:

```
/*   File: strsrch.c
     Other Source Files: strtab.c
     Header Files: strtab.h
     This program searches for a string (key) in a set of strings
     in a two dimensional array. It prints the index where key is found.
     It prints -1, if the string is not found.
*/

#include <stdio.h>
#define MAX 10
#include "strtab.h"

main()
{    int k;
     char names[MAX][SIZE] = { "John Jones", "Sheila Smith",
                                    "John Smith", "Helen Kent"};

     printf("***String Search for John Smith***\n\n");
     k = srchstrtab(names, 4, "John Smith");
     printf("John Smith found at index %d\n", k);
}
```

Figure 9.15: Driver for String Search Program

```
/*   File: strtab.c - continued */
#include <string.h>
/*   Searches a string table strtab[][] of size n for a string key. */
int srchstrtab(char strtab[][SIZE], int n, char key[])
{    int i;

     for (i = 0; i < n; i++)
          if (strcmp(strtab[i], key) == 0)
               return i;
     return -1;
}
```

Figure 9.16: Code for `srchstrtab()`

```
/*   File: strsort.c
     Other Source Files: strtab.c
     Header Files: strtab.h
     This program sorts a set of strings in a two dimensional array.
     It prints the unsorted and the sorted set of strings.
*/

#include <stdio.h>
#define MAX 10
#include "strtab.h"

main()
{    int n;
     char names[MAX][SIZE] = { "John Jones", "Sheila Smith",
                                    "John Smith", "Helen Kent"};

     printf("***String Array - unsorted and sorted***\n\n");
     printf("Unsorted ");
     printstrtab(names, 4);

     sortstrtab(names, 4);
     printf("Sorted ");
     printstrtab(names, 4);
}
```

Figure 9.17: Driver for Sorting Strings Program

```
***String Search for John Smith***

John Smith found at index 2
```

Our next task calls for sorting a set of strings.

SORTSTR: Sort a set of strings. Print strings in unsorted and in sorted order.

The algorithm is again very simple and we implement it in the program driver. The driver simply calls on `sortstrtab()` to sort the strings and prints the strings, first unsorted and then sorted. A prototype for `sortstrtab()` is included in file `strtab.h` and the driver is shown in Figure 9.17. An array of strings is initialized in the declaration and the unsorted array is printed. Then, the array is sorted, and the sorted array is printed.

Sorting of an array of strings is equally straight forward. Let us assume, the array of strings is to be sorted in increasing ASCII order, i.e. `a` is less than `b`, `b` is less than `c`, `A` is less than `a`, and so on. We will use the selection sort algorithm from Chapter 10. Two nested loops are needed; the inner loop moves the largest string in an array of some effective size to the highest index in the array, and the outer loop repeats the process with a decremented effective size until the effective size is one. The function is included in file `strtab.c` and shown in Figure 9.18. The function

```
/*    File: strtab.c - continued */
/*    Sorts an array of strings. The number of strings in the
      array is lim.
*/
void sortstrtab(char strtab[][SIZE], int lim)
{    int i, eff_size, maxpos = 0;
     char tmp[SIZE];

     for (eff_size = lim; eff_size > 1; eff_size--) {
          for (i = 0; i < eff_size; i++)

               if (strcmp(strtab[i], strtab[maxpos]) > 0)
                    maxpos = i;

          strcpy(tmp, strtab[maxpos]);
          strcpy(strtab[maxpos], strtab[eff_size-1]);
          strcpy(strtab[eff_size - 1], tmp);
     }
}
```

Figure 9.18: Code for `sortstrtab()`

is similar to the numeric selection sort function, except that we now use `strcmp()` to compare strings and `strcpy()` to swap strings. A sample session is shown below:

```
***String Array - unsorted and sorted***

Unsorted Names are:
John Jones
Sheila Smith
John Smith
Helen Kent

Sorted Names are:
Helen Kent
John Jones
John Smith
Sheila Smith
```

In our example program, the entire strings are compared. If we wish to sort by last name, we could modify our function to find and compare only the last names.

9.4 Arrays of Pointers

As seen in the last example, sorting an array of strings requires swapping the strings which can require copying a lot of data. For efficiency, it is better to avoid actual swapping of data whenever a data item is large, such as a string or an entire data base record. In addition, arrays may be needed in more than one order; for example, we may need an exam scores array sorted by Id number and by weighted scores; or, we may need strings in both an unsorted form and a sorted form. In either of these cases, we must either keep two copies of the data, each sorted differently, or find a more efficient way to store the data structure. The solution is to use pointers to elements of the array and swap pointers. Consider some examples:

```
int data1, data2, *ptr1, *ptr2, *save;

data1 = 100; data2 = 200;
ptr1 = &data1; ptr2 = &data2;
```

We could swap the values of the data and store the swapped values in `data1` and `data2` or we could simply swap the values of the pointers:

```
save = ptr1;
ptr1 = ptr2;
ptr2 = save;
```

We have not changed the values in `data1` and `data2`; but `ptr1` now accesses `data2` and `ptr2` access `data1`. We have swapped the pointer values so they point to objects in a different order. We can apply the same idea to strings:

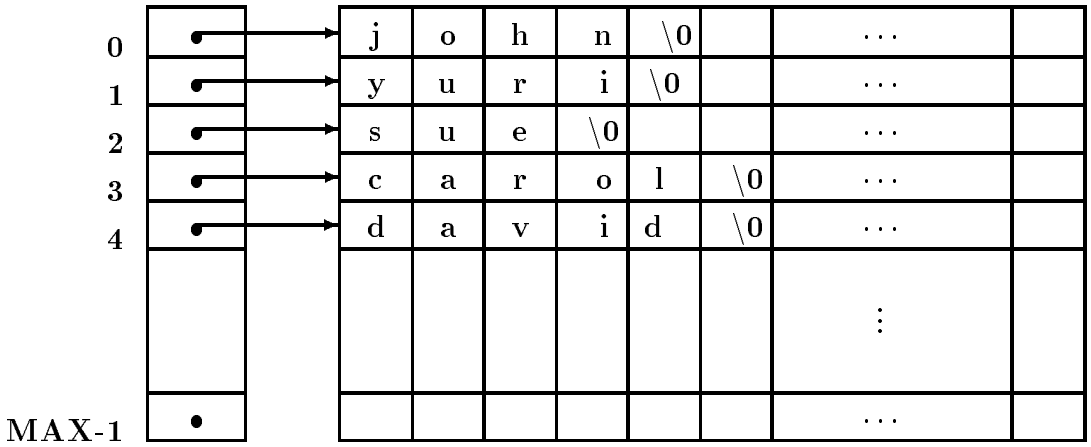
```
char name1[] = "John";
char name2[] = "Dave";
char *p1, *p2, *save;

p1 = name1;
p2 = name2;
```

Pointers `p1` and `p2` point to strings `name1` and `name2`. We can now swap the pointer values so `p1` and `p2` point to `name2` and `name1`, respectively.

In general, an array of pointers can be used to point to an array of data items with each element of the pointer array pointing to an element of the data array. Data items can be accessed either directly in the data array, or indirectly by dereferencing the elements of the pointer array. The advantage of a pointer array is that the pointers can be reordered in any manner without moving the data items. For example, the pointer array can be reordered so that the successive elements of the pointer array point to data items in sorted order without moving the data items. Reordering pointers is relatively fast compared to reordering large data items such as data records or strings. This approach saves a lot of time, with the additional advantage that the data items remain available in the original order. Let us see how we might implement such a scheme.

STRPTRS: Given an array of strings, use pointers to order the strings in sorted form, leaving the array unchanged.

We will use an array of character pointers to point to the strings declared as follows:



captionUnsorted Pointers to Strings

```
char * nameptr[MAX];
```

The array, `nameptr[]`, is an array of size `MAX`, and each element of the array is a character pointer. It is then possible to assign character pointer values to the elements of the array; for example:

```
nameptr[i] = "John Smith";
```

The string `"John Smith"` is placed somewhere in memory by the compiler and the pointer to the string constant is then assigned to `nameptr[i]`. It is also possible to assign the value of any string pointer to `nameptr[i]`; for example, if `s` is a string, then it is possible to assign the pointer value `s` to `nameptr[i]`:

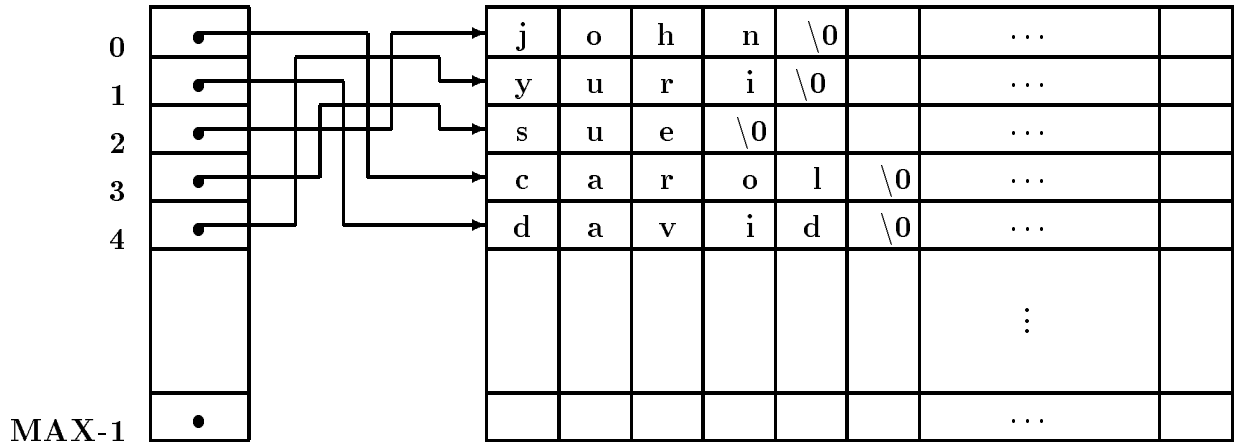
```
nameptr[i] = s;
```

In particular, we can read strings into a two dimensional array, `names[][]`, and assign each string pointer, `names[i]` to the i^{th} element of the pointer array, `nameptr[]`:

```
for (i = 0; i < MAX && gets(names[i]); i++)
     nameptr[i] = names[i];
```

The strings can then be accessed either by `names[i]` or by `nameptr[i]` as seen in Figure 9.4. We can then reorder the pointers in `nameptr[]` so that they successively point to the strings in sorted order as seen in Figure 9.4. We can then print the strings in the original order by accessing them through `names[i]` and print the strings in sorted order by accessing them through `nameptr[i]`. Here is the algorithm:

```
while not end of file and array not exhausted,
     read a string
     store it in an array of strings and
     assign the string to an element of a pointer array

access the array of strings and print them out
access the array of pointers and print strings that point to
```



captionSorted Pointers to Strings

The program driver, including a prototype for `sortptrs()` is shown in Figure 9.19. It declares a two dimensional array of strings, `names[][]`, and an array of character pointers, `nameptr[]`. It then reads strings into `names[][]`, and assigns each string pointer `names[i]` to `nameptr[i]`. The function `sortptrs()` is then called to reorder `nameptr[]` so the successive pointers of the array point to the strings in sorted order. Finally, strings are printed in original unsorted order by accessing them through `names[i]` and in sorted order by accessing them through `nameptr[i]`.

The function `sortptrs()` uses the selection sort algorithm modified to access data items through pointers. It repeatedly moves the pointer to the largest string to the highest index of an effective array. The implementation of sorting using pointers to strings is shown in Figure 9.20. The algorithm determines `maxpos`, the index of the pointer to the largest string. The pointer at `maxpos` is then moved to the highest index in the effective array. The array size is then reduced, etc.

Sample Session:

```
***Arrays of Pointers - Sorting by Pointers***

Enter one name per line, EOF to terminate
john
yuri
sue
carol
david
^D

The unsorted names are:
john
yuri
sue
carol
david
```

```
/*   File: ptraray.c
     This program uses an array of pointers. Elements of the array
     point to strings. The pointers are reordered so that they
     point to the strings in sorted order. Unsorted and sorted
     strings are printed out.
*/

#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define MAX  10          /* max number of names */
#define SIZE 31          /* size of names plus one for NULL */
void sortptrs(char * nameptr[], int n);

main()
{    int i;              /* counter */
     int n;              /* number of names read */
     char names[MAX][SIZE]; /* 2-d array of names */
     char *nameptr[MAX]; /* array of ptrs - used to point to names */

     printf("***Arrays of Pointers - Sorting by Pointers***\n\n");
     /* read the names into the 2-d array */
     printf("Enter one name per line, ");
     printf("EOF to terminate\n");

     for (n = 0; gets(names[n]) && n < MAX; n++)
          nameptr[n] = names[n];   /* assign string pointer */
                                   /* to a char pointer in the */
                                   /* array of pointers. */

     if (n == MAX)
          printf("\n***Only %d names allowed***\n", MAX);

     printf("The unsorted names are:\n");
     /* print the names */
     for (i = 0; i < n; i++)
          puts(names[i]);          /* access names in stored array.*/

     sortptrs(nameptr, n);         /* sort pointers */
     printf("The sorted names are:\n");
     for (i = 0; i < n; i++)       /* print sorted names, */
          puts(nameptr[i]);        /* accessed via array of pointers. */
}
```

Figure 9.19: Driver for Sorting Pointer Array Program

```
/*   File: ptraray.c  - continued */
/*   The elements of the array of pointers nameptr[] point to
     strings. The pointer array is reordered so the pointers
     point to strings in sorted order. The function uses selection
     sort algorithm.
*/

void sortptrs(char * nameptr[], int n)
{    int i, eff_size, maxpos = 0;
     char *tmpptr;

     for (eff_size = n; eff_size > 1; eff_size--) {
          for (i = 0; i < eff_size; i++)

               if (strcmp(nameptr[i],nameptr[maxpos]) > 0)
                    maxpos = i;

          tmpptr = nameptr[maxpos];
          nameptr[maxpos] = nameptr[eff_size-1];
          nameptr[eff_size-1] = tmpptr;
     }
}
```

Figure 9.20: Code for `sortptrs()`

```
The sorted names are:
carol
david
john
sue
yuri
```

Reordering of pointers, so they point to data items in sorted order, is referred to as *sorting by pointers*. When the data items are large, such as data records or strings, this is the preferred way of sorting because it is far more efficient to move pointers than it is to move entire data records.

9.5 An Example: Linear Algebraic Equations

As our final example program using two dimensional arrays in this chapter, we develop a program to solve systems of simultaneous linear equations. A set of linear algebraic equations, also called simultaneous equations, occur in a variety of mathematical applications in science, engineering, economics, and social sciences. Examples include: electronic circuit analysis, econometric analysis, structural analysis, etc. In the most general case, the number of equations, n, may be different from the number of unknowns, m; thus, it may not be possible to find a unique solution. However, if n equals m, there is a good chance of finding a unique solution for the unknowns.

Our next task is to solve a set of linear algebraic equations, assuming that the number of equations equals the number of unknowns:

LINEQNS: Read the coefficients and the right hand side values for a set of linear equations; solve the equations for the unknowns.

The solution of a set of linear equations is fairly complex. We will first review the process of solution and then develop the algorithm in small parts. As we develop parts of the algorithm, we will implement these parts as functions. The driver will just read the coefficients, call on a function to solve the equations, and call a function to print the solution.

Let us start with an example of a set of three simultaneous equations in three unknowns: x_1, x_2, and x_3.

$$1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 = 6$$

$$2 \cdot x_1 + 3 \cdot x_2 + 1 \cdot x_3 = 6$$

$$1 \cdot x_1 + 0 \cdot x_2 + 2 \cdot x_3 = 3$$

We can use arrays to represent such a set of equations; a two dimensional array to store the coefficients, a one dimensional array to store the values of the unknowns when solved, and another one dimensional array to store the values on the right hand side. Later, we will include the right hand side values as an additional column in the two dimensional array of coefficients. Each row of the two dimensional array stores the coefficients of one of the equations. Since the array index in C starts at 0, we will assume the unknowns are the elements `x[0]`, `x[1]`, and `x[2]`. Similarly, the elements in row zero are the coefficients in the equation number 0, the elements in row one are for equation number one, and so forth.

Then using arrays, a general set of n linear algebraic equations with m unknowns may be expressed as shown below:

```
a[0][0] * x[0] +  a[0][1]*x[1] + ...+  a[0][m - 1] * x[m - 1]  =  y[0]
a[1][0] * x[0] +  a[1][1]*x[1] + ...+  a[1][m - 1] * x[m - 1]  =  y[1]
...
a[n-1][0]*x[0] +   ...              +  a[n-1][m - 1]*x[m - 1]  =  y[n-1]
```

The unknowns and the right hand side are assumed to be elements of one dimensional arrays: `x[0], x[1],..., x[m - 1]` and `y[0], y[1],..., y[n - 1]`, respectively. The coefficients are assumed to be elements of a two dimensional array: `a[i][j]` for $i = 0, \ldots, n-1$ and $j = 0, \ldots, m-1$. The coefficients of each equation correspond to a row of the array. For our discussion in this section, we assume that m equals n. With this assumption, it is possible to find a unique solution of these equations unless the equations are linearly dependent, i.e. some equations are linear combinations of others.

A common method for solving such equations is called the *Gaussian elimination* method. The method eliminates (i.e. makes zero) all coefficients below the main diagonal of the two dimensional array. It does so by adding multiples of some equations to others in a systematic way. The elimination makes the array of new coefficients have an upper triangular form since the lower triangular coefficients are all zero.

The modified equivalent set of n equations in $m = n$ unknowns in the upper triangular form have the appearance shown below:

```
a[0][0]*x[0]+  a[0][1]*x[1] +    ...          +   a[0][n-1]  *x[n-1] =   y[0]
               a[1][1]*x[1] +    ...          +   a[1][n-1]  *x[n-1] =   y[1]
                              a[2][2]*x[2]..+   a[2][n-1]  *x[n-1] =   y[2]
                                                a[n-1][n-1]*x[n-1]=  y[n-1]
```

The upper triangular equations can be solved by *back substitution*. Back substitution first solves the last equation which has only one unknown, `x[n-1]`. It is easily solved for this value — `x[n-1] = y[n-1]/a[n-1][n-1]`. The next to the last equation may then be solved — since `x[n-1]` has been determined already, this value is substituted in the equation, and this equation has again only one unknown, `x[n-2]`. The unknown, `x[n-2]`, is solved for, and the process continues backward to the next higher equation. At each stage, the values of the unknowns solved for in the previous equations are substituted in the new equation leaving only one unknown. In this manner, each equation has only one unknown which is easily solved for.

Let us take a simple example to see how the process works. For the equations:

```
1 * x[0] + 2 * x[1] + 3 * x[2] = 6
2 * x[0] + 3 * x[1] + 1 * x[2] = 6
1 * x[0] + 0 * x[1] + 2 * x[2] = 3
```

We first reduce to zero the coefficients in the first column below the main diagonal (i.e. array index zero). If the first equation is multiplied by -2 and added to the second equation, the coefficient in the second row and first column will be zero:

```
1 * x[0] + 2 * x[1] + 3 * x[2] =  6
0 * x[0] - 1 * x[1] - 5 * x[2] = -6
1 * x[0] + 0 * x[1] + 2 * x[2] =  3
```

Similarly, if the first equation is multiplied by -1 and added to the third equation, the coefficient in the third row and first column will be zero:

```
1 * x[0] + 2 * x[1] + 3 * x[2] =  6
0 * x[0] - 1 * x[1] - 5 * x[2] = -6
0 * x[0] - 2 * x[1] - 1 * x[2] = -3
```

Coefficients in the first column below the main diagonal are now all zero, so we do the same for the second column. In this case, the second equation is multiplied by a multiplier and added to equations below the second; thus, multiplying the second equation by -2 and adding to the third makes the coefficient in the second column zero:

```
1 * x[0] + 2 * x[1] + 3 * x[2] = 6
0 * x[0] - 1 * x[1] - 5 * x[2] = -6
0 * x[0] + 0 * x[1] + 9 * x[2] = 9
```

We now have equivalent equations with an upper triangular form for the non-zero coefficients. The equations can be solved backwards — the last equation gives us `x[2]` = 1. Substituting the value of `x[2]` in the next to the last equation and solving for `x[1]` gives us `x[1]` = 1. Finally, substituting `x[2]` and `x[1]` in the first equation gives us `x[0]` = 1.

From the above discussion, we can see that a general algorithm involves two steps: modify the coefficients of the equations to an upper triangular form, and solve the equations by back substitution.

Let us first consider the process of modifying the equations to an upper triangular form. Since only the coefficients and the right hand side values are involved in the computations that modify the equations to upper triangular form, we can work with these items stored in an array with n rows and $n + 1$ columns (the extra column contains the right hand side values).

Let us assume the process has already reduced to zero the first $k - 1$ columns below the main diagonal, storing the modified new values of the elements in the same elements of the array. Now, it is time to reduce the k^{th} lower column to zero (by lower column, we mean the part of the column below the main diagonal). The situation is shown in below:

```
a[0][0]   a[0][1]   ...          a[0][k]      ...          a[0][n]
0         a[1][1]   ...          a[1][k]      ...          a[1][n]
0         0         a[2][2]...   a[2][k]      ...          a[2][n]
...       ...       ...          ...          ...          ...
0         0         0...         a[k][k]      a[k][k+1]..  a[k][n]
0         0         0...         a[k+1][k]    ...          a[k+1][n]
0         0         0...         a[k+2][k]    ...          a[k+2][n]
...       ...       ...          ...          ...          ...
0         0         0...         a[n-1][k]    ...          a[n-1][n]
```

The n^{th} column represents the right hand side values with `a[i][n]` equal to `y[i]`. We multiply the k^{th} row by an appropriate multiplier and add it to each row with index greater than k. Assuming that `a[k][k]` is non-zero, the k^{th} row multiplier for addition to the i^{th} row ($i > k$) is:

```
-a[i][k] / a[k][k]
```

The k^{th} row multiplied by the above multiplier and added to the i^{th} row will make the new `a[i][k]` zero. The following loop will reduce to zero the lower k^{th} column:

```
/*   Algorithm: process_column
     Reduces lower column k to zero.
*/
for (i = k + 1; i < n; i++) {      /* process rows k+1 to n-1 */
  m = - a[i][k] / a[k][k]          /* multiplier for ith row */

  for (j = k; j <= n; j++)         /* 0 thru k-1 cols. are zero */
    a[i][j] += m * a[k][j];        /* add kth row times m */
                                   /* to ith row. */
}
```

However, before we can use the above loop to reduce the lower k^{th} column to zero, we must make sure that `a[k][k]` is non-zero. If the current `a[k][k]` is zero, all we need to do is exchange this k^{th} row with any higher indexed row with a non-zero element in the k^{th} column. After the exchange of the two rows, the new `a[k][k]` will be non-zero. The above loop is then used to reduce the lower k^{th} column to zero. The non-zero element, `a[k][k]` used in the multiplier is called a *pivot*.

So, there are two steps involved in modifying the equations to upper triangular form: for each row find a pivot, and reduce the corresponding lower column to zero. If a non-zero pivot element is not found, then one or more equations are linear combinations of others, the equations are called *linearly dependent*, and they cannot be solved.

Figures 9.22 and 9.23 show the set of functions that convert the first n rows and columns of an array to an upper triangular form. These and other functions use a user defined type, `status`, with possible values `ERROR` returned if there is an error, and `OK` returned otherwise. The type `status` is defined as follows:

```
typedef enum {ERROR, OK} status;
```

We also assume a maximum of `MAX` equations, so the two dimensional array must have `MAX` rows and `MAX+1` columns. Figure 9.21 includes the header file with the defines and function prototypes used in the program. Since precision is important in these computations, we have used formal parameters of type `double`. The two dimensional arrays can store coefficients for a maximum of `MAX` equations (rows) and have `MAX + 1` columns to accommodate the right hand side values.

The function `uptriangle()` transforms coefficients of the equations to an upper triangular form. For each `k` from 0 through `n-1`, it calls `findpivot()` to find the pivot in the k^{th} column. If no pivot is found, `findpivot()` will return an `ERROR` (`findpivot()` is called even for the $(n-1)^{st}$ column even though there is no lower $(n-1)^{st}$ column to test if `a[n-1][n-1]` is zero). If `findpivot()` returns `OK`, then `uptriangle()` calls `process_col()` to reduce the lower k^{th} column to zero. We have included debug statements in `process_col()` to help track the process. The function `pr2adbl()` prints the two dimensional array — we will soon write this function.

The function `findpivot()` calls on function `findnonzero()` to find a non-zero pivot in column `k` if `a[k][k]` is zero. If a pivot is found, it swaps the appropriate rows and returns `OK`. Otherwise, it reurns `ERROR`. The function `findnonzero()` merely scans the lower column `k` for a non-zero element. It either returns the row in which it finds a non-zero element or it returns -1 if no such element is found. Rows of the array are swapped by the function `swaprows()` which also includes a debug statement to prints the row indices of the rows being swapped.

When `uptriangle()` returns with `OK` status, the array will be in upper triangular form. The next step in solving the equations is to employ back substitution to find the values of the unknowns.

```
/*   File: gauss.h */
typedef enum {ERROR, OK} status;
#define DEBUG
#define MAX 10           /* maximum number of equations */

status uptriangle(double a[][MAX + 1], int n);
void process_col(double a[][MAX + 1], int k, int n);
status findpivot(double a[][MAX + 1], int k, int n);
int findnonzero(double a[][MAX + 1], int k, int n);
void swaprows(double a[][MAX + 1], int k, int j, int n);
status gauss(double a[][MAX + 1], double x[], int n);
int getcoeffs(double a[][MAX + 1]);
void pr2adbl(double a[][MAX + 1], int n);
void pr1adbl(double x[], int n);
```

Figure 9.21: Header File for Gauss Functions

We now examine the back substitution process. As we saw earlier, we must solve equations backwards starting at index $n-1$ and proceeding to index 0. The i^{th} equation in upper triangular form looks like this:

```
a[i][i]*x[i] + a[i][i+1]*x[i+1] + ... + a[i][n-1]*x[n-1] = a[i][n]
```

Recall, in our representation, the right hand side is the n^{th} column of the two dimensional array. For each index i, we must sum all contributions from those unknowns already solved for, i.e. those `x[i]` with index greater than i. This is the following sum:

```
sum = a[i][i+1]*x[i+1] + ... + a[i][n-1]*x[n-1]
```

We then subtract this sum from the right hand side, `a[i][n]`, and divide the result by `a[i][i]` to determine the solution for `x[i]`. The algorithm is shown below:

```
/*   Algorithm: Back_Substitution */
for (i = n - 1; i >= 0;  i--) {          /* go backwards */
     sum = 0;

     for (j = i + 1; j <= n - 1; j++)    /* sum all contributions from */
          sum += a[i][j] * x[j];         /* x[j] with j > i */
     x[i] = (a[i][n] - sum) / a[i][i];   /* solve for x[i] */
}
```

We can now write the function `gauss()` that solves a set of equations by the Gaussian elimination method which first calls on `uptriangle()` to convert the coefficients to upper triangular form. If this succeeds, then back substitution is carried out to find the solutions. As with other functions, `gauss()` returns `OK` if successful, and `ERROR` otherwise. The code is shown in Figure 9.24. The code is straight forward. It incorporates the back substitution algorithm after the function call to `uptriangle()`. If the function call returns `ERROR`, the equations cannot be solved

```
/* File: gauss.c */
#include <stdio.h>
#include "gauss.h"

/*   Implements the Gauss method to transform equations to
     an upper triangular form.
*/
status uptriangle(double a[][MAX + 1], int n)
{    int i, j, k;

     for (k = 0; k < n; k++) {
          if (findpivot(a, k, n) == OK)
               process_col(a, k, n);
          else
               return ERROR;
     }
     return OK;
}

/*   Zeros out coefficients in column k below the main diagonal. */
void process_col(double a[][MAX + 1], int k, int n)
{    int i, j;
     double m;

     for (i = k + 1; i < n; i++) {
          m = -a[i][k] / a[k][k];
          for (j = k; j <= n; j++)
               a[i][j] += m * a[k][j];
          #ifdef DEBUG
               printf("Multiplier for row %d is %6.2f\n", i, m);
               pr2adbl(a, n);
          #endif
     }
}
```

Figure 9.22: Code for Functions to do Gaussian Elimination

```
/*   Finds a non-zero pivot element in column k and row with
     index >= k.
*/
status findpivot(double a[][MAX + 1], int k, int n)
{    int j;
     void swaprows();

     if (a[k][k] == 0) {
          j = findnonzero(a, k, n);
          if (j < 0)
               return ERROR;
          else
               swaprows(a, k, j, n);
          #ifdef DEBUG
               printf("Rows %d and %d swapped\n", k, j);
          #endif
     }
     return OK;
}

/*   Scans the rows with index >= k for the first non-zero element
     in the kth column of the array 'a' of size n.
*/
int findnonzero(double a[][MAX + 1], int k, int n)
{    int i;

     for (i = k; i < n; i++)
          if (a[i][k])
               return(i);
     return(-1);
}

/*   Swaps the kth and the jth rows in the array 'a' with n rows. */
void swaprows(double a[][MAX + 1], int k, int j, int n)
{    int i;
     double temp;

     for (i = k; i <= n; i++) {
          temp = a[k][i];
          a[k][i] = a[j][i];
          a[j][i] = temp;
     }
}
```

Figure 9.23: Code for Functions to do Gaussian Elimination — continued

```
/*   File: gauss.c - continued */
/*   Transforms equations to upper triangular form using Gauss
     method. Then, solves equations, one at a time.
*/
status gauss(double a[][MAX + 1], double x[], int n)
{    int i, j;
     double sum;

     if (uptriangle(a, n) == ERROR) {
          printf("Dependent equations - cannot be solved\n");
          return ERROR;
     }

     for (i = n - 1; i >= 0;  i--) {
          sum = 0;

          for (j = i + 1; j <= n - 1; j++)
               sum += a[i][j] * x[j];

          if (a[i][i])
               x[i] = (a[i][n] - sum) / a[i][i];
          else
               return ERROR;

     }
     return OK;
}
```

Figure 9.24: Code for `gauss()`

and `gauss()` returns `ERROR`. Otherwise, `gauss()` proceeds with back substitution and stores the result in the array `x[]`. Since all `a[i][i]` must be non-zero at this point, we do not really need to test if `a[i][i]` is zero before using it as a divisor; however, we do so as an added precaution.

We are almost ready to use the function `gauss()` in a program. Before we can do so; however, we need some utility functions to read and print data. Here are the descriptions of these functions:

`getcoeffs()`: reads the coefficients and the right hand side values into an array; it returns the number of equations.

`pr2adbl()`: prints an array with n rows and $n + 1$ columns.

`pr1adbl()`: prints a solution array.

All these functions use data of type double. The code is shown in Figure 9.25.

Finally, we are ready to write a program driver as shown in Figure 9.26. The driver first reads coefficients and the right hand side values for a set of equations and then calls on `gauss()` to solve the equations. During the debug phase, both the original data and the transformed upper triangular version are printed. Finally, if the equations are solved with success, the solution is printed. Otherwise, an error message is printed. During debugging, the macro `DEBUG` is defined in `gauss.h` so that we can track the process. The program loops as long as there are equations to be solved. In each case, it gets coefficients using `getcoeffs()` and solves them using `gauss()`. During debug, the program uses `pr2adbl()` to print the original array and the array after gauss transformation. If the solution is possible, the program prints the solution array using `pr1adbl()`. Here are several example equation solutions:

Sample Session:

```
***Simultaneous Equations - Gauss Elimination Method***

Number of equations, zero to quit:  2
Type coefficients and right side of each row
Row 0:  1 3 2
Row 1:  3 5 2

Original equations are:
   1.00  3.00  2.00
   3.00  5.00  2.00

Multiplier for row 1 is -3.00
   1.00   3.00   2.00
   0.00  -4.00  -4.00

Equations after Gauss Transformation are:
   1.00   3.00   2.00
   0.00  -4.00  -4.00

Solution is:
-1.00
1.00
```

```
/*   File: gauss.c - continued */
/*   Function gets the coefficients and the right hand side of equations.
*/
int getcoeffs(double a[][MAX + 1])
{    int i, j, n;

     printf("Number of equations, zero to quit: ");
     scanf("%d", &n);

     if (n)
          printf("Type coefficients and right side of each row\n");

     for (i = 0; i < n; i++) {
          printf("Row %d: ", i);

          for (j = 0; j <= n; j++)
               scanf("%lf", &a[i][j]);
     }
     return n;
}

/*   Prints coefficients and right side of equations */
void pr2adbl(double a[][MAX + 1], int n)
{    int i, j;

     for (i = 0; i < n; i++) {

          for (j = 0; j <= n; j++)
               printf("%10.2f  ", a[i][j]);

          printf("\n");
     }
}

/*   Prints the solution array */
void pr1adbl(double x[], int n)
{    int i;

     for (i = 0; i < n; i++)
          printf("%10.2f\n", x[i]);
}
```

Figure 9.25: Code for Utility Functions for Gauss Program

```
/*   File: gauss.c
     Header Files: gauss.h
     This program solves a number of simultaneous linear algebraic
     equations using the Gauss elimination method. The process repeats
     itself until number of equations is zero.
*/

main()
{    double a[MAX][MAX + 1];  /* coefficients and right hand side */
     double x[MAX];           /* solution */
     int n;                   /* number of equations */
     status soln;             /* status of solution, OK or ERROR */

     printf("***Simultaneous Equations***\n\n");
     while (n = getcoeffs(a)) {
          printf("\nOriginal equations are:\n");
          #ifdef DEBUG
               pr2adbl(a, n);
          #endif

          soln = gauss(a, x, n);
          #ifdef DEBUG
               printf("\nEquations after Gauss Transformation are:\n");
               pr2adbl(a, n);
          #endif

          if (soln == OK) {
               printf("\nSolution is:\n");
               pr1adbl(x, n);
          }

          else printf("Equations cannot be solved\n");
     }
}
```

Figure 9.26: Driver Program for Gaussian Elimination

```
Number of equations, zero to quit:  3
Type coefficients and right side of each row
Row 0:  1 2 3 4
Row 1:  4 3 2 1
Row 2:  0 7 2 5

Original equations are:
   1.00  2.00  3.00  4.00
   4.00  3.00  2.00  1.00
   0.00  7.00  2.00  5.00

Multiplier for row 1 is -4.00
   1.00   2.00    3.00    4.00
   0.00  -5.00  -10.00  -15.00
   0.00   7.00    2.00    5.00

Multiplier for row 2 is -0.00
    1.00   2.00    3.00    4.00
    0.00  -5.00  -10.00  -15.00
   -0.00   7.00    2.00    5.00

Multiplier for row 2 is 1.40
   1.00   2.00    3.00    4.00
   0.00  -5.00  -10.00  -15.00
   0.00   0.00  -12.00  -16.00

Equations after Gauss Transformation are:
   1.00   2.00    3.00    4.00
   0.00  -5.00  -10.00  -15.00
   0.00   0.00  -12.00  -16.00

Solution is:
-0.67
0.33
1.33

Number of equations, zero to quit:  3
Type coefficients and right side of each row
Row 0:  1 2 3 4
Row 1:  2 4 6 8
Row 2:  3 1 7 9

Original equations are:
   1.00  2.00  3.00  4.00
   2.00  4.00  6.00  8.00
   3.00  1.00  7.00  9.00

Multiplier for row 1 is -2.00
```

```
   1.00  2.00  3.00  4.00
   0.00  0.00  0.00  0.00
   3.00  1.00  7.00  9.00

Multiplier for row 2 is -3.00
   1.00   2.00   3.00   4.00
   0.00   0.00   0.00   0.00
   0.00  -5.00  -2.00  -3.00
Rows 1 and 2 swapped

Multiplier for row 2 is 0.00
   1.00   2.00   3.00   4.00
   0.00  -5.00  -2.00  -3.00
   0.00  -0.00  -0.00  -0.00
Dependent equations - cannot be solved

Equations after Gauss Transformation are:
   1.00   2.00   3.00   4.00
   0.00  -5.00  -2.00  -3.00
   0.00  -0.00  -0.00  -0.00
Equations cannot be solved
Number of equations, zero to quit:  0
```

The first two sets of equations are solvable; the last set is not because the second equation in the last set is a multiple of the first equation. Thus these equations are linearly dependent and they cannot be solved uniquely. In this case, after the $zero^{th}$ lower column is reduced to zero, `a[1][1]` is zero. A pivot is found in row 2, rows 1 and 2 are swapped, and lower column 1 is reduced to zero. However, `a[2][2]` is now zero, and there is no unique way to solve these equations.

If the coefficients are such that the equations are almost but not quite linearly dependent, the solution can be quite imprecise. An improvement in precision may be obtained by using an element with the *largest* absolute value as the pivot. Implementation of an improved version of the method is left as an exercise.

9.6 Common Errors

1. Failure to specify ranges of smaller dimensional arrays in declaration of formal parameters. All but the range of the first dimension must be given in a formal parameter declaration. Example:

   ```
   init2(int aray2[][])
   {
        ...
   }
   ```

   Error! `aray2` is a pointer to a two dimensional array, i.e. it points to an object that is a one-dimensional array, `aray2[0]`. Without a knowledge of the size of the object, `aray2[0]`,

it is not possible to access `aray2[1]`, `aray2[2]`, etc. Consequently, one must specify the number of integer objects in `aray2[0]`:

```
init2(int aray2[][COLS])
{ ...
}
```

Correct! `aray2[0]` has `COLS` objects. It is possible to advance the pointer, `aray2` correctly to the next row, etc.

2. Failure to pass arguments correctly in function calls:

```
init2(aray2[MAX][COLS]);
init2(aray2[][COLS]);
init2(aray2[][]);
```

All of the above are errors. A two dimensional array *name* is passed in a function call:

```
init2(aray2);
```

3. Confusion between pointers to different types of objects. For example, in the above, `aray2` points to an *array* object, `aray2[0]`, whereas `aray2[0]` points to an `int` object. The expression `aray2 + 1` points to `aray2[1]`, whereas `aray2[0] + 1` points to `aray2[0][1]`. In the first case the pointer is increased by `COLS` integer objects, whereas in the second case the pointer is increased by one integer object.

4. Confusion between arrays of character strings and arrays of character pointers:

```
char table[MAX][SIZE], *ptraray[MAX];
```

The first declares table to be a two dimensional array that can be used to store an array of strings, one each in `table[0]`, `table[1]`, `table[i]`, etc. The second declares `ptraray` to be an array, each element of which is a `char *`. Read the declaration from the end: `[MAX]` says it is an array with `MAX` elements; `ptraray` is the name of the array; `char *` says each element of the array is a `char *`. Properly initialized with strings stored in `table[][]`, `table[i]` can point to a string. Properly initialized with pointers to strings, `ptraray[i]` can also point to a string. However, `table[MAX][SIZE]` provides memory space for the strings, whereas `ptraray[MAX]` provides memory space only for pointers to strings. Both pointers may be used in a like manner:

```
puts(table[i]);
puts(ptraray[i]);
```

They will both print the strings pointed to by the pointers.

9.7 Summary

In this chapter we have seen that, in C, the concept of an array can be extended to arrays of multi-dimensions. In particular, a two dimensional array is represented as a one dimensional array, each of whose elements, themselves, are one dimensional arrays, i.e. and array or arrays. Similarly, a three dimensional array is an array whose elements are each 2 dimensional arrays (an array of arrays of arrays). We have seen how such arrays are declared within programs and how they are organized in memory (row major order). We have seen how we can access the elements of multi dimensional arrays using the subscripting notation and the correspondence between this notation and pointer values. Because for higher dimensional arrays, the pointer expressions may get complicated and confusing, in general, most programs use the subscripting notations for arrays of two dimensions or more. We have also shown that when passing arrays to functions, the size of all dimensions beyond the first must be specified in the formal parameter list of the function so that the location of all elements can be calculated by the compiler.

Throughout the chapter we have seen applications for which a two dimensional data structure provides a convenient and compact way of organizing information. These have included data base applications, such as our payroll and student test score examples, as well as using two dimensional arrays to store an array of strings. We have seen how we can then use this later data structure to search and sort arrays of strings, and have shown that for this data type, as well as other large data types, it is often more efficient to work with arrays of pointers when reordering such data structures.

Finally, we have developed a rather large application using 2D arrays — solutions to simultaneous linear equations usinf Gaussian elimination. This is one algorithm for doing computations in the realm of linear algebra; several additional examples common in engineering problems are presented in Chapter 15.

One last point to remember about multi-dimensional arrays: this data structure is a very useful way to organize a large collection of data into one common data structure; however, all of the data items in this structure must be of the same type. In the next chapter we will see another compound data type provided in C which does not have such a restriction — the *structure*.

9.8 Exercises

Given the following declaration:

```
int x[10][20];
```

Explain what each of the following represent:

1. `x`
2. `x + i`
3. `*(x + i)`
4. `*(x + i) + j`
5. `*(*(x + i) + j)`
6. `x[0]`
7. `x[i]`
8. `x[i] + j`
9. `*(x[i] + j)`

Find and correct errors if any. What does the program do in each case?

10.
```
main()
{    int x[5][10];

     init(x[][]);
}

void init(int a[][])
{    int i, j;

     for (i = 0; i < 10; i++)
          for (j = 0; j < 5; j++)
               a[i][j] = 0;
}
```

11.
```
main()
{    int x[5][10];

     init(x[][]);
}

void init(int *a)
{    int i, j;
```

```
        for (i = 0; i < 10; i++)
             for (j = 0; j < 5; j++) {
                  *a = 0;
                  a++;
    }
```

12.
```
    main()
    {    char s[5][100];

         read_strings(s);
         print_strings(s);
    }

    read_strings(char s[][100])
    {
         for (i = 0; i < 5; i++) {
              gets(*s);
              s++;
         }
    }

    print_strings(char s[][100])
    {
         while (*s) {
              puts(s);
              s++;
         }
    }
```

9.9 Problems

1. Read id numbers, project scores, and exam scores in a two dimensional array from a file. Compute the averages of each project and exam scores; compute and store the weighted average of the scores for each id number.

2. Repeat 1, but sort and print the two dimensional array by weighted average in decreasing order. Sort and print the array by id numbers in increasing order. Use an array of pointers to sort.

3. Repeat 2, but plot the frequency of each weighted score.

4. Combine 1-3 into a menu-driven program with the following options: read names, id numbers, and scores from a file; add scores for a new project or exam; save scores in a file; change existing scores for a project or an exam for specified id numbers; delete a data record; add a data record; compute averages; sort scores in ascending or descending order by a primary key, e.g. id numbers, weighted scores, etc.; compute weighted average; plot frequency of weighted scores; help; quit.

5. Write a function that uses binary search algorithm to search an array of strings.

6. Write a function that sorts strings by selection sort in either increasing or decreasing order.

7. Write a program that takes a string and breaks it up into individual words and stores them.

8. Repeat 7 and keep track of word lengths. Display the frequency of different word lengths.

9. Repeat 7, but store only new words that occur in a string. If the word has already been stored, ignore it.

10. Write a function that checks if the set of words in a string, `s`, represents a subset of the set of words in a second string, `t`. That is, the words of `s` are all contained in `t`, with `t` possibly containing additional words.

11. Write a menu-driven spell check program with the following options: read a dictionary from a file; spell check a text file; add to dictionary; delete from dictionary; display text buffer; save text buffer; help; quit.

    The dictionary should be kept sorted at all times and searched using binary search. Use an array of pointers to sort when new entries are inserted. In the spell check option, the program reads in lines of text from a file. Each word in a line is checked with the dictionary. If the word is present in the dictionary, it is ignored. Otherwise, the user is asked to make a decision: replace the word or add it to the dictionary. Either replace the word with a new word in the line or add the word to dictionary. Each corrected line is appended to a text buffer. At the quit command, the user is alerted if the text buffer has not been saved.

12. Write a simple macro processor. It reads lines from a source file. Ignoring leading white space, each line is examined to see if it is a control line starting with a symbol # and followed by a word "define". If it is, store the defined identifier and the replacement string. Each line is examined for the possible occurrence of each and every defined identifier; if a defined

identifier occurs in a line, replace it with the replacement string. The modified line must be examined again to see if a defined identifier exists; if so, replace the identifier with a string, etc.

13. Write a lexical scanner, `scan()`, which calls `nexttok()` of Problem 42 to get the next token from a string. Each new token or symbol of type identifier, integer, or float that is found is stored in a symbol table. The token is inserted in the table if and only if it was not already present. A second array keeps track of the type of each token stored at an index. The function `scan()` uses `srcharay()` to search the array of tokens, uses `inserttok()` to insert a token in an array, and uses `inserttype()` to insert type of a token.

    The function `scan()` returns a token in a string, type of the token, and index where the token is stored in the array. If the array is filled, a message saying so must be displayed.

    Write a program driver to read strings repeatedly. For each string, call `scan()` to get a token. As `scan()` returns, print the token, its type, and index. Repeat until an end of string token is reached. When the end of file is encountered, print each of the tokens, its type, and index.

14. Write routines for drawing lines and rectangles. Write a program that draws a specified composite figure using a character `'*'`. Allow the user to specify additions to the figure and display the figure when the user requests.

15. Modify 14 to a menu-driven program that allows: draw horizontal and vertical lines, horizontally oriented rectangles, filled rectangles, display figure, help, and quit.

16. Write a program that plays a game of tic-tac-toe with the user. The game has three rows and three columns. A player wins when he succeeds in filling a row or a column or a diagonal with his mark, `'0'`. The program uses `'*'`. Write and use the following functions:

    `init_board()`: initialize the board

    `display_board()`: dispays the board

    `enter_move()`: for user to enter a move in row and col

    `state_of_game()`: test state, finish or continue

17. Modify the Gauss Method so that a pivot with the largest magnitude is used in converting the array of coefficients to an upper triangular form.

18. Modify 17 to a menu-driven program that allows the following commands: Get coefficients, display coefficients, solve equations, display solution, verify solution, help, and quit. Write and use functions `get_coeffs()`, `display_coeffs()`, `solve_eqns()`, `display_soln()`, `verify_soln()`, `help()`.

19. Modify 18 so that the input data is in the form:

```
a00 x0 + a01 x1 + a02 x2 = b1
```

20. Modify 19 so that display coefficients displays equations in the above form.

21. Write a simple menu driven editor which allows the following commands: text insert, display text, delete text, delete lines, insert lines, find string, find word, replace string, replace word, help, and quit. A window should display part of the text when requested.

PART II

Chapter 10

Sorting and Searching

One very common application for computers is storing and retrieving information. For example, the telephone company stores information such as the names, addresses and phone numbers of its customers. When you dial directory assistance to get the phone number for someone, the operator must look up that particular piece of information from among all of data that has been stored. Taken together, all of this information is one form of a *data base* which is organized as a collection of *records*. Each record consists of several *fields*, each containing one piece of information, such as the name, address, phone number, id number, social security number or part number, etc..

As the amount of information to be stored and accessed becomes very large, the computer proves to be a useful tool to assist in this task. Over the years, as computers have been applied to these types of tasks, many techniques and algorithms have been developed to efficiently maintain and process information in data bases. In this chapter, we will develop and implement some of the simpler instances of these algorithms. The processes of "looking up" a particular data record in the data base is called *searching*. We will look at two different search algorithms; one very easy to implement, but inefficient, the other much more efficient. As we will see, in order to do an efficient search in a data base, the records must be maintained in some *order*. For example, consider the task of finding the phone number in the phone book of someone whose name you know, as opposed to trying to find the name of someone whose phone number you know in the same book.

The process of ordering the records in a data base is called *sorting*. We will discuss three sorting algorithms and their implementation in this chapter, as well. Sorting and searching together constitute a major area of study in computational methods. We present some of these methods here to introduce this area of computing as well as to make use of some of the programming techniques we have developed in previous chapters.

As we develop these algorithms, we use a very simple data base of records consisting of single integers only. We conclude the chapter by applying the searching and sorting techniques to our payroll data base with records consisting of multiple numeric fields. In Chapter 9 we will see how these same algorithms can be applied to string data types described in Chapter 11.

10.1 Finding a Data Item — The Search Problem

Suppose we have a collection of data items of some specific type (e.g. integers), and we wish to determine if a particular data item is in the collection. The particular data item we want to find

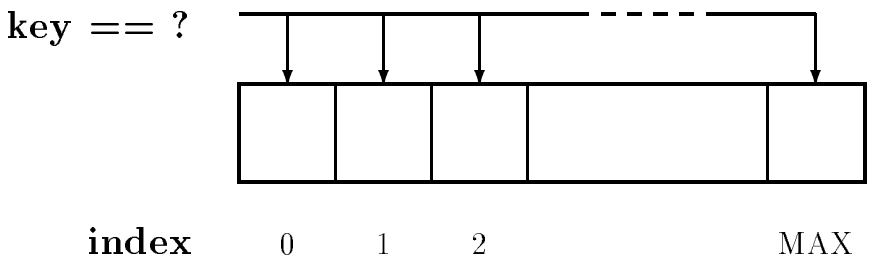


Figure 10.1: Sequential Search

is called the *key* and out task is to search the records in the data base to find one which "matches" the key.

The first decision we must make is how to represent the collection of data items. In Chapter 7 we saw a data structure which could hold a collection of data items all of the same type: the array. So we can consider our data base of integer values to be stored in an array. Our task then becomes:

Task:

SRCH0: Search an array for an index where the key is located; if key is not present, print a message. Repeat until an end of file is entered for the key.

In this task, we choose to return the index where the key is located because this index will allow us to retrieve the entire record in the case where our array is part of a database. The simplest approach to determine if a key is present in an array is to make an exhaustive search of the array. Start with the first element, if the key matches, we are done; otherwise move on to the next element and compare the key, and so on. We simply traverse the array in sequence from the first element to the last as shown in Figure 10.1. Each element is compared to the key. If the key is found in the array, the corresponding array index is returned. If the item is not found in the array, an invalid index, say -1, is returned. This type of search is called *Sequential Search* or *Linear Search* because we sequentially examine the elements of the array. In the worst case, the number of elements that must be compared with the key is linearly proportional to the size of the array.

Linear search is not the most efficient way to search for an item in a collection of items; however, it is very simple to implement. Moreover, if the array elements are arranged in random order, it is the only reasonable way to search. In addition, efficiency becomes important only in large arrays; if the array is small, there aren't many elements to search and the amount of time it takes is not even noticed by the user. Thus, for many situations, linear search is a perfectly valid approach. Here is a linear search algorithm which returns the index in the array where key is found or -1 if key is not found in the array:

```
initialize index i to 0
traverse the array until exhausted
     if array[i] matches key
          return i;
return -1.
```