

## Practical - 02

### Strings, Array Pointers and function pointers

#### Relationship between Arrays and Pointers

Let an array representation as shown below :

| index | 0   | 1   | arr<br>2 | array<br>3 | 4   |     |
|-------|-----|-----|----------|------------|-----|-----|
| ...   | 3   | 5   | 7        | 9          | 11  | ... |
| 296   | 300 | 304 | 308      | 312        | 316 | 320 |
|       |     |     |          |            |     | 324 |

int = 4 bytes

- 'arr' serves two purposes here, first it is the name of the array and second, arr itself represents the base address of the array i.e. 300 in the above case, if we print the value in arr then it will print the address of the first element in the array.
- As the array name arr itself represents the base address of the array, then by default arr acts as a pointer to the first element of the array.
- arr is the same as &arr and &arr[0] in C Language.
- If we use dereferencing operator (\*) on any of the above representations of array address, we will get the value of the very first element of the array.

```
#include <stdio.h>
int main() {
    // array declaration and initialization
    int arr[5] = {3, 5, 7, 9, 11};
    // printing the addresses and values represented by arr, &arr
    and &arr[0]
    printf("arr : %u, Value : %d\n", arr, *arr);
    printf("&arr : %u, Value : %d\n", &arr, *(arr));
    printf("&arr[0] : %u, Value : %d\n", &arr[0], *( &arr[0]));
    return 0;
}
```

In a C Program, we denote array elements as arr[i], where i is the index value. Below is a similar syntax in terms of pointers of how we can represent the array elements using the dereferencing operator (\*) on the array name i.e. using the pointers property of the array.

\*(arr + i)

```

#include <stdio.h>
int main()
{
    // array declaration and initialization
    int arr[5] = {2, 4, 6, 8, 10}, i;
    for(i = 0; i < 5; i++)
    {
        // printing the elements address and value at
        // arr[i] using *(arr + i) syntax
        printf("[index %d] Address : %u, Value : %d\n", i, (arr +
i), *(arr + i));
    }
    return 0;
}

```

|       | arr  |      |      |      |      |
|-------|------|------|------|------|------|
| index | 0    | 1    | 2    | 3    | 4    |
|       | 2    | 4    | 6    | 8    | 10   |
|       | 1000 | 1004 | 1008 | 1012 | 1016 |
|       |      |      |      |      | 1020 |

- (arr + i) represents the address of the value at index i, so \*(arr + i) will give the value at ith index (address(arr + i) = address(arr[i])), it is used to print the addresses of the array elements as the value of i changes from 0-4.
- \* is a dereferencing operator used for printing the value at the provided address. \*(arr + i) will print the values of the array at consecutive addresses as the value of i changes from 0-4.

Note : From the above example we can conclude that, &arr[0] is equal to arr and arr[0] is equal to \*arr. Similarly,

- &arr[1] is equal to (arr + 1) and arr[1] is equal to \*(arr + 1).
- &arr[2] is equal to (arr + 2) and arr[2] is equal to \*(arr + 2) and so on.
- ...
- Finally, we can write the above expressions in a fundamental form:
- &arr[i] is equal to (arr + i) and arr[i] is equal to \*(arr + i).

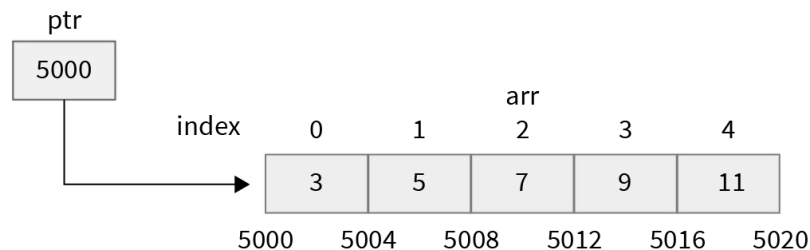
Note : When the array name arr is an operand of sizeof() operator or the & (address-of) unary operator i.e. sizeof(arr) and &arr respectively, then the array name arr refers to the whole array object, thus sizeof(arr) gives us the size of the entire array in bytes and &arr covers the whole array because as we know the array name arr generally means the base address of the array, so arr and &arr are equivalent but arr + 1 and &arr + 1 will not be equal if the array size is more than 1, arr + 1 gives the address of the next element in the array, while &arr + 1 gives the address of the element that is next to the last element of the array (&arr covers the whole array).

## Pointer to Arrays

In a pointer to an array, we just have to store the base address of the array in the pointer variable. We know in the arrays that the base address of an array can be represented in three forms, let us see the syntax of how we can store the base address in a pointer variable:

```
int arr[5] = {10, 20, 30, 40, 50};
int (*ptr);
ptr = &arr;
//ptr = &arr[0];
//ptr = arr;
```

```
#include <stdio.h>
int main()
{
    // array declaration and initialization
    int arr[5] = {3, 5, 7, 9, 11}, i;
    // both `arr` and `&arr` return the address of the first
    element of the array.
    int *ptr = arr;
    // printing the elements of array using addition arithmetic
    on pointer
    for(i = 0; i < 5; i++){
        printf("%d ", *(ptr + i));
    }
    return 0;
}
```



- $(ptr + i)$  will give the address of the array elements as the value of  $i$  changes from 0-4 as  $address(ptr + i) = address(arr[i])$ .
- $*$  is the dereferencing operator used for printing the value at the provided address.  $*(ptr + i)$  will print the values of the array as the value of  $i$  changes.

The pointer can be used to access the array elements, accessing the whole array using pointer arithmetic makes the accessing faster.

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

(a + i) points to i<sup>th</sup> 1-D array.

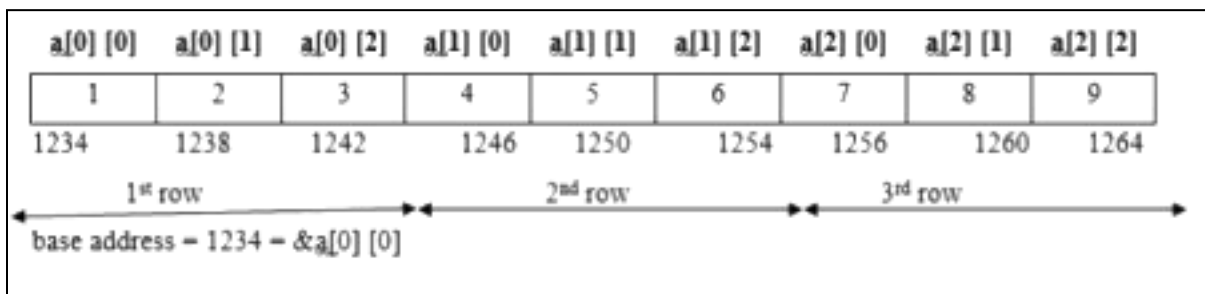
\*(a+i) points to the base address of the i<sup>th</sup> 1-D array

\*(a + i) points to the address of the 0<sup>th</sup> element of the 1-D array. So, \*(a + i) + 1 points to the address of the 1<sup>st</sup> element of the 1-D array \*(a + i) + 2 points to the address of the 2<sup>nd</sup> element of the 1-D array

\*(a + i) + j points to the base address of j<sup>th</sup> element of i<sup>th</sup> 1-D array.

On dereferencing \*(a + i) + j we will get the value of j<sup>th</sup> element of i<sup>th</sup> 1-D array.

\*( \*(a + i) + j) by using this expression we can find the value of j<sup>th</sup> element of i<sup>th</sup> 1-D array.



Example:

```
#include <stdio.h>
int main()
{
    // array declaration and initialization
    int a[5] = {3, 5, 7, 9, 11}, i;

    // Valid in case of arrays but not valid in case of single
    integer values.
    int *ptr = a;

    // All representations prints the base address of the array
    printf("ptr : %u, &a[0] : %u, a : %u, &a : %u\n", ptr,
    &a[0], a, &a);

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

```

    // printing address values
    printf("[index %d] Address : %u\n", i, (ptr + i));
}

printf("\n");

for (i = 0; i < 5; i++)
{
    // Accessing array values through pointer
    // a[i] = *(a + i) = *(ptr + i) = *(i + a) = a[i]
    printf("[index %d] Value : %d %d %d %d\n", i, *(a + i),
*(ptr + i), *(i + a), a[i]);
}

printf("\n");

// Gives address of next byte after array's last element
printf("&a : %u, &a + 1 : %u\n", &a, &a + 1);

// Gives the address of the next element
printf("a : %u, a + 1 : %u\n", a, a + 1);

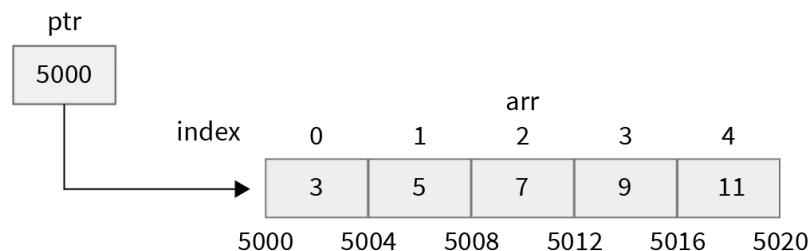
// Gives value at index 1
printf("*(a + 1) : %d\n", *(a + 1));

// Gives (value at index 0) + 1
printf("*a + 1 : %d\n", *a + 1);

// Gives (value at index 0) / 2, we can't perform *(p / 2)
or *(p * 2)
printf("(*ptr / 2) : %d\n", (*ptr / 2));

return 0;
}

```



- As we know, `ptr`, `&a[0]`, `a` and `&a` are representing the same address, so all representations

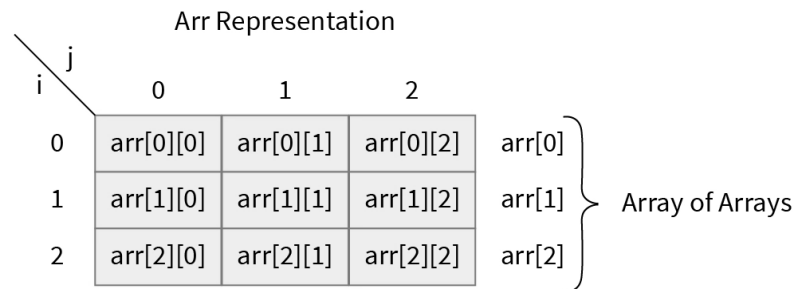
print the same address value in the output.

- First for loop ( $i = 0$  to  $4$ ) is used to print the address of all the array elements.
- Second for loop ( $i = 0$  to  $4$ ) is used to demonstrate that  $a[i] = *(a + i) = *(ptr + i) = *(a + i) = a[i]$ . All these representations of the array elements are equivalent to each other.
- $\&arr + 1$  gives the address of the element that is next to the last element ( $\&arr + 1$  covers the whole array) while  $a + 1$  gives the address of the second element of the array.
- $*(a + 1)$  prints the value at index 1 in the output and is equivalent to  $a[1]$ .
- $*a + 1$  prints the (value at [index 0]) + 1 and is equivalent to  $a[0] + 1$ .
- *( $*ptr / 2$ ) prints the (value at [index 0]) / 2, we can't perform division or multiplication operations on pointers directly. ( $*(p / 2)$  or  $*(p * 2)$  respectively).*

Multi-dimensional arrays are defined as an array of arrays. 2-D arrays consist of 1-D arrays, while 3-D arrays consist of 2-D arrays as their elements.

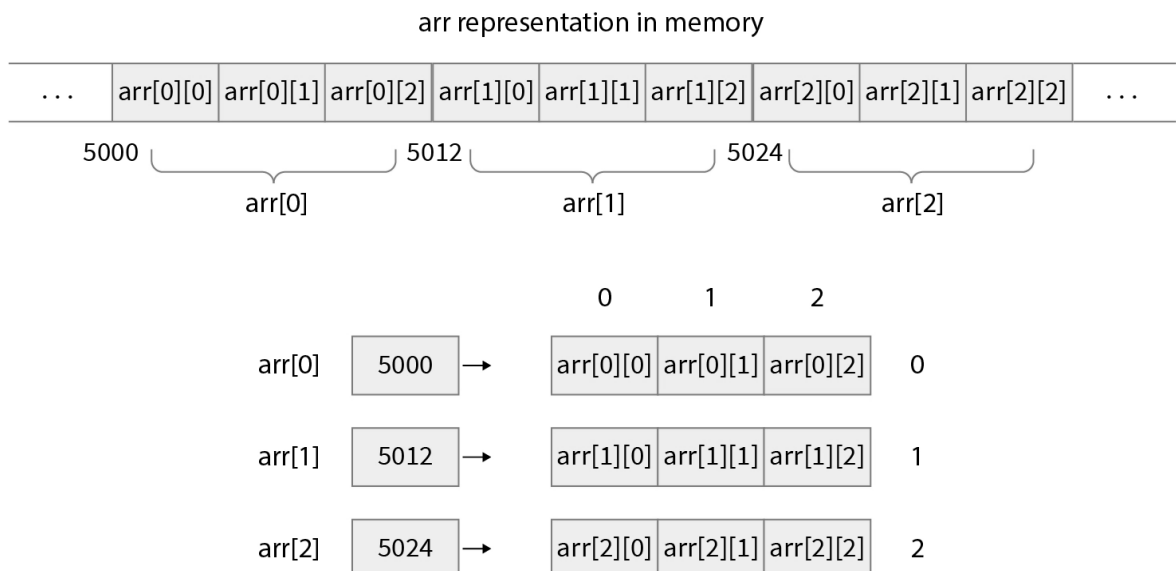
## Pointer to 2D array

A 2-D array is an array of arrays, we can understand 2-D arrays as they are made up of  $n$  1-D arrays stored in a linear manner in the memory. 2-D arrays can also be represented in a matrix form. In the matrix form, there are rows and columns



Here one perspective is, array contains 3 1-D arrays as its element, so array name `arr` acts as a pointer to the 1st 1-D array i.e. `arr[0]` and not to the first element of the array i.e. `arr[0][0]`. As we know our system's memory is organized in a sequential manner so it is not possible to store a 2-D array in rows and columns fashion, they are just used for the logical representation of 2-D arrays.

So it can be said that, we have combined 3 1-D arrays that are stored in the memory to make a 2-D array, here `arr[0]`, `arr[1]`, `arr[2]` represents the base address of the respective arrays. So, `arr[0]`, `arr[1]` and `arr[2]` act as a pointer to these arrays and we can access the 2-D arrays using the above array pointers.



## Syntax for representing 2-D array elements :

$*(arr + i) + j$

Note :  $*(arr + i) + j$  represents the element of an array arr at the index value of ith row and jth column; it is equivalent to the regular representation of 2-D array elements as arr[i][j].

```
#include <stdio.h>
int main()
{
    int arr[3][3] = {{2, 4, 6},
                    {0, 1, 0},
                    {3, 5, 7}};

    int i, j;

    // the below statement is wrong because
    // arr will return the address of a first 1-D array.
    // int *ptr = arr;
    // int *ptr = &arr[0]; is correct or we can write
    &arr[1], &arr[2].

    printf("Addresses : \n");
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%u[%d%d] ", (*(arr + i) + j), i,
j));
        }
        printf("\n");
    }

    printf("Values : \n");
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%d[%d%d] ", (*(arr + i) + j), i,
j));
        }
        printf("\n");
    }

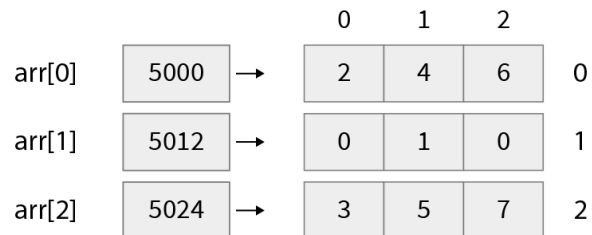
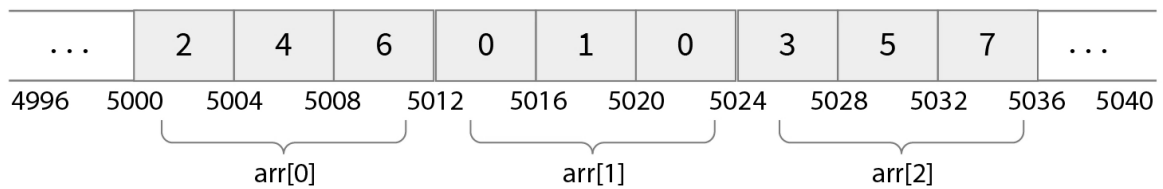
    return 0;
}
```



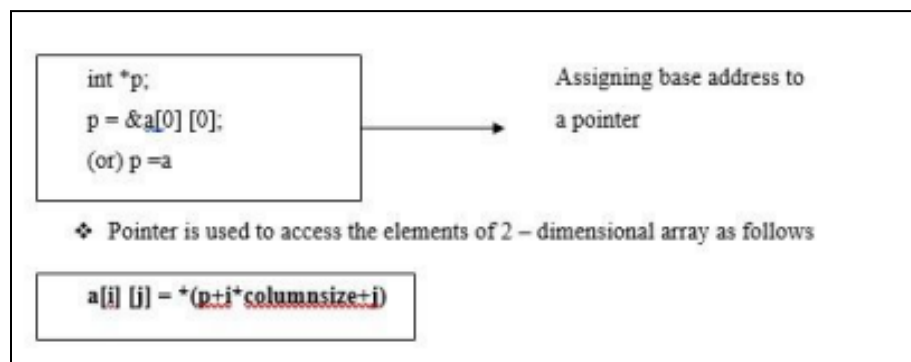
Arr Representation

| i \ j |   |   |   |
|-------|---|---|---|
|       | 0 | 1 | 2 |
| 0     | 2 | 4 | 6 |
| 1     | 0 | 1 | 0 |
| 2     | 3 | 5 | 7 |

arr representation in memory



- We have used  $*(arr + i) + j$  to print the address and  $*(*(arr + i) + j)$  to print the value of the array elements in the output.
- We can see that all the address values are separated by a 4 bytes difference.



```

#include<stdio.h>
int main()
{
    int arr[3][4] = {
                        {11,22,33,44},
                        {55,66,77,88},
                        {11,66,77,44}
                    };

    int i, j;
    for(i = 0; i < 3; i++) {
        printf("Address of %d th array %u \n",i , *(arr + i));
        for(j = 0; j < 4; j++) {
            printf("arr[%d][%d]=%d\n", i, j, *( *(arr + i) + j) );
        }
        printf("\n\n");
    }
    return 0;
}

```

### Output

```

Address of 0 th array 2686736
arr[0][0]=11
arr[0][1]=22
arr[0][2]=33
arr[0][3]=44

```

```

Address of 1 th array 2686752
arr[1][0]=55
arr[1][1]=66
arr[1][2]=77
arr[1][3]=88

```

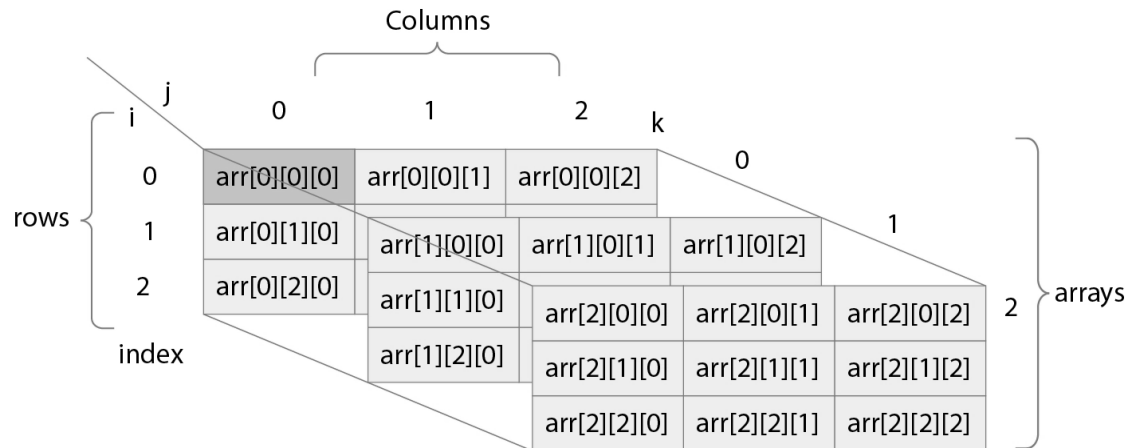
```

Address of 2 th array 2686768
arr[2][0]=11
arr[2][1]=66
arr[2][2]=77
arr[2][3]=44

```

## Pointer to 3D Arrays in C

When the elements of an array are 2-D arrays, then the array formed is known as 3-Dimensional Array. 3-Dimensional arrays can also be known as array of matrices. Below is a representation of how a 3-D array looks like.



### Syntax for Representing 3-D array elements :

$$*(*(*(arr + i) + j) + k)$$

$*(*(*(arr + i) + j) + k)$  represents the element of an array arr at the index value of ith row and jth column of the kth array in the array arr; it is equivalent to the regular representation of 3-D array elements as arr[i][j][k].

### Example:

```
#include <stdio.h>

int main()
{
    int arr[3][3][3] = {{1, 2, 3, 4, 5, 6, 7, 8, 9},
                        {2, 4, 6, 8, 10, 12, 14, 16, 18},
                        {3, 5, 7, 9, 11, 13, 15, 17, 19}};

    int i, j, k;

    // the below statement is wrong because
    // arr will return the address of a first 1-D array.
    // int *ptr = arr;
    // int *ptr = &arr[0][0]; is correct or we can write
    &arr[1][0], &arr[2][0].
```

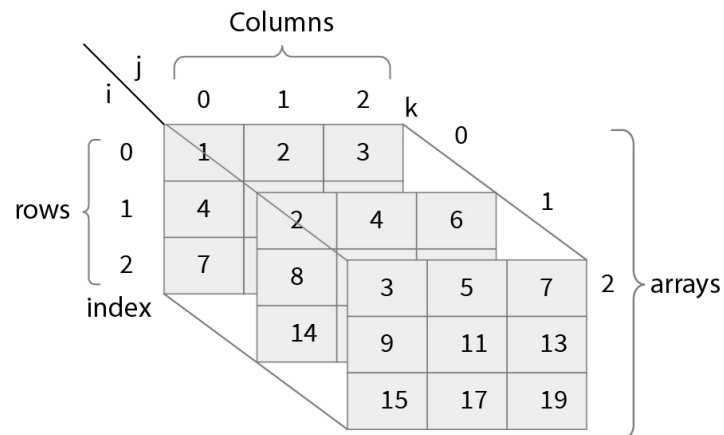
```

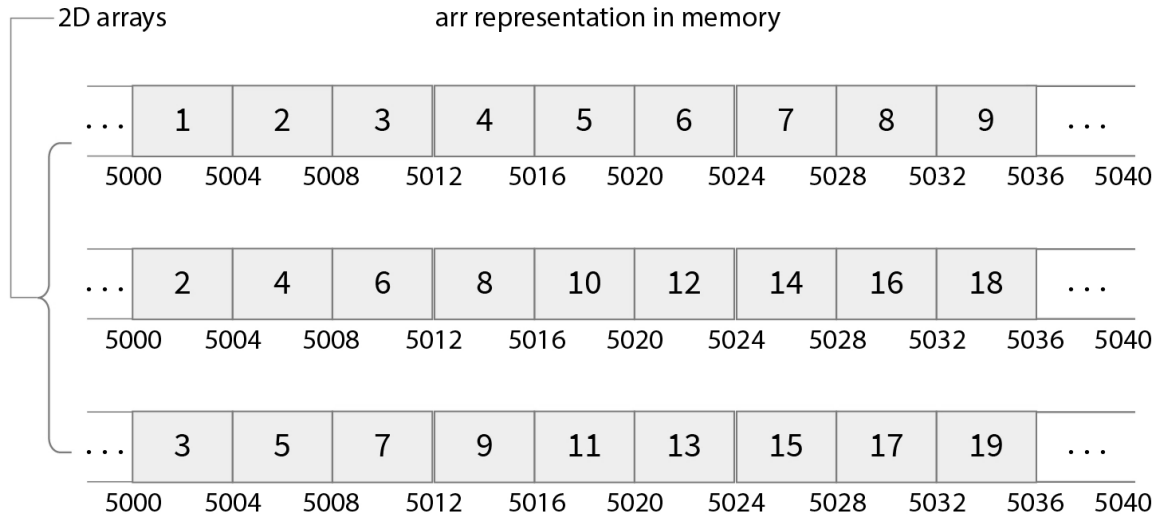
printf("Addresses : \n");
for(i = 0; i < 3; i++)
{
    for(j = 0; j < 3; j++)
    {
        for(k = 0; k < 3; k++)
        {
            printf("%u[%d%d%d] ", (*(arr + i) + j)
+ k), i, j, k);
        }
        printf("\n");
    }
    printf("\n");
}

printf("Values : \n");
for(i = 0; i < 3; i++)
{
    for(j = 0; j < 3; j++)
    {
        for(k = 0; k < 3; k++)
        {
            printf("%d[%d%d%d] ", (*(arr + i) +
j) + k), i, j, k);
        }
        printf("\n");
    }
    printf("\n");
}

return 0;
}

```





- We have used  $((*(arr + i) + j) + k)$  to print the address and  $((*(arr + i) + j) + k)$  to print the value of the array elements in the output.
- We can see that all the address values are separated by a 4 bytes difference.

## Array of Pointers in C

Arrays are collections of elements stored in contiguous memory locations. An array of pointers is similar to any other array in C Language. It is an array which contains numerous pointer variables and these pointer variables can store address values of some other variables having the same data type.

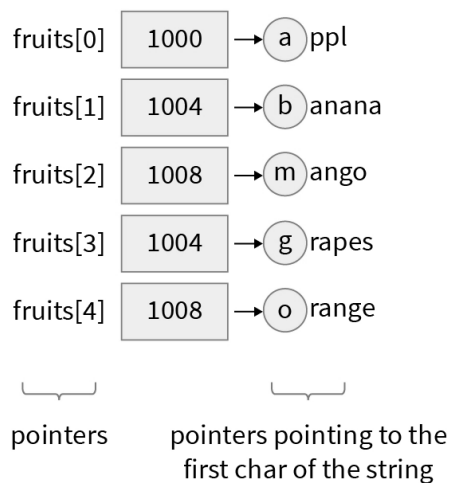
### Syntax to declare a pointer array :

data\_type (\*array\_name)[sizeof\_array];

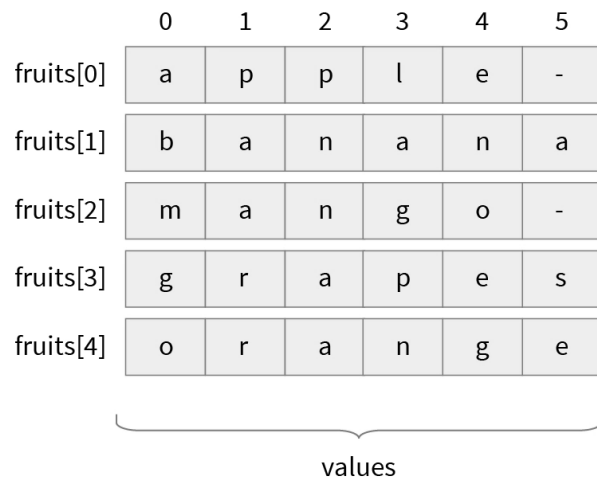
```
#include <stdio.h>
int main()
{
    char    *fruits[5]    =    {"apple",    "banana",    "mango",
    "grapes", "orange"}, i;

    for(i = 0; i < 5; i++)
    {
        printf("%s\n", fruits[i]);
    }
    return 0;
}
```

Using char pointer array  
char fruits[5];



Using 2D char array  
char fruits[5][6];

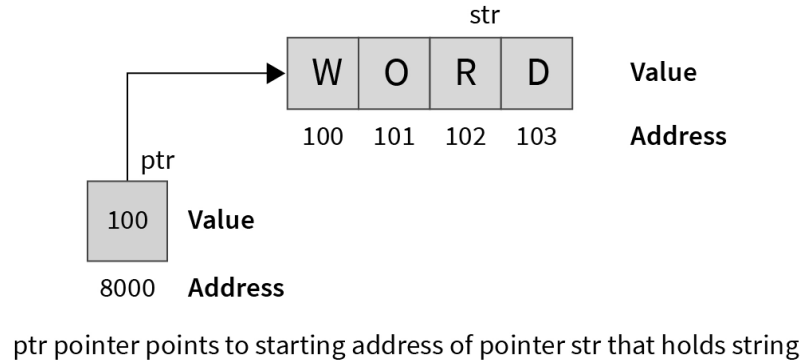


## Strings and Pointers

String is a char data type that stores the sequence of characters in an array. A string in C always ends with a null character (`\0`), which indicates the termination of the string. Pointer to string in C can be used to point to the starting address of the array, the first character in the array. These pointers can be dereferenced using the asterisk `*` operator to identify the character stored at the location. 2D arrays and pointer variables both can be used to store multiple strings.

C allows users to store words and sentences with the help of the char data type. Character data type stores only a single character, but a person's name has more than one character. We can create an array of characters to store such data with more than one character in C. Such data type that stores a sequence of characters in an array is called string.

Strings are collections of characters and can be stored using arrays in C. Pointer to string in C can point to the starting address of the array that is the first character in the array. Pointers can be dereferenced using the asterisk `*` operator to identify characters stored at a location.



```
char str[7] = {'S', 't', 'r', 'i', 'c', 'g', '\0'}; //  
String  
str[4] = 'n'; // String
```

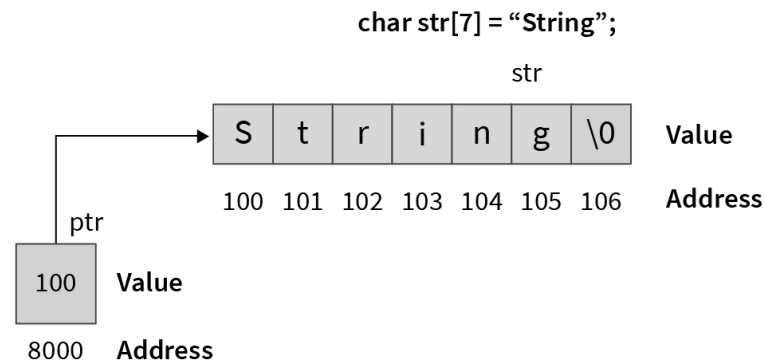
We don't have to explicitly add a null character in the string as the compiler automatically adds it. We can also use string literals to set the value of arrays. A string literal is a sequence of characters enclosed in double quotation marks (`" "`).

```
/* string literal */  
char *string_literal = "This is a string literal."
```

When we create an array, the variable name points to the address of the first element of the array. The other way to put this is the variable name of the array points to its starting position in the memory. We can create a character pointer to string in C that points to the starting address of the character array. This pointer will point to the starting address of the string, that is the first character of the string, and we can dereference the pointer to access the value of the string.

```
// character array storing the string 'String'  
char str[7] = "String";
```

```
// pointer storing the starting address of the
// character array str
char *ptr = str;
```



ptr pointer points to starting address of pointer str that holds the string

An array is a contiguous block of memory, and when pointers to string in C are used to point them, the pointer stores the starting address of the array. Similarly, when we point a char array to a pointer, we pass the base address of the array to the pointer. The pointer variable can be dereferenced using the asterisk \* symbol in C to get the character stored in the address.

```
char arr[] = "Hello";
char *ptr = arr;

printf("%c ", *ptr);           // H
printf("%c ", *(ptr + 1));     // e
printf("%c ", *(ptr + 2));     // l
printf("%c ", *(ptr + 3));     // l
printf("%c ", *(ptr + 4));     // o
```

OR

```
while (*ptr != '\0') {
    // the current character is not \0
    // so we will print the character
    printf("%c", *ptr);

    // move to the next character.
    ptr++;
}
```

In C, a string can be referred to either using a character pointer or as a character array. In the following code we are assigning the address of the string str to the pointer ptr.



```
char *ptr = str;
```

```
int main(void) {  
    // string variable  
    char str[6] = "Hello";  
    // pointer variable  
    char *ptr = str;  
    // print the string  
    while(*ptr != '\0') {  
        printf("%c", *ptr);  
        //move the ptr pointer to the  
        //next memory location  
        ptr++;  
    }  
    return 0;  
}
```

To store the string in a pointer variable, we need to create a char type variable and use the asterisk \* operator to tell the compiler the variable is a pointer.

```
// storing string using an array  
char arr[] = "ThisIsString\0";  
  
// storing string using a pointer  
char *str = "ThisIsString\0";
```

Example:

```
#include<stdio.h>  
  
int main() {  
    // creating a pointer variable to store the value of  
    // our string  
    char *strPtr = "HelloWorld";  
  
    // temporary pointer to iterate over the string  
    char *temp = strPtr;  
  
    // creating a while loop till we don't find  
    // a null character in the string  
    while (*temp != '\0') {  
        // the current character is not \0  
        // so we will print the character  
        printf("%c", *temp);  
    }
```

```

        // move the temp pointer to the next memory location
        temp++;
    }
    return 0;
}

```

## Array of String

We can use a two-dimensional array to store multiple strings, as shown below. Here, the compiler adds a null character at the end of every string if not mentioned explicitly. The strings can have variable sizes, as shown, but the size of the largest string must be less than (or equal to inclusive of null character) the column size of the 2-D array.

```

char str[4][12] = {
    "String",
    "Topics",
    "Pointers",
    "World"
}

```

|   |   |   |   |   |    |    |   |    |  |  |  |
|---|---|---|---|---|----|----|---|----|--|--|--|
| S | t | r | i | n | g  | \0 |   |    |  |  |  |
| T | o | p | i | c | \0 |    |   |    |  |  |  |
| P | o | i | n | t | e  | r  | s | \0 |  |  |  |
| W | o | r | l | d | \0 |    |   |    |  |  |  |

For 2-D arrays, both the dimensions of the array need to be defined at the time of variable declaration, and our strings don't have to be of the same length.

To solve the problem of memory wastage, we can use pointers of size four that can be used to store strings of variable size. In this case, each string takes the memory equal to the string length (inclusive of the null character), preventing memory wastage like in the case of a 2-D array. Here, `str[i]` represents the base address of the *i*th string.

```

char *str[4] = {
    "String",
    "Topics",
    "Pointers",
    "World"
};

```

In this case, we are using a pointer variable `str` of size four, because of which we are only allocating space equal to the length of the individual string



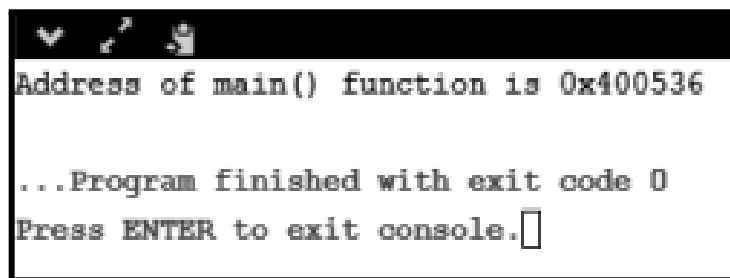
## Function Pointer

As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

eg.

```
#include <stdio.h>
int main()
{
    printf("Address of main() function is %p",main);
    return 0;
}
```

## Output



```
#include <stdio.h>

void test() {
    // test function that does nothing
    return ;
}

int main() {
    int a = 5;
    // printing the address of variable a
    printf("Address of variable = %p\n", &a);

    // printing the address of function main()
    printf("Address of a function = %p", test);
    return 0;
}
```

## Declaration of a function pointer

Till now, we have seen that the functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.

### Syntax of function pointer

return type (\*ptr\_name)(type1, type2...);

For example:

```
int (*ip) (int);
```

In the above declaration, \*ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

```
float (*fp) (float);
```

Our next step is to assign the address of a function to the function pointer.

```
float (*fp) (int , int); // Declaration of a function pointer.
```

```
float func( int , int ); // Declaration of function.
```

```
fp = func; // Assigning address of func to the fp pointer.
```

Calling an above function using a usual way is given below:

```
result = func(a , b); // Calling a function using usual ways.
```

Calling a function using a function pointer is given below:

```
result = (*fp)( a , b); // Calling a function using function pointer.
```

OR

```
result = fp(a , b);
```

Example:

```
int (*pointer) (int); // function pointer declaration
int areaSquare (int); // function declaration
pointer = areaSquare;
```

```
int length = 5;
```

```
// Different ways to call the function
```

```
// 1. using function name
int area = areaSquare(length);
```

```
// 2. using function pointer (a)
int area = (*pointer)(length);
```

```
// 3. using function pointer (b)
int area = pointer(length);
```

## Array of function pointers

Arrays are data structures that store collections of identical data types. Like any other data types we can create an array to store function pointers in C. Function pointers can be accessed from their indexes like we access normal array values `arr[i]`. This way we are creating an array of function pointers, where each array element stores a function pointer pointing to different functions.

Example:

```
#include<stdio.h>
float add(int, int);
float multiply(int,int);
float divide(int,int);
float subtract(int,int);
int main() {
    int a, b;
    float (*operation[4])(int, int);
    operation[0] = add;
    operation[1] = subtract;
    operation[2] = multiply;
    operation[3] = divide;
    printf("Enter two values ");
    scanf("%d%d", &a, &b);
    float result = (*operation[0])(a, b);
    printf("Addition (a+b) = %.1f\n", result);
    result = (*operation[1])(a, b);
    printf("Subtraction (a-b) = %.1f\n", result);
    result = (*operation[2])(a, b);
    printf("Multiplication (a*b) = %.1f\n", result);
    result = (*operation[3])(a, b);
    printf("Division (a/b) = %.1f\n", result);
    return 0;
}
float add(int a, int b) {
    return a + b;
}
float subtract(int a, int b) {
    return a - b;
}
float multiply(int a, int b) {
    return a * b;
}
float divide(int a, int b) {
    return a / (b * 1.0);
}
```

## Functions Using Pointer Variables

C allows pointers to be passed in as function arguments and also returns pointers from the function. To pass pointers in the function, we simply declare the function parameter as a pointer type. When functions have their pointer type arguments, the changes made on them inside the function persist even after the program exists function scope because the changes are made on the actual address pointed by the pointer.

```
#include<stdio.h>

int* increment(int a) {
    int *b;
    *b = a;
    *b += 1; // incrementing the value

    return b; // returning pointer from the function
}

int main() {
    int num = 5;

    int *b = increment(num); // ERROR SEGMENTATION FAULT
    printf("Incremented value = %d", *b);

    return 0;
}
```

### Safe ways to return a pointer from a function

- Return variables are either created using the keyword static or created dynamically at run time because such variables exist in memory beyond the scope of the called function.
- Use arguments that are passed by their reference because such functions exist in the calling function scope.

Another way to use function pointers is by passing them to other functions as a function argument. We also call such functions as callback functions because receiving function calls them back.

### Example:

```
#include<stdio.h>

int conditionalSum(int a, int b,int (*ptr)()) {
    // modify the arguments according to the condition
    // of the function ptr points to
    a = (*ptr)(a);
    b = (*ptr)(b);
    return a + b;
}

int square(int a) {
    // function return square power of a number
    return a * a;
}
```

```

int cube(int a) {
    // function return cubic power of a number
    return a * a * a;
}
int main() {
    int (*fp)(int);
    // point function pointer to function square()
    fp = square;

    // sum = 2^2 + 3^2, as fp points to function square()
    int sum = conditionalSum(2, 3, fp);
    printf("Square sum = %d\n", sum);

    // point function pointer to function cube()
    fp = cube;

    // sum = 2^3 + 3^3, as fp points to function cube()
    sum = conditionalSum(2, 3, fp);
    printf("Cubic sum = %d", sum);
    return 0;
}

```

## Output

```

Square sum = 13
Cubic sum = 35

```



## Exercise

1. Write a C program to count vowels and consonants in a string using a pointer
  2. Write a C program to compare two strings using pointers.
  3. Write a C program to create a function to accept a number and check whether it is even or odd. Call the function using a pointer to function.
  4. Create a function in C to accept an integer and function pointer as argument. Write a C program with two functions, one which increments the int argument by 2 and the other which decrements the int argument by 2. Demonstrate the use of passing pointer to function argument using these three functions. User will enter an integer number and select if he wants to add or subtract the number.
  5. Write a C Program to check whether the string entered by the user is a palindrome string or not using a pointer.
  6. Create a two dimensional array to store sales data of Quarter (Q1, Q2, Q3 and Q4) and for four companies (C1, C2, C3, C4). Accept the quarter and company from the user and display the sales figure. Use a pointer to the 2D array and suitable pointer arithmetic.
- 

## Practice Exercise:

- Write a C program to copy a string to another without using strcpy function. Use pointers for both the strings.
- Write a C program to copy one array to another using pointers. The size of both the arrays may be different.
- Create an array of 10 float numbers. Pass them to a function named average. Accept a number m from the user in the function. Calculate the average of the first m numbers of the array and return to the caller.
- Write a C program to accept a string and a character from the user. Now using a pointer delete all the occurrences of the character from the string and return the corrected string. Make it using a user defined function.