Shastri 4th Semester

Computer Science

Unit: 4th

ARRAY IN C LANGUAGE

An array in C is a collection of variables of the same type, stored in contiguous memory locations. It allows you to store multiple values in a single variable and access them using an index. The index starts from 0.

For example, you can create an array of integers called **numbers** that can hold 5 integers by writing:

Example

```
int numbers[5];
```

then store values in each element of the array by specifying the index in square brackets:

Example

```
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

To access a specific element of the array, you can use the index in square brackets as well:

Example

```
printf("The second element of the array is: %d", numbers[1]);
```

This will output "The second element of the array is: 20".

You can also initialize the array when you define it:

Example

```
int numbers[] = {10, 20, 30, 40, 50};
```

You can use loops to iterate through the elements of an array, for example, a for loop:

Example

```
for(int i=0;i<5;i++)

printf("%d ",numbers[i]);
```

Arrays are useful when you need to work with a large number of similar data and can make your code more efficient and easier to read.

TYPES OF ARRAYS

There are several **types of arrays** in C, the most common are:

1. **One-dimensional arrays**: These are the most basic type of arrays in C. It can store multiple values of the same data type, in a single variable. It allows you to store a fixed number of elements, and each element is identified by its index. Here's an example of a one-dimensional array of integers:

Example

```
int numbers[5] = {1, 2, 3, 4, 5};
```

One-dimensional Arrays: Array manipulation; Searching insertion

In C language, an array is a collection of elements of the same type, stored in contiguous memory locations. A one-dimensional array is a linear collection of elements, where each element can be accessed by its index.

Here are some common operations that can be performed on one-dimensional arrays:

- Array manipulation: This involves manipulating the elements of an array, such as traversing, sorting, or reversing the array. For example, the following code shows how to traverse an array and print its elements:

Example

```
int myArray[5] = {1, 2, 3, 4, 5};

 int size = sizeof(myArray) / sizeof(myArray[0]);

 for (int i = 0; i < size; i++)
```

```
{
printf("myArray[%d] = %d\n", i, myArray[i]);
}
```

- Searching: This involves finding an element in an array, given its value or index. For example, the following code shows how to search for an element in an array using linear search:

Example

```
int myArray[5] = {1, 2, 3, 4, 5};
int size = sizeof(myArray) / sizeof(myArray[0]);
 int searchValue = 3;
int index = -1;
for (int i = 0; i < size; i++)
{
 if (myArray[i] == searchValue)
 {
index = i; break; }
}
if (index != -1)
{
printf("The value %d is found at index %d\n", searchValue, index);
 }
 else
{
 printf("The value %d is not found in the array\n", searchValue);
}
```

- Insertion: This involves inserting an element into an array at a specific position. For example, the following code shows how to insert an element into an array at index 2:

Example

```
int myArray[5] = {1, 2, 3, 4, 5};
 int size = sizeof(myArray) / sizeof(myArray[0]);
 int newValue = 6; int index = 2;
for (int i = size - 1; i >= index; i--)
{
myArray[i + 1] = myArray[i];
}
 myArray[index] = newValue; size++;
```

The above insertion operation increases the size of the array by one, but in C language arrays are static, which means once the size of an array is defined it cannot be changed. It's possible to use dynamic memory allocation to overcome this limitation.

C language: Deletion of an element from an array

deletion of an element from an array refers to removing an element from a specific position in the array and shifting the remaining elements to fill the gap.

Here is an example of how to delete an element from an array at a specific position:

Example

```
int myArray[5] = {1, 2, 3, 4, 5};
int size = sizeof(myArray) / sizeof(myArray[0]);
int index = 2;
for (int i = index; i < size - 1; i++)
{
```

```
 myArray[i] = myArray[i + 1];
 }
size--;
```

In this example, an element located at index 2 is deleted. The for loop starts from the element located at the index specified for deletion and copies the next element to the current position, doing this for all the remaining elements, thus effectively deleting the element at the specified index.

It's worth noting that, the above deletion operation decreases the size of the array by one, but in C language arrays are static, which means once the size of an array is defined it cannot be changed. It's possible to use dynamic memory allocation to overcome this limitation.

It's important to note that, the deletion of an element from an array can also be done by using a library function like memmove() which is a part of the string.h library.

It's also worth noting that, when you delete an element from an array, the indexes of the elements following the deleted element change.

C Language: Finding the largest/smallest element in an array

Finding the largest or smallest element in an array is a common task that can be achieved using different algorithms. Here are two examples of how to find the largest and smallest element in an array using C language:

- Finding the largest element:

Example

```
int myArray[5] = {1, 2, 3, 4, 5};
 int size = sizeof(myArray) / sizeof(myArray[0]);
int largest = myArray[0];
for (int i = 1; i < size; i++)
{
 if (myArray[i] > largest)
{
```

```
largest = myArray[i];
}
 }
printf("The largest element is %d\n", largest);
```

In this example, we initialize a variable "largest" with the first element of the array, and then use a for loop to traverse through the array. In each iteration, we compare the current element of the array with the "largest" variable, if the current element is greater than the "largest" variable, we update the "largest" variable with the current element. At the end of the loop, the "largest" variable contains the largest element of the array.

- Finding the smallest element:

Example

```
int myArray[5] = {1, 2, 3, 4, 5};
int size = sizeof(myArray) / sizeof(myArray[0]);
int smallest = myArray[0];
for (int i = 1; i < size; i++)
 {
if (myArray[i] < smallest)
 {
smallest = myArray[i];
}
} printf("The smallest element is %d\n", smallest);
```

In this example, we initialize a variable "smallest" with the first element of the array and then use a for loop to traverse through the array. In each iteration, we compare the current element of the array with the "smallest" variable, if the current element is smaller than the "smallest" variable, we update the "smallest" variable with the current element. At the end of the loop, the "smallest" variable contains the smallest element of the array. both the above examples use a simple linear search algorithm

that has a time complexity of O(n), where n is the number of elements in the array. There are other algorithms like sorting that can be used to find the largest and smallest elements in an array with better time complexity.

2. **Multi-dimensional arrays**: These arrays can store multiple arrays and are also called a matrix. Each element in a multi-dimensional array is identified by its row and column number. They are also known as two-dimensional arrays or 2D arrays. Here's an example of a 2-dimensional array of integers:

Example

```
int matrix[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Two-dimensional arrays, Addition/Multiplication of two matrices

a two-dimensional array is an array of arrays. It is an array where each element is itself an array. Two-dimensional arrays are often used to represent matrices, which are used in many mathematical and scientific applications.

Here are examples of how to perform the addition and multiplication of two matrices using a two-dimensional array in C language:

- Addition of two matrices:

Example

```
#define ROW 3
#define COL 3
void addMatrices(int A[][COL], int B[][COL], int C[][COL])
{
for (int i = 0; i < ROW; i++)
 {
 for (int j = 0; j < COL; j++)
 { C[i][j] = A[i][j] + B[i][j];}
}
 }
```

In this example, we have defined two matrices A and B of size ROW x COL and a resultant matrix C. The function addMatrices takes these matrices as input and performs the addition of the corresponding elements of A and B and stores the result in C.

- Multiplication of two matrices:

Example

```
#define ROW1 3
 #define COL1 4
 #define ROW2 4
 #define COL2 2
 void multiplyMatrices(int A[][COL1], int B[][COL2], int C[][COL2])
{
for (int i = 0; i < ROW1; i++)
{
 for (int j = 0; j < COL2; j++)
{
 C[i][j] = 0; for (int k = 0; k < COL1; k++)
 {
 C[i][j] += A[i][k] * B[k][j];
}
 }
 }
 }
```

In this example, we have defined two matrices A and B of size ROW1 x COL1 and ROW2 x COL2 respectively, and a resultant matrix C. The function multiply matrices take these matrices as input and perform the multiplication of the matrices A and B and stores the result in C.

It's worth noting that, the number of columns of the first matrix must be equal to the number of rows of the second matrix for multiplication to be possible. Also, for matrices A, B and

Null terminated strings as array of characters

In C language, a string is a sequence of characters, and a null-terminated string is a special kind of string that is terminated with a null character ('\0'). A null-terminated string is often represented as an array of characters in C.

Here is an example of how to create and use a null-terminated string in C:

Example

```
char myString[] = "Hello, world!";

printf("The string is: %s\n", myString);
```

In this example, we have created a null-terminated string named "myString" and assigned the value "Hello, world!" to it. The string is an array of characters, and the last element of the array is the null character '\0', which indicates the end of the string.

We can also use the **%s** format specifier with the **printf()** function to print a null-terminated string. The **printf()** function automatically detects the null character and stops printing the characters after it.

We can also use the string library function to manipulate the null-terminated string like:

- strlen(string) which returns the length of the string
- strcat(string1,string2) which concatenates two strings
- strcmp(string1,string2) which compares two strings

It's worth noting that, In C language, a string is not a built-in data type, but an array of characters that ends with a null character. It's also worth noting that, when working with strings, we should be careful to avoid buffer overflows and other security vulnerabilities.

3. **Array of Array**: A Array of the array is an array of arrays, where each element of the main array is another array. They are also called arrays of arrays. The size of the sub-arrays can be different from each other.

Example

int Var[3][] = { {1, 2, 3}, {4, 5, 6, 7}, {8, 9}};

4. **Dynamic Array**: C programming language doesn't have built-in support for dynamic arrays but with the help of pointers, we can implement a dynamic array. Here, the user can add as many elements as they want to the array. The number of elements of the dynamic array can be changed at runtime.

All of these types of arrays have different use cases and each has its own set of advantages and disadvantages. Arrays are powerful tools for storing and manipulating data, but it's important to choose the right type of array for the specific task at hand to take advantage of their strengths and avoid their weaknesses.

some examples of arrays in C that demonstrate how they can be used:

1. **One-dimensional array example**: This example shows how to create and initialize a one-dimensional array of integers and how to access its elements:

   Example

   int numbers[5] = {1, 2, 3, 4, 5};

   printf("The first element of the array is: %d\n", numbers[0]);

   printf("The second element of the array is: %d\n", numbers[1]);

   This will output "The first element of the array is: 1" and "The second element of the array is: 2".

2. **Multi-dimensional array example**: This example shows how to create and initialize a 2-dimensional array of integers and how to access its elements using nested loops:

   Example

   int matrix[2][3] = { {1, 2, 3}, {4, 5, 6}};

   for (int i = 0; i < 2; i++) { for (int j = 0; j < 3; j++)

```
 {
printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
}
 }
```

This will output:

Example

```
matrix[0][0] = 1
 matrix[0][1] = 2
 matrix[0][2] = 3
matrix[1][0] = 4
matrix[1][1] = 5
 matrix[1][2] = 6
```

3. **Array of Array example**: This example shows how to create and initialize a var array and how to access its elements using nested loops:

Example

```
int var[3][] = { {1, 2, 3}, {4, 5, 6, 7}, {8, 9}};
 for (int i = 0; i < 3; i++)
{
 for (int j = 0; j < (sizeof(var[i])/sizeof(var[i][0])); j++)
 {
 printf("var[%d][%d] = %d\n", i, j, var[i][j]);
}
 }
```

This will output:

```
var[0][0] = 1
```

var[0][1] = 2

var[0][2] = 3

var[1][0] = 4

var[1][1] = 5

var[1][2] = 6

var[1][3] = 7

var[2][0] = 8

var[2][1] = 9

Dynamic Array example: This example shows how to create a dynamic array using pointers and malloc function and how to access its elements using pointer arithmetic

dynamic array in C:

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
 {
   // create a dynamic array of 5 integers
   int* dynamicArray = (int*)
malloc (5 * sizeof(int));
   // set the values of the array
   dynamicArray[0] = 10;
   dynamicArray[1] = 20;
   dynamicArray[2] = 30;
   dynamicArray[3] = 40;
   dynamicArray[4] = 50;
```

```
    // print the values of the array
    printf("The first element of the array is: %d\n", *dynamicArray);
    printf("The second element of the array is: %d\n", *(dynamicArray + 1));
    // resize the array to hold 10 integers
    dynamicArray = (int*) realloc(dynamicArray, 10 * sizeof(int));
    // set the values of the additional elements
    dynamicArray[5] = 60;
    dynamicArray[6] = 70;
    dynamicArray[7] = 80;
    dynamicArray[8] = 90;
    dynamicArray[9] = 100;
    // print the values of the additional elements
    printf("The sixth element of the array is: %d\n", *(dynamicArray + 5));
    printf("The seventh element of the array is: %d\n", *(dynamicArray + 6));
    // free the memory allocated for the array
    free(dynamicArray);
    return 0;
}
```

In this example, we first create a dynamic array of 5 integers by allocating memory using the malloc function. We then set the values of the array using pointer arithmetic. Next, we resize the array to hold 10 integers using the realloc function and set the values of the additional elements using pointer arithmetic. Finally, we print the values of the array and free the memory allocated for the array using the free function. It's important to note that while dynamic arrays are more flexible than static arrays, they also require more memory overhead. To ensure good performance, you should use dynamic arrays only when necessary and make sure to carefully manage the memory that you allocate.

Some Questions for practice

1. What is an array in C?
2. How do you declare an array in C?
3. What are the different types of arrays in C?
4. How do you initialize an array in C?
5. How do you access elements of an array in C?
6. How do you use the index operator in C arrays?
7. What is the difference between a one-dimensional and a two-dimensional array in C?
8. How do you use a multi-dimensional array in C?
9. How do you pass an array to a function in C?
10. How do you return an array from a function in C?
11. How do you use the sizeof operator with arrays in C?
12. How do you use the for loop with arrays in C?
13. How do you use the while loop with arrays in C?
14. How do you use the do-while loop with arrays in C?
15. How do you use the strlen function with arrays in C?
16. How do you use the memset function with arrays in C?
17. How do you use the memcpy function with arrays in C?
18. How do you use the qsort function with arrays in C?
19. How do you use the bsearch function with arrays in C?
20. What are the best practices for using arrays in C?