Chapter 8

8.1 Declaring and Referencing Arrays
8.2 Array Subscripts
8.3 Using For Loops for Sequential Access
8.4 Using Array Elements as Function Arguments
8.5 Array Arguments
8.6 Searching and Sorting an Array
8.7 Multidimensional Arrays
8.9 Common Programming Errors

Introduction

- Simple data types use a single memory cell to store a variable
- Collections of data should be logically grouped
- Example: 75 students in the class should we declare 75 separate variables to hold grades?
- Grouping related data items together into a single composite data structure is done using an array

Declaring Arrays I

- An array is a collection of two or more adjacent memory cells, called array elements
- All elements in an array are associated with a single variable name
- Each element is individually accessed using indices

Declaring Arrays II

- To set up an array in memory, we declare both the name of the array and the number of cells associated with it:
  ```
  double my_first_array[8];
  int students[10];
  ```
- This instructs C to associate 8 memory cells of type double with the name my_first_array
- This instructs C to associate 10 memory cells of type int with the name students
- These memory cells will be adjacent to each other in memory

Referencing Array Elements I

- To process the data stored in an array, each individual element is associated to a reference value
- By specifying the array name and identifying the element desired, we can access a particular value
- The sub scripted variable x[0] (read as x sub zero) may be used to reference the first element
**Referencing Array Elements II**

- Other elements can be accessed similarly: `x[1], x[2], ...`
- `myArray[0] = 8;`
- `printf("value of second element=%d",myArray[1]);`
- `scanf("input a number: %d",&anotherArray[9]);`  
  - For an array of size `n`, we index `0, 1, ..., n-1`
  - An array size must be an integer (no such thing as half an element)

---

**Referencing Array Elements I**

**Pitfall**

- Take care that you do not reference an index outside the array:
  1. `double grades[7];`
  2. `...`
  3. `printf("75 th grade is %f\n", grades[74]);` ← Illegal
  4. `printf("76 th grade is %f\n", grades[75]);` ← Illegal
  5. `printf("-1th grade is %f\n", grades[-1]);` ← Illegal
  6. `int i;`  
  7. `for (i =0; i <76; i ++)`  
  8. `printf("%d-th grade is %f\n", (i+1), grades[i]);`  
  9. ↑ Illegal on last iteration

---

**Array Initialization**

- You can declare multiple arrays along with regular variables:
  - `double cactus[5], needle, pins[7];`
- We can initialize a simple variable when we declare it:
  - `int sum = 0;`
- Same with arrays:
  1. `int array[SIZE];`
  2. `for (i =0; i < SIZE ; i ++)`  
  3. `array[i] = 0;`

---

**Array Declaration & Initialization**

- We can declare and initialize an array  
- If we initialize when we declare, we can omit the size
  1. `int primeNumbersLessThanHundred[] = {`
  2. `2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,`
  3. `41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,`
  4. `89, 97 };`

---

**Using for Loops for Sequential Access**

- Elements of an array are processed in sequence, starting with element zero.
- This processing can be done easily using an indexed for loop: a counting loop whose loop control variable runs from zero to one less than the array size.
- Using the loop counter as an array index (subscript) gives access to each array element in turn.
  1. `for(i=0; i < SIZE; i++) {`
  2. `printf("%d ",array[i]);`
  3. `}

---

**Using Array Elements as Function Arguments**

- You can use scanf with array elements just like with regular variables
  1. `int x[10];`
  2. `int i = 0;`
  3. `scanf("%d", &x[i]);`
  4. `printf("Hey, I read %d\n", x[i]);`
Arrays as Arguments

- You can also use entire arrays as function arguments
- Passing arrays as arguments to a function means:
  - The function can access any value in the array
  - The function can change any value in the array
- Syntax: specify an array as a parameter by using the square brackets:
  \[ \text{int sum(int array[], int size);} \]
- Note: what is actually being passed is a pointer to the first element of
  the array!
- We could equivalently define:
  \[ \text{int sum(int *array, int size);} \]

Full Example

```c
#include <stdio.h>

int sum (int array[], int size);

int main (void)
{
    int foo[] = {1,2,3,4,5,6,7,8,9,10} , i;
    int sum (int array[], int size);
    printf("sum of all array elements is %d\n",sum(foo, 10));
    return 0;
}
```

Searching and Sorting an Array

Two common problems with array processing:
1. Searching - Finding the index of a particular element in an array
2. Sorting - rearranging array elements in a particular order

Arrays as Input Arguments

- Since arrays are passed by reference, functions can modify their values
- Sometimes, we would like to pass arrays as arguments, but do not want
to change their values.
- We can do this by using the `const` quantifier in the function declaration:
  `int sum(const int foo[], int size)...`
- Specifies to the compiler that the array is to be used only as an input
- The function does not intend to modify the array
- The compiler enforces this: any attempt to change an array element in
  the function as an error

Formal Array Parameter

- It was necessary to pass an additional variable `size` to `sum`
- An array does not have an explicit size associated with it
- C does not allocate space in memory for arrays, the operating system
does at runtime
- As programmers, we are responsible for:
  - Memory management,
  - for keeping track of the size of an array and
  - for ensuring that we do not access memory outside the array
- If a function accesses an array, it needs to be told how big it is

Returning an Array Result

- C only allows us to return a single item
- It is not possible to return an array (a collection of items)
- We can, however, return a `pointer` to an array
- We cannot return a pointer to a `local array` (dangerous, undefined
  behavior)
- Requires knowledge of `dynamic memory` and `malloc`
- More later, for now: declare an array large enough for your purposes

Arrays as Arguments

- You can also use entire arrays as function arguments
- Passing arrays as arguments to a function means:
  - The function can access any value in the array
  - The function can change any value in the array
- Syntax: specify an array as a parameter by using the square brackets:
  \[ \text{int sum(int array[], int size);} \]
- Note: what is actually being passed is a pointer to the first element of
  the array!
- We could equivalently define:
  \[ \text{int sum(int *array, int size);} \]
Searching an Array

**Input:** hello
1. Assume the target has not been found //hello
2. Start with the initial array element, a[0];
3. While the target is not found and there are more array elements do
4. 
5. If the current element matches array then
6. 
7. Advance to next array element
8. 
9. If flag is set to true then
10. 
11. Return the array index
12. 
13. Return -1 to indicate not found

**Algorithm 1:** Searching Algorithm

Sorting an Array - Selection

1. foreach index value i = 0, ..., n−2 do
2. Find the index of the smallest element in the subarray a[i], ..., a[n−1];
3. Swap the smallest element with the element stored at index i;
4. end

**Algorithm 2:** Selection Sort Algorithm

Sorting an Array - Bubble Sort

1. While j ≤ n−1 do
2. 
3. If a[j] > a[j+1] then
4. 
5. end

**Algorithm 3:** Bubble Sort Algorithm
Multidimensional Arrays I

- A multidimensional array is an array with two or more dimensions
- Two-dimensional arrays represent tables of data, matrices, and other two-dimensional objects
- Declare multidimensional arrays similar to regular arrays:
  ```c
  int myArray[10][20];
  ```
- This declares a 10 × 20 sized array
- Interpretation: 10 rows, 20 columns

Multidimensional Arrays II

- Each row/column is still indexed 0, . . . , n − 1 and 0, . . . , m − 1
- Last row, last column: `myArray[9][19] = 29;`
- When iterating over a multidimensional array, use nested for loops

```c
int a[10][10];
for (i = 0; i < 10; i++)
for (j = 0; j < 10; j++)
a[i][j] = 1 + i + j;
```

Initialization of Multidimensional Arrays

You can initialize multidimensional arrays when declaring

```c
char tictactoe[][3] = {
{' ',' ',' '},
{' ',' ',' '},
{' ',' ',' '}
};
```

This would initialize a 3 × 3 the array with all blank spaces.

Dynamic Memory

- `int array[10];` is a static declaration
- The size is fixed for the life of the program
- Often, you don’t know how large of an array you’ll need
- Not practical or possible to declare a “large enough” array for all purposes
- C does not allow you to declare an array size using a variable

Common Programming Errors

- Most common error: out-of-range access error
- Segmentation fault, Bus error
- Error may not be caught in some situations: unexpected results
- Use correct syntax when passing arrays as parameters
Dynamic Memory

```
1 int n;
2 printf("Enter the size of the array: ");
3 scanf("%d", &n);
4 int array[n]; ← Bad
```

- May compile
- May even work (sometimes)
- Behavior is not defined in C

Dynamic Memory I

How-to in C

- The C function `malloc` (memory allocation) can be used to allocate memory.
- `malloc` takes one argument: the number of bytes to be allocated
- `malloc` returns a generic pointer to the allocated memory
- Returns `NULL` if it failed
- Can cast the generic pointer to the proper type (unnecessary in C, but required in C++)
- C function `sizeof` gives the number of bytes of each type of variable (system dependant!)

Dynamic Memory II

```
1 int n = 10;
2 int * myDynamicArray = NULL ;
3 myDynamicArray = ( int *) malloc (n * sizeof ( int ));
4 myDynamicArray[0] = 1;
5 myDynamicArray[9] = 42;
```

Dynamic Memory I

Garbage Collection

- If and when you are done using the memory, you should free it up so that it can be reused
- The C function `free` deallocates memory (frees it up):
  `free(myDynamicArray);`
- Note that all information in `myDynamicArray` will be lost

Dynamic Memory I

Memory Management

- Be careful that you don’t cause dangling pointers or memory leaks
- If you allocate memory, but lose a pointer to it, then the memory is effectively lost
- The memory is still being used, but you cannot access it
- This is a memory leak
Dynamic Memory I
Memory Management

```c
int * my_array;
my_array = (int *) malloc(10 * sizeof(int));
my_array = NULL;
```

- The memory location pointing to that 40 bytes is now lost
- The memory will be unavailable (until the program ends).

Dynamic Memory I
Multi-Dimensional Arrays

- To declare a dynamic multi-dimensional array, you need to use pointers to pointers
- Each row (or column) needs a call to malloc via a loop

```c
int **myMatrix = NULL;
myMatrix = (int **) malloc(10 * sizeof(int *));
int i=0;
for(i=0; i<10; i++)
myMatrix[i] = (int *) malloc(10 * sizeof(int));
myMatrix[9][9] = 10;
```

Exercises
Write the following functions and write a main driver program to test them.
- `void printArray(int *array, int size)` — prints the elements of an integer array
- `void printMatrix(int **array, int rows, int columns)` — prints the elements of an integer array
- `double average(int *array, int size)` — computes the average of all elements in the array
- `int *onesArray(int size)` — returns a pointer to a dynamically allocated integer array all initialized to 1
- `int *sortedCopy(int *array, int size)` — returns (a pointer to) a sorted copy of array
- `int **identityMatrix(int n)` — returns a pointer to an $n \times n$ integer array, initialized to the identity matrix