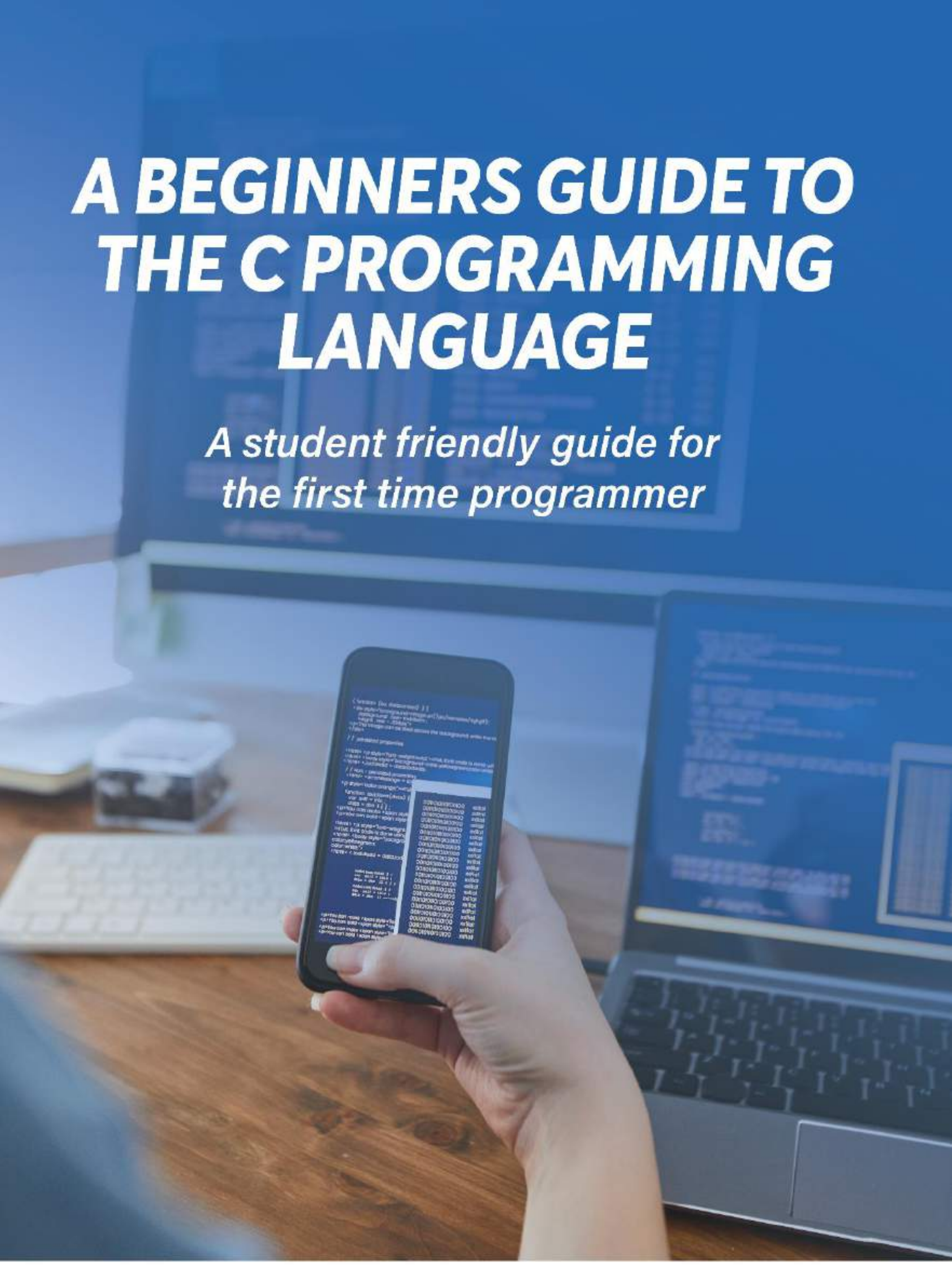
A BEGINNERS GUIDE TO THE C PROGRAMMING LANGUAGE

A student friendly guide for the first time programmer



- Sanjeevan D'Souza

Elite IT Academy

Beginners Guide to Programming in 'C'

A student-friendly guide to learning the 'C' Programming Language

Sanjeevan D'Souza

CHAPTER FOURTEEN
ARRAYS IN 'C'

An important aspect of designing good programs is to organize data efficiently. This means that memory is not wasted, data can be easily accessed with minimal code and is consistent. i.e., does not overflow and maintains the required precision or accuracy.

So, programmers spend a big part of program design, ensuring that the data is well-organized. So far, we have stored data only in individual variables. In this and later chapters, we will see how to store and access data in collections efficiently.

Arrays

An array is a collection of similar data of the same data type. If you need multiple variables for the same kind of data, you can create a single array to access this collection.

For example, if we wanted to store the age of 100 students, we can declare 100 variables to store each student's age. If we do this, then the code required to process these variables would need to be repeated for each variable, and it would be tedious to process them, as shown below

```
int age1, age2, age3;
int sum = 0;
double avg;

printf(“Input Age: “);
scanf(“%d”, &age1);
sum += age1;
```

```
8.
9.  printf("Input Age: ");
10. scanf("%d", &age2);
11. sum += age2;
12.
13. printf("Input Age: ");
14. scanf("%d", &age3);
15. sum += age3;
16.
17. avg = sum/3.0;
18. printf("Sum is %7.2f\n", avg);
```

In this example, we have considered only three *age* variables instead of 100.

We first declare them in line 1.

It's important to note that the variables are not necessarily created contiguously in memory when you declare variables. Each variable can theoretically be anywhere in memory, and we can access the values within the variable by directly referencing the variable name, as is done in line numbers 6,7, 10, 11, etc. This addressing is called *Direct Addressing*.

Also, because we have three distinct variables, lines 6-8 must be repeated multiple times for each variable name to perform the program's function. It would result in many lines of code being written, and most of the code would be duplicated over & over again, making the program longer and challenging to maintain.

The three *age* variables are logically the same. i.e., they contain *age* information, and the same operations are performed on them.

This is an excellent case to use arrays. An *array* is used to declare a collection of logically similar variables of the same data type, where you want to perform the same operations on all of them.

An array is allocated contiguous memory.

The syntax of an array declaration is shown below

```
Data_type array_name [number of elements];
```

```
2.  Data_type array_name [number of elements] =
                             {initializer list};
3.  Data_type array_name[] = {initializer list};
```

Example:

```
1.  int age[100];
2.  int age[100] = {23,25,20,22,23};
3.  int age[] = {23,25,20,22,23};
```

The first line declares an *int* array called *age,* with 100 elements. The elements in the *array* are uninitialized if it is declared locally or zero if declared globally.

The second line declares an *int* array called *age*, with 100 elements. Only the first five elements of the array are initialized. The remaining are uninitialized if declared locally or zero if declared globally.

The third line declares an *int* array called *age* but does not specify the number of elements in the array. Instead, it has an *initializer list*. In this case, the size of the array is the number of initializers in the initializer list, which is 5.

An array occupies a contiguous block of memory. The size of the block is the product of the number of elements in the array and the size of the array data type.

In line 1, the size of the array is 4 * 100 = 400 bytes

In line 2, the size of the array is 4 * 100 = 400 bytes

In line 3, the size of the array is 4 * 5 = 20 bytes

The number within the square brackets ([ ]) is the number of elements in the array. You can access each element of the array by using the *base-indexed* form of addressing, as shown below

```
Array_name[index]
```

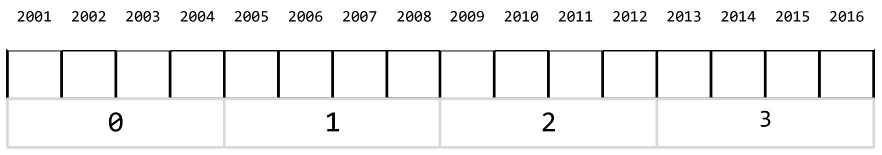
The array name is the starting address of the array. The index provides the element number being accessed. The index can range from 0 for the 1st element of the array, to n-1 where n is the number of array elements.

Note that the first element of the array has an index of 0.

So, the *base indexed* form of addressing is as follows

Base Address + (sizeof each element of the array * index)

We can visualize an array in memory, as shown below



The last row in the diagram shows the index to each element of the array. The top line shows the address of each contiguous byte in the *array*. Each cell in the middle row will contain the data of the element in the *array*.

So, if the array is called *age* and the data type is *int*, each array element is 4 bytes. The start of the array is at address 2001. You can address the 3rd element of the array as follows

```
Age[2]
```

This will result in the calculation of the starting address of the 3rd element of the array as follows

```
2001 + (4 * 2) = 2009
```

As you can see, 2009 is the starting address of the 3rd element of the array.

The advantage of using arrays and the *base-indexed* form of addressing in arrays is that the array elements can be processed efficiently using loops.

Let us see an example of this.

Problem

Input the age of 10 students and find the largest and smallest age.

Pseudocode

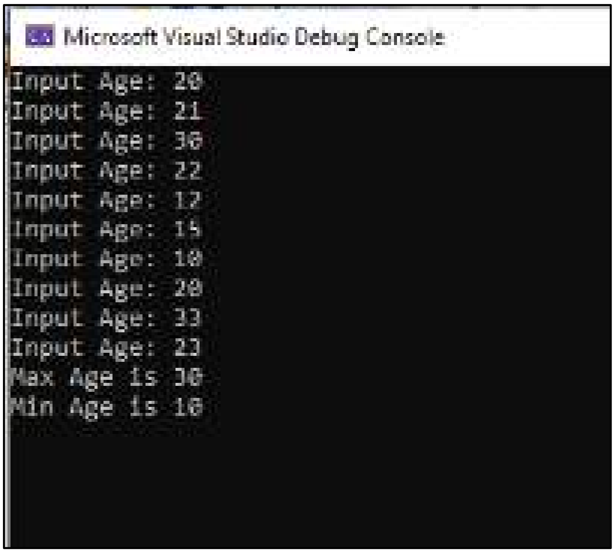
```
1.  Initialize MaxAge, MinAge to zero
2.  Initialize Age to 0
3.  Initialize an array Ages[10]
4.  For num = 0; num < 10; num++
5.      Input “Enter Age: “;
6.      Get Age
7.      Save Age to array Ages[num]
8.  EndFor
9.  Store first element of the Ages array in MaxAge and MinAge
10. For num = 1; num < 10; num++
11.     If (maxAge < ages[num])
12.         maxAge = ages[num];
13.     If (minAge > ages[num])
14.         minAge = ages[num];
15. EndFor
16. Print maxAge & minAge
17. End
```

Program

```c
#include <stdio.h>
#define MAX(a, b) (a > b) ? a : b
#define MIN(a, b) (a < b) ? a : b

int main(int argc, char* argv[])
{
    int maxAge, minAge;
    int age;
    int ages[10];
    int num;

    for (num = 0; num < 10; num++)
    {
        printf("Input Age: ");
        scanf("%d", &age);
14.         ages[num] = age;
15.     }

16.     maxAge = minAge = ages[0];

17.     for (num = 1; num < 8; num++)
18.     {
19.         maxAge = MAX(maxAge, ages[num]);
20.         minAge = MIN(minAge, ages[num]);
21.     }

22.     printf("Max Age is %d\n", maxAge);
23.     printf("Min Age is %d\n", minAge);
24. }
```

In the example above, the array *ages* is declared as an array of 10 integers. The array is initialized in the for loop in line 14, with data entered from the keyboard. The array is then read one element at a time, and compared with the *max* and *min* values, to find the largest and smallest values.

Output



```
Input Age: 20
Input Age: 21
Input Age: 30
Input Age: 22
Input Age: 12
Input Age: 15
Input Age: 10
Input Age: 20
Input Age: 33
Input Age: 23
Max Age is 30
Min Age is 10
```

The array elements are accessed using index values in variables as well as constants. Note the following

1. Arrays in 'C' have zero-based indexes.

2. 'C' does not do any bounds checking when elements of the array are accessed. Thus, you can access memory locations that are not allocated to the elements of the array, causing *run-time errors* in your program

3. The array elements can be accessed randomly. i.e., any element of the array can be accessed

Multi-dimensional arrays

Arrays declared in the previous sections are single-dimensional arrays. It means it is a single row of multiple columns. Each column contains an element of the array. As we have seen, this row occupies contiguous memory.

A multi-dimensional array is an array of arrays.

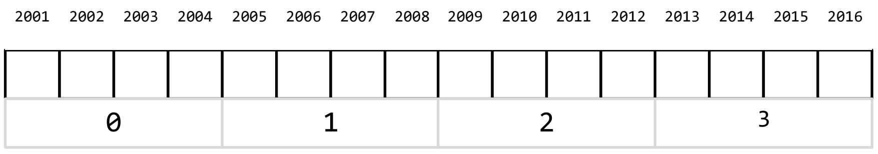
A 2-dimensional array can be considered to be an array of one-dimensional arrays. In this array, there are multiple rows, and each row contains multiple columns. Note that all rows will have the same number of columns.

A 3-dimensional array is an array of two-dimensional arrays.

Below diagrams show how these multi-dimensional arrays are represented

Single dimensional array

```
int arrname[4];
```



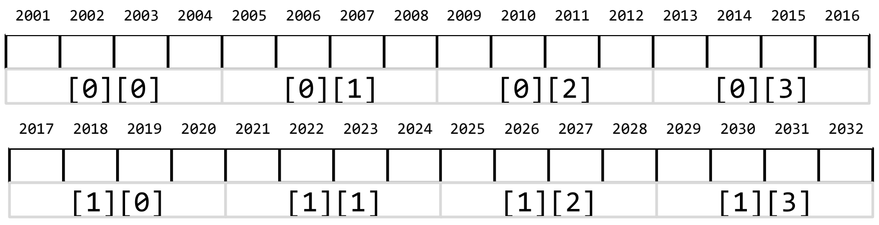
Here the array declared is a single-dimensional array with four elements. The array name is the starting address of the array, *arrname* with one subscript refers to an element of the array.

2- dimensional array

```
int arrname[2][4];
```

This array declaration declares a 2- dimensional array, with two rows and four columns. Just like a single-dimensional array, it is allocated contiguous bytes of memory. To access an element of the array, you need to specify two indexes, The row index and the column index. As shown below

arrname[0][2] – Access the first row, 3rd element

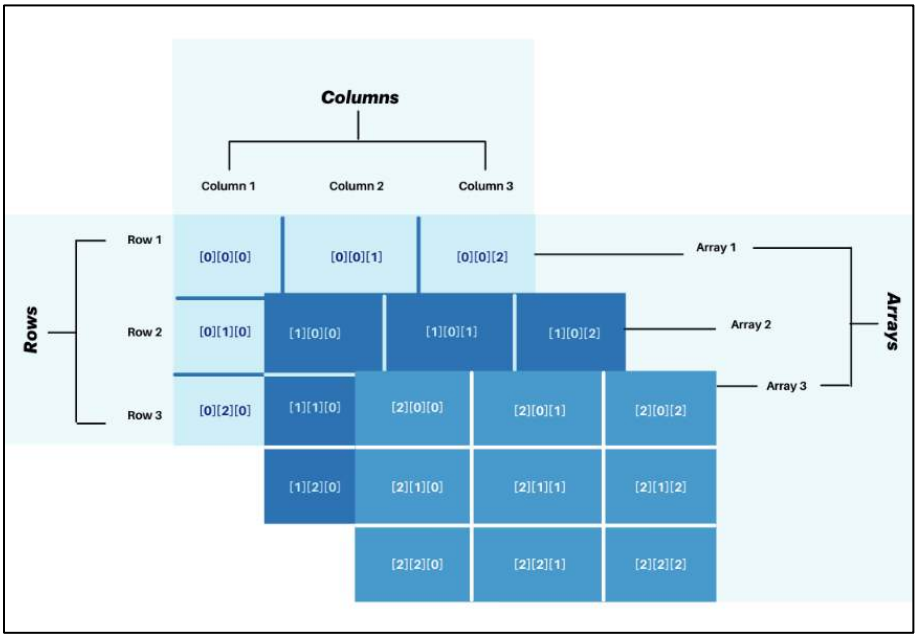


Here, you require two subscripts to access an element of the array. The array name refers to the starting address of the array, array name and one subscript (i.e., *arrname*[1]) refers to the starting address of a row, and array name and two subscripts refer to an element of the array.

3-dimensional arrays

```
int arrname[2][2][4];
```

A 3-dimensional array is an array of two-dimensional arrays. It requires three indexes to reference an element of the array. A three-dimensional array can be considered to be a cube. Here each two-dimensional array is placed one behind the other, as in a cube.



A 3-dimensional array is also allocated contiguous memory and is represented in memory, as shown below.

2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016
[0][0][0]				[0][0][1]				[0][0][2]				[0][0][3]			

2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032
[0][1][0]				[0][1][1]				[0][1][2]				[0][1][3]			

2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048
[1][0][0]				[1][0][1]				[1][0][2]				[1][0][3]			

2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063	2064
[1][1][0]				[1][1][1]				[1][1][2]				[1][1][3]			

To access an element in a 3-dimensional array, you need to have three indexes, the 1st index is the index to the 2-dimensional array, the 2nd index is the row of the 2-dimensional array being accessed, and the 3rd index is the element of the row.

Initializing a multi-dimensional array

A multi-dimensional array is initialized as shown below

```
int ages [2][3] = {
                   {20,30,40},
                   {21,31,41}
                  };
```

Here each row is initialized with its initializer list, contained within an outer initializer list. A comma separates each item in the initializer list (,).

Let's demonstrate the usage of a two-dimensional array with a more complex example.

Problem

Input the marks of 10 students in the 1st and 2nd grades, and display the largest and smallest marks for each grade.

Pseudocode

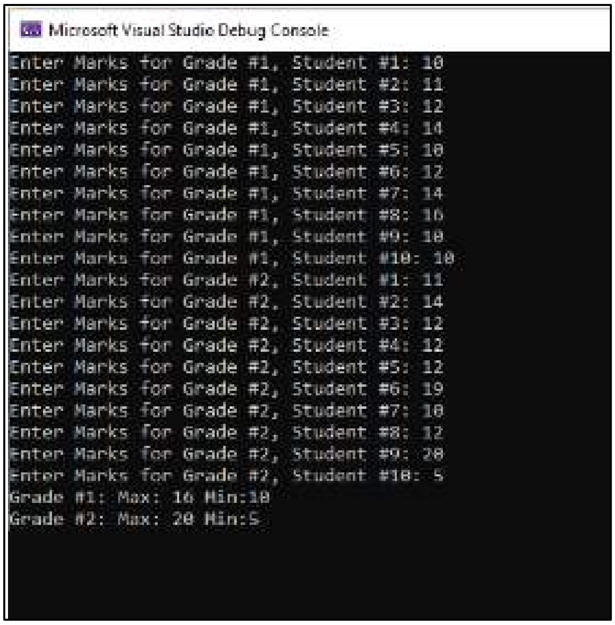
```
int marks[2][10]; /* Array to store marks for 10
                     students, and 2 grades */
int maxMarks[2];
int minMarks[2];
int grade, student, marksVal;

For (grade = 0; grade < 2; grade++) /* For 2 grades */
     /* Nested loop for 10 students in each grade */
     For (student = 0; student < 10; student++)
         Input "Enter Marks: ";
         Get marksVal
         Save marksVal in marks[grade][student]
     EndFor
EndFor

For (grade = 0; grade < 2; grade++) /* for 2 grades */
     maxMarks[grade] = marks[grade][0];
     For (student = 1; student < 10; student++)
```

```
        If (maxMarks[grade] < marks[grade][student])
            maxMarks[grade] = marks[grade][student];
        If (minMarks[grade] < marks[grade][student])
            minMarks[grade] = marks[grade][student];
EndFor

For (grade = 0; grade < 2; grade++)
    Print Grade, maxMarks[grade] & minMarks[grade]
EndFor

End
```

Program

```c
#include <stdio.h>
#define MAX(a, b) (a > b) ? a : b
#define MIN(a, b) (a < b) ? a : b


int main(int argc, char* argv[])
{
    /* 2-dim array to store the marks for 10 students in 2 grades
    */

    int marks[2][10];
    int maxMarks[2]; /* Array to store largest marks */
    int minMarks[2]; /* Array to store lowest marks */
    int grade, student, marksVal;

    /* Input the marks of 10 students in 2 grades, and store in
    /  array The nested loop below, is used to input the marks for
    /  each grade The outer loop is for the 2 grades
    /  The inner loop is for the 10 students
    */
    for (grade = 0; grade < 2; grade++)
    {
        for (student = 0; student < 10; student++)
        {
            printf("Enter Marks for Grade #%d, Student #%d: ",
                 grade + 1, student + 1);
            scanf("%d", &marksVal);
            marks[grade][student] = marksVal;
        }
    }

    /* Find the largest and smallest marks in each grade
    /  The nested loop below, is used to find the max and min
    /  values for each grade The outer loop is for the 2 grades
    /  The inner loop is for the 10 students
    */
    for (grade = 0; grade < 2; grade++)
    {
        minMarks[grade] = maxMarks[grade] = marks[grade][0];
        for (student = 1; student < 10; student++)
        {
            maxMarks[grade] = MAX(maxMarks[grade],
                                  marks[grade][student]);
            minMarks[grade] = MIN(minMarks[grade],
                                  marks[grade][student]);
        }
    }

    /* Display the max and min values for each grade */
    for (grade = 0; grade < 2; grade++)
    {
        printf("Grade #%d: Max: %d Min:%d\n", grade + 1,
                        maxMarks[grade], minMarks[grade]);
    }
}
```

Output


Microsoft Visual Studio Debug Console

```
Enter Marks for Grade #1, Student #1: 10
Enter Marks for Grade #1, Student #2: 11
Enter Marks for Grade #1, Student #3: 12
Enter Marks for Grade #1, Student #4: 14
Enter Marks for Grade #1, Student #5: 10
Enter Marks for Grade #1, Student #6: 12
Enter Marks for Grade #1, Student #7: 14
Enter Marks for Grade #1, Student #8: 16
Enter Marks for Grade #1, Student #9: 10
Enter Marks for Grade #1, Student #10: 10
Enter Marks for Grade #2, Student #1: 11
Enter Marks for Grade #2, Student #2: 14
Enter Marks for Grade #2, Student #3: 12
Enter Marks for Grade #2, Student #4: 12
Enter Marks for Grade #2, Student #5: 12
Enter Marks for Grade #2, Student #6: 19
Enter Marks for Grade #2, Student #7: 10
Enter Marks for Grade #2, Student #8: 12
Enter Marks for Grade #2, Student #9: 20
Enter Marks for Grade #2, Student #10: 5
Grade #1: Max: 16 Min:10
Grade #2: Max: 20 Min:5
```

As shown in the example above, processing a 2-dimensional array typically involves nested loops, the outer loop to process each row, and the inner loop to process the elements in each row.

Note that loop control statements, like *break* and *continue*, will work within the context of the loop it is written in.

Summary

The simplest form of a *data structure* is an *array*. It is a collection of multiple variables of the same data type that is functionally the same. Arrays are generally processed using loops to iterate across each element of the array. This chapter has seen how to declare an array, initialize it, and iterate through it.

In the next chapter, we will understand how to create user-defined data types using *structures* and *unions*.