

Fundamentals of Computer Programming

UNIT IV : Pointers and Array

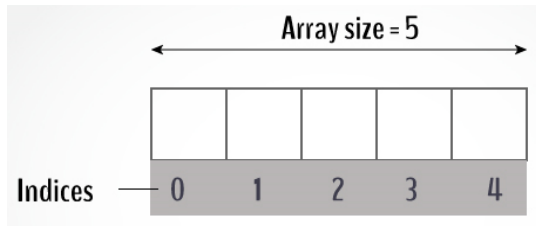
Array

An array is a variable that can store multiple values.

Syntax: `dataType arrayName[arraySize];`

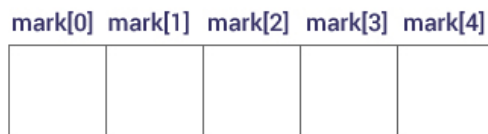
Example: `int data[100];`

The size and type of an array cannot be changed once it is declared.



Access 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. These arrays are called one-dimensional arrays.



- 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.

Array initialization

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

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

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.

| mark[0] | mark[1] | mark[2] | mark[3] | mark[4] |
|---------|---------|---------|---------|---------|
| 19 | 10 | 8 | 17 | 9 |

Change Value of Array elements

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

```
// make the value of the third element to -1
```

```
mark[2] = -1;
```

```
// make the value of the fifth element to 0
```

```
mark[4] = 0;
```

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]);
```

Disadvantages of using an Array

- It is mandatory to determine the size of array to store the elements in it.
- As array elements are stored at consecutive memory locations, so insertion and deletion of an element is difficult/time consuming.
- There is a certain chance of memory wastage/shortage.
- Array size is static in nature so size of array cannot be altered.

Multidimensional array

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.

| | Column 1 | Column 2 | Column 3 | Column 4 |
|-------|----------|----------|----------|----------|
| Row 1 | x[0][0] | x[0][1] | x[0][2] | x[0][3] |
| Row 2 | x[1][0] | x[1][1] | x[1][2] | x[1][3] |
| Row 3 | x[2][0] | x[2][1] | x[2][2] | x[2][3] |

Example: A three-dimensional (3d) array. float y[2][4][3];

Initializing a multidimensional 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};
```

Example: 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)
```

Fundamentals of Computer Programming

```
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

Row/column major formats

In computing, row-major order and column-major order describe methods for storing multidimensional arrays in linear memory. Following standard matrix notation, rows are identified by the first index of a two-dimensional array and columns by the second index. Row-major order is used in C; column major order is used in Fortran and Matlab.

Row-major order

In row-major storage, a multidimensional array in linear memory is accessed such that rows are stored one after the other. It is the approach used by the C programming language as well as many other languages, with the notable exceptions of Fortran and MATLAB. When using row-major order, the difference between addresses of array cells in increasing rows is larger than addresses of cells in increasing columns. For example, consider this 2×3 array:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Declaring this array in C as

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

would find the array laid-out in linear memory as:

1 2 3 4 5 6

The difference in offset from one column to the next is 1 and from one row to the next is 3. The linear offset from the beginning of the array to any given element A[row][column] can then be computed as:

$\text{offset} = \text{row} * \text{NUMCOLS} + \text{column}$

Where NUMCOLS is the number of columns in the array. The above formula only works when using the C convention of labeling the first element 0. In other words, row 1, column 2 in matrix A, would be represented as A[0][1]

Column-major order

Column-major order is a similar method of flattening arrays onto linear memory, but the columns are listed in sequence. The programming languages Fortran, MATLAB, Octave and R use column-major ordering. The array if stored in linear memory with column-major order would look like the following:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

1 4 2 5 3 6

With columns listed first. The memory offset could then be computed as:

offset = row + column*NUMROWS

where NUMROWS represents the number of rows in the array—in this case, 2. Treating a row-major array as a column-major array is the same as transposing it. Because performing a transpose requires data movement, and is quite difficult to do in-place for non-square matrices, such transpositions are rarely performed explicitly.

Pointers

Pointers and address

Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc. A pointer can also be used to refer another pointer, function. A pointer can be incremented/decremented, i.e., to point to the next/ previous memory location.

Pointer Syntax : data_type *var_name; Example : int *p; char *p;

Where, * is used to denote that “p” is pointer variable and not a normal variable.

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

Example:

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

```
int main()
```

```
{
```

```
    int *ptr, q;
```

```
    q = 50;
```

Fundamentals of Computer Programming

```
/* address of q is assigned to ptr */  
ptr = &q;  
/* display q's value using ptr variable */  
printf("%d", *ptr);  
return 0;  
}
```

Output: 50

Working of Pointer

If we declare a variable `v` of type `int`, `v` will actually store a value.

`int v=0;` // `v` is equal to zero now.

However, each variable, apart from value, also has its address (or, simply put, where it is located in the memory). The address can be retrieved by putting an ampersand (&) before the variable name.

`&v`

If you print the address of a variable on the screen, it will look like a totally random number (moreover, it can be different from run to run).

Variable: A value stored in a named storage/memory address

Pointer: A variable that points to the storage/memory address of another variable

Initialize a pointer

After declaring a pointer, we initialize it with a variable address. If pointers are not uninitialized and used in the program, the results are unpredictable and potentially disastrous.

To get the address of a variable, we use the ampersand (&) operator, placed before the name of a variable whose address we need. Pointer initialization is done with the following syntax.

```
pointer = &variable;
```

| Operator | Meaning |
|----------|--|
| * | Serves 2 purpose 1. Declaration of a pointer 2. Returns the value of the referenced variable |
| & | Serves only 1 purpose Returns the address of a variable |

Advantages of Pointers

Fundamentals of Computer Programming

- Pointers are useful for accessing memory locations.
- Pointers provide an efficient way for accessing the elements of an array structure.
- Pointers are used for dynamic memory allocation as well as deallocation.
- Pointers are used to form complex data structures such as linked list, graph, tree, etc.

Disadvantages of Pointers

- Pointers are a little complex to understand.
- Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all.
- If an incorrect value is provided to a pointer, it may cause memory corruption.
- Pointers are also responsible for memory leakage.
- Pointers are comparatively slower than that of the variables.
- Programmers find it very difficult to work with the pointers; therefore it is programmer's responsibility to manipulate a pointer carefully.

Types of a pointer

Null pointer

We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.

Following program illustrates the use of a null pointer:

```
#include <stdio.h>
int main()
{
    int *p = NULL;      //null pointer
    printf("The value inside variable p is:\n%x",p);
    return 0;
}
```

Output:

The value inside variable p is:
0

Void Pointer

In C programming, a void pointer is also called as a generic pointer. It does not have any standard data type. A void pointer is created by using the keyword void. It can be used to store an address of any variable.

Following program illustrates the use of a void pointer:

```
#include <stdio.h>
int main()
{
```


Fundamentals of Computer Programming

```
void *p = NULL;    //void pointer
printf("The size of pointer is:%d\n",sizeof(p));
return 0;
}
```

Output:

The size of pointer is:4

Wild pointer

A pointer is said to be a wild pointer if it is not being initialized to anything. These types of pointers are not efficient because they may point to some unknown memory location which may cause problems in our program and it may lead to crashing of the program. One should always be careful while working with wild pointers.

Following program illustrates the use of wild pointer:

```
#include <stdio.h>
int main()
{
    int *p; //wild pointer
    printf("\n%d",*p);
    return 0;
}
```

Output

timeout: the monitored command dumped core
sh: line 1: 95298 Segmentation fault timeout 10s main

Other types of pointers in 'c' are as follows:

- Dangling pointer
- Complex pointer
- Near pointer
- Far pointer
- Huge pointer

Direct and Indirect Access Pointers

In C, there are two equivalent ways to access and manipulate a variable content

- Direct access: we use directly the variable name
- Indirect access: we use a pointer to the variable

Let's understand this with the help of program below

```
#include <stdio.h>
/* Declare and initialize an int variable */
int var = 1;
/* Declare a pointer to int */
int *ptr;
int main( void )
{
```

Fundamentals of Computer Programming

```
/* Initialize ptr to point to var */
ptr = &var;
/* Access var directly and indirectly */
printf("\nDirect access, var = %d", var);
printf("\nIndirect access, var = %d", *ptr);
/* Display the address of var two ways */
printf("\n\nThe address of var = %d", &var);
printf("\nThe address of var = %d\n", ptr);
/*change the content of var through the pointer*/
*ptr=48;
printf("\nIndirect access, var = %d", *ptr);
return 0;
}
```

After compiling the program without any errors, the result is:

Direct access, var = 1

Indirect access, var = 1

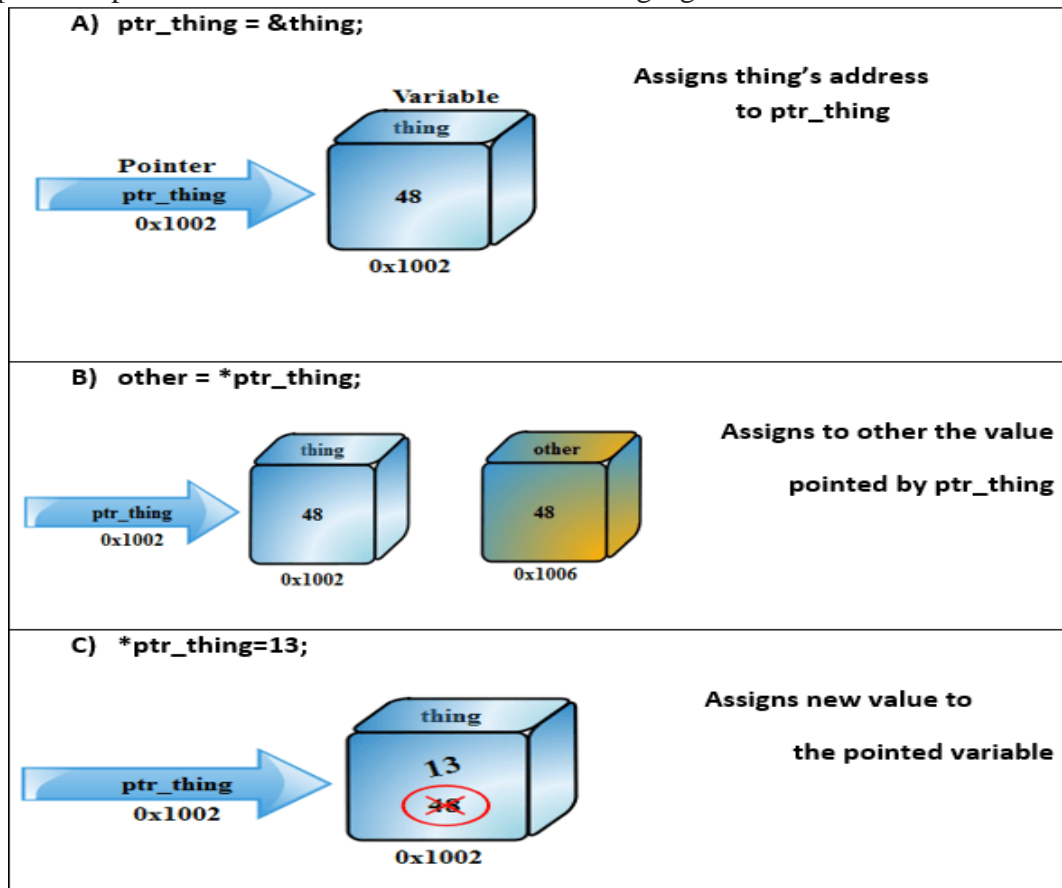
The address of var = 4202496

The address of var = 4202496

Indirect access, var = 48

Pointers Arithmetic

The pointer operations are summarized in the following figure



Pointer Operations

Fundamentals of Computer Programming

Priority operation (precedence)

When working with pointers, we must observe the following priority rules:

- The operators * and & have the same priority as the unary operators (the negation!, the incrementation++, decrement--).
- In the same expression, the unary operators *, &!, ++, - are evaluated from right to left.

If a P pointer points to an X variable, then * P can be used wherever X can be written.

The following expressions are equivalent:

```
int X =10
```

```
int *P = &Y;
```

For the above code, below expressions are true

| Expression | Equivalent Expression |
|------------|-----------------------|
| Y=*P+1 | Y=X+1 |
| *P=*P+10 | X=X+10 |
| *P+=2 | X+=2 |
| ++*P | ++X |
| (*P)++ | X++ |

In the latter case, parentheses are needed: as the unary operators * and ++ are evaluated from right to left, without the parentheses the pointer P would be incremented, not the object on which P points.

Below table shows the arithmetic and basic operation that can be used when dealing with pointers

| Operation | Explanation |
|-----------------------------------|---|
| Assignment | int *P1,*P2 P1=P2; P1 and P2 point to the same integer variable |
| Incrementation and decrementation | Int *P1; P1++;P1-- ; |
| Adding an offset (Constant) | This allows the pointer to move N elements in a table. The pointer will be increased or |

Fundamentals of Computer Programming

| | |
|--|---|
| | decreased by N times the number of byte (s) of the type of the variable. $P1+5$; |
|--|---|

Pointers and Arrays

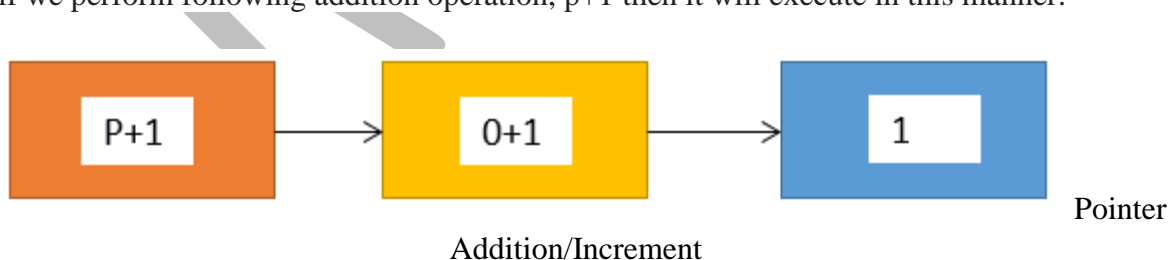
Traditionally, we access the array elements using its index, but this method can be eliminated by using pointers. Pointers make it easy to access each array element.

```
#include <stdio.h>
int main()
{
    int a[5]={1,2,3,4,5}; //array initialization
    int *p; //pointer declaration
    /*the ptr points to the first element of the array*/
    p=a; /*We can also type simply ptr==&a[0] */
    printf("Printing the array elements using pointer\n");
    for(int i=0;i<5;i++) //loop for traversing array elements
    {
        printf("\n%x",*p); //printing array elements
        p++; //incrementing to the next element, you can also write p=p+1
    }
    return 0;
}
```

Output

1
2
3
4
5

Adding a particular number to a pointer will move the pointer location to the value obtained by an addition operation. Suppose p is a pointer that currently points to the memory location 0 if we perform following addition operation, $p+1$ then it will execute in this manner:



Since p currently points to the location 0 after adding 1, the value will become 1, and hence the pointer will point to the memory location 1.

Pointers and Strings

A string is an array of char objects, ending with a null character '\0'. We can manipulate strings using pointers. Here is an example that explains this section

Fundamentals of Computer Programming

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str[]="Hello!";
    char *p;
    p=str;
    printf("First character is:%c\n",*p);
    p=p+1;
    printf("Next character is:%c\n",*p);
    printf("Printing all the characters in a string\n");
    p=str; //reset the pointer
    for(int i=0;i<strlen(str);i++)
    {
        printf("%c\n",*p);
        p++;
    }
    return 0;
}
```

Output

First character is:H

Next character is:e

Printing all the characters in a string

H

e

l

l

o

Another way to deal strings is with an array of pointers like in the following program:

```
#include <stdio.h>
int main(){
    char *materials[ ] = { "iron", "copper", "gold"};
    printf("Please remember these materials :\n");
    int i ;
    for (i = 0; i < 3; i++) {
        printf("%s\n", materials[ i ]);}
    return 0;}
```

Output:

Please remember these materials:

iron

copper

gold

Summary

- A pointer is nothing but a memory location where data is stored.
- A pointer is used to access the memory location.
- There are various types of pointers such as a null pointer, wild pointer, void pointer and other types of pointers.
- Pointers can be used with array and string to access elements more efficiently.
- We can create function pointers to invoke a function dynamically.
- Arithmetic operations can be done on a pointer which is known as pointer arithmetic.
- Pointers can also point to function which make it easy to call different functions in the case of defining an array of pointers.
- When you want to deal different variable data type, you can use a typecast void pointer.

Pointers and Function Arguments

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will effect the original variable.

Example: Swapping two numbers using Pointer

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
    int m = 10, n = 20;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);
    swap(&m, &n); //passing address of m and n to the swap function
    printf("After Swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}
/*pointer 'a' and 'b' holds and points to the address of 'm' and 'n' */
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Output:

```
m = 10
n = 20
After Swapping:
m = 20
```

n = 10

Functions returning Pointer variables

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>
int* larger(int*, int*);
void main()
{
    int a = 15;
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger", *p);
}
int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}
```

Output:

92 is larger

Character Pointers and Functions

Since text strings are represented in C by arrays of characters, and since arrays are very often manipulated via pointers, character pointers are probably the most common pointers in C.

```
char *pmessage;
pmessage = "now is the time";
pmessage = "hello, world";
```

two different ways that string literals like "now is the time" are used in C. In the definition

```
char amessage[] = "now is the time";
```

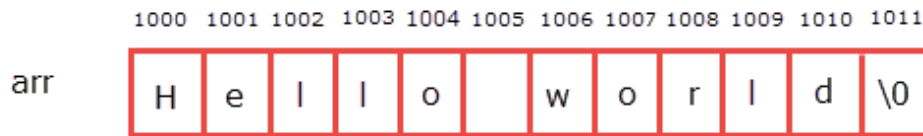
the string literal is used as the initializer for the array amessage. amessage is here an array of 16 characters. In the definition

```
char *pmessage = "now is the time";
```

Fundamentals of Computer Programming

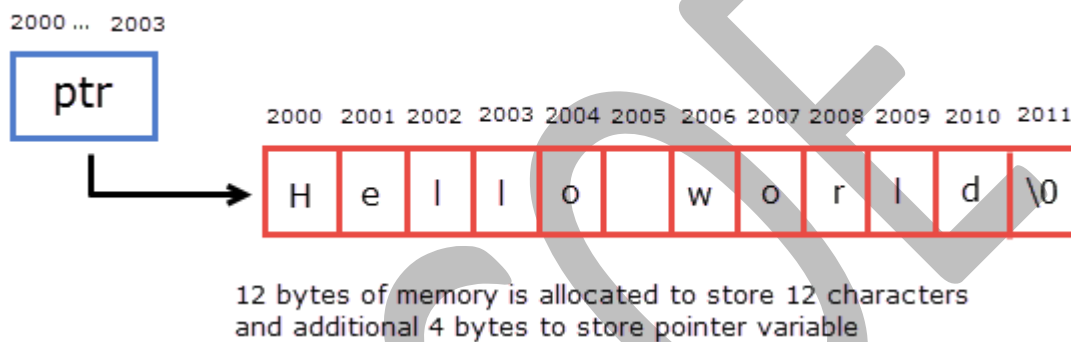
on the other hand, the string literal is used to create a little block of characters somewhere in memory which the pointer pmessage is initialized to point to.

```
char arr[] = "Hello World"; // array version
```



12 bytes of memory is allocated to store 12 characters

```
char ptr* = "Hello World"; // pointer version
```



The type of both the variables is a pointer to char or (char*), so you can pass either of them to a function whose formal argument accepts an array of characters or a character pointer.

Pointer Arrays

Declaration of an array of pointers to an integer –

```
int *ptr[MAX];
```

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value.

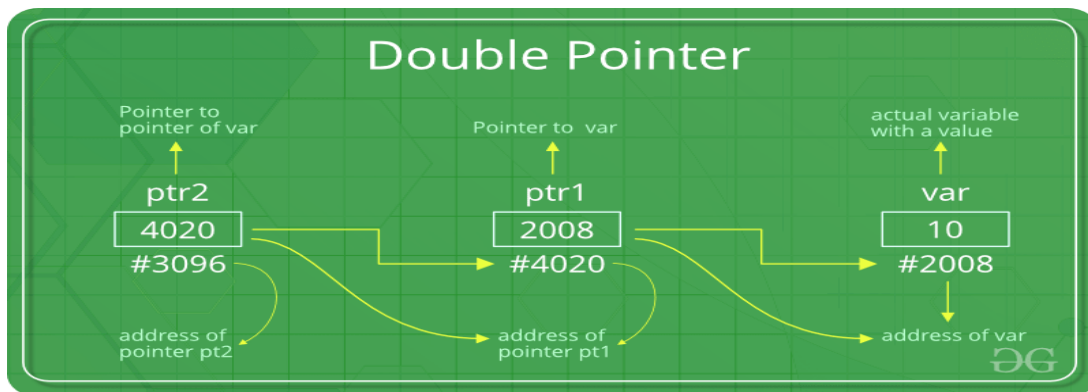
```
#include <stdio.h>
const int MAX = 3;
int main () {
    int var[] = { 10, 100, 200};
    int i, *ptr[MAX];
    for ( i = 0; i < MAX; i++) {
        ptr[i] = &var[i]; /* assign the address of integer. */
    }
    for ( i = 0; i < MAX; i++) {
        printf("Value of var[%d] = %d\n", i, *ptr[i] );
    }
    return 0;
}
```


Pointer to Pointer

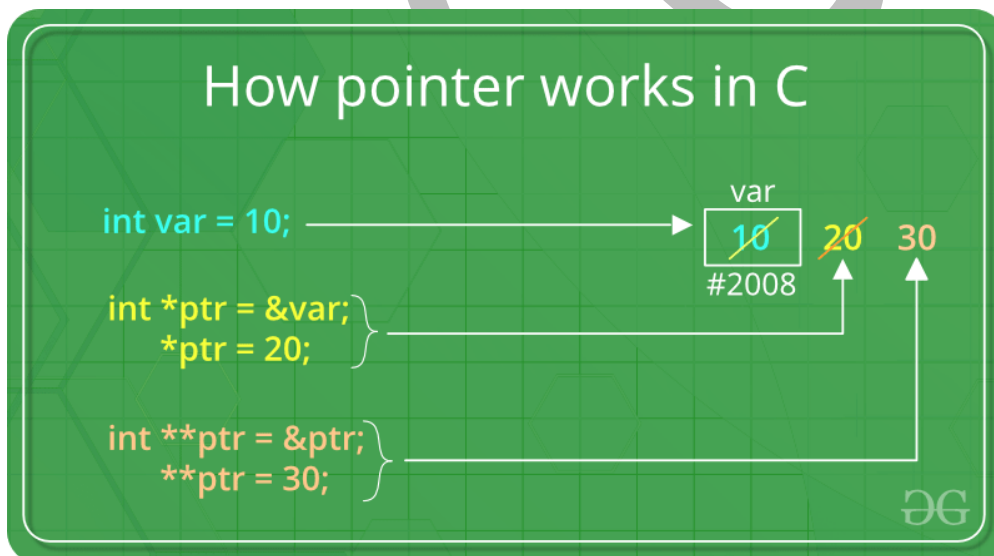
Declaring Pointer to Pointer is similar to declaring pointer in C. The difference is we have to place an additional '*' before the name of pointer.

Syntax:

```
int **ptr; // declaring double pointers
```



The above diagram shows the memory representation of a pointer to pointer. The first pointer ptr1 stores the address of the variable and the second pointer ptr2 stores the address of the first pointer.



// C program to demonstrate pointer to pointer

```
int main()
```

```
{
```

```
    int var = 789;
```

```
    // pointer for var
```

```
    int *ptr2;
```

```
    // double pointer for ptr2
```

```
    int **ptr1;
```

```
    // storing address of var in ptr2
```

Fundamentals of Computer Programming

```
ptr2 = &var;
// Storing address of ptr2 in ptr1
ptr1 = &ptr2;
// Displaying value of var using
// both single and double pointers
printf("Value of var = %d\n", var );
printf("Value of var using single pointer = %d\n", *ptr2 );
printf("Value of var using double pointer = %d\n", **ptr1);
return 0;
}
```

Output:

Value of var = 789

Value of var using single pointer = 789

Value of var using double pointer = 789

Command line arguments

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

```
int main(int argc, char *argv[])
```

argc (ARGument Count) denotes the number of arguments to be passed and argv [] (ARGument Vector) denotes to a pointer array that is pointing to every argument that has been passed to your program.

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

```
#include <conio.h>
```

```
int main(int argc, char *argv[])
{
    int i;
    if( argc >= 2 )
    {
        printf("The arguments supplied are:\n");
        for(i = 1; i < argc; i++)
        {
            printf("%s\t", argv[i]);
        }
    }
    else
    {
        printf("argument list is empty.\n");
    }
}
```

Fundamentals of Computer Programming

```
}  
return 0;  
}
```

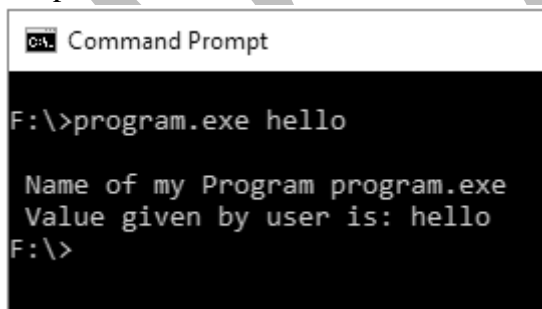
argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.

Example:

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

```
int main( int argc, char *argv [] )  
{  
    printf(" \n Name of my Program %s \t", argv[0]);  
  
    if( argc == 2 )  
    {  
        printf("\n Value given by user is: %s \t", argv[1]);  
    }  
    else if( argc > 2 )  
    {  
        printf("\n Many values given by users.\n");  
    }  
    else  
    {  
        printf(" \n Single value expected.\n");  
    }  
}
```

Output:



```
Command Prompt  
F:\>program.exe hello  
  
Name of my Program program.exe  
Value given by user is: hello  
F:\>
```

Question Bank

1. Consider following declaration and Give the values of following expressions:

```
int x = 10, y = 10;
int *p1 = &x, *p2 = &y;
(*p1) ++
- - (*p2)
*p1 + (*p2)
++ (*p2) - *p1
```

2. What is the output of following?

```
int m[2];
(m + 1) = 100;
*m = * (m+1)
printf("%d", m[0]);
```

3. Give the output of following program:

```
int f(char *p);
main()
{
char str[ ] = "MKICS";
printf("%d", f(str));
}
int f(char *p)
{
Char *q = p;
while (*++p);
return (p-q);
}
```

Short Questions (2- 3 Marks)

4. What is subscript in array? What is the use of it?
5. Define array.
6. Explain Multi-dimensional character array with example.
7. How to pass array to User defined function. Give proper example.
8. Explain need of array variable. What is the difference between character array and numeric array.

Questions (6-7 Marks)

9. Explain the concept of pointers in detail. OR Write a note on Pointer
10. Explain Pointer to Array
11. Explain Arithmetic Operation on pointer.
12. Explain Pointer to Function with proper example.
13. Write following Differences
 - a. 1. Character Array Vs. Numeric Array
14. Define array. Explain the basic concept of array.
15. How to initialize array? Explain with example and write its uses.
16. Write a program to print number from 1 to 10 using array and find sum of them.
17. Write a program to find maximum and minimum number from an array of 10 elements.
18. Define 2-D array with its syntax and explain how to initialize 2-D array.
19. How 1-D and 2-D array elements are stored in memory/ Explain with example.
20. What are strings? Explain with example.
21. What are string handling functions? Explain all with their uses.

Fundamentals of Computer Programming

22. Write a program to addition of two 2-D array.
23. Write a program to multiplication of two 2-D array.
24. Write a program to find length of a given string using strlen() function.
25. Write a program to copy one string into another string using strcpy() function.
26. Write a program to concatenate two string using strcat() function.
27. Write a program to compare two string using strcmp() function.

RSCHOOL