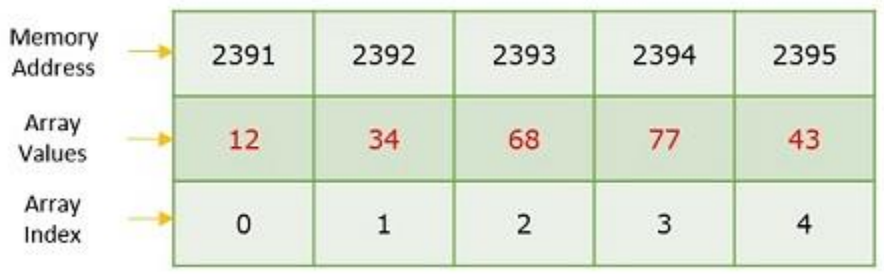
What is an Array?

An array is a type of linear data structure that is defined as a collection of elements with same or different data types. They exist in both single dimension and multiple dimensions. These data structures come into picture when there is a necessity to store multiple elements of similar nature together at one place.



The difference between an array index and a memory address is that the array index acts like a key value to label the elements in the array. However, a memory address is the starting address of free memory available.

Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Syntax

Creating an array in **C** and **C++** programming languages −

```
data_type array_name[array_size]={elements separated by commas}
or,
data_type array_name[array_size];
```

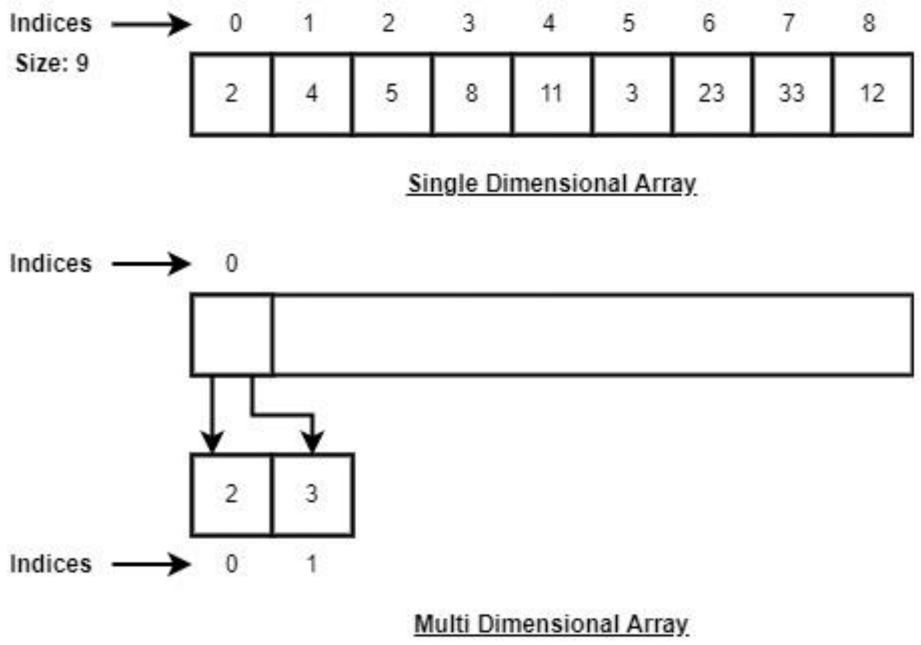
Creating an array in **Java** programming language −

```
data_type[] array_name = {elements separated by commas}
or,
data_type array_name = new data_type[array_size];
```

Array Representation

Arrays are represented as a collection of buckets where each bucket stores one element. These buckets are indexed from '0' to 'n-1', where n is the size of that particular array. For example, an array with size 10 will have buckets indexed from 0 to 9.

This indexing will be similar for the multidimensional arrays as well. If it is a 2-dimensional array, it will have sub-buckets in each bucket. Then it will be indexed as array_name[m][n], where m and n are the sizes of each level in the array.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 9 which means it can store 9 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 23.

Basic Operations in Arrays

The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update. These operations are usually performed to either modify the data in the array or to report the status of the array.

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.
- **Display** − Displays the contents of the array.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	**Default Value**
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Array - Insertion Operation

In the insertion operation, we are adding one or more elements to the array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. This is done using input statements of the programming languages.

Algorithm

Following is an algorithm to insert elements into a Linear Array until we reach the end of the array −

```
1. Start
2. Create an Array of a desired datatype and size.
3. Initialize a variable 'i' as 0.
4. Enter the element at ith index of the array.
5. Increment i by 1.
6. Repeat Steps 4 & 5 until the end of the array.
7. Stop
```

Example

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

C

Open Compiler

```
#include <stdio.h>
int main(){
  int LA[3] = {}, i;
  printf("Array Before Insertion:\n");
  for(i = 0; i < 3; i++)
    printf("LA[%d] = %d \n", i, LA[i]);
  printf("Inserting Elements.. \n");
  printf("The array elements after insertion :\n"); // prints array values
  for(i = 0; i < 3; i++) {
    LA[i] = i + 2;
    printf("LA[%d] = %d \n", i, LA[i]);
  }
  return 0;
}
```

Output

```
Array Before Insertion:
LA[0] = 0
LA[1] = 0
LA[2] = 0
Inserting elements..
Array After Insertion:
LA[0] = 2
LA[1] = 3
LA[2] = 4
LA[3] = 5
LA[4] = 6
```

For other variations of array insertion operation, click here.

Array - Deletion Operation

In this array operation, we delete an element from the particular index of an array. This deletion operation takes place as we assign the value in the consequent index to the current index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the K^th^ position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following are the implementations of this operation in various programming languages −

C

Open Compiler

```c
#include <stdio.h>
void main(){
  int LA[] = {1,3,5};
  int n = 3;
  int i;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++)
    printf("LA[%d] = %d \n", i, LA[i]);
  for(i = 1; i<n; i++) {
    LA[i] = LA[i+1];
    n = n - 1;
  }
  printf("The array elements after deletion :\n");
  for(i = 0; i<n; i++)
    printf("LA[%d] = %d \n", i, LA[i]);
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
The array elements after deletion :
LA[0] = 1
LA[1] = 5
```

Array - Search Operation

Searching an element in the array using a key; The key element sequentially compares every value in the array to check if the key is present in the array or not.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following are the implementations of this operation in various programming languages −

C C++JavaPython

Open Compiler

```c
#include <stdio.h>
void main(){
  int LA[] = {1,3,5,7,8};
  int item = 5, n = 5;
  int i = 0, j = 0;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
  for(i = 0; i<n; i++) {
    if( LA[i] == item ) {  {1,3,5,7,8}
      printf("Found element %d at position %d\n", item, i+1);
    }
  }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Array - Traversal Operation

This operation traverses through all the elements of an array. We use loop statements to carry this out.

Algorithm

Following is the algorithm to traverse through all the elements present in a Linear Array −

```
1  Start
2. Initialize an Array of certain size and datatype.
3. Initialize another variable ‘i’ with 0.
4. Print the ith value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End
```

Example

Following are the implementations of this operation in various programming languages −

C C++JavaPython

Open Compiler

```
#include <stdio.h>
int main(){
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n = 5;
  int i = 0, j = n;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Array - Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following are the implementations of this operation in various programming languages −

C C++JavaPython

Open Compiler

```c
#include <stdio.h>
void main(){
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5, item = 10;
  int i, j;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
  LA[k-1] = item;
  printf("The array elements after updation :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Array - Display Operation

This operation displays all the elements in the entire array using a print statement.

Algorithm

Consider LA is a linear array with N elements. Following is the algorithm to display an array elements.

```
1. Start
2. Print all the elements in the Array
3. Stop
```

Example

Following are the implementations of this operation in various programming languages −

Open Compiler

```c
#include <stdio.h>
int main(){
  int LA[] = {1,3,5,7,8};
  int n = 5;
  int i;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Multidimensional Arrays in C

Last Updated : 11 Mar, 2024

[redacted]

Prerequisite: Arrays in C

A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays is generally stored in row-major order in the memory.

The ***general form of declaring N-dimensional arrays*** is shown below.

Syntax:

```
data_type array_name[size1][size2]....[sizeN];
```

- **data_type**: Type of data to be stored in the array.
- **array_name**: Name of the array.
- **size1, size2,…, sizeN**: Size of each dimension.

Examples:

Two dimensional array: int two_d[10][20];

Three dimensional array: int three_d[10][20][30];

Size of Multidimensional Arrays:

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

- The array **int x[10][20]** can store total (10*20) = 200 elements.
- Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

To get the size of the array in bytes, we multiply the size of a single element with the total number of elements in the array.

For example:

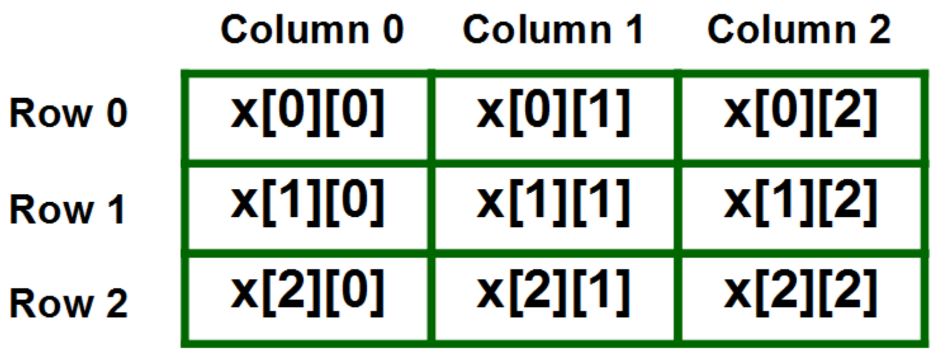
- Size of array **int x[10][20]** = 10 * 20 * 4 = 800 bytes.   (where int = 4 bytes)
- Similarly, size of **int x[5][10][20]** = 5 * 10 * 20 * 4 = 4000 bytes.   (where int = 4 bytes)

The most commonly used forms of the multidimensional array are:

1. **Two Dimensional Array**
2. **Three Dimensional Array**

Two-Dimensional Array in C

A **two-dimensional array** or **2D array** in C is the simplest form of the multidimensional array. We can visualize a two-dimensional array as an array of one-dimensional arrays arranged one over another forming a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and the column number ranges from 0 to (y-1).



Graphical Representation of Two-Dimensional Array of Size 3 x 3

Declaration of Two-Dimensional Array in C

The basic form of declaring a 2D array with **x** rows and **y** columns in C is shown below.

Syntax:

```
data_type array_name[x][y];
```

where,

- **data_type:** Type of data to be stored in each element.
- **array_name:** name of the array
- **x:** Number of rows.
- **y:** Number of columns.

We can declare a two-dimensional integer array say 'x' with 10 rows and 20 columns as:

Example:

```
int x[10][20];
```

Note: In this type of declaration, the array is allocated memory in the stack and the size of the array should be known at the compile time i.e. size of the array is fixed. We can also create an array dynamically in C by using methods mentioned here.

Initialization of Two-Dimensional Arrays in C

The various ways in which a 2D array can be initialized are as follows:

1. **Using Initializer List**
2. **Using Loops**

1. Initialization of 2D array using Initializer List

We can initialize a 2D array in C by using an initializer list as shown in the example below.

First Method:

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

The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in order: the first 4 elements from the left will be filled in the first row, the next 4 elements in the second row, and so on.

Second Method (better):

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

This type of initialization makes use of nested braces. Each set of inner braces represents one row. In the above example, there is a total of three rows so there are three sets of inner braces. The advantage of this method is that it is easier to understand.

Note: The number of elements in initializer list should always be less than or equal to the total number of elements in the array.

We can also declare the array without defining the size of the row if we are using list initialization. The compiler will automatically deduce the size of the array in this case:

```
data_type array_name[][y] = {...} ;
```

It is still compulsory to define the number of columns.

2. Initialization of 2D array using Loops

We can use any C loop to initialize each member of a 2D array one by one as shown in the below example.

Example:

```
int x[3][4];


for(int i = 0; i < 3; i++){
    for(int j = 0; j < 4; j++){
        x[i][j] = i + j;
    }
}
```

This method is useful when the values of each element have some sequential relation.

Accessing Elements of Two-Dimensional Arrays in C

Elements in 2D arrays are accessed using row indexes and column indexes. Each element in a 2D array can be referred to by:

Syntax:

```
array_name[i][j]
```

where,

- **i:** The row index.
- **j:** The column index.

Example:

```
int x[2][1];
```

The above example represents the element present in the third row and second column.

***Note**: In arrays, if the size of an array is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is 2+1 = 3. To output all the elements of a Two-Dimensional array we can use nested for loops. We will require two 'for' loops. One to traverse the rows and another to traverse columns.*

For printing the whole array, we access each element one by one using loops. The order of traversal can be row-major order or column-major order depending upon the requirement. The below example demonstrates the row-major traversal of a 2D array.

Example:

- C

```
// C Program to print the elements of a
// Two-Dimensional array

#include <stdio.h>

int main(void)
{
    // an array with 3 rows and 2 columns.
    int x[3][2] = { { 0, 1 }, { 2, 3 }, { 4, 5 } };

    // output each array element's value
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            printf("Element at x[%i][%i]: ", i, j);
            printf("%d\n", x[i][j]);
        }
    }

    return (0);
}

// This code is contributed by sarajadhav12052009
```

Output

```
Element at x[0][0]: 0

Element at x[0][1]: 1

Element at x[1][0]: 2

Element at x[1][1]: 3

Element at x[2][0]: 4

Element at x[2][1]: 5
```

Time Complexity: O(N*M) , where N(here 3) and M(here 2) are numbers of rows and columns respectively.

Space Complexity:O(1)

How 2D Arrays are Stored in the Memory?

The elements of the 2-D array have to be stored contiguously in memory. As the computers have linear memory addresses, the 2-D arrays must be linearized so as to enable their storage. There are two ways to achieve linearization of array elements:

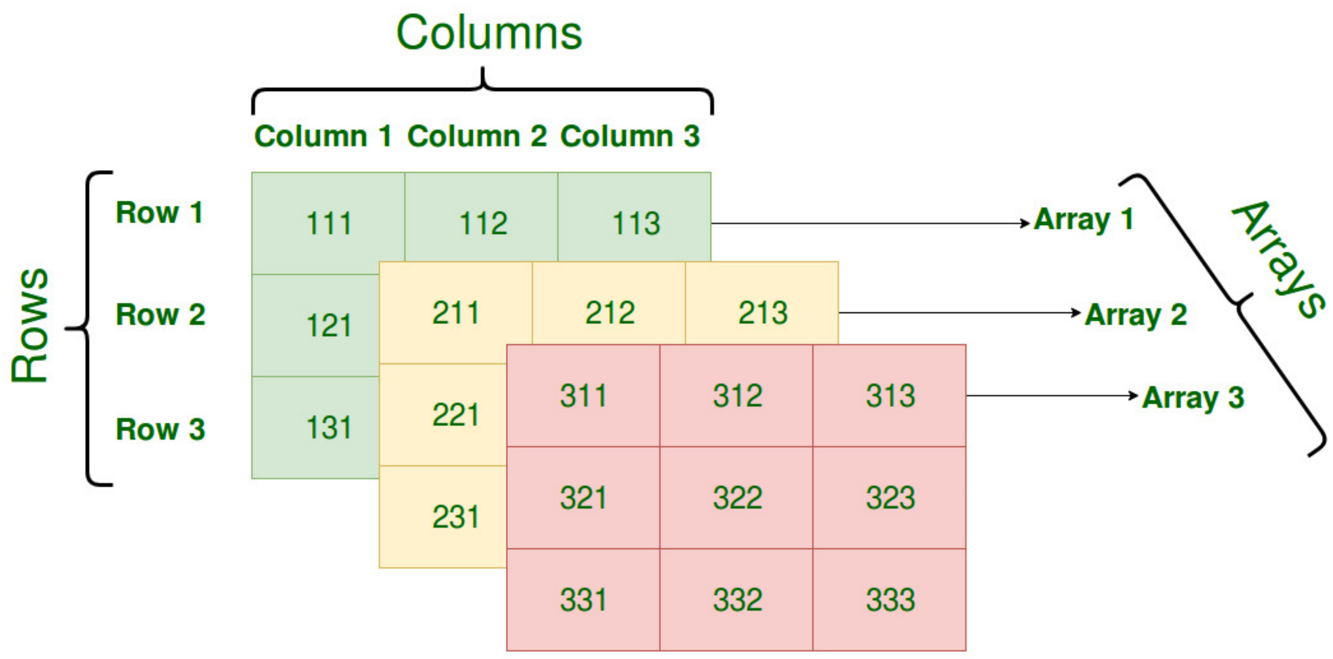
- **Row-major-** The linearization technique stores firstly the first row of the array, then the second row of the array, then the third row, and so on. (i.e. elements are stored row-wise. Rows are listed on the basis of columns)

- **Column-major**– This linearization technique stores first the first column, then the second column, then the third column, and so on i.e. (elements are stored column-wise. Columns are listed on the basis of rows)

The computer does not keep track of the addresses of all the elements of the array but does keep track of the Base Address (starting address of the very first element) and calculates the addresses of the elements when required.

Three-Dimensional Array in C

A **Three Dimensional Array** or **3D** array in C is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.



Graphical Representation of Three-Dimensional Array of Size 3 x 3 x 3

Declaration of Three-Dimensional Array in C

We can declare a 3D array with **x** 2D arrays each having **y** rows and **z** columns using the syntax shown below.

Syntax:

```
data_type array_name[x][y][z];
```

- **data_type:** Type of data to be stored in each element.
- **array_name:** name of the array
- **x:** Number of 2D arrays.
- **y:** Number of rows in each 2D array.
- **z:** Number of columns in each 2D array.

Example:

```
int array[3][3][3];
```

Initialization of Three-Dimensional Array in C

Initialization in a 3D array is the same as that of 2D arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

A 3D array in C can be initialized by using:

1. **Initializer List**
2. **Loops**

Initialization of 3D Array using Initializer List

Method 1:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15, 16, 17, 18, 19,
                 20, 21, 22, 23};
```

Method 2(Better):

```
int x[2][3][4] =
 {
   { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
   { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
 };
```

Again, just like the 2D arrays, we can also declare the 3D arrays without specifying the size of the first dimensions if we are using initializer list for initialization. The compiler will automatically deduce the size of the first dimension. But we still need to specify the rest of the dimensions.

```
data_type array_name[][y][z] = {....};
```

Initialization of 3D Array using Loops

It is also similar to that of 2D array with one more nested loop for accessing one more dimension.

```
int x[2][3][4];

for (int i=0; i<2; i++) {
    for (int j=0; j<3; j++) {
        for (int k=0; k<4; k++) {
            x[i][j][k] = (some_value);
        }
    }
}
```

Accessing elements in Three-Dimensional Array in C

Accessing elements in 3D Arrays is also similar to that of 2D Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in 3D Arrays.

Syntax:

```
array_name[x][y][z]
```

where,

- **x:** Index of 2D array.
- **y:** Index of that 2D array row.
- **z:** Index of that 2D array column.

- C

```c
// C program to print elements of Three-Dimensional Array

#include <stdio.h>

int main(void)
{
    // initializing the 3-dimensional array
    int x[2][3][2] = { { { 0, 1 }, { 2, 3 }, { 4, 5 } },
                       { { 6, 7 }, { 8, 9 }, { 10, 11 } } };

    // output each element's value
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                printf("Element at x[%i][%i][%i] = %d\n", i,
                       j, k, x[i][j][k]);
            }
        }
    }
    return (0);
}
```

Output

```
Element at x[0][0][0] = 0

Element at x[0][0][1] = 1

Element at x[0][1][0] = 2

Element at x[0][1][1] = 3

Element at x[0][2][0] = 4

Element at x[0][2][1] = 5

Element at x[1][0][0] = 6

Element at x[1][0][1] = 7

Element at x[1][1][0] = 8

Element at x[1][1][1] = 9

Element at x[1][2][0] = 10

Element at x[1][2][1] = 11
```

Array of Pointers in C

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Note: *It is important to keep in mind the operator precedence and associativity in the array of pointers declarations of different type as a single change will mean the whole different thing. For example, enclosing *array_name in the parenthesis will mean that array_name is a pointer to an array.*

Example:

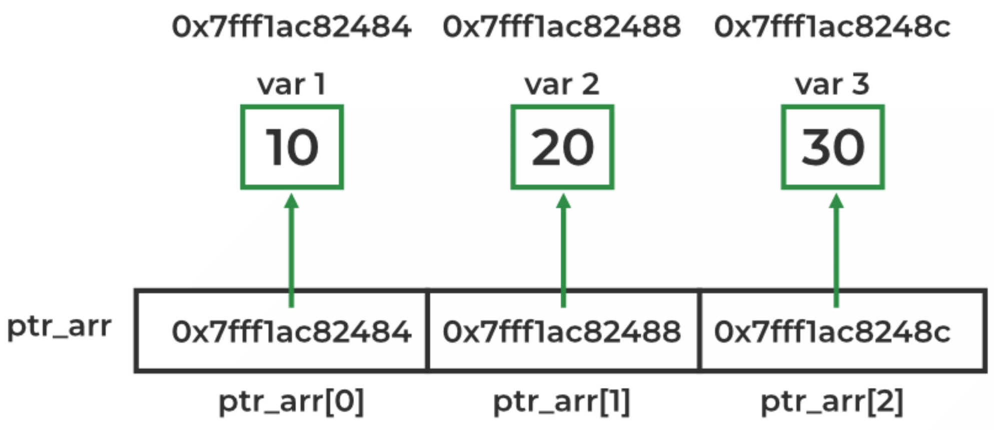
C

```c
// C program to demonstrate the use of array of pointers
#include <stdio.h>

int main()
{
    // declaring some temp variables
    int var1 = 10;
    int var2 = 20;
    int var3 = 30;

    // array of pointers to integers
    int* ptr_arr[3] = { &var1, &var2, &var3 };

    // traversing using loop
    for (int i = 0; i < 3; i++) {
        printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
    }

    return 0;
}
```

Output

```
Value of var1: 10    Address: 0x7fff1ac82484

Value of var2: 20    Address: 0x7fff1ac82488

Value of var3: 30    Address: 0x7fff1ac8248c
```

Explanation:



As shown in the above example, each element of the array is a pointer pointing to an integer. We can access the value of these integers by first selecting the array element and then dereferencing it to get the value.

Array of Pointers to Character

One of the main applications of the array of pointers is to store multiple strings as an array of pointers to characters. Here, each pointer in the array is a character pointer that points to the first character of the string.

Syntax:

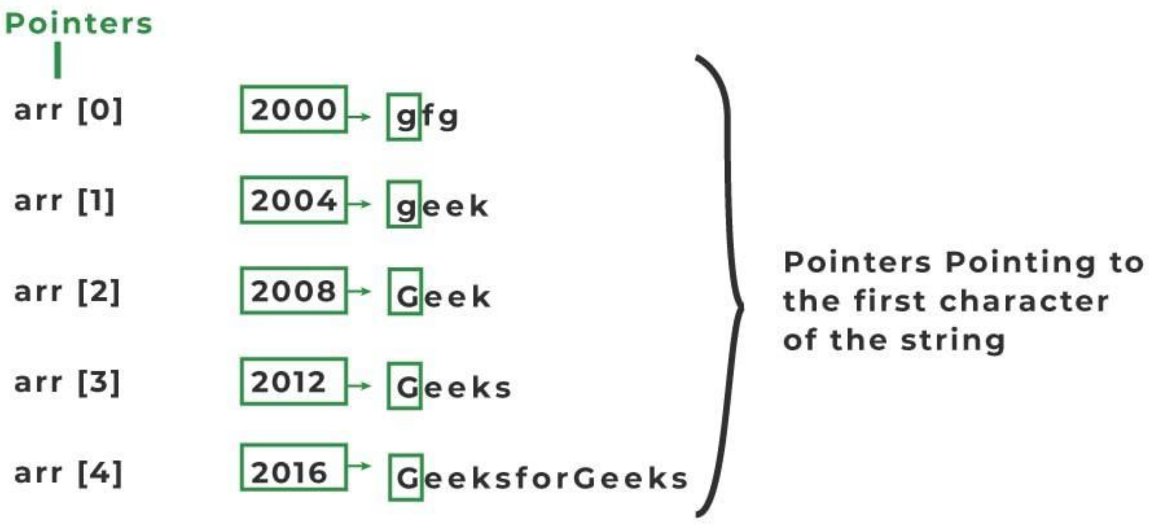
```
char *array_name [array_size];
```

After that, we can assign a string of any length to these pointers.

Example:

C

```
char* arr[5]
   = { "gfg", "geek", "Geek", "Geeks", "GeeksforGeeks" }
```



***Note:** Here, each string will take different amount of space so offset will not be the same and does not follow any particular order.*

This method of storing strings has the advantage of the traditional array of strings. Consider the following two examples:

Example 1:

C

```
// C Program to print Array of strings without array of pointers
#include <stdio.h>
int main()
{
   char str[3][10] = { "Geek", "Geeks", "Geekfor" };


   printf("String array Elements are:\n");
```

```
    for (int i = 0; i < 3; i++) {
        printf("%s\n", str[i]);
    }

    return 0;
}
```

Output

```
String array Elements are:

Geek

Geeks

Geekfor
```

In the above program, we have declared the 3 rows and 10 columns of our array of strings. But because of predefining the size of the array of strings the space consumption of the program increases if the memory is not utilized properly or left unused. Now let’s try to store the same strings in an array of pointers.

Example 2:

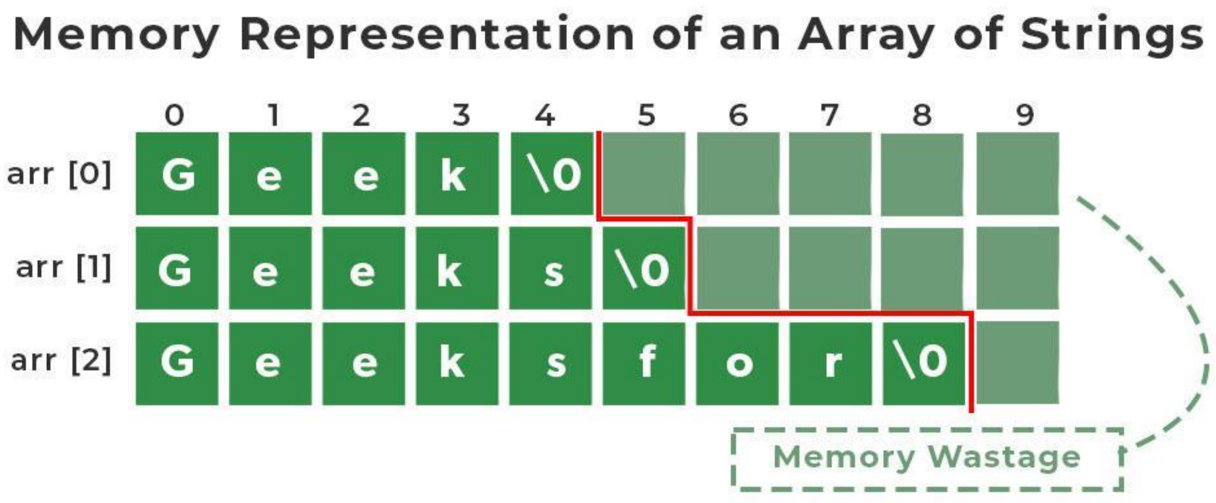
C

```c
// C Program to print Array of strings with array of pointers
#include <stdio.h>

int main() {
    // Declare an array of pointers to characters
    char* arr[] = { "geek", "Geeks", "Geeksfor" };

    // Print each string and its address
    for (int i = 0; i < 3; i++) {
        printf("%s\n", arr[i]);
    }

    // Print addresses of each string
    for (int i = 0; i < 3; i++) {
        printf("Address of arr[%d]: %p\n", i, (void*)arr[i]);
    }

    return 0;
}
```

Output

```
geek

Geeks

Geeksfor

Address of arr[0]: 0x400634

Address of arr[1]: 0x400639

Address of arr[2]: 0x40063f
```

Here, the total memory used is the memory required for storing the strings and pointers without leaving any empty space hence, saving a lot of wasted space. We can understand this using the image shown below.



The space occupied by the array of pointers to characters is shown by solid green blocks excluding the memory required for storing the pointer while the space occupied by the array of strings includes both solid and light green blocks.

C Dynamic Memory Allocation

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 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.

The expression results in a `NULL` pointer if the memory cannot be allocated.

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`.

Example 1: malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

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

int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) malloc(n * sizeof(int));

  // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);

  // deallocating the memory
  free(ptr);

  return 0;
}
```
Run Code

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

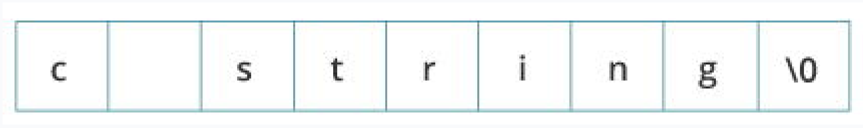
Here, we have dynamically allocated the memory for `n` number of `int`.

C Programming Strings

In C programming, a string is a sequence of characters terminated with a null character `\0`. For example:

```c
char c[] = "c string";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character `\0` at the end by default.

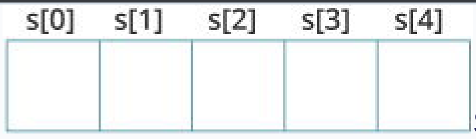


Memory Diagram

How to declare a string?

Here's how you can declare strings:

```c
char s[5];
```



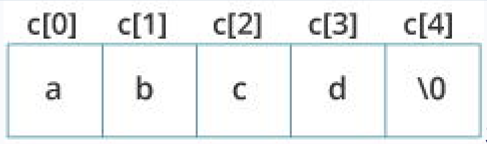
String Declaration in C

Here, we have declared a string of 5 characters.

How to initialize strings?

You can initialize strings in a number of ways.

```
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



String Initialization in C

Let's take another example:

```
char c[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is `'\0'`) to a `char` array having 5 characters. This is bad and you should never do this.

Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```
char c[100];
c = "C programming";  // Error! array type is not assignable.
```

Note: Use the strcpy() function to copy the string instead.

Read String from the user

You can use the `scanf()` function to read a string.
The `scanf()` function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example 1: scanf() to read a string

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Even though `Dennis Ritchie` was entered in the above program, only `"Dennis"` was stored in the `name` string. It's because there was a space after `Dennis`.
Also notice that we have used the code `name` instead of `&name` with `scanf()`.

```
scanf("%s", name);
```

This is because `name` is a `char` array, and we know that array names decay to pointers in C.
Thus, the `name` in `scanf()` already points to the address of the first element in the string, which is why we don't need to use `&`.

How to read a line of text?

You can use the `fgets()` function to read a line of string. And, you can use `puts()` to display the string.

Example 2: fgets() and puts()

```c
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);  // read string
    printf("Name: ");
    puts(name);    // display string
    return 0;
}
```

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Here, we have used `fgets()` function to read a string from the user.

`fgets(name, sizeof(name), stdlin); // read string`

The `sizeof(name)` results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the `name` string.

To print the string, we have used `puts(name);`.

Note: The `gets()` function can also be to take input from the user. However, it is removed from the C standard.

It's because `gets()` allows you to input any length of characters. Hence, there might be a buffer overflow.

Passing Strings to Functions

Strings can be passed to a function in a similar way as arrays. Learn more about passing arrays to a function.

Example 3: Passing string to a Function

```c
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str);     // Passing string to a function.
    return 0;
}
void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

Strings and Pointers

Similar like arrays, string names are "decayed" to pointers. Hence, you can use pointers to manipulate elements of the string. We recommended you to check C Arrays and Pointers before you check this example.

Example 4: Strings and Pointers

```c
#include <stdio.h>

int main(void) {
  char name[] = "Harry Potter";

  printf("%c", *name);     // Output: H
  printf("%c", *(name+1));   // Output: a
  printf("%c", *(name+7));   // Output: o

  char *namePtr;

  namePtr = name;
  printf("%c", *namePtr);     // Output: H
  printf("%c", *(namePtr+1));   // Output: a
  printf("%c", *(namePtr+7));   // Output: o
}
```

C struct

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.

Define Structures

Before you can create structure variables, you need to define its data type. To define a struct, the `struct` keyword is used.

Syntax of struct

```
struct structureName {
  dataType member1;
  dataType member2;
  ...
};
```

For example,

```
struct Person {
  char name[50];
  int citNo;
  float salary;
};
```

Here, a derived type `struct Person` is defined. Now, you can create variables of this type.

Create struct Variables

When a `struct` type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.
Here's how we create structure variables:

```
struct Person {
  // code
};

int main() {
  struct Person person1, person2, p[20];
  return 0;
}
```

Another way of creating a `struct` variable is:

```
struct Person {
    // code
} person1, person2, p[20];
```

In both cases,

- `person1` and `person2` are `struct Person` variables
- `p[]` is a `struct Person` array of size 20.

Access Members of a Structure

There are two types of operators used for accessing members of a structure.

1. `.` - Member operator
2. `->` - Structure pointer operator (will be discussed in the next tutorial)

Suppose, you want to access the `salary` of `person2`. Here's how you can do it.

```
person2.salary
```

Example 1: C structs

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

// create struct with person1 variable
struct Person {
  char name[50];
  int citNo;
  float salary;
} person1;

int main() {
  // assign value to name of person1
  strcpy(person1.name, "George Orwell");

  // assign values to other person1 variables
  person1.citNo = 1984;
  person1. salary = 2500;

  // print struct variables
  printf("Name: %s\n", person1.name);
  printf("Citizenship No.: %d\n", person1.citNo);
  printf("Salary: %.2f", person1.salary);

  return 0;
}
```
Run Code

Output

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

In this program, we have created a `struct` named `Person`. We have also created a variable of `Person` named `person1`.

In `main()`, we have assigned values to the variables defined in `Person` for the `person1` object.

```
strcpy(person1.name, "George Orwell");
person1.citNo = 1984;
person1. salary = 2500;
```

Notice that we have used `strcpy()` function to assign the value to `person1.name`.

This is because `name` is a `char` array (C-string) and we cannot use the assignment operator `=` with it after we have declared the string.

Finally, we printed the data of `person1`.

C Pointers to struct

Here's how you can create pointers to structs.

```c
struct name {
    member1;
    member2;
    .
    .
};

int main()
{
    struct name *ptr, Harry;
}
```

Here, `ptr` is a pointer to `struct`.

Example: Access members using Pointer

To access members of a structure using pointers, we use the `->` operator.

```c
#include <stdio.h>
struct person
{
   int age;
   float weight;
};

int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);
    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);

    return 0;
}
```
Run Code

In this example, the address of `person1` is stored in the `personPtr` pointer using `personPtr = &person1;`.

Now, you can access the members of `person1` using the `personPtr` pointer.

By the way,

- `personPtr->age` is equivalent to `(*personPtr).age`
- `personPtr->weight` is equivalent to `(*personPtr).weight`

Dynamic memory allocation of structs

Before you proceed this section, we recommend you to check C dynamic memory allocation.

Sometimes, the number of struct variables you declared may be insufficient. You may need to allocate memory during run-time. Here's how you can achieve this in C programming.

Example: Dynamic memory allocation of structs

```
#include <stdio.h>
#include <stdlib.h>
struct person {
   int age;
   float weight;
   char name[30];
};

int main()
{
   struct person *ptr;
   int i, n;

   printf("Enter the number of persons: ");
   scanf("%d", &n);

   // allocating memory for n numbers of struct person
   ptr = (struct person*) malloc(n * sizeof(struct person));

   for(i = 0; i < n; ++i)
   {
       printf("Enter first name and age respectively: ");

       // To access members of 1st struct person,
       // ptr->name and ptr->age is used

       // To access members of 2nd struct person,
       // (ptr+1)->name and (ptr+1)->age is used
       scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
   }

   printf("Displaying Information:\n");
   for(i = 0; i < n; ++i)
       printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);

   return 0;
}
```
Run Code

When you run the program, the output will be:

```
Enter the number of persons:  2
Enter first name and age respectively:  Harry 24
Enter first name and age respectively:  Gary 32
Displaying Information:
Name: Harry      Age: 24
Name: Gary       Age: 32
```

In the above example, `n` number of struct variables are created where `n` is entered by the user.
To allocate the memory for `n` number of `struct person`, we used,

```
ptr = (struct person*) malloc(n * sizeof(struct person));
```

Then, we used the `ptr` pointer to access elements of `person`.