Chapter 1

Arrays

1.1 Motivation

Let us start this chapter with a "fun" brain exercise. SUDOKU is an original Japanese number puzzle. It is presented as a table, as shown below, which is made up of 9 rows and 9 columns. Some spaces are blank, while some are filled up with an integer ranging from 1 to 9. Within a SUDOKU table there are "regions". Each region is made up of 3 rows and 3 columns. Regions are drawn bounded by thick lines.

The objective of the puzzle is to complete the table by filling up the missing numbers. The rule is very simple: a number cannot be repeated in the same column, or same row or region. Have fun solving the puzzle below!

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

Solving SUDOKU puzzles requires sound logic. It can be both entertaining, and intellectually fulfilling. SUDOKU became a big craze several years ago in many countries other than Japan. Today, you will find SUDOKU puzzles together with crossword puzzles in newspapers and magazines.

The popularity of SUDOKU prompted companies, and even university research laboratories to develop algorithms and computer programs for automatically generating and solving SUDOKU puzzles. There are a lot of websites offering daily SUDOKU puzzles with different levels of difficulty – catering from young children to adults alike. SUDOKU puzzles are available even in mobile devices such as cellular phones.

There are a host of other puzzles, games, and other computing problems requiring the use of a tabular structure. Examples include:

a. contact numbers in a phone book

b. ID numbers of students enrolled in a class

c. entries in an accounting ledger

d. chess board game

e. matrix for a system of linear equations

In this chapter, we will learn how we can represent and manipulate lists and tables such as SUDOKU puzzles in a computer using the C programming language.

1.2 What is an Array?

An array is a group of elements of the same data type, i.e., it is homogeneous. An array is characterized by four attributes, namely: (i) name, (ii) element type, (iii) size, and (iv) dimension.

The array's *name* is a user–defined identifier which is used to refer to the array's individual elements. The *size* attribute is a positive integer indicating the number of elements in the array. The *element type* is the data type of the elements in the array. The data type can be a simple data type (i.e., `char`, `int`, `float`, `double`) or a user–defined type (for example, structure). An array can be one–dimensional or multi–dimensional (i.e., dimension is greater than one).

Oh yeah, you can define custom data types in C called structures.

Multidimensional arrays are just arrays with arrays inside them. And even more the higher the dimension.

The number of bytes needed by an array are dependent on two factors, namely (i) number of elements and (ii) size of each element. These are both known during compile time – thus, a contiguous block of memory space for an array are set aside using *static memory allocation*. An array's size remains fixed, i.e., the number of array elements cannot be decreased or increased, during run–time.

Q: Ooh, so you cannot remove/append to arrays during runtime, as opposed to programming languages like Python?

In the following discussions, we first deal with one–dimensional arrays which are also commonly referred to as "lists" . The concepts will then be extended to two–dimensional arrays, and higher–dimensional arrays.

1.3 One–Dimensional Array

For brevity we will write 1D to mean one dimension. We can graphically represent a 1D array as a vertical list of elements as shown in the example below. The first element is 8, while the last element is 3. Think of the 1D array drawn in this manner as one of the columns, for example, of a SUDOKU table. In general, the data type, number of elements and the value of each element of a 1D array will be dependent on the problem being solved.

8
9
7
2
5
1
6
4
3

Alternatively, a 1D array can also be drawn as a horizontal list of elements as shown in the following example. In this case, think of the 1D array as one of the rows of the SUDOKU table.

8	9	7	2	5	1	6	4	3

1D arrays are not limited to integer elements. Illustrated below are arrays of characters and floating point values.

'C'	'O'	'M'	'P'	'R'	'O'	'2'

42.75	-10.23	63.54	89.75	101.23

1.3.1 1D Array Declaration

The syntax for declaring a 1D array in the C programming language is as follows:

```
<data type> <name> [<size>]
```

where `data type` is the element's data type, `name` is the array's name, `size` is the array size, and a single pair of square brackets means that the array is one–dimensional.

Some example declarations are:

```
char string[10];   /* 1D array of 10 characters */
int A[5];          /* 1D array of 5 integers */
float Value[50];   /* 1D array of 50 floats */
double Data[100];  /* 1D array of 100 doubles */
```

By default, array elements are uninitialized. It is the programmer's responsibility to initialize individual elements.

Several arrays of the same data type can be declared on the same line. For example:

```
int X[5], Y[10], Z[20];  /* three 1D arrays */
```

Good programming practice, however, recommends that variables be declared one at a time per line of code for better readability.

1.3.2 1D Array Element Definition

In C, a variable declaration tells the compiler how much memory will be needed by a variable based on its type. By default, the value of the variable is garbage. A *variable definition*, on the other hand, indicates not only the memory requirements of a variable, but also its initial value. Just like variables of simple data types, an array may be defined.[1] The syntax is as follows:

```
<data type> <name> [<size>] = {<list of values>}
```

where the `<list of values>` is the set of values to be assigned to array elements in the order that they appeared. The values need to be separated by comma and placed inside a pair of curly brackets. Examples of definitions are:

```
char coursecode[7] = {'C', 'O', 'M', 'P', 'R', 'O', '2'};
int M[3] = {10, 20, 30};
double F[5] = {42.75, -10.23, 63.54, 89.75, 101.23};
```

Note that it is not necessary to specify all the array elements. In the example array definition below, only the first two elements of array A, i.e., A[0] and A[1] are given. The remaining elements A[2] to A[9] are automatically initialized to 0. For array S, the elements S[4] to S[7] are also 0 (ASCII code 0). For array Z, the elements Z[2] to Z[19] are 0.0f (single precision floating point zero).

```
int A[10] = {1, 2};
char S[8] = {'G', 'o', 'o', 'd'};
float Z[20] = {-5.5, -2.0};
```

[1] Actually, the array elements may be defined.

1.3.3 Referencing an Element in a 1D Array

The syntax for referencing or accessing a particular array element is:

```
<name> [<index>]
```

where `index` is a whole number, and its range of values is from `0` to `size - 1`. Note that some textbooks and C compilers such as `gcc` use the term *subscript* instead of *index*. In this document, these two words mean the same thing.

Examples of valid array references based on the arrays declared above are:

```
string[0] /* refers to the 1st element of array string */
A[4]      /* refers to the last element of array A */
Value[9]  /* refers to the 10th element of array Value */
F[2]      /* refers to the 3rd element of array F */
```

Examples of invalid array references are:

```
Data[-1]  /* semantic error: negative index */
X[5]      /* semantic error: index is more than size - 1 */
Y[1.25]   /* syntax error: index is not a whole number */
```

Listing 1.1 shows a very simple example program incorporating what we learned so far: (i) array declaration and (ii) how to reference array elements. There are two arrays in the example program, namely `coursecode` and `A`. The elements of `coursecode` are defined, while those of `A` are assigned after the array declaration.

Listing 1.1: Array declaration and accessing elements

```
1  #include <stdio.h>
2  int main()
3  {
4      char coursecode[7] = {'C', 'O', 'M', 'P', 'R', 'O', '2'};
5      int A[5];
6      int i;
7
8      /* note: for loop is often used together with arrays */
9      for (i = 0; i <  7; i++)
10         printf("%c", coursecode[i]);
11     printf("\n");
```

```
12
13      /* initialize elements of array A */
14      A[0] = 3;
15      A[1] = -5;
16      A[2] = 1000;
17      A[3] = 123;
18      A[4] = 47;
19
20      for (i = 0; i < 5; i++)
21          printf("Value of A[%d] is %d.\n", i, A[i]);
22
23      return 0;
24  }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.1. What is the output of the program?

2. Change the initial values of the elements of array `A` to any integer value that you want. Run the program and see the corresponding output.

3. It is possible to assign the result of an expression as initial value to array elements. For example, change the initialization of the first element of array `A` to `A[0] = 60 / 2 + 5;`. Initialize the remaining array elements (i.e., `A[2]` to `A[4]`) using any valid expression that you want. Run and test the program to see the result.

4. Valid array indices must be from `0` to `size - 1`. Use of indices outside of this range will result into a semantic error. Change the second `for` loop initialization to `i = -2`, and the condition to `i < 8`. Compile the program. Is there a compilation error? If there is no error, run the program. Is there any run–time error?

5. Use the original program in Listing 1.1. Find out what will happen if the statement: `A[5] = 789;` is inserted before the `for` loop. Compile the program. Is there a compilation error? If there is no error, run the program. Is there any run–time error?

6. Edit the `for` loop in original program in Listing 1.1 such that the output is a list of array elements in reversed order, i.e., from the last element down to the first element.

Array elements are usually initialized to zero. Listing 1.2 shows how this can be done using a `for` loop.

Listing 1.2: Initializing elements to zero

```
1  #include <stdio.h>
2  int main()
3  {
4      int A[5];  /* integer array A with size of 5 */
5      int i;     /* i will be used as array index */
6
7      /* initialize all array elements to zero */
8      for (i = 0; i < 5; i++)
9          A[i] = 0;
10
11     /* print the array element values */
12     for (i = 0; i < 5; i++)
13         printf("Value of A[%d] is %d\n", i, A[i]);
14
15     return 0;
16 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.2. What is the output of the program?

2. It is possible to initialize the array elements with values other than zero. Edit the program by changing `A[i] = 0;` to `A[i] = (i + 1) * 5;`. Based on this, what will be the values of the array elements? Confirm your answer by running the program and taking note of the printed results.

3. Modify the program such that the values of the elements (from the first to the last) will be initialized as: -10, -8, -6, -4, and -2 respectively.

4. It is possible to initialize the array elements using `scanf()`. Edit the original program in Listing 1.2 by changing `A[i] = 0;` to `scanf("%d", &A[i]);`. Run and test the program to see if it works properly. Try to input a sequence of numbers that are not necessarily in any pattern, for example: 5, -3, 11, 2, 1000.

1.3.4 Working with multiple 1D arrays

Some programming problems require manipulation of multiple arrays within the same function as illustrated in Listing 1.3. The program declares two integer arrays named `A` and `B`, both of size 5. Thereafter, elements of array `A` are initialized via `scanf()`. The second `for` loop copies the elements of array `A` to array `B`. The last `for` loop outputs the values of array `B`.

Listing 1.3: Copying elements of array `A` to array `B`

```
1  #include <stdio.h>
2  int main()
3  {
4      int A[5];
5      int B[5];
6      int i;      /* i will be used as array index */
7
8      /* input elements of array A */
9      for (i = 0; i < 5; i++) {
10         printf("Input value of element %d: ", i);
11         scanf("%d", &A[i]);
12     }
13
14     /* copy elements of A to B */
15     for (i = 0; i < 5; i++)
16         B[i] = A[i];
17
18     /* output elements of B */
19     for (i = 0; i < 5; i++)
20         printf("Value of element %d is %d.\n", i, B[i]);
21
22     return 0;
23 }
```

At this point in time, an alert reader is probably asking, *"Can we not assign ALL elements of array* `A` *to* `B` *by just one assignment statement of the form* `B = A`*?* In the C programming language, operations on arrays have to be done on a per–element basis. It is syntactically incorrect to manipulate one whole array. Thus, an array–to–array assignment such as `B = A`, or the addition of two arrays such as `A + B` is not defined!

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.3. What is the output of the program?
2. Edit the code such that the program will copy the elements of `A` are copied to `B` in reversed order. That is, the last element of `A` is copied as the first element of `B` and that the first element of `A` is copied as the last element of `B`.
3. Verify that an array–to–array assignment is not allowed by inserting `B = A;` immediately after the first `for` loop. What is the compilation error?
4. Write a new program that will declare 3 integer arrays, say with the names `A`, `B`, and `C` which will store 5 elements each. Initialize the contents of array `A` using a `for` loop. Afterwards, initialize the contents of `B` using another loop. In a third loop, compute `C[i] = A[i] + B[i]`. Finally, output the contents of array `C` in another loop.

1.3.5 1D array and parameter passing

Can we pass a whole **array** as parameter to a function? The answer to this question is NO, and the reason was already explained above, i.e., *array–to-array assignment is not defined in the C programming language.*

A more detailed explanation is given as follows. Consider, for the meantime, a function whose prototype is `void Test(int x);`. It can be called as `Test(y)` where variable `y` contains some integer value. Based from what we learned about parameter passing (in CCPROG1), the *value* of the actual parameter `y` will be passed and *copied* to formal parameter `x`. The same concept, i.e., *copying the value of the actual parameter to the formal parameter*, does not extend to arrays because, as mentioned again above, *array–to–array to assignment is not defined.*

There is a reason why a whole array is not passed as parameter. For argument's sake, let us imagine, for the meantime, that arrays can be passed as parameters. What can happen when you have an array parameter that has a VERY BIG size, for example, 5 million double data type integers? (1) There might not be enough memory for the allocation of the formal parameter. (2) Even if memory was successfully allocated for a VERY BIG array, copying the elements from the actual to the formal parameter *may* take some precious time. This can cause slow program execution especially if the function involved will be called many times, for example, inside the body of a loop with 1000 iterations. Due to these concerns

in both the dimensions of memory space and time, whole arrays are not passed as parameter in C.

It is, however, possible to pass the (value of the) address of the array as function parameter. Thereafter, the array elements can be accessed by dereferencing the memory addresses of the individual elements.

We describe the "array notation" for accomplishing this task as illustrated in Listing 1.4.

The input of array elements are done in function `void InputElements(int A[], int n)`. The first formal parameter, i.e., `int A[]` tells the compiler that `A` is the *name* of a group of integer elements.[2] Note that it is not necessary to write (in–between the square brackets) a constant value corresponding to the actual array size. This function is called inside `main()` as `InputElements(A, 5)`.

Listing 1.4: 1D array and parameter passing

```
1  #include <stdio.h>
2  void InputElements(int A[], int n)
3  {
4      int i;
5
6      for (i = 0; i < n; i++) {
7          printf("Input the value of element %d: ", i);
8          scanf("%d", &A[i]);
9      }
10 }
11
12 void PrintElements(int A[], int n)
13 {
14     int i;
15
16     for (i = 0; i < n; i++)
17         printf("Value of element %d = %d.\n", i, A[i]);
18 }
19
20 int main()
21 {
22     int A[5];
23
24     InputElements(A, 5);
25     PrintElements(A, 5);
```

[2]The square brackets `[]` is the indicator that A is the name associated to a group.

```
26
27      return 0;
28  }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.4. What is the output of the program?

2. Modify the program such that the array size is for 10 integer elements.

3. Implement a new function `void ReversedPrintElements(int A[], int n)`. This function will print the elements of the array in reversed sequence, i.e., from the last element down to the first element. Call this function inside `main()`.

4. Verify that an array–to–array assignment is not allowed by inserting `B = A;` immediately after the first `for` loop. What is the compilation error?

5. Write a function `int GetSum(int A[], int n)` that will compute and return the sum of the elements in array `A`.

6. Write a function `int CountPositive(int A[], int n)` that will count and return the value of the number of positive integer elements in array `A` with `n` elements. Note that the value 0 is neither positive nor negative.

7. Write a function `int CountNegative(int A[], int n)` that will count and return the value of the number of negative integer elements in array `A` with `n` elements.

8. Write a function `int Search(int A[], int n, int key)` that will search `key` in array `A`. If it is in the list, the function should return the index of the element matching the key value, otherwise it should return a value of -1. Assume that the array elements are unique, i.e., no two values are the same. For example, let the original integer array contain the elements 10, -5, 23, -8, and 74. The function call `Search(A, 5, 23)` should return 2 which means that the search key 23 was found in index 2 of array A. On the other hand, the function call `Search(A, 5, 99)` should return -1 which means that the search key 99 was not found in array A.

9. Write a function `void RotateRight(int A[], int n, int x)` which will rotate the elements of array `A` towards the right by `x` number of positions. Assume that `x > 0`. For example, let the original integer array contain the

elements 10, -5, 23, -8, and 74. The rotated list after calling `RotateRight(A, 5, 1)` will contain the elements 74, 10, -5, 23, -8. Using the original array A, calling `RotateRight(A, 5, 3)` will result into 23, -8, 74, 10, -5.

10. Same as the previous problem except that rotation is towards the left. Implement function `void RotateLeft(int A[], int n, int x)`. Assume that `x > 0`.

11. Write a function `void CopyArray(int A[], int B[], int n)` that will copy the contents of array `A` to array `B`. Assume that both arrays A and B can hold up to n number of elements.

12. Write a function `void ReversedCopyArray(int A[], int B[], int n)` that will copy the contents of array `A` in reversed order to array `B`.

1.4 Relationship Between Arrays and Pointers

To a beginning programmer, the codes in Listing 1.4 present an illusion, i.e., it seems that what is being passed as parameter is the **whole** array `A`. As was explained above, this is really not true.

So what is it that is actually passed as parameter if it is not the array itself? The answer is: the **name** of the array. We quote, verbatim, the following from page 99 of (Kernighan & Ritchie, 1988)

> "...the name of an array is a synonym for the location of the initial element, ..."

Furthermore, on the same page, (Kernighan & Ritchie, 1988) wrote

> "When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable and so an array name parameter is a pointer, that is, a variable containing an address."

As indicated by the quotations above, arrays and pointers are very much related to each other. The name of the array corresponds to the address of the first element, i.e., the base address. We can easily verify this by printing the addresses of the array elements. We present two versions; the first one uses the address–of operator as shown in Listing 1.5. There is really nothing new about this program since we all know the meaning and how to use the `&` operator.

Listing 1.5: Display array element addresses using &

```
#include <stdio.h>
int main()
{
    int A[5];
    int i;
    for (i = 0; i < 5; i++)
       printf("Address of A[%d] is %p\n", i, &A[i]);

    return 0;
}
```

The second version, shown in Listing 1.6, is more interesting! Recall that the name of the array is synonymous with the base address. To get the memory address of an array element, we simply need to add the base address with the element's index. In the case of array `A`, the address of element `i` is computed as `A + i`.

Listing 1.6: Display array element addresses using address arithmetic

```
#include <stdio.h>
int main()
{
    int A[5];
    int i;
    for (i = 0; i < 5; i++)
       printf("Address of A[%d] is %p\n", i, A + i);

    return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 1.5. Run the program more than once. Take note of the addresses of the elements. What do you notice?
   Their addresses increment by 4.
2. Encode and run the program in Listings 1.6. Verify that the result of `A + i` is a memory address. Run the program more than once. Take note of the addresses of the elements.
   Like 1, their addresses increment by 4.
3. For both programs, try changing the data type of the array to (a) `char`, (b) `float` and (c) `double`. Check again the addresses of each array element.

Data Type	Bytes
char	1
int	4
float	4
double	8

The concept presented in the previous discussion is very important, we highlight the equivalence:

```
&A[i] == A + i
```

Applying the dereference operation on both sides of the equality symbol yields:

```
*&A[i] == *(A + i)
```

Since `*` and `&` cancels each other, the final equivalence relationship is given by:

```
A[i] == *(A + i)
```

`A[i]` is the *array indexing notation* for accessing the `i`'th element of array `A`. The *pointer dereference notation* for achieving the same effect is `*(A + i)`. The following is a verbatim quote from page 99 of (Kernighan & Ritchie, 1988)

> "In evaluating `a[i]`, C converts it to `*(a + i)` immediately; the two forms are equivalent."

Most beginning programmers find the pointer notation difficult, so they prefer to use the array indexing notation for accessing elements. With lots of practice and more experience, a programmer will become proficient in both forms.

We illustrate in Listing 1.7 how to access array elements by dereferencing the element address `A + i`.

Listing 1.7: Access array elements via dereference operator

```
1  #include <stdio.h>
2  int main()
3  {
4      int A[5];
5      int i;
6
7      /* initialize array elements */
8      for (i = 0; i < 5; i++)
9          *(A + i) = (i + 1) * 5;
10
```

```
11      for (i = 0; i < 5; i++)
12          printf("A[%d] = %d\n", i, *(A + i));
13
14      return 0;
15  }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.7.

2. Refer back to the problems given in the previous sections. Convert all the programs you have written from an array indexing notation to a pointer dereference form. Verify that the semantics are the same by testing and running your programs.

1.5 Two–Dimensional Array

For brevity we will write 2D to mean two dimension. A 2D array is a collection of homogeneous elements organized in a set of rows and columns. The more common term for 2D array is *table*. In mathematics, the term *matrix* is also used to refer to a 2D array. If the number of rows is the same as the number of columns, the matrix is called a *square matrix*. An example of a square matrix is a 9x9 SUDOKU puzzle we encountered at the start of this chapter.[3]

Consider a 2x3 array of integer shown below:

50	19	48
35	64	70

In the C programming language, rows and columns are indexed starting from 0. Thus, the first and second rows are indexed as row 0 and row 1 respectively. The same is true for columns. Thus, the columns in the example above are numbered as column 0, column 1 and column 2. A pair of row and column indices are needed to refer to an array element. The value of the array element at row 1, column 2 is 70.

We show in the next example a 5x5 square matrix of characters; the single quotes were omitted for brevity. The character at row 2, column 3 is `'D'`. Can you find combinations of letters that form words in English (similar to the game Boggle or Word Factory)?

R	I	A	H	C
B	A	G	I	A
A	R	O	D	T
T	E	N	S	E
S	I	G	N	G

1.5.1 2D Array Declaration

The syntax for declaring a 2D array in the C programming language is as follows:

```
<data type> <name> [<row size>][column size]
```

[3]In English,the notation `9x9` is read as "9 by 9" where the first number is the number of rows and the second number is the number of columns.

where `data type` is the element's data type, `name` is the array's name, `row size` is the number of rows, `column size` is the number of columns, and that two pairs of square bracket means that the array is 2D.

The total number of elements in the array is equal to the product of `row size` with `column size`. For example, a 2D array made of 2 rows and 3 columns has 6 elements. A SUDOKU puzzle has 81 elements.

Just like a 1D array, a block of contiguous memory space for storing the 2D array elements will be allocated using static memory allocation.

Some example declarations are:

```
char Boggle[5][5];   /* 2D character array with 5 rows, 5 columns */
int Table[2][3];     /* 2D integer array with 3 rows, 2 columns */
int Sudoku[9][9];    /* for Sudoku puzzle of 9 rows, 9 columns */
float Matrix[4][3];  /* 2D float matrix 4 rows, 3 columns */
```

1.5.2 2D Array Element Definition

Elements of a 2D array can be defined as follows

```
<data type> <name> [<row size>][<column size>] = {<list of values>}
```

where the `<list of values>` is the set of values to be assigned to array elements in the order that they appeared. The values need to be separated by commas and placed inside a pair of curly brackets.

The following are example of 2D array element definitions. The `Table` array and `Boggle` definitions correspond to the 2D arrays graphically represented in Section 1.5.

```
int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };

char Boggle[5][5] = { {'R', 'I', 'A', 'H', 'C'},
                      {'B', 'A', 'G', 'I', 'A'},
                      {'A', 'R', 'O', 'D', 'T'},
                      {'T', 'E', 'N', 'S', 'E'},
                      {'S', 'I', 'G', 'N', 'G'} };
```

1.5.3 Referencing an Element in a 2D Array

The syntax for referencing or accessing a particular array element in a 2D array is:

```
<name> [<row index>][<column index>]
```

where `row index` and `column index` are the indices of the element. The `row index` is an integer from `0` to `row size - 1`, while `column index` is from `0` to `column size - 1`. Examples of valid array references, based on the arrays declared above, are:

```
Boggle[0][0]    /* refers to element at row 0, column 0 */
Table[1][0]     /* refers to element at row 1, column 0 */
Sudoku[2][5]    /* refers to element at row 2, column 5 */
Sudoku[8][8]    /* refers to last element at row 8, column 8 */
Matrix[2][1]    /* refer to element at row 2, column 1 */
```

Examples of invalid array references are:

```
Boggle[-1][0]  /* semantic error: negative index */
Sudoku[0][9]   /* semantic error: column index is
                  more than column size - 1 */
Table[3.4][1]  /* syntax error: row index is not a whole number */
```

1.6 Mapping a 2D Array into the Primary Memory

We think of 2D arrays as two–dimensional in concept. It should be noted, however, that physically, it is mapped internally (i.e., stored internally) as a collection of 1D arrays into the primary memory. The C programming language in particular maps 2D array elements into the primary memory in *row–major order*.

Consider, for example, the 2D array defined with the name `M` below.

```
int M[2][3] = { {'A', 'B', 'C'}, {'X', 'Y', 'Z'} };
```

The 2D graphical representation of `M` is

'A'	'B'	'C'
'X'	'Y'	'Z'

In the softcopy of this document, we used red color as a visual cue to indicate the elements of row 0, while blue color indicates elements of row 1.

In the primary memory, `M` is mapped in row–major order. This means that the elements are ordered from row 0 to row 1 (in increasing row index). Within the same row, elements are ordered from column 0 to column 2 (in increasing column index). The corresponding graphical representation of the 2D array in row–major order is as follows:

'A'
'B'
'C'
'X'
'Y'
'Z'

More detailed information about array `M` with regards to the element ordering and the corresponding addresses are shown below.

Element	Address	Value
`M[0][0]`	`&M[0][0]`	'A'
`M[0][1]`	`&M[0][1]`	'B'
`M[0][2]`	`&M[0][2]`	'C'
`M[1][0]`	`&M[1][0]`	'X'
`M[1][1]`	`&M[1][1]`	'Y'
`M[1][2]`	`&M[1][2]`	'Z'

We use the codes in Listing 1.8 to demonstrate that 2D array elements are really mapped in row–major order. The program shows both brute force and nested `for` loop approach when working with 2D array elements.

Listing 1.8: Row–major order demo program

```
#include <stdio.h>
#include <stdio.h>
int main()
{
    int M[2][3] = { {'A', 'B', 'C'}, {'X', 'Y', 'Z'} };
    int i, j;

    printf("Brute force approach:\n");
    /* display row 0 element addresses */
    printf("&M[0][0] = %p\n", &M[0][0]);
    printf("&M[0][1] = %p\n", &M[0][1]);
    printf("&M[0][2] = %p\n", &M[0][2]);

   /* display row 1 element addresses */
   printf("&M[1][0] = %p\n", &M[1][0]);
   printf("&M[1][1] = %p\n", &M[1][1]);
   printf("&M[1][2] = %p\n", &M[1][2]);

   printf("\n");
   printf("Nested for loop approach:\n");
   for (i = 0; i < 2; i++)
      for (j = 0; j  < 3; j++)
          printf("&M[%d][%d] = %p\n", i, j, &M[i][j]);

   return 0;
}
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 1.8.
2. Modify the program by changing the number of rows from 2 to 4. Define the new elements using any value that you want.
3. Immediately after the nested `for` loop, insert the following codes:

   ```
   printf("\n");
   for (i = 0; i < 4; i++)
       printf("M + %d = %p\n", M + i); /* what does M + i represent? */
   ```

   Take note of the printed values. What do you think does the expression `M + i` represent?

4. Using the program from the previous problem, change the element data type from `int` to (a) `char`, (b) `float` and (c) `double`. Take note of the addresses of the elements.

Listing 1.9 shows a very simple example program incorporating what we learned so far: (i) 2D array definition, and (ii) how to reference array elements. It is very important to notice that a loop within a loop, i.e., a nested loop, is usually present when manipulating elements of a 2D array.

Listing 1.9: Accessing 2D array elements

```
1  #include <stdio.h>
2  int main()
3  {
4      int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };
5      int row; /* row index */
6      int col; /* column index */
7
8
9      /* print elements of array named as Table */
10     /* note that a double loop is commonly used when
11        accessing elements of a 2D array */
12     for (row = 0; row < 2; row++) {
13         for (col = 0; col < 3; col++)
14             printf("%d ", Table[row][col]);
15
16         printf("\n");
17     }
18     return 0;
19 }
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.9.

2. Edit the program by changing the array `Table` definition to an array declaration. Thereafter, insert new codes that will initialize the elements of `Table` as indicated in the drawing below:

0	0	0
0	0	0

3. Do the same requirements as item 2 above, except that the initial values for `Table` should be set as follows:

10	20	30
40	50	60

4. Write a new program that will define a 2D array named `Boggle` following the example in Section 1.5.2. Print the elements. The first line of output should be the elements of row 0; next line of output should be the elements of row 1, and so on.

5. Write a new program that will declare a 2D array named as `MyTable` with 4 rows and 3 columns with double data type elements. Thereafter, input the elements of the `MyTable` using `scanf()`. Finally, output the elements of the array similar to the sequencing mentioned in the previous problem. Print 2 digits only after the decimal point.

1.6.1 2D Array and Parameter Passing

Similar to the discussion in "1D Array and Parameter Passing" in subsection 1.3.5, the `name` of the 2D array can be passed as function parameter.

We show how to do this in Listing 1.10. The function `void Out2DArray(int Table[][3])` prints the elements in row–major order. The formal parameter `int Table[][3]` indicates that the name `Table` is the base address of a 2D array. There is no need to specify any value between the first pair of square brackets, but it is necessary to put the column size in the second pair of square brackets.
Why???

Listing 1.10: 2D Array and parameter passing

```
#include <stdio.h>
void Output2DArray(int Table[][3])
{
     int i;    /* row index */
     int j;    /* column index */

    for (i = 0; i < 2; i++) {
          for (j = 0; j < 3; j++)
              printf("%d ", Table[i][j]);

          printf("\n");
     }
}

int main()
{
     int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };

     Output2DArray(Table);
     return 0;
}
```

We modify the program by introducing a new function for input of the `Table` elements. The function prototype for this function is `void Input2DArray(int Table[][3])`. We pass the name of the array as parameter as shown in Listing 1.11. Notice that both input and output of array elements are done using row–major order.

Listing 1.11: Input array elements in another function

```
#include <stdio.h>

void Input2DArray(int Table[][3])
{
    int i;    /* row index */
    int j;    /* column index */

    for (i = 0; i < 2; i++) {
          for (j = 0; j < 3; j++)  {
              printf("Input element %d, %d: ", i, j);
              scanf("%d", &Table[i][j]);
          }
          printf("\n");
```

```
14      }
15  }
16
17  void Output2DArray(int Table[][3])
18  {
19      int i;    /* row index */
20      int j;    /* column index */
21
22     for (i = 0; i < 2; i++) {
23          for (j = 0; j < 3; j++)
24              printf("%d ", Table[i][j]);
25
26          printf("\n");
27      }
28  }
29
30  int main()
31  {
32      int Table[2][3];
33
34      Input2DArray(Table);
35      Output2DArray(Table);
36      return 0;
37  }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listings 1.10 and 1.11.
2. Modify the program by considering a 2D array with more rows and columns. For example, change row size to 4 and column size to 5.
3. Modify the programs by changing the data type from `int` to `double`.

In the preceding example programs, all the array elements were referenced inside a nested `for` loop. Is it possible to access only the elements of a particular row or column? Listing 1.12 shows how this can be done. Function `void OutputRowElements(int Table[][3], int row)` references only the elements of a specified row – the value of which is passed as a parameter. In this case, there is no need to use a nested loop. Only one loop will be needed to generate the indices of the column numbers within a specified row.

Listing 1.12: Access elements of one row only

```
#include <stdio.h>

void OutputRowElements(int Table[][3], int row)
{
   int col;   /* column index */

   for (col = 0; col < 3; col++)
       printf("%d ", Table[row][col]);

   printf("\n");
}

int main()
{
    int Table[2][3] = { {50, 19, 48}, {35, 64, 70} };
    int row;

    printf("Input row number of the elements you want to print: ");
    scanf("%d", &row);

    OutputRowElements(Table, row);
    return 0;
}
```

▶ Self–Directed Learning Activity ◀

1. Encode and run the program in Listing 1.12. What is the output when `row = 0`? when `row = 1`? when `row` is more than `1`, for example, `row = 2`?
2. Write a new function `void OutputColumnElements(int Table[][3], int col)` which will output only the elements of a particular column from the first row to the last row. For example, the call `OutputColumnElements(Table, 1)` will print 19 followed by 64 using the `Table` definition in Listing 1.12.
3. Write a function `int GetRowSum(int Table[][3], int row)` that will compute and return the sum of the elements in a specified row. For example: `GetRowSum(Table, 1)` will return a value of `169` (computed as `35 + 64 + 70`).
4. Write a function `int GetColumnSum(int Table[][3], int col)` that will compute and return the sum of the elements in a specified column. For example: `GetColumnSum(Table, 2)` will return a value of `118` (computed as `48 + 70`).

1.7 Representative Problems

In this section we discuss representative problems which naturally requires 1D and 2D arrays.

1.7.1 Sorting

The *sorting* problem takes as input a list of numbers. Its output is a list of the same numbers but re–arranged into either increasing or decreasing order. For example, if the original list is {`40, 30, 10, 50, 20`}, the sorted list in increasing order is {`10, 20, 30, 40, 50`}, and the sorted list in decreasing order is {`50, 40, 30, 20, 10`}.

There are several sorting algorithms in the literature. Some of these are bubble sort, straight selection sort, insertion sort, merge sort, shell sort and quicksort. We will discuss one sorting algorithm only, specifically, straight selection sort.[4]

In an increasing order of elements, the first element should be the smallest element, and the last element is the largest. We use the following as an example of an initially unsorted array.

40	30	10	50	20

The algorithm is described below.

1. Assume, for the meantime, that `A[0]` is the "current" smallest value in the (unsorted) list. Determine which, among the remaining elements, is the element with the least value compared with `A[0]`. If there is such an element, swap it with `A[0]`.

   Using the sample array above, `A[0]` is 40. We find among the remaining elements the least value compared with `A[0]`. This value corresponds to 10 which has an index of 2. We then swap `A[0]` with `A[2]`. The resulting array after swapping is

10	30	40	50	20

[4] A more detailed discussion of sorting algorithms will be given in a subject called Data Structures and Algorithms.

where the element in red color is the element that should rightfully occupy the first position, while the element in blue was the original element occupying said position.

2. We then repeat the same logic as above for the remaining elements `A[1]` to `A[4]`. Let `A[1]` be the current smallest element. Among the remaining elements, the one with least value compared with `A[1]` is `A[4]`. We swap these two elements resulting to:

10	20	40	50	30

3. Continuing with this logic, the contents of the arrays will change as follows:

10	20	30	50	40

10	20	30	40	50

Notice that we repeated the same instructions for four times on five elements. In general, it takes a maximum of `n-1` iterations to sort `n` elements using straight selection sort.

We show in Listing 1.13 the implementation of the sorting algorithm.

Listing 1.13: Straight selection sort

```
1  #include <stdio.h>
2
3  void Swap (int *px, int *py)
4  {
5      int temp;
6
7      temp = *px;
8      *px = *py;
9      *py = temp;
10 }
11
12 void Sort(int A[], int n)
13 {
14     int i;
15     int j;
16     int min; /* index of the smallest value */
17
18     for (i = 0; i < n-1; i++) { /* note: n-1 steps only */
```

```
19          min = i;
20
21          for (j = i + 1; j < n; j++) {
22              if (A[min]  > A[j])
23                  min = j;
24          }
25          Swap(&A[i], &A[min]);
26      }
27  }
28
29  void Output1DArray(int A[], int n)
30  {
31      int i;
32      for (i = 0; i < n; i++)
33          printf("%d\n", A[i]);
34  }
35
36  int main()
37  {
38      int A[5] = {40, 20, 30, 50, 10};
39
40      printf("Array before sorting: \n");
41      Output1DArray(A, 5);
42
43      Sort(A, 5);  /* sort the array elements */
44      printf("\n");
45      printf("Array after sorting: \n");
46      Output1DArray(A, 5);
47
48      return 0;
49  }
```

► Self–Directed Learning Activity ◄

1. Encode and run the program in Listing 1.13. Next, edit the program by increasing the size of the array from 5 to 10. Initialize the contents using an integer value that you want. Execute the program to see the results. Change the contents several more times and see the corresponding results.

2. Write a new program to implement a sort function that performs straight selection sorting resulting to a list that is in decreasing order.

3. Visit `http://people.cs.ubc.ca/~harrison/Java/sorting-demo.html` to

see a visualization/animation of different sorting algorithms including straight selection sort.

1.7.2 Matrix Algebra

Matrices (containing numeric values) are used in many problems such as in solving a system of simultaneous equation. There is even a specific area in mathematics called Matrix Algebra that studies matrices and operations on matrices.

We will describe simple matrix operations only. In the following discussions, let `s` represent a scalar value and `A`, `B` and `C` represent matrices.

The basic operations on matrices are:

1. Scalar multiplication: `B = s * A`

   This multiplies each element of `A` with a scalar value `s`. More specifically, `B[i][j] = s * A[i][j]`.

2. Matrix Addition: `C = A + B`

   This operation adds the elements of `A` with `B`, and stores the result into matrix `C`. The per–element relationship is given by `C[i][j] = A[i][j] + B[i][j]`. Note that the matrices should have the same operand.

Functions which implement these matrix operations are given below. Assume that arrays are 5x5.

```
void ScalarMultiplication(int B[][5], int A[][5], int s)
{
     int i, j;

     for (i = 0; i < 5; i++)
         for (j = 0; j < 5; j++)
              B[i][j] = s * A[i][j];
}

void MatrixAddition(int C[][5], int A[][5], int B[][5])
{
     int i, j;
```

```
        for (i = 0; i < 5; i++)
            for (j = 0; j < 5; j++)
                C[i][j] = A[i][j] + B[i][j];
}
```

► **Self–Directed Learning Activity** ◄

1. Encode and incorporate the codes above in a program. Create your own `main()` function which will contain the declarations for the matrices and other variables. The `main()` function should call `ScalarMultiplication()` and `MatrixAddition()`. Print the contents of the resulting matrix.

2. Create a new function `void MatrixSubtraction(int C[][5], int A[][5], int B[][5])` that will compute `C = A - B`, i.e., the difference between matrix `A` and `B`.

1.8 Experiments for the Curious

Let us try some experiments. We will deliberately commit mistakes and see what will happen when we compile and run the codes.

1. Can we declare a 1D array with a negative size? For example, will the declaration `int A[-5];` be accepted by the compiler?

2. Can we declare a 1D array with a size of 0? For example, will `int A[0];` be accepted by the compiler? If this was accepted by your compiler, do the following: declare array `A` then assign 5 to `A`, i.e., `A[0] = 5;`. Compile and then run the program. Is there any run–time error? If you answered yes, what is the run–time error?

3. Can we define a 1D array with a size of 0? For example, will the definition `int A[0] = 5;` be accepted by the compiler?

4. Can we declare a 1D array with a size of 1? For example, will `int A[1];` be accepted by the compiler? Does it make sense to declare a 1D array of just one element?

5. Recall the that the name of the array is synonymous with the base address. Is it possible to increment the array's name in the function where it was declared? For example, will the following program work? Give a reason why or why not it will work.

```
int main()
{
    int A[5];

    A++;   /* will this work? */
    return 0;
}
```

6. Try next the following program. Will it work? Why or why not?

```
void Test(int A[], int n)
{
     A[0] = 5;
     A++;        /* will this work? */
     *A = 10;
}

int main()
{
    int A[5];

    printf("A[0] = %d\n", A[0]);
    printf("A[1] = %d\n", A[1]);
    return 0;
}
```

7. Is it possible to declare a 2D array with 1 row and several columns? For example, is the declaration `int A[1][3];` syntatically correct? Does it make sense to declare an array with just one row or just one column?

8. Assume an array with 3 rows and 5 columns. What do you think will happen if it was passed to a function which accepts fewer columns? Test this with the following function:

```
void Test(int M[][2])
{
     int i, j;
     for (i = 0; i < 3; i++)
         for (j = 0; j < 2; j++) {
             printf("%d ", M[i][j]);

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

1.9 Chapter Summary

The key ideas presented in this chapter are summarized below:

- An array is a finite group of homogeneous elements.
- An array is characterized by four attributes, namely: name, size, element type and dimension.
- Memory space for arrays are allocated using static memory allocation.
- An array element is accessed by indicating its name and its correponding index. 2D arrays require two indices, namely the row index and column index.
- The name of the array corresponds to its base address. It can be passed as function parameter.
- Elements of a 1D array can also be accessed via pointer dereference.

Problems for Chapter 1

Problem 1.1. Write a function `int IsIncreasingOrder(int A[], int n)` which will return 1 if `A[i] < A[i+1]` for `i = 0` to `n-2`, otherwise it will return 0. Assume that elements are unique, i.e, no two elements have the same value.

Problem 1.2. Write a function `int CountOdd(int A[], int n)` which will count and return the number of odd values in array `A`.

Problem 1.3. Write a function `int Minimum(int A[], int n)` which will determine and return the smallest element in array `A`.

Problem 1.4. Write a function `int Maximum(int A[], int n)` which will determine and return the largest element in array `A`.

Problem 1.5. Write a function `int Sum(int A[], int n)` which will determine and return the sum of all the elements in array `A`.

Problem 1.6. Write a function `float Average(int A[], int n)` which will determine and return the average of the elements in array `A`. Note that the function is of type `float`.

Problem 1.7 Write a function `int CountUpper(char S[], int n)` which will count the number of upper case letters in array `S`. For example, assume an array `S` defined as follows: `char S[7] = {'C', 'o', 'M', 'p', 'r', 'o', '2'};` A call to `CountUpperCase(S, 7)` will return 2 since there are two upper case letters, namely, 'C' and 'M'.

Problem 1.8 Write a function `void ConvertUpper(char S[], int n)` which will convert all letters in the array to upper case. For example, `ConverUpper(S, 7)` using the array `S` defined in the previous problem will result into a modified array `S` containing 'C', 'O', 'M', 'P', 'R', 'O', '2'.

Problem 1.9. Write a function `void MaxCopy(int C[], int A[], int B[], int n)`. Assume that arrays `A` and `B` contain values. Determine which of the two values between `A[i]` and `B[i]` is higher. The higher value is assigned to `C[i]` for `i = 0` to `n-1`.

Problem 1.10. An identity matrix is a square matrix whose main diagonal elements are all 1, and the remaining elements are all 0. Write a function `int IsIdentityMatrix(int M[][5])` that will return 1 if the 5x5 matrix `M` is an identity matrix; otherwise, it should return 0.

Problem 1.11 Find out (for example, using Google search) what is the *transpose* of a matrix. Write a function `void TransposeMatrix(int M[][5], int T[][5])` that will compute and store the transpose of a given matrix `M` to matrix `T`.

Problem 1.12 Find out how to compute the product of two matrices, say `A` and `B`. Please see the following site: `http://people.hofstra.edu/Stefan_Waner/realWorld/tutorialsf1/frames3_2.html`. Implement a function for multiplying matrices `A` with `B` with the result stored in matrix `C`. The function prototype is `void MatrixMultiply(int C[][5], int A[][5], int B[][5]);`.

Problem 1.13. You were familiarized with row–major order in this chapter. Find out what is column–major order. Explain in not more than two sentences what is column–major order.

Problem 1.14. Assume a 2D array `M` of 3 rows and 5 columns. Write a function `void PrintColumnMajorOrder(int M[][5])` which will print the values of the array in column major order. Print one number only per line of output.

References

Consult the following resources for more information.

1. comp.lang.c Frequently Asked Questions. `http://c-faq.com/`

2. Kernighan, B. and Ritchie, D. (1998). *The C Programming Language, 2nd Edition.* Prentice Hall.