SDN150S – SOFTWARE DESIGN 1

ARRAYS

DEPARTMENT OF ELECTRICAL ELECTRONIC AND COMPUTER ENGINEERING
CAPE PENINSULA UNIVERSITY OF TECHNOLOGY

Cape Peninsula University of Technology

OUTLINE

- What is an Array?
- Declaring, Initializing, & Accessing Array Elements
- Arrays Concepts
- Multidimensional Arrays
- Two-dimensional & Three-dimensional Arrays
- Passing Arrays to Function
- Return Array from Function
- Pointer to an Array
- Array Exercises

WHAT IS AN ARRAY?

- An Array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type.
- Although an array is used to store a collection of data, it is often more useful to think of an array as a collection of variables of the same type.
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.
- A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

DECLARING ARRAYS (1 OF 2)

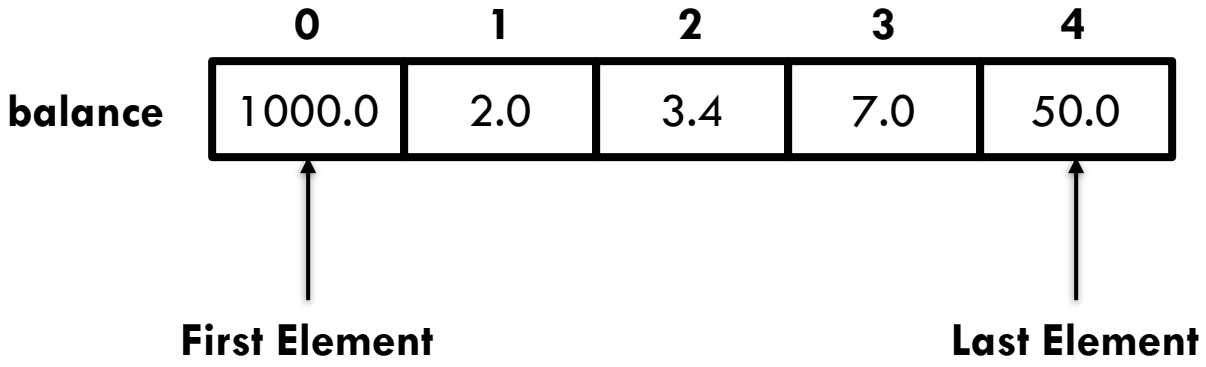
- To declare an array in C, a programmer specifies the ***type of the elements*** and the ***number of elements*** required by an array as show below:
  - `type arrayName [ arraySize ];`
- This is called a single-dimensional array. The ***arraySize*** must be an integer constant greater than zero and ***array type*** can be any valid C data type.
- For example, to declare a 10-element array called balance of type double, use this statement: `double balance[10];`
- Here, balance is a variable array which is sufficient to hold up to 10 double numbers.

INITIALIZING ARRAYS (1 OF 2)

- You can initialize an array in C either one by one or using a single statement as shown below, respectively:
  - `double balance[0] = 1000.0;`
  - `double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};`
- Note that the number of values between braces { } cannot be larger than the number of elements declared for the array (number between square brackets [ ]).
- If you omit the size of the array, you then have an array just big enough to hold the size of the initialized array created as shown:
  - `double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};`
- Note that you will create exactly the same array as you did in the previous example.

INITIALIZING ARRAYS (2 OF 2)

- All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1.
- As already discussed, the diagram below shows the pictorial representation of the array balance:



ACCESSING ARRAY ELEMENTS (1 OF 2)

- An element is accessed by indexing the array name.
- This is done by placing the index of the element within square brackets after the name of the array. For example:
  - `double salary = balance[5];`
- The above statement will take the 5th element from the array and assign the value to salary variable as shown in the code below.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.    double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
5.    double salary=balance[4]; // accessing the array
6.    printf("R%.2f",salary);   // print output
7.    return 0;
8.  }
```

OUTPUT
R50.00

ACCESSING ARRAY ELEMENTS (2 OF 2)

- The following example shows how to use all the three above-mentioned concepts viz. declaration, Initialization, and accessing arrays:

```
1.  #include <stdio.h>
2.  int main ()
3.  {
4.      int n[ 10 ]; /* n is an array of 10 integers */
5.      int i,j;
6.      /* initialize elements of array n to 0 */
7.      for ( i = 0; i < 10; i++ )
8.      {
9.      /* set element at location i to i + 100 */
10.         n[ i ] = i + 100;
11.     }
12.     /* output each array element's value */
13.     for (j = 0; j < 10; j++ )
14.     {
15.         printf("Element[%d] = %d\n", j, n[j] );
16.     }
17.     return 0;
18. }
```

OUTPUT

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

ARRAYS CONCEPTS

- The following concepts are important aspects of an array that a C programmer should know:

Concept	Description
Multidimensional arrays	C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Return array from a function	C allows a function to return an array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

MULTIDIMENSIONAL ARRAYS

- C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:
  - `type name[size1][size2]...[sizeN];`
- For example, If we want to visualize a two-dimensional (2D) array, we can visualize it as int arr[3][3].
- it means a 2D array of type integer having 3 rows and 3 columns. Shown as a simple matrix below

```
1. int arr[3][3];  //2D array containing 3 rows and 3 columns
```

```
1 2 3
4 5 6
7 8 9
    3x3
```

TWO-DIMENSIONAL ARRAYS (1 OF 3)

- A 2D array is a list of one-dimensional arrays. To declare a 2D integer array of size [size1][size2], you would write something as follows:
  - `type arr[size1][size2];`
  - Where type can be any valid C data type and arr will be a valid arrayName.
  - A 2D array (size1 & size2) can be considered as a table that has a number of rows and columns.
  - Example int arr[3][4] can be depicted as:

	Column 0	**Column 1**	**Column 2**	**Column 3**
Row 0	`arr[0][0]`	`arr[0][1]`	`arr[0][2]`	`arr[0][3]`
Row 1	`arr[1][0]`	`arr[1][1]`	`arr[1][2]`	`arr[1][3]`
Row 2	`arr[2][0]`	`arr[2][1]`	`arr[2][2]`	`arr[2][3]`

TWO-DIMENSIONAL ARRAYS (2 OF 3)

Initializing 2D Arrays:

- Multidimensional arrays may be initialized by specifying bracketed values for each row. For example the 3 by 4 array is initialized as follows:

```
1. int arr[3][4] = {
2. {0, 1, 2, 3} , /* initializers for row indexed by 0 */
3. {4, 5, 6, 7} , /* initializers for row indexed by 1 */
4. {8, 9, 10, 11} /* initializers for row indexed by 2 */
5. };
```

- The nested braces, which indicate the intended row, are optional. Thus the following initialization is equivalent to the previous example:

```
1. int arr[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

TWO-DIMENSIONAL ARRAYS (3 OF 3)

Accessing 2D Array Elements:

- An element in a 2D array is accessed by using the subscripts, i.e., row index and column index of the array.
  - For example, the following program uses a nested loop to handle and access a 2D array:

```
1. #include <stdio.h>
2. int main ()
3. {
4.      /* an array with 5 rows and 2 columns*/
5.      int arr[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
6.      int i, j;
7.      /* output each array element's value */
8.      for ( i = 0; i < 5; i++ )
9.      {
10.         for ( j = 0; j < 2; j++ )
11.         {
12.             printf("arr[%d][%d] = %d\n", i,j, arr[i][j] );
13.         }
14.     }
15.     int val = arr[2][1]; /* accessing the array */
16.     printf("val = %d",val);
17.     return 0;}
```

OUTPUT

```
arr[0][0] = 0
arr[0][1] = 0
arr[1][0] = 1
arr[1][1] = 2
arr[2][0] = 2
arr[2][1] = 4
arr[3][0] = 3
arr[3][1] = 6
arr[4][0] = 4
arr[4][1] = 8
val = 4
```

THREE-DIMENSIONAL ARRAYS (1 OF 3)

- In a three-dimensional (3D) array the syntax can be write as:
  - type arrayName[block_size][row_size][column_size];
  - Example: `int arr[3][3][3]`
  - Where int specifies the array data type and arr indicate the array name.
  - First dimension representing the block size (total number of 2D arrays),
  - Second dimension represents the rows of 2D arrays.
  - Third dimension represents the columns of 2D arrays.
- `int arr[3][3][3];  //3D array`

block(1) 1 2 3
4 5 6
7 8 9
3x3

block(2) 10 11 12
13 14 15
16 17 18
3x3

block(3) 19 20 21
22 23 24
25 25 27
3x3

THREE-DIMENSIONAL ARRAYS (2 OF 3)

Types of Declaration for a 3D array:

- int arr[2][3][3]; // No assignment

```
block(1)  1221 -543 3421        block(2)  654 5467 -878     //all values are
          3342 6543 4221                  456 1567 7890    //garbage values
          -564 4566 -345                  567 6561 2433
```

- Int arr[2][3][3]={}; // Empty assignment

```
block(1)  0 0 0       block(2)  0 0 0 // 0 will be stored
          0 0 0                 0 0 0
          0 0 0                 0 0 0
```

- Int arr[3][2][2]={0,1,2,3,4,5,6,7,8.9,3,2}; // List assignment

```
block(1)  0 1       block(2)  4 5       block(3)  8 9
          2 3                 6 7                 3 2
```

- Int arr[3][3][3]= {{{10,20,30},{40,50,60},{70,80,90}},
  {{11,22,33},{44,55,66},{77,88,99}},
  {{12,23,34},{45,56,67},{78,89,90}}};
  // block assignment

see example in previous slide

THREE-DIMENSIONAL ARRAYS (3 OF 3)

- **Inserting 3D Array Elements:**

```
1. #include<stdio.h>
2. int i,j,k; //variables for nested for loops
3. int main(){
4. int arr[2][3][3]; //array declaration
5. printf("enter the values in the array: \n");
6. for(i=1;i<=2;i++) //represents block
7. {
8. for(j=1;j<=3;j++) //represents rows
9. {
10.for(k=1;k<=3;k++) //represents columns
11.{
12.printf("the value at arr[%d][%d][%d]: ",i,j,k);
13.scanf("%d",&arr[i][j][k]);
14.}}}
15.printf("printing the values in array: \n");
16.for(i=1;i<=2;i++)
17.{
18.for(j=1;j<=3;j++)
19.{
20.for(k=1;k<=3;k++)
21.{
22.printf("%d ",arr[i][j][k]);
23.if(k==3)
24.{
25.printf("\n");
26.}}}
27.printf("\n");
28.}
29.return 0;}
```

OUTPUT

```
enter the values in the array:
the value at arr[1][1][1]: 1
the value at arr[1][1][2]: 2
the value at arr[1][1][3]: 3
the value at arr[1][2][1]: 4
the value at arr[1][2][2]: 5
the value at arr[1][2][3]: 6
the value at arr[1][3][1]: 7
the value at arr[1][3][2]: 8
the value at arr[1][3][3]: 9
the value at arr[2][1][1]: 09
the value at arr[2][1][2]: 34
the value at arr[2][1][3]: 56
the value at arr[2][2][1]: 78
the value at arr[2][2][2]: 89
the value at arr[2][2][3]: 12
the value at arr[2][3][1]: 34
the value at arr[2][3][2]: 26
the value at arr[2][3][3]: 76
printing the values in array:
1 2 3
4 5 6
7 8 9

9 34 56
78 89 12
34 26 76
```

PASSING ARRAYS TO FUNCTIONS (1 OF 2)

- When passing a single or multidimensional array as an argument in a function, you would have to declare a formal parameter in one of three ways.
- Note that either way will produces similar results since they all tell the compiler that an integer pointer is going to be received.

AS POINTER:	AS SIZED ARRAY:	AS UNSIZED ARRAY:

`void myFunction(int *param)`
`{`
 	
`// body of function`
 	
`}`
	`void myFunction(int param[10])`
	`{`
	
	`// body of function`
	
	`}`
		`void myFunction(int param[])`
		`{`
		
		`// body of function`
		
		`}`

PASSING ARRAYS TO FUNCTIONS (2 OF 2)

- The example below shows a function that takes an array as an argument along with another argument and based on the passed arguments, to returns the average of the numbers passed:

```
1.  double getAverage(int arr[], int size)
2.  {
3.  int i;
4.  double avg;
5.  double sum;
6.  for (i = 0; i < size; ++i)
7.  {
8.  sum += arr[i];
9.  }
10. avg = sum / size;
11. return avg;
12. }
```

```
1.  #include <stdio.h>
2.  /* function declaration */
3.  double getAverage(int arr[], int size);
4.  int main ()
5.  {
6.  /* an int array with 5 elements */
7.  int balance[5] = {1000, 2, 3, 17, 50};
8.  double avg;
9.  /* pass pointer to the array as an argument */
10. avg = getAverage( balance, 5 ) ;
11. /* output the returned value */
12. printf( "Average value is: %f ", avg );
13. return 0;
14. }
```

OUTPUT

Average value is: 214.400000

RETURN ARRAY FROM A FUNCTION (1 OF 2)

- C programming does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.
- If you want to return a single-dimensional array from a function, you would have to declare a function returning a pointer as shown:

```
int * myFunction()
{
// body of function
}
```

- Usually, C does not return the address of a local variable to outside of the function, so you would have to define the local variable as static variable.

RETURN ARRAY FROM A FUNCTION (2 OF 2)

- The example below considers a function that generate 10 random numbers and return them using an array from a function as follows:

```c
#include <stdio.h>
/* function to generate and return random numbers */
int * getRandom( )
{
static int r[10];
int i;
/* set the seed */
srand( (unsigned)time( NULL ) );
for ( i = 0; i < 10; ++i)
{
r[i] = rand();
printf( "r[%d] = %d\n", i, r[i]);
}
return r;
}
```

```c
/* main function to call above defined function */
int main ()
{
/* a pointer to an int */
int *p;
int i;
p = getRandom();
for ( i = 0; i < 10; i++ )
{
printf( "*(p + %d) : %d\n", i, *(p + i));
}
return 0;
}
```

OUTPUT

```
r[0] = 1557439844
r[1] = 782405888
r[2] = 870051935
r[3] = 1343986372
r[4] = 1065976373
r[5] = 531241767
r[6] = 2114540085
r[7] = 378826792
r[8] = 1505349601
r[9] = 1571411560
*(p + 0) : 1557439844
*(p + 1) : 782405888
*(p + 2) : 870051935
*(p + 3) : 1343986372
*(p + 4) : 1065976373
*(p + 5) : 531241767
*(p + 6) : 2114540085
*(p + 7) : 378826792
*(p + 8) : 1505349601
*(p + 9) : 1571411560
```

POINTER TO AN ARRAY (1 OF 3)

- An array name is a constant pointer to the first element of the array double balance[5].
- This means during declaration, the array named ***balance*** will be pointer to &balance[0], which is the address of the first element of the array.

```
double *p;
double balance[10];
p = balance;
```

- It is therefore okay to use array names as constant pointers, and vice versa.
- Since *(balance + 4) is a legitimate way of accessing the data at balance[4].

POINTER TO AN ARRAY (2 OF 3)

- Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2), and so on. The example below show all the concepts discussed above:

```
1. #include <stdio.h>
2. int main ()
3. {
4. /* an array with 5 elements */
5. double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
6. double *p;
7. int i;
8. p = balance;
9. /* output each array element's value */
10.printf( "Array values using pointer\n");
11.for ( i = 0; i < 5; i++ )
12.{
13.printf("*(p + %d) : %.2f\n", i, *(p + i) );
14.}
15.printf( "Array values using balance as address\n");
16.for ( i = 0; i < 5; i++ )
17.{
18.printf("*(balance + %d) : %.2f\n", i, *(balance + i) );
19.}
20.return 0;
21.}
```

OUTPUT

```
Array values using pointer
*(p + 0) : 1000.00
*(p + 1) : 2.00
*(p + 2) : 3.40
*(p + 3) : 17.00
*(p + 4) : 50.00
Array values using balance as
address
*(balance + 0) : 1000.00
*(balance + 1) : 2.00
*(balance + 2) : 3.40
*(balance + 3) : 17.00
*(balance + 4) : 50.00
```

ARRAY EXERCISES

1. Write a program in C to store elements in an array and print it first use 1D-array and then a 2D-array structure using the values below:

   ```
   {45.5,76.2,89.0,21.4,34.4,90.0,23.2,45.2,87.4,29.5,37.3,22.2}
   ```

2. Write a program in C to separate odd and even integers from 1 – 30 in separate arrays

3. Write a C function to count all the negative elements in the array below.

   ```
   {-23,8,34,28,-9,-11,67,45,-87}
   ```

4. Write a C program to calculate determinant of the 3 x 3 matrix used in question 3.

5. Write a C program to add two matrices using multi-dimensional arrays.

6. Passing a sized array to a function, write a C program to find the maximum number in an array of elements.