Array of Structures in C

In C Programming, Structures are used to group different data types to organize the data in structural way and Arrays are used to group same data type values. In this article we will show you the Array of Structures in C concept with one practical example.

For example, we are storing employee details such as name, id, age, address and salary. Normally we group them as employee structure with the above mentioned members. We can create the structure variable to access or modify the structure members. A company may have 10 to 100 employee, how about storing the same for 100 employees?

In C Programming, We can easily solve the above mentioned problem by combining 2 powerful concepts Arrays and Structures in C. We can create the employee structure with the above mentioned members and then instead of creating the structure variable, we create the array of structure variable.

Declaring C Array of Structures at structure Initialization

```
/* Array of Structures in C Initialization */
struct Employee
{
  int age;
  char name[50];
  int salary;
} Employees[4] = {
                 {25, "Suresh", 25000},
                 {24, "Tutorial", 28000},
                 {22, "Gateway", 35000},
                 {27, "Mike", 20000}
                 };
```

Here, Employee structure is used for storing the employee details such as age, name and salary. We created the array of structures variable Employees [4] (with size 4) at the declaration time only. We also initialized the values of each and every structure member for all the 4 employees.

From the above,

```
Employees[0] = {25, "Suresh", 25000}

Employees[1] = {24, "Tutorial", 28000}

Employees[2] = {22, "Gateway", 35000}

Employees[3] = {27, "Mike", 20000}
```

Declaring C Array of Structures in main() Function

```
/* Array of Structures in C Initialization */
struct Employee
{
  int age;
  char name[50];
  int salary;
};
```

Within the main() function, Create the Employee Structure Variable

```
struct Employee  Employees[4];
Employees[4] = {
                 {25, "Suresh", 25000},
                 {24, "Tutorial", 28000},
                 {22, "Gateway", 35000},
                 {27, "Mike", 20000}
                 };
```

Array of Structures in C Example

This program for Array of Structures in C will declare the student structure and displays the information of N number of students.

```
/* Array of Structures in C example */

#include <stdio.h>

struct Student
{
  char Student_Name[50];
  int C_Marks;
  int DataBase_Marks;
  int CPlus_Marks;
  int English_Marks;
};

int main()
{
  int i;
  struct Student Students[4] =
                {
                 {"Suresh", 80, 70, 60, 70},
                 {"Tutorial", 85, 82, 65, 68},
                 {"Gateway", 75, 70, 89, 82},
                 {"Mike", 70, 65, 69, 92}
                 };

  printf(".....Student Details....");
  for(i=0; i<4; i++)
  {
    printf("\n Student Name = %s", Students[i].Student_Name);
    printf("\n First Year Marks = %d", Students[i].C_Marks);
    printf("\n Second Year Marks = %d", Students[i].DataBase_Marks);
    printf("\n First Year Marks = %d", Students[i].CPlus_Marks);
    printf("\n Second Year Marks = %d", Students[i].English_Marks);
  }

 return 0;
}
```

Dynamic Memory Allocation

Allocating memory

There are two ways that memory gets allocated for data storage:

1. Compile Time (or static) Allocation
    - Memory for named variables is allocated by the compiler
    - Exact size and type of storage must be known at compile time
    - For standard array declarations, this is why the size has to be constant
2. Dynamic Memory Allocation
    - Memory allocated "on the fly" during run time
    - dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
    - Exact amount of space or number of items does not have to be known by the compiler in advance.
    - For dynamic memory allocation, pointers are crucial

Dynamic Memory Allocation

- We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"
- For this reason, dynamic allocation requires two steps:
    1. Creating the dynamic space.
    2. Storing its **address** in a **pointer** (so that the space can be accesed)
- To dynamically allocate memory in C++, we use the **new** operator.
- De-allocation:
    - Deallocation is the "clean-up" of space being used for variables or other data storage
    - Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)

- It is the programmer's job to deallocate dynamically created space
- To de-allocate dynamic memory, we use the `delete` operator

C – Dynamic memory allocation

DYNAMIC MEMORY ALLOCATION IN C:

The process of allocating memory during program execution is called dynamic memory allocation.

DYNAMIC MEMORY ALLOCATION FUNCTIONS IN C:

C language offers 4 dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

Function	Syntax
malloc ()	malloc (number *sizeof(int));
calloc ()	calloc (number, sizeof(int));
realloc ()	realloc (pointer_name, number * sizeof(int));
free ()	free (pointer_name);

1. MALLOC() FUNCTION IN C:

- malloc () function is used to allocate space in memory during the execution of the program.
- malloc () does not initialize the memory allocated during execution. It carries garbage value.
- malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

EXAMPLE PROGRAM FOR MALLOC() FUNCTION IN C:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    char *mem  allocation;
    /* memory is allocated dynamically */
    mem_allocation = malloc( 20 * sizeof(char) );
    if( mem_allocation== NULL )
    {
      printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
      strcpy( mem_allocation,"fresh2refresh.com");
    }
    printf("Dynamically allocated memory content : " \
          "%s\n", mem_allocation );
    free(mem_allocation);
}
```

2. CALLOC() FUNCTION IN C:

- calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't.

EXAMPLE PROGRAM FOR CALLOC() FUNCTION IN C:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = calloc( 20, sizeof(char) );
    if( mem_allocation== NULL )
    {
       printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
       strcpy( mem_allocation,"fresh2refresh.com");
    }
       printf("Dynamically allocated memory content   : " \
             "%s\n", mem_allocation );
       free(mem_allocation);
}
```

3. REALLOC() FUNCTION IN C:

- realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4. FREE() FUNCTION IN C:

- free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

EXAMPLE PROGRAM FOR REALLOC() AND FREE() FUNCTIONS IN C:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation,"fresh2refresh.com");
   }
   printf("Dynamically allocated memory content  : " \
         "%s\n", mem_allocation );
   mem_allocation=realloc(mem_allocation,100*sizeof(char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
      strcpy( mem_allocation,"space is extended upto " \
                        "100 characters");
   }
   printf("Resized memory : %s\n", mem  allocation );
   free(mem_allocation);
}
```

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

Static memory allocation	Dynamic memory allocation
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can't be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(20*sizeof(int));	Same as malloc () function int *ptr;ptr = (int*)calloc(20, sizeof(int));

Write C code to dynamically allocate one, two and three dimensional arrays (using malloc())

Write C code to dynamically allocate one, two and three dimensional arrays (using malloc())

Its pretty simple to do this in the C language if you know how to use C pointers. Here are some example C code snipptes….

One dimensional array

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int i, count = 0, no_of_elements=5;

        // Allocating dynamic array
        int *array = malloc(no_of_elements * sizeof(int));

        // Accessing one dimensional array
        for (i = 0; i<no_of_elements; i++)
            array[i] = ++count;  // OR *(arr+i) = ++count

        for (i = 0; i <  no_of_elements; i++)
            printf("%d ", array[i]);

        return 0;
}
```

Two dimensional array

There are more than one method to allocate two dimensional array dynamically but we use only pointer method.

```
#include <stdio.h>
#include <stdlib.h>

#define ROW      3
#define COLUMN   4

int main()
{
   int i, j, count=0;

        // Allocating two dimensional dynamic array
   int **arr = (int **)malloc(ROW * sizeof(int *));
   for (i=0; i<ROW; i++)
       arr[i] = (int *)malloc(COLUMN * sizeof(int));

   // Accessing two dimensional array
   for (i = 0; i <  ROW; i++)
   {
        for (j = 0; j < COLUMN; j++)
        {
                arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count
                printf("%d ", arr[i][j]);
        }
   }

   return 0;
}
```

Three dimensional array

```
#include<stdio.h>
#include<malloc.h>

#define AXIS_X 3
#define AXIS  Y 4
#define AXIS_Z 5

int main()
{
```

```
10          int ***p,i,j,k,count=0;
11
12          // Allocating three dimensional dynamic array
13          p=(int ***) malloc(AXIS_X * sizeof(int ***));
14
15          for(i=0;  i<AXIS_X;  i++)
16          {
17                      p[i]=(int **)malloc(AXIS_Y * sizeof(int *));
18
19                      for(j=0;  j<AXIS_Y;  j++)
20                      {
21                                  p[i][j]=(int *)malloc(AXIS_Z * sizeof(int));
22                      }
23          }
24
25          // Accessing three dimensional array
26          for(k=0;  k<AXIS  Z;  k++)
27          {
28                      for(i=0;  i<AXIS_X;  i++)
29                      {
30                                  for(j=0;  j<AXIS_Y;  j++)
31                                  {
32                                              *(*(*(p+i)+j)+k)= ++count;   // OR *(*(*(p+i)+j)+k) = ++count
33                                              printf("%d ",p[i][j][k]);
34                                  }
35                      }
36          }
37
38
39          return 0;
40 }
```

Program to add two polynomials

Given two polynomials represented by two arrays, write a function that adds given two polynomials.

Example:

```
Input:  A[] = {5, 0, 10, 6}
        B[] = {1, 2, 4}
Output: sum[] = {6, 2, 14, 6}


The first input array represents "5 + 0x^1 + 10x^2 + 6x^3"
The second array represents "1 + 2x^1 + 4x^2"
And Output is "6 + 2x^1 + 14x^2 + 6x^3"
```

Addition is simpler than multiplication of polynomials. We initialize result as one of the two polynomials, then we traverse the other polynomial and add all terms to the result.

```
add(A[0..m-1], B[0..n-1])
1) Create a sum array sum[] of size equal to maximum of 'm' and 'n'
2) Copy A[] to sum[].
3) Traverse array B[] and do following for every element B[i]
          sum[i] = A[i] + B[i]
4) Return sum[].
```

The following is implementation of above algorithm.

```
// Simple C++ program to add two polynomials
#include <iostream>
using namespace std;

// A utility function to return maximum of two integers
int max(int m, int n) {  return (m > n)? m: n; }

// A[] represents coefficients of first polynomial
// B[] represents coefficients of second polynomial
// m and n are sizes of A[] and B[] respectively
int *add(int A[], int B[], int m, int n)
{
   int size = max(m, n);
   int *sum = new int[size];

   // Initialize the sum polynomial
   for (int i = 0; i<m; i++)
     sum[i] = A[i];

   // Take ever term of first polynomial
   for (int i=0; i<n; i++)
       sum[i] += B[i];

   return sum;
}

// A utility function to print a polynomial
void printPoly(int poly[], int n)
{
    for (int i=0; i<n; i++)
    {
       cout << poly[i];
       if (i != 0)
        cout << "x^" << i ;
       if (i != n-1)
       cout << " + ";
    }
}


// Driver program to test above functions
int main()
{
    // The following array represents polynomial 5 + 10x^2 + 6x^3
    int A[] = {5, 0, 10, 6};
```

```
    // The following array represents polynomial 1 + 2x + 4x^2
    int B[] = {1, 2, 4};
    int m = sizeof(A)/sizeof(A[0]);
    int n = sizeof(B)/sizeof(B[0]);

    cout << "First polynomial is \n";
    printPoly(A, m);
    cout << "\nSecond polynomial is \n";
    printPoly(B, n);

    int *sum = add(A, B, m, n);
    int size = max(m, n);

    cout << "\nsum polynomial is \n";
    printPoly(sum, size);

    return 0;
}
```

Output:

```
First polynomial is

5 + 0x^1 + 10x^2 + 6x^3

Second polynomial is

1 + 2x^1 + 4x^2

Sum polynomial is

6 + 2x^1 + 14x^2 + 6x^3
```

Time complexity of the above algorithm and program is O(m+n) where m and n are orders of two given polynomials.