**VISION INSTITUTE OF MANAGEMENT**

**C PROGRAMMING**

**UNIT-1(ARRAYS)**

* **Arrays:**
* C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but 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 can 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.

First element Last element

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number[ 0] | Number [1] | Number[ 2] | Number[3] | ------------- |

* **Declaring Arrays:**

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

 *Datatype arrayName [ arraySize ];*

This is called a single-dimensional array. The arraySize must be an integer constant greater than zero and 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];*

Now balance is a variable array which is sufficient to hold up to 10 double numbers.

* **Initializing Arrays:**

It is possible to initialize an array during declaration. For example,

int mark[5] = {19, 10, 8, 17, 9};

 The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

 If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

 *double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};*

* **You can also initialize an array like this-**

 int mark[] = {19, 10, 8, 17, 9};

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.

 

Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

## Input and Output Array Elements:

Here's how you can take input from the user and store it in an array element.

 // take input and store it in the 3rd element

​ scanf("%d", &mark[2]);

 // take input and store it in the ith element

 scanf("%d", &mark[i-1]);

* **Accessing Array Elements:**
* You can access elements of an array by indices.
* Suppose you declared an array mark as above. The first element is mark[0], the second element is mark[1] and so on.



* **Few keynotes**:
* Arrays have 0 as the first index, not 1. In this example, mark[0] is the first element.
* If the size of an array is n, to access the last element, the n-1 index is used. In this example, mark[4].
* Suppose the starting address of mark[0] is **2120d**. Then, the address of the mark[1] will be **2124d**. Similarly, the address of mark[2] will be **2128d** and so on.
This is because the size of a float is 4 bytes.

Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

#include<stdio.h>

int main ()

{

 int n[ 10 ]; /\* n is an array of 10 integers \*/

 int i , j;

 /\* initialize elements of array n to 0 \*/

for ( i = 0; i < 10; i++ )

 {

 n[ i ] = i + 100; /\* set element at location i to i + 100 \*/

 }

 /\* output each array element's value \*/

 for (j = 0; j < 10; j++ )

{

 printf("Element[%d] = %d\n", j, n[j] );

}

 return 0;

}

When the above code is compiled and executed, it produces the following result:

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

## Example 1: Calculate Average:

// Program to find the average of n numbers using arrays

#include <stdio.h>

int main()

{

 int marks[10], i, n, sum = 0, average;

 printf("Enter number of elements: ");

 scanf("%d", &n);

 for(i=0; i<n; ++i)

 {

 printf("Enter number%d: ",i+1);

 scanf("%d", &marks[i]);

 // adding integers entered by the user to the sum variable

 sum += marks[i];

 }

 average = sum/n;

 printf("Average = %d", average);

 return 0;

}

**Output:**

Enter n: 5

Enter number1: 45

Enter number2: 35

Enter number3: 38

Enter number4: 31

Enter number5: 49

Average = 39

Here, we have computed the average of n numbers entered by the user.

* **Multi-Dimensional in C:**

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

 type name[size1][size2]...[sizeN];

**For example-** the following declaration creates a three dimensional 5, 10, 4 integer array:

 int ptr[5][10][4];

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

float x[3][4];

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.

 

Similarly, you can declare a three-dimensional (3d) array. For example,

float y[2][4][3];

Here, the array y can hold 24 elements.

* **Two-Dimensional Arrays:**

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x, y you would write something as follows:

 datatype arrayName [ x ][ y ];

Where datatype can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be think as a table which will have x number of rows and y number of columns.

Here is how you can initialize two-dimensional and three-dimensional arrays:

### Initialization of a 2d array:

// Different ways to initialize two-dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};

**Let us check below program where we have used nested loop to handle a two dimensional array:**

#include<stdio.h>

Int main()

{

 /\* an array with 5 rows and 2 columns\*/

 int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

int i, j;

 /\* output each array element's value \*/

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

 {

 for ( j = 0; j < 2; j++ )

 {

 printf("a[%d][%d] = %d\n", i,j, a[i][j] );

 }

 }

 return 0;

}

When the above code is compiled and executed, it produces the following result:

a[0][0]: 0

 a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

 a[3][1]: 6

a[4][0]: 4

 a[4][1]: 8

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

**Program 1: WAP to display the elements of a matrix of order 2X2.**

#include<stdio.h>

#include<conio.h>

Void main()

{

 int a[2][2],i,j;

Printf(“enter the value in matrix:”);

for(i=0;i<2;i++)

{

 for(j=0;j<2;j++)

 {

 scanf(“%d”,&a[i][j]);

 }

}

Printf(“Enter 2X2 matrix:”);

for(i=0;i<2;i++)

{

 for(j=0;j<2;j++)

 {

 printf(“%d”, a[i][j]);

 }

 printf(“/n”);

}

getch();

}

### Initialization of a 3d array:

You can initialize a three-dimensional array in a similar way like a two-dimensional array. Here's an example,

int test[2][3][4] = {

 {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},

 {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};

### Example 2: Sum of two matrices:

// C program to find the sum of two matrices of order 2\*2

#include <stdio.h>

int main()

{

 float a[2][2], b[2][2], result[2][2];

 // Taking input using nested for loop

 printf("Enter elements of 1st matrix\n");

 for (int i = 0; i < 2; ++i)

 for (int j = 0; j < 2; ++j)

 {

 printf("Enter a%d%d: ", i + 1, j + 1);

 scanf("%f", &a[i][j]);

 }

 // Taking input using nested for loop

 printf("Enter elements of 2nd matrix\n");

 for (int i = 0; i < 2; ++i)

 for (int j = 0; j < 2; ++j)

 {

 printf("Enter b%d%d: ", i + 1, j + 1);

 scanf("%f", &b[i][j]);

 }

// adding corresponding elements of two arrays

 for (int i = 0; i < 2; ++i)

 for (int j = 0; j < 2; ++j)

 {

 result[i][j] = a[i][j] + b[i][j];

 }

 // Displaying the sum

 printf("\nSum Of Matrix:");

 for (int i = 0; i < 2; ++i)

 for (int j = 0; j < 2; ++j)

 {

 printf("%.1f\t", result[i][j]);

 if (j == 1)

 printf("\n");

 }

 return 0;

}

**Output:**

Enter elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2 0.5

-0.9 25.0

# Pass arrays to a function in C:

In C programming, you can pass en entire array to functions. Before we learn that, let's see how you can pass individual elements of an array to functions.

### Passing individual array elements:

Passing array elements to a function is similar to [passing variables to a function](https://www.programiz.com/c-programming/c-user-defined-functions).

### Example 1: Passing an array:

#include <stdio.h>

void display(int age1, int age2)

{

 printf("%d\n", age1);

 printf("%d\n", age2);

}

int main()

{

 int ageArray[] = {2, 8, 4, 12};

 // Passing second and third elements to display()

 display(ageArray[1], ageArray[2]);

 return 0;

}

**Output:**

8

4

### Example 2: Passing arrays to functions:

// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>

float calculateSum(float age[]);

int main() {

 float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};

 // age array is passed to calculateSum()

 result = calculateSum(age);

 printf("Result = %.2f", result);

 return 0;

}

float calculateSum(float age[]) {

 float sum = 0.0;

 for (int i = 0; i < 6; ++i) {

 sum += age[i];

 }

 return sum;

}

**Output:**

Result = 162.50

To pass an entire array to a function, only the name of the array is passed as an argument.

result = calculateSum(age);

However, notice the use of [] in the function definition.

float calculateSum(float age[])

{

... ..

}

This informs the compiler that you are passing a one-dimensional array to the function.

* **Passing Multidimensional Arrays to a Function:**

To pass multidimensional arrays to a function, only the name of the array is passed to the function(similar to one-dimensional arrays).

### Example 3: Passing two-dimensional arrays:

#include <stdio.h>

void displayNumbers(int num[2][2]);

int main()

{

 int num[2][2];

 printf("Enter 4 numbers:\n");

 for (int i = 0; i < 2; ++i)

 for (int j = 0; j < 2; ++j)

 scanf("%d", &num[i][j]);

 // passing multi-dimensional array to a function

 displayNumbers(num);

 return 0;

}

void displayNumbers(int num[2][2])

{

 printf("Displaying:\n");

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

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

 printf("%d\n", num[i][j]);

 }

 }

}

**Output:**

Enter 4 numbers:

2

3

4

5

Displaying:

2

3

4

5

**Note:** In C programming, you can pass arrays to functions, however, you cannot return arrays from functions.

* **Sorting Algorithms:**
* We have seen in the last lecture that having a sort arrays can make it easier to do search. This suggests that it may be important to be able to take an unsorted an array and rearrange it so it’s sorted!
* There are many different algorithms for sorting: bucket sort, bubble sort, insertion sort, selection sort, heap sort, etc. This is testimony to the importance and complexity of the problem, despite its apparent simplicity. In this lecture we discuss selection sort, which is one of the simplest algorithms.
* **Sorting data:**
* There are three approaches to sorting arrays: selection sort, insertion sort, and bubble sort. As you will notice, whereas searching involves a single for loop and visiting each array location, sorting involves nested for loops, and n-1 passes through the array.
* In a **selection sort**, we start with the first position in the array, find the smallest value in a first pass through the array, and swap the value in the first position in the array with the value in the position of the newly found smallest value; then we take each subsequent position in the array, find the smallest value in the remaining array, and swap position.
* In an **insertion sort**, you start with one item, take a new item and sort the two items relative to each other, then take a new item and sort the three items relative to each other (swapping the new item with consecutive values until it is no longer lower, and thus inserting it in that position), and so on. It is like sorting a deck of cards with your hands.
* In a **bubble sort**, you swap neighbors; the larger items drop down while the smaller ones bubble up, in n-1 passes through the array.

**Unit 2(Pointers)**

* **C- Pointers:**
* Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer.
* As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand & operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

#include<stdio.h>

int main ()

 {

 int var1;

char var2[10];

printf("Address of var1 variable: %x\n", &var1 );

printf("Address of var2 variable: %x\n", &var2 );

 return 0;

}

When the above code is compiled and executed, it produces result something as follows:

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

* **What Are Pointers?**

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address.

### Pointer Syntax:

Here is how we can declare pointers.

int\* p;

Here, we have declared a pointer p of int type.

You can also declare pointers in these ways.

int \*p1;

int \* p2;

Let's take another example of declaring pointers.

int\* p1, p2;

Here, we have declared a pointer p1 and a normal variable p2.

### Assigning addresses to Pointers:

Let's take an example.

int\* pc, c;

c = 5;

pc = &c;

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

* **Get Value of Thing Pointed by Pointers:**

To get the value of the thing pointed by the pointers, we use the \* operator. For example:

int\* pc, c;

c = 5;

pc = &c;

printf("%d", \*pc); // Output: 5

Here, the address of c is assigned to the pc pointer. To get the value stored in that address, we used \*pc.

**Note:***In the above example,*pc*is a pointer, not \*pc. You cannot and should not do something like \*pc = &c;*

By the way, \* is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

### Changing Value Pointed by Pointers:

Let's take an example.

Int \* pc, c;

c = 5;

pc = &c;

c = 1;

printf("%d", c); // Output: 1

printf("%d", \*pc); // Output: 1

We have assigned the address of c to the pc pointer.

Then, we changed the value of c to 1. Since pc and the address of c is the same, \*pc gives us 1.

Let's take another example.

int\* pc, c;

c = 5;

pc = &c;

\*pc = 1;

printf("%d", \*pc); // Output: 1

printf("%d", c); // Output: 1

We have assigned the address of c to the pc pointer.

Then, we changed \*pc to 1 using \*pc = 1;. Since pc and the address of c is the same, c will be equal to 1.

Let's take one more example.

int\* pc, c, d;

c = 5;

d = -15;

pc = &c; printf("%d", \*pc); // Output: 5

pc = &d; printf("%d", \*pc); // Output: -15

Initially, the address of c is assigned to the pc pointer using pc = &c;. Since c is 5, \*pc gives us 5

Then, the address of d is assigned to the pc pointer using pc = &d;. Since d is -15, \*pc gives us -15.

### Example: Working of Pointers:

Let's take a working example.

#include <stdio.h>

int main()

{

 int\* pc, c;

 c = 22;

 printf("Address of c: %p\n", &c);

 printf("Value of c: %d\n\n", c); // 22

 pc = &c;

 printf("Address of pointer pc: %p\n", pc);

 printf("Content of pointer pc: %d\n\n", \*pc); // 22

 c = 11;

 printf("Address of pointer pc: %p\n", pc);

 printf("Content of pointer pc: %d\n\n", \*pc); // 11

 \*pc = 2;

 printf("Address of c: %p\n", &c);

 printf("Value of c: %d\n\n", c); // 2

 return 0;

}

**Output:**

Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

Content of pointer pc: 22

Address of pointer pc: 2686784

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2

* **NULL Pointers in C:**

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

#include<stdio.h>

int main ()

 {

 int \*ptr = NULL;

printf("The value of ptr is : %x\n", ptr );

 return 0;

}

When the above code is compiled and executed, it produces the following result:

The value of ptr is 0

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null zero value, it is assumed to point to nothing.

**Example:**

## Find the Largest Element in an array:

#include <stdio.h>

int main() {

 int i, n;

float arr[100];

printf("Enter the number of elements (1 to 100): ");

scanf("%d", &n);

for (i = 0; i < n; ++i)

{

printf("Enter number%d: ", i + 1);

scanf("%f", &arr[i]);

}

// storing the largest number to arr[0]

for (i = 1; i < n; ++i)

{

if (arr[0] < arr[i])

 arr[0] = arr[i];

}

printf("Largest element = %.2f", arr[0]);

return 0;

}

**Output:**

Enter the number of elements (1 to 100): 5

Enter number1: 34.5

Enter number2: 2.4

Enter number3: -35.5

Enter number4: 38.7

Enter number5: 24.5

Largest element = 38.70

This program takes n number of elements from the user and stores it in arr[].

* **Dynamic Memory Management:**
* We can dynamically allocate memory in your C program using standard library functions: malloc(), calloc(), free() and realloc().
* As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.
* Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
* To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file.

## C malloc()-

The name "malloc" stands for memory allocation.

The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a [pointer](https://www.programiz.com/c-programming/c-pointers) of void which can be casted into pointers of any form.

### Syntax of malloc():

ptr = (castType\*) malloc(size);

**Example:**

ptr = (float\*) malloc(100 \* sizeof(float));

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

## C calloc()-

The name "calloc" stands for contiguous allocation.

The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.

### Syntax of calloc()-

ptr = (castType\*)calloc(n, size);

**Example:**

 ptr = (float\*) calloc(25, sizeof(float));

The above statement allocates contiguous space in memory for 25 elements of type float.

## C free():

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

### Syntax of free():

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

## C realloc()-

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

### Syntax of realloc():

ptr = realloc(ptr, x);

Here, ptr is reallocated with a new size x.

**ASSIGNMENTS OF UNIT 1 AND UNIT 2:**

1. Write a program to store and print 12 values entered by the user.
2. Write a program to find the multiplication of two matrices.
3. Write the relationship between Pointers and Arrays with example.
4. WAP to find the transpose of matrix.
5. WAP to calculate the average of array elements.