Dynamic memory allocation in C

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

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

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.

realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

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

Let's see the example of malloc() function.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
                printf("Sorry! unable to allocate memory");
                exit(0);
            }
```

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

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Let's see the example of calloc() function.

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

```c
#include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc

    if(ptr==NULL)
    {
            printf("Sorry! unable to allocate memory");
            exit(0);
        }
        printf("Enter elements of array: ");
        for(i=0;i<n;++i)
        {   \
            scanf("%d",ptr+i);
            sum+=*(ptr+i);
        }
        printf("Sum=%d",sum);
        free(ptr);
      return 0;
      }
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, **new**-size)

```
#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 malloc 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]);
        }
    }

    // Get the new size for the array
    n = 10;
    printf("\n\nEnter the new size of the array: %d\n", n);

    // Dynamically re-allocate memory using realloc()
    ptr = (int*)realloc(ptr, n * sizeof(int));

    // Memory has been successfully allocated
    printf("Memory successfully re-allocated using realloc.\n");

    // Get the new elements of the array
    for (i = 5; 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]);
    }

    free(ptr);
}
```

```
    return 0;

}
```

Output

```
Enter number of elements: 5

Memory successfully allocated using calloc.

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



Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. free(ptr)

```
#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.
```

C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. **int** n = 10;
2. **int*** p = &n; // Variable p of type pointer is pointing to the address of the v ariable n of type integer.

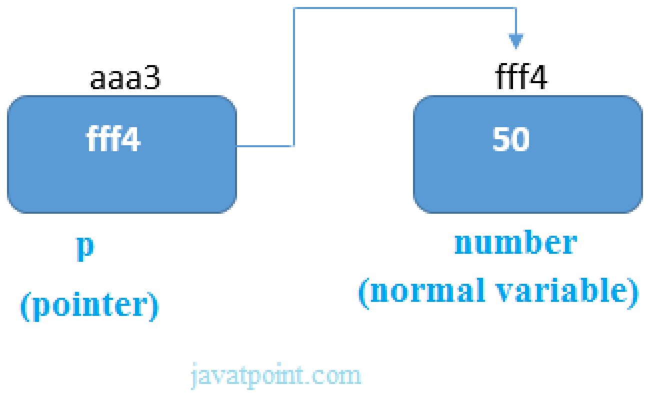
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. **int** *a;//pointer to int
2. **char** *c;//pointer to char

Pointer Example

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

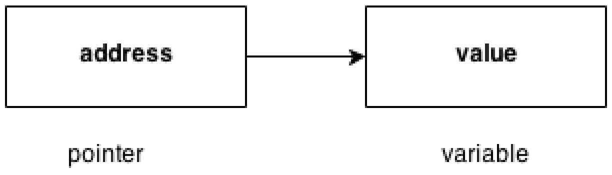
1. **int** arr[10];
2. **int** *p[10]=&arr; // Variable p of type pointer is pointing to the address of a
   n integer array arr.

Pointer to a function

1. **void** show (**int**);
2. **void**(*p)(**int**) = &display; // Pointer p is pointing to the address of a functio
   n

Pointer to structure

1. **struct** st {
2.   **int** i;
3.   **float** f;
4. }ref;
5. **struct** st *p = &ref;



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>
int main(){
int number=50;
printf("value of number is %d, address of number is %u",number,&number);

return 0;
}
```

Output

```
value of number is 50, address of number is fff4
```

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

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

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
#include<stdio.h>
int main(){
int a=10,b=20,*p1=&a,*p2=&b;

printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);

        return 0;
        }
```

Output

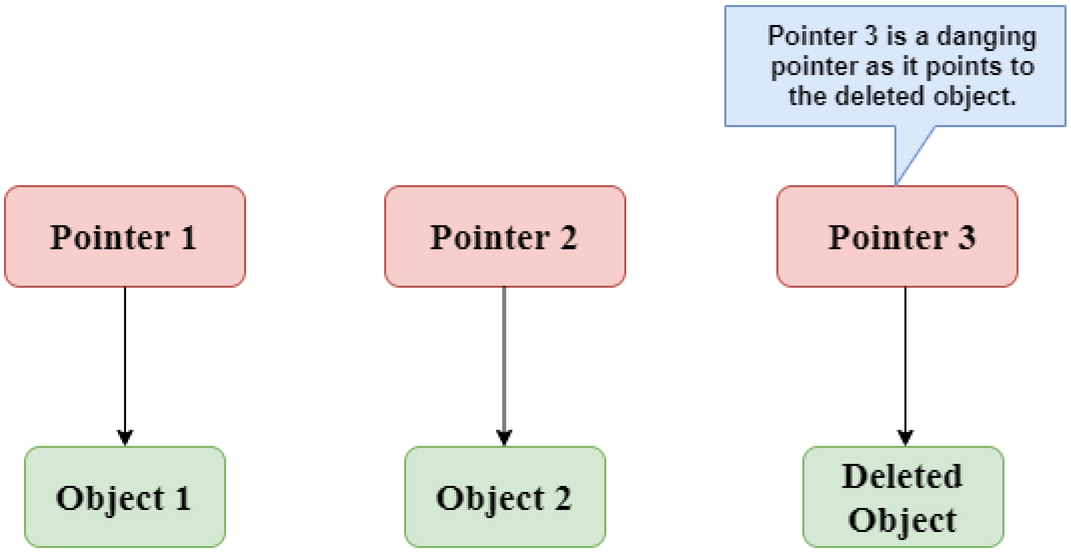
```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

Dangling Pointers in C

The most common bugs related to pointers and memory management is dangling/wild pointers. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Dangling pointer occurs at the time of the object destruction when the object is deleted or de-allocated from memory without modifying the value of the pointer. In this case, the pointer is pointing to the memory, which is de-allocated. The dangling pointer can point to the memory, which contains either the program code or the code of the operating system. If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash. If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.

Let's observe the following examples.


Pointer 3 is a danging pointer as it points to the deleted object.
Pointer 1
Pointer 2
Pointer 3
Object 1
Object 2
Deleted Object

In the above figure, we can observe that the **Pointer 3** is a dangling pointer. **Pointer 1** and **Pointer 2** are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. **Pointer 3** is a dangling pointer as it points to the de-allocated object.

Let's understand the dangling pointer through some C programs.

Using free() function to de-allocate the memory.

```
#include <stdio.h>
int main()
{
  int *ptr=(int *)malloc(sizeof(int));
```

```
    int a=560;
    ptr=&a;
    free(ptr);
    return 0;
}
```

In the above code, we have created two variables, i.e., *ptr and a where 'ptr' is a pointer and 'a' is a integer variable. The *ptr is a pointer variable which is created with the help of **malloc()** function. As we know that malloc() function returns void, so we use int * to convert void pointer into int pointer.