## UNIT IV – POINTERS IN C

Pointers - Passing arguments by address - Dynamic Memory Allocation – Pointer arithmetic - Pointers and one dimensional array - Pointers and Multi-Dimensional Array: Array of Pointers, Pointer to Pointer, Pointer to an array - void Pointer - Pointer to function.

### 4.1 POINTERS:

A pointer is a variable that stores the memory address of another variable. Instead of holding a data value directly, a pointer holds the location where that value is stored.

C pointer is the derived data type that is used to store the address of another variable and can also be used to access and manipulate the variable's data stored at that location. The pointers are considered as derived data types.

### a) Why Use Pointers?

1. **Efficiency**: Pointers can be more efficient for certain operations, especially when dealing with large data structures. Instead of copying entire structures, you can pass around pointers.
2. **Dynamic Memory Management**: Pointers allow for dynamic memory allocation, meaning you can allocate memory during runtime using functions like malloc or new.
3. **Data Structures**: Many complex data structures like linked lists, trees, and graphs rely on pointers to link elements together.

### b) Basic Pointer Syntax

**1. Declaration**: To declare a pointer, you use the * operator. For example:

```
int *p;       // p is a pointer to an int
```

Example of Valid Pointer Variable Declarations

Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
```

```
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

**2.Initialization**: You can initialize a pointer to the address of a variable using the address-of operator &:

```
                    int a = 10;
p = &a;        // p now holds the address of a
```

Example

Here is an example of pointer initialization –

```
int x = 10;
int *ptr = &x;
```

Here, **x** is an integer variable, **ptr** is an integer pointer. The pointer **ptr** is being initialized with **x**.

**c) Referencing and Dereferencing Pointers**

A pointer references a location in memory. Obtaining the value stored at that location is known as **dereferencing the pointer**.

In C, it is important to understand the purpose of the following two operators in the context of pointer mechanism −

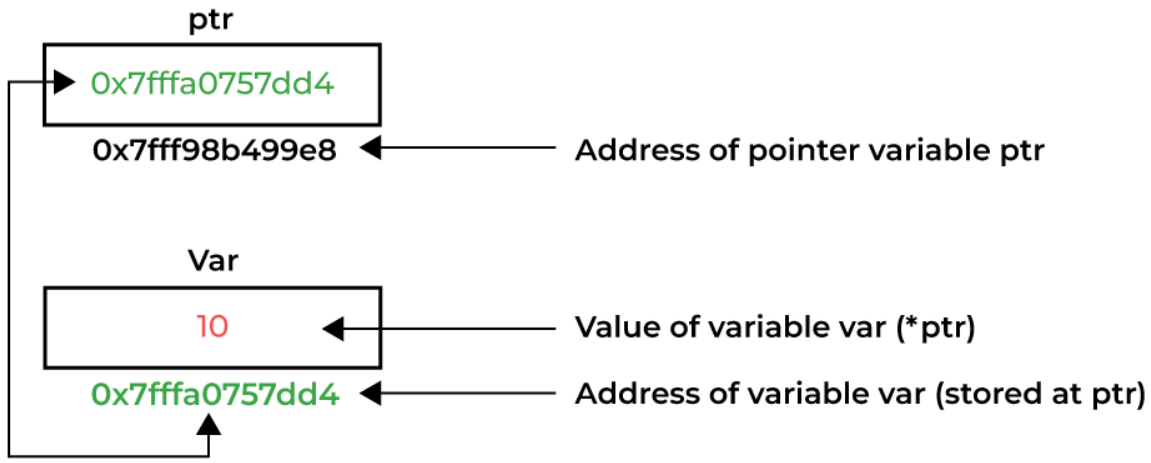
- **The & Operator** − It is also known as the "Address-of operator". It is used for Referencing which means taking the address of an existing variable (using **&**) to set a pointer variable.
- **The * Operator** − It is also known as the "dereference operator". **Dereferencing** a pointer is carried out using the *** operator** to get the value from the memory address that is pointed by the pointer.

Pointers are used to pass parameters by reference. This is useful if a programmer wants a function's modifications to a parameter to be visible to the function's caller. This is also useful for returning multiple values from a function.

**Dereferencing**: To access or modify the value at the address stored in a pointer, you use the dereference operator *:

```
int value = *p; // value is now 10
*p = 20; // a is now 20
```

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same **(*) dereferencing operator** that we used in the pointer declaration.




**How to use pointers**

We can use pointers with any type of variable such as integer, float, string, etc. You can also use pointers with derived data types such as array, structure, union, etc.

Example

In the below example, we are using pointers for getting values of different types of variables.

```c
#include <stdio.h>

int main() {
  int x = 10;
  float y = 1.3f;
  char z = 'p';

  // Pointer declaration and initialization
  int * ptr_x = & x;
  float * ptr_y = & y;
  char * ptr_z = & z;

  // Printing the values
  printf("Value of x = %d\n", * ptr_x);
  printf("Value of y = %f\n", * ptr_y);
  printf("Value of z = %c\n", * ptr_z);

  return 0;
}
```

***Output***

Value of x = 10

Value of y = 1.300000

Value of z = p

## 4.2 PASSING ARGUMENTS BY ADDRESS

**Basics of Argument Passing**

1. **Pass by Value**: When you pass an argument by value, the function receives a copy of the variable. Any changes made to this variable inside the function do not affect the original variable.
2. **Pass by Address**: When you pass an argument by address (or by reference), you provide the function with the memory address of the variable. This allows the function to access and modify the original variable.

How It Works

1. **Using Pointers**: In languages like C/C++, you use pointers to pass by address. A pointer is a variable that stores the memory address of another variable.
2. **Function Definition**: When defining a function that takes an argument by address, you use a pointer type for that argument.

```
void modifyValue(int *ptr) {
    *ptr = 10;  // Dereference the pointer to modify the original value
}
```

**3.Calling the Function**: When calling the function, you pass the address of the variable using the address-of operator (&).

```
int main() {
int num = 5;
modifyValue(&num);  // Pass the address of num
printf("%d\n", num);  // Output will be 10
return 0;
}
```

**Advantages of Passing by Address**

1. **Efficiency**: Passing large data structures (like arrays or structs) by address can be more efficient than passing by value because it avoids copying large amounts of data.
2. **Modification**: Functions can modify the original variable, which can be useful for returning multiple values from a function or changing the state of an object.
3. **Dynamic Memory Management**: It allows functions to manage dynamic memory by passing pointers, which can be allocated and freed inside the function.
4. Passing the pointers to the function means the memory location of the variables is passed to the parameters in the function, and then the operations are performed. The function definition accepts these addresses using pointers, addresses are stored using pointers.

**Program to swap two numbers by using pass by reference method**

**// C program to swap two values using pass by reference**

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

```
 void swap(int* a, int* b)
{
 int temp;
 temp = *a;
 *a = *b;
 *b = temp;
}
 int main()
{
 int a = 10, b = 20;
 printf("Values before swap function are: %d, %d\n",
        a, b);
 swap(&a, &b);
 printf("Values after swap function are: %d, %d",
        a, b);
 return 0;
}
```

**Output**

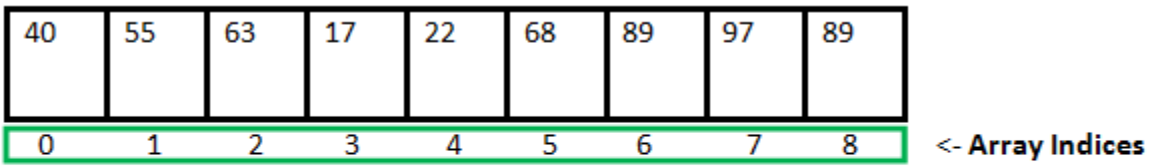
```
Values before swap function are: 10, 20

Values after swap function are: 20, 10
```

## 4.3 DYNAMIC MEMORY ALLOCATION

### Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.




**Array Length = 9**
**First Index = 0**
**Last Index = 8**

As can be seen, the length (size) of the array above is 9. But what if there is a requirement to change this length (size)? For example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.
  - This procedure is referred to as **Dynamic Memory Allocation in C**.
  - Dynamic memory allocation using **malloc()**, **calloc()**, **free()**, and **realloc()** is essential for efficient memory management in C.

**Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

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

### 1.malloc ( ) METHOD

The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast

into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

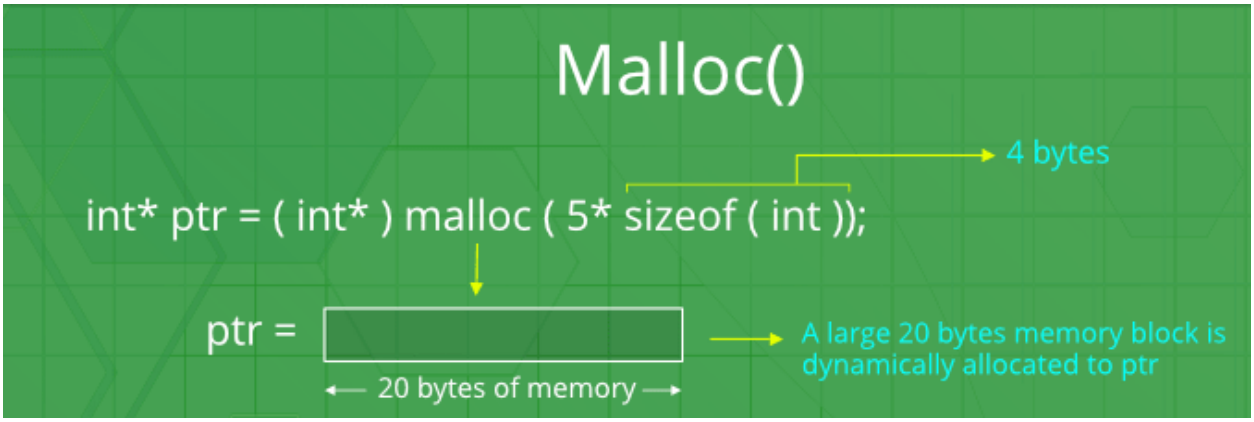
**Syntax of malloc() in C**

```
ptr = (cast-type*) malloc(byte-size)
```

**For Example:**

***ptr = (int*) malloc(100 * sizeof(int));***

*Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.*




If space is insufficient, allocation fails and returns a NULL pointer.

**Example of malloc() in C**

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;
   // Get the number of elements for the array
   printf("Enter number of elements:");
   scanf("%d",&n);
   printf("Entered number of elements: %d\n", n);
   // Dynamically allocate memory using malloc()
```

```
    ptr = (int*)malloc(n * sizeof(int));
    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");
        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
    return 0;
}
```

**Output**

```
Enter number of elements:7
Entered number of elements: 7
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7,
```

**2). <u>calloc() method</u>**

1. **"calloc"** or **"contiguous allocation"** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

Syntax of calloc() in C

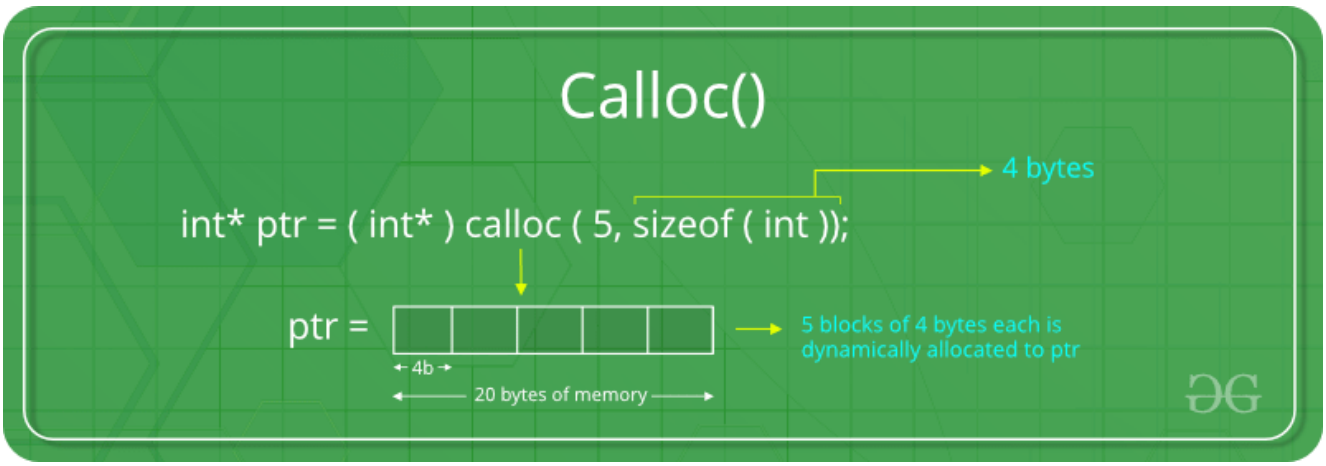
ptr = (cast-type*)calloc(n, element

here, n is the no. of elements and element-size is the size of each element.

**For Example:**

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

*This statement allocates contiguous spa*




If space is insufficient, allocation fails and returns a NULL pointer.

**Example of calloc() in C**

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);
    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));
```

```
    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");
        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
    return 0;
}
```

**Output**

```
Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

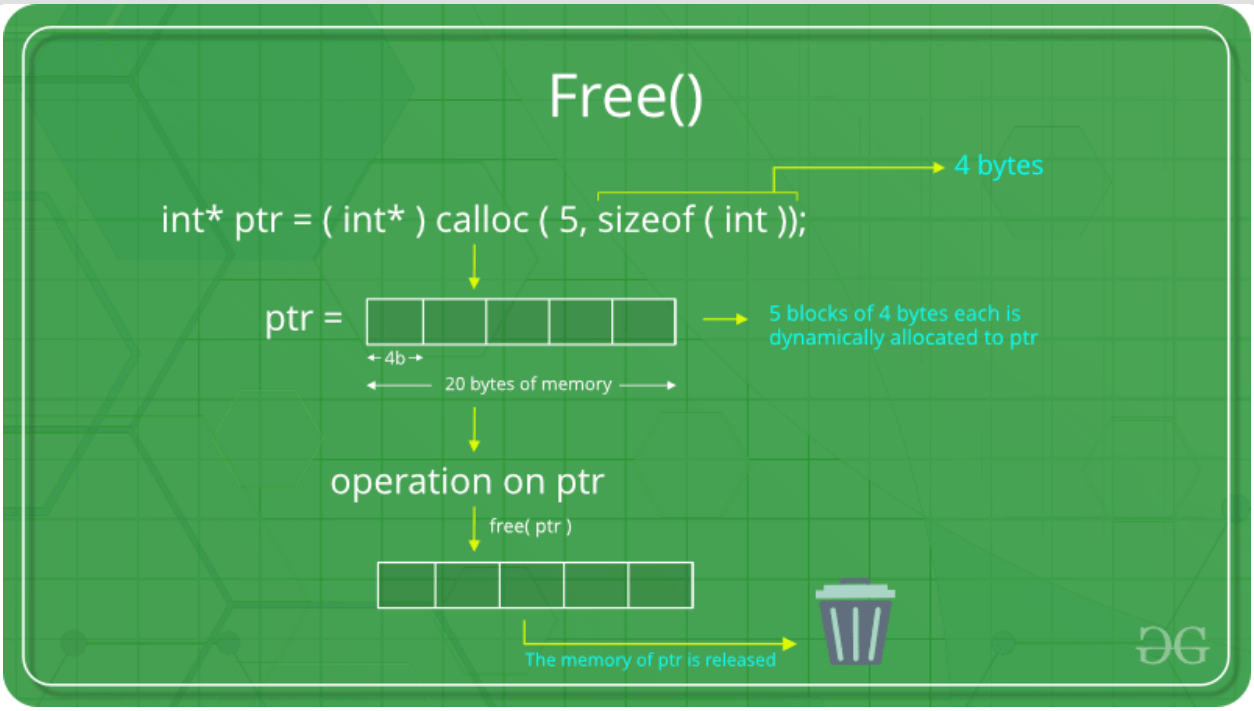
### 3) free() method

**"free"** method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used,

whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

**Syntax of free() in C**

```
free(ptr);
```




**Example of free() in C**

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;
    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);
    // Dynamically allocate memory using malloc()
```

```
    ptr = (int*)malloc(n * sizeof(int));
    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));
    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");
        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");
        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");
        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }
    return 0;
}
```

**Output**

```
Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.
```

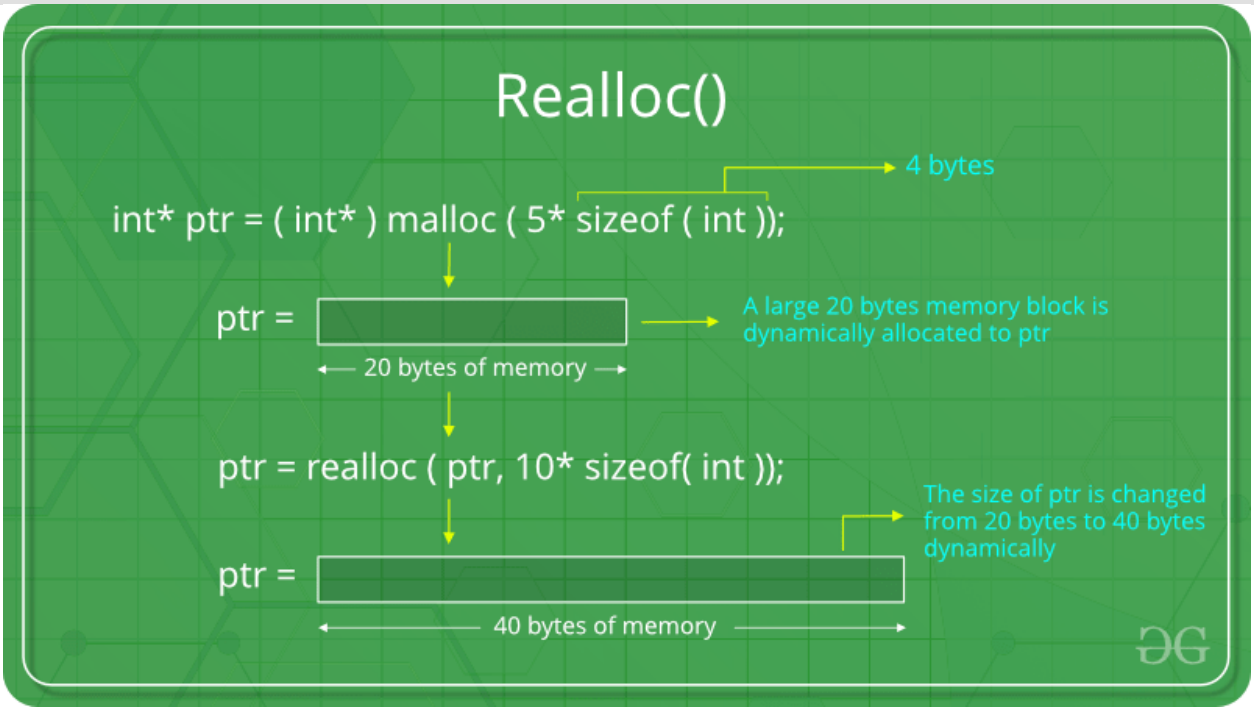
**4). realloc() method**

**"realloc"** or **"re-allocation"** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

**Syntax of realloc() in C**

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.




```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int index = 0, i = 0, n,
      *marks; // this marks pointer hold the base address
            // of  the block created
   int ans;
   marks = (int*)malloc(sizeof(
```

```
    int)); // dynamically allocate memory using malloc
// check if the memory is successfully allocated by
// malloc or not?
if (marks == NULL) {
    printf("memory cannot be allocated");
}
else {
    // memory has successfully allocated
    printf("Memory has been successfully allocated by "
           "using malloc\n");
    printf("\n marks = %pc\n",
           marks); // print the base or beginning
                   // address of allocated memory
    do {
        printf("\n Enter Marks\n");
        scanf("%d", &marks[index]); // Get the marks
        printf("would you like to add more(1/0): ");
        scanf("%d", &ans);

        if (ans == 1) {
            index++;
            marks = (int*)realloc(
                marks,
                (index + 1)
                    * sizeof(
                        int)); // Dynamically reallocate
                               // memory by using realloc
            // check if the memory is successfully
            // allocated by realloc or not?
            if (marks == NULL) {
                printf("memory cannot be allocated");
            }
```

```
            else {
                printf("Memory has been successfully "
                    "reallocated using realloc:\n");
                printf(
                    "\n base address of marks are:%pc",
                    marks); ////print the base or
                            ///beginning address of
                            ///allocated memory
            }
        }
    } while (ans == 1);
    // print the marks of the students
    for (i = 0; i <= index; i++) {
        printf("marks of students %d are: %d\n ", i,
            marks[i]);
    }
    free(marks);
  }
  return 0;
}
```

## 4.4 POINTER ARITHMETIC IN C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement

- Addition
- Subtraction
- Comparison

## 1. Increment using pointers

- We know that "++" and "--" are used as the increment and decrement operators in C. They are unary operators, used in prefix or postfix manner with numeric variable operands, and they increment or decrement the value of the variable by one.
- Assume that an integer variable "x" is created at address 1000 in the memory, with 10 as its value. Then, "x++" makes the value of "x" as 11.

Let's see the example of incrementing pointer variable on 64-bit architecture. #include<stdio.h>

```
int main(){
int number=50;        int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by 4 bytes.
    return 0;
}
```

*Address of p variable is 3214864300*
*After increment: Address of p variable is 3214864304*

**Traversing an array by using pointer**

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

```
    int *p = arr;
  int i;
  printf("printing array elements...\n");
  for(i = 0; i< 5; i++)
  {       printf("%d  ",*(p+i));
     }
   }
```

**Output**

*printing array elements...*
*1 2 3 4 5*

Pointers can be incremented or decremented, which allows for traversing arrays:

**2. <u>Decrementing Pointer in C</u>**

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

1. new_address= current_address - i * size_of(data type)

Let's see the example of decrementing pointer variable on 64-bit OS.

**Example**

```
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
    p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
```

```
    printf
    ("After decrement: Address of p variable is %u \n",p); // P will now point to the immidiate pr
    evious location.
}
```

**Output**

*Address of p variable is 3214864300*
*After decrement: Address of p variable is 3214864296*

**3) Pointer Addition**

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

Let's see the example of adding value to pointer variable on 64-bit architecture.

**syntax**

```
new_address= current_address + (number * size_of(data type))
```

**Example**

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

int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
    printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
    printf("After adding 3: Address of p variable is %u \n",p);
    return 0;
    }
```

**Output**

*Address of p variable is 3214864300*

*After adding 3: Address of p variable is 3214864312*

**4) Pointer Subtraction**

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

1. new_address= current_address - (number * size_of(data type))

```
#include<stdio.h>
int main(){
    int number=50;
int *p;//pointer to int
    p=&number;//stores the address of number variable
    printf("Address of p variable is %u \n",p);
    p=p-3; //subtracting 3 from pointer variable
    printf("After subtracting 3: Address of p variable is %u \n",p);
return 0;
    }
```

**Output**

*Address of p variable is 3214864300*
*After subtracting 3: Address of p variable is 3214864288*

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

```
int arr[] = {1, 2, 3, 4};
int *p = arr; // p points to the first element
printf("%d", *p); // Outputs 1
p++; // Moves to the next integer
printf("%d", *p); // Outputs 2
```

**Null Pointers**

A null pointer is a pointer that does not point to any valid memory location. It's often used to signify that the pointer is not initialized:

```
int *p = NULL;
```

**Access and Manipulate Values using Pointer**

The value of the variable which is pointed by a pointer can be accessed and manipulated by using the pointer variable. You need to use the asterisk (*) sign with the pointer variable to access and manipulate the variable's value.

Example

In the below example, we are taking an integer variable with its initial value and changing it with the new value.

```c
#include <stdio.h>

int main() {
  int x = 10;

  // Pointer declaration and initialization
  int * ptr = & x;

  // Printing the current value
  printf("Value of x = %d\n", * ptr);

  // Changing the value
  * ptr = 20;

  // Printing the updated value
  printf("Value of x = %d\n", * ptr);

  return 0;
}
```

***Output***

Value of x = 10

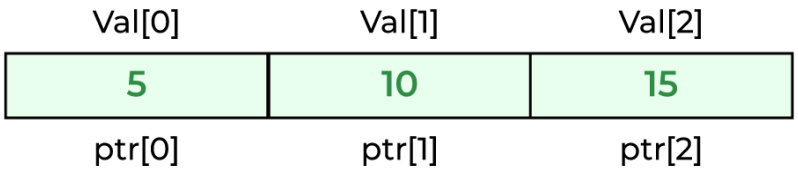
Value of x = 20

## 4.5 POINTERS AND ONE DIMENSIONAL ARRAYS

In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

If we assign this value to a non-constant pointer of the same type, then we can access the elements of the array using this pointer.

**Example 1: Accessing Array Elements using Pointer with Array Subscript**



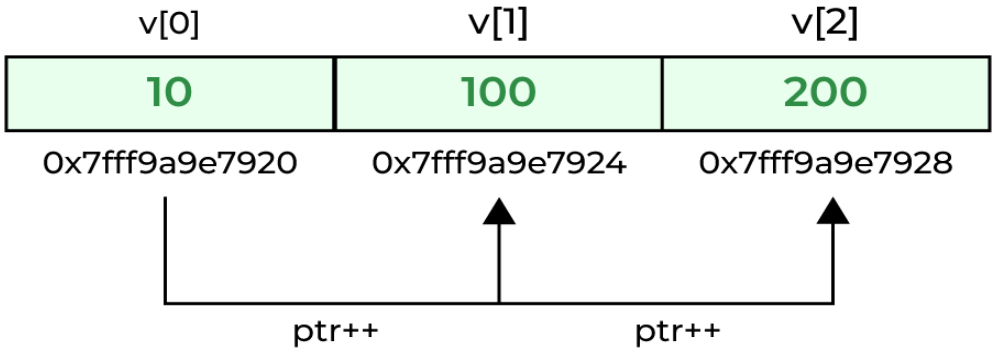

```
// C Program to access array elements using pointer
#include <stdio.h>
void p_array()
{
    int val[3] = {5,10,15};
    int* ptr;
    ptr = val;
    printf("Elements of the array are: ");
   printf("%d, %d, %d", ptr[0], ptr[1], ptr[2]);
    return;
}
int main()
{
    p_array();
    return 0;
}
```

**Output**

```
Elements of the array are: 5, 10, 15
```

Not only that, as the array elements are stored continuously, we can pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

**Example 2: Accessing Array Elements using Pointer Arithmetic**




```
// C Program to access array elements using pointers
#include <stdio.h>
int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    int* ptr_arr = arr;
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr_arr++);
    }
    return 0;
}
```

**Output**

```
1 2 3 4 5
```

**Explain pointers and one-dimensional array in C language**

**Pointers and one-dimensional arrays**

- The compiler allocates Continuous memory locations for all the elements of the array.
- The base address = first element address (index 0) of the array.
    - For Example − int a [5] = {10, 20,30,40,50};

Elements

The five elements are stored as follows −

| Elements | a[0] | a[1] | a[2] | a[3] | a[4] |
|---|---|---|---|---|---|
| Value | 10 | 20 | 30 | 40 | 50 |
| Address | 1000 | 1004 | 1008 | 1012 | 1016 |

base address

a= &a[0]=1000

- If ‘p’ is declared as integer pointer, then, an array ‘a’ can be pointed by the following assignment −

p = a;

(or) p = &a[0];

- Every value of 'a' is accessed by using p++ to move from one element to another element. When a pointer is incremented, its value is increased by the size of the datatype that it points to. This length is called the "scale factor".
- The relationship between ‘p’ and 'a' is explained below −

P = &a[0] = 1000

P+1 = &a[1] = 1004

P+2 = &a[2] = 1008

P+3 = &a[3] = 1012

P+4 = &a[4] = 1016

- Address of an element is calculated using its index and the scale factor of the datatype. An example to explain this is given herewith.

Address of a[3] = base address + (3* scale factor of int)

= 1000 + (3*4)

= 1000 +12

= 1012

- Pointers can be used to access array elements instead of using array indexing.
- *(p+3) gives the value of a[3].

- a[i] = *(p+i)

**Example program**

Following is the C program for pointers and one-dimensional arrays −

```
#include<stdio.h>
main ( ){
  int a[5];
  int *p,i;
  printf ("Enter 5 lements");
  for (i=0; i<5; i++)
    scanf ("%d", &a[i]);
  p = &a[0];
  printf ("Elements of the array are");
  for (i=0; i<5; i++)
    printf("%d", *(p+i));
}
```

**Output**

When the above program is executed, it produces the following result −

Enter 5 elements: 10 20 30 40 50
Elements of the array are : 10 20 30 40 50

## 4.6 POINTERS AND MULTI-DIMENSIONAL ARRAYS

If a one-dimensional array is like a list of elements, a two-dimensional array is like a table or a matrix.The elements in a 2D array can be considered to be logically arranged in rows and columns. Hence, the location of any element is decided by two indices, its row number and column number. Both row and column indexes start from "0".

```
int arr[2][2];
```

Such an array is represented as −

| | **Col0** | **Col1** | **Col2** |
|---|---|---|---|
| Row0 | arr[0][0] | arr[0][1] | arr[0][2] |
| Row1 | arr[1][0] | arr[1][1] | arr[1][2] |
| Row2 | arr[2][0] | arr[2][1] | arr[2][2] |

It may be noted that the tabular arrangement is only a logical representation. The compiler allocates a block of continuous bytes. In C, the array allocation is done in a row-major manner, which means the elements are read into the array in a row−wise manner.

Here, we declare a 2D array with three rows and four columns (the number in the first square bracket always refers to the number of rows) as −

```
int arr[3][4] = {
   {1, 2,  3,  4},
   {5, 6,  7,  8},
   {9, 10, 11, 12}
};
```

The compiler will allocate the memory for the above 2D array in a row−wise order. Assuming that the first element of the array is at the address 1000 and the size of type "int" is 4 bytes, the elements of the array will get the following allocated memory locations −

| | **Row 0** | | | | **Row 1** | | | | **Row 2** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Value** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| **Address** | 1000 | 1004 | 1008 | 1012 | 1016 | 1020 | 1024 | 1028 | 1032 | 1036 | 1040 | 1044 |

We will assign the address of the first element of the array num to the pointer ptr using the address of & operator.

```
int *ptr = &arr[0][0];
```

**Example 1**

If the pointer is incremented by 1, it moves to the next address. All the 12 elements in the "3×4" array can be accessed in a loop as follows −

```
#include <stdio.h>
int main(){
   int arr[3][4] = {
      {1, 2,  3,  4},
      {5, 6,  7,  8},
      {9, 10, 11, 12},
   };
    int *ptr = &arr[0][0];
   int i, j, k = 0;
   for (i = 0; i < 3; i++){
      for (j = 0; j < 4; j++){
         printf("%d   ", *(ptr + k));
         k++;
      }
      printf("\n");
   }
      return 0;
}
```

**Output**

When you run this code, it will produce the following output −

```
1   2   3   4
5   6   7   8
9   10   11   12
```

In general, the address of any element of the array by with the use the following formula −

```
add of element at ith row and jth col = baseAddress + [(i * no_of_cols + j) * sizeof(array_type)]
```

In our 3×4 array,

```
add of arr[2][4] = 1000 + (2*4 + 2)*4 = 1044
```

You can refer to the above figure and it confirms that the address of "arr[3][4]" is 1044.

## 4.7 ARRAY OF POINTERS

### What is an Array of Pointers?

Just like an integer array holds a collection of integer variables, an **array of pointers** would hold variables of pointer type. It means each variable in an array of pointers is a pointer that points to another address.

The name of an array can be used as a pointer because it holds the address to the first element of the array. If we store the address of an array in another pointer, then it is possible to manipulate the array using pointer arithmetic.

### Create an Array of Pointers

To create an array of pointers in C language, you need to declare an array of pointers in the same way as a pointer declaration. Use the data type then an asterisk sign followed by an identifier (array of pointers variable name) with a subscript ([]) containing the size of the array.

In an array of pointers, each element contains the pointer to a specific type.

### Example of Creating an Array of Pointers

The following example demonstrates how you can create and use an array of pointers. Here, we are declaring three integer variables and to access and use them, we are creating an array of pointers. With the help of an array of pointers, we are printing the values of the variables.

```
#include <stdio.h>

int main() {
  // Declaring integers
  int var1 = 1;
```

```
  int var2 = 2;
  int var3 = 3;

  // Declaring an array of pointers to integers
  int *ptr[3];

  // Initializing each element of
  // array of pointers with the addresses of
  // integer variables
  ptr[0] = &var1;
  ptr[1] = &var2;
  ptr[2] = &var3;

  // Accessing values
  for (int i = 0; i < 3; i++) {
    printf("Value at ptr[%d] = %d\n", i, *ptr[i]);
  }

  return 0;
}
```

**Output**

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

Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

There may be a situation when we want to maintain an array that can store pointers to an "int" or "char" or any other data type available.

**(a) <u>An Array of Pointers to Integers</u>**

Here is the 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.

**Example**

The following example uses three integers, which are stored in an array of pointers, as follows −

```
#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;
}
```

**Output**

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

**b) An Array of Pointers to Characters**

You can also use an array of pointers to character to store a list of strings as follows −

```
#include <stdio.h>

const int MAX = 4;

int main(){

   char *names[] = {
      "Zara Ali",
```

```
    "Hina Ali",
    "Nuha Ali",
    "Sara Ali"
  };

  int i = 0;

  for(i = 0; i < MAX; i++){
    printf("Value of names[%d] = %s\n", i, names[i]);
  }

  return 0;
}
```

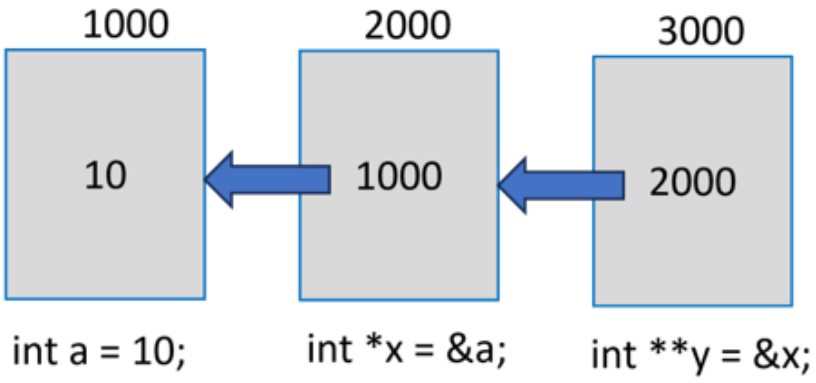
**Output**

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

Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali

## 4.8 POINTER TO POINTER

We may have a pointer variable that stores the address of another pointer itself.




In the above figure, "a" is a normal "int" variable, whose pointer is "x". In turn, the variable stores the address of "x".

Note that "y" is declared as "int **" to indicate that it is a pointer to another pointer variable. Obviously, "y" will return the address of "x" and "*y" is the value in "x" (which is the address of "a").

To obtain the value of "a" from "y", we need to use the expression "**y". Usually, "y" will be called as the **pointer to a pointer**.

## Example

Take a look at the following example −

```c
#include <stdio.h>

int main(){

   int var = 10;
   int *intptr = &var;
   int **ptrptr = &intptr;

   printf("var: %d \nAddress of var: %d \n\n",var, &var);
   printf("inttptr: %d \nAddress of inttptr: %d \n\n", intptr, &intptr);
   printf("var: %d \nValue at intptr: %d \n\n", var, *intptr);
   printf("ptrptr: %d \nAddress of ptrtptr: %d \n\n", ptrptr, &ptrptr);
   printf("intptr: %d \nValue at ptrptr: %d \n\n", intptr, *ptrptr);
   printf("var: %d \n*intptr: %d \n**ptrptr: %d", var, *intptr, **ptrptr);

   return 0;
}
```

## *Output*

Run the code and check its output −

var: 10

Address of var: 951734452

inttptr: 951734452
Address of inttptr: 951734456

var: 10
Value at intptr: 10

ptrptr: 951734456
Address of ptrtptr: 951734464
intptr: 951734452
Value at ptrptr: 951734452

var: 10
*intptr: 10
**ptrptr: 10

## 4.9 VOID POINTER

A void pointer is a pointer that has no associated data type with it. A void pointer can hold an address of any type and can be typecasted to any type.

**Example of Void Pointer in C**

```
// C Program to demonstrate that a void pointer can hold the address of any type-castable type
#include <stdio.h>
int main()
{
    int a = 10;
    char b = 'x';
    // void pointer holds address of int 'a'
    void* p = &a;
    // void pointer holds address of char 'b'
```

```
    p = &b;
}
```

**Properties of Void Pointers**

**1. void pointers cannot be dereferenced.**

**Example**

The following program doesn't compile.

**// C Program to demonstrate that a void pointer cannot be dereferenced**

```
#include <stdio.h>
int main()
{
    int a = 10;
    void* ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

**Output**

```
Compiler Error: 'void*' is not a pointer-to-object type
```

The below program demonstrates the usage of a void pointer to store the address of an integer variable and the void pointer is typecasted to an integer pointer and then dereferenced to access the value. The following program compiles and runs fine.

**// C program to dereference the void pointer to access the value**

```
#include <stdio.h>
int main()
{
    int a = 10;
    void* ptr = &a;
    // The void pointer 'ptr' is cast to an integer pointer using '(int*)ptr' Then, the value is dereferenced with `*(int*)ptr` to get the value at that memory location
    printf("%d", *(int*)ptr);
```

```
    return 0;
}
```

**Output**

```
10
```

## 4.10 POINTER TO FUNCTION

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

**Example program**

```
#include <stdio.h>
// A normal function with an int parameter and void return type
void fun(int a)
{
printf("Value of a is %d\n", a);
}
int main()
{
    void (*fun_ptr)(int);           // fun_ptr is a pointer to function fun()
    fun_ptr = &fun;
    (*fun_ptr)(10);    // Invoking fun() using fun_ptr
    return 0;
}
```

**Output**

```
Value of a is 10
```

**Following are some interesting facts about function pointers.**

**1)** Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

**2)** Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

**3)** A function's name can also be used to get functions' address