



**Department of Computer Science and Engineering,
PES University, Bangalore, India**

**Lecture Notes
Problem Solving With C
UE24CS151B**

***Lecture #8
Dynamic Memory Management in C***

**By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU**

**Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)**

Unit #: 3**Unit Name: Text Processing and User-Defined Types****Topic: Dynamic Memory Management in C**

Course objectives: The objective(s) of this course is to make students

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Introduction

Dynamic memory management refers to the **ability to allocate, reallocate, and free memory during the execution of a program**. Unlike static memory allocation, where the size and structure of memory are fixed at compile-time, dynamic memory **allows programs to be more flexible and efficient, adapting to varying data sizes or user inputs at runtime**. This is particularly useful for working with data structures such as **linked lists, trees, dynamic arrays, and graphs**, where the amount of memory required cannot be determined in advance.

Few points to think before we proceed:

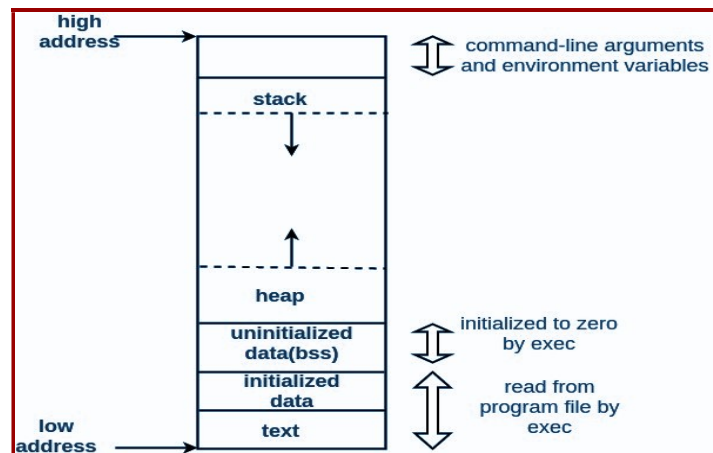
- In case of an array, memory is allocated before the execution time. Is there over utilization or under utilization of memory? - Yes
- Can we take the size of the array at runtime and request to allocate memory for the array at runtime? – This is Variable length Array(VLA).

It is not a good idea to use this as there is no functionality in VLA to check the non availability of the space. If the size is large, **results in code crash without any intimation**.

- Can we avoid this by allocating memory whenever and how much ever we require using some of the functions? – Yes, Use DMM functions

Memory Layout of C Program

The Operating System does the memory management job for C Program. It helps in the **allocation and deallocation of memory blocks** either during **compile-time** or during the **run-time**. When the **memory is allocated during compile-time**, it is **stored** in the **Static Memory** and it is **known** as **Static Memory Allocation**, and when the **memory is allocated during run-time**, it is stored in the **Dynamic Memory** and it is known as **Dynamic Memory Allocation**. When a C program is executed, its memory is divided into several distinct segments, each with a specific purpose. Understanding this layout is essential for effective use of memory, especially when working with pointers and dynamic memory. Refer to the diagram in the next page for the pictorial representation of Memory Layout.



Text segment

It contains **machine code** of the **compiled program**. Usually, it is sharable so that only a single copy needs to be in memory for frequently executed programs, such as **text editors**, **C compiler**, **shells**, and so on. The text segment of an **executable object file** is usually **read-only segment** that prevents a program from being accidentally modified.

Initialized Data Segment

Stores all **global, static, constant, and external variables** - declared with `extern` keyword that are initialized beforehand. It is **not read-only**, since the values of the variables can be changed at run time. This segment can be further classified into **initialized read-only area** and **initialized read-write area**.

```
#include <stdio.h>
const char Entity[]="PES University"; /* global variable stored in Initialized Data Segment in read-only area*/
char Faculty[]="Nitin V Pujari"; /* global variable stored in Initialized Data Segment in read-write area*/
int main()
{
    static int Value = 11; /* static variable stored in Initialized Data Segment*/
    return 0;
}
```

Uninitialized Data Segment(bss)

Data in this segment is initialized to arithmetic 0 before execution of the program. Uninitialized data starts at the end of the data segment and contains all **global variables and static variables that are initialized to 0 or do not have explicit initialization in source code**.

```
#include <stdio.h>
char Entity; /* Uninitialized variable stored in bss*/
char Faculty; /* Uninitialized variable stored in bss*/
int main()
{
    static int Value; /* Uninitialized static variable stored in bss */
    return 0;
}
```

Heap Segment

It is the segment where dynamic memory allocation usually takes place. When some more memory need to be allocated using malloc and calloc function, heap grows upward. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

```
#include <stdio.h>

int main()
{
    int *Value=(int*)malloc(sizeof(int)); /* memory allocating in heap segment */
    return 0;
}
```

Stack Segment

Is used to **store all local variables and is used for passing arguments to the functions** along with the return address of the instruction which is to be executed after the function call is completed. Local variables have a scope to the block where they are defined in, they are created when control enters into the block. All recursive function calls are added to stack.

Note: The stack and heap are traditionally located at opposite ends of the process's virtual address space

C does **not provide any operator for dynamic memory management**. Instead, it relies on library functions, known as **memory management functions**, to allocate and release memory during program execution. These functions are declared in the **stdlib.h** header file.

1. malloc():

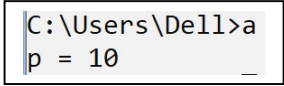
It **allocates the specified number of bytes and returns a void pointer to the first byte** of the allocated memory block. Does not initialize the memory and contains **garbage values**.

Prototype: **void *malloc(size_t size);**

If the allocation is successful, it returns a valid memory address. If it fails (e.g., due to insufficient memory), it returns NULL.

Coding Exampe_1: Dynamic allocation for an integer

```
int main() {  
    int* p = (int*)malloc(sizeof(int)); //dynamic allocation for one integer  
    //address of that location is returned in p  
    *p = 10;  
    printf("p = %d", *p);  
}
```



Coding Exampe_2: Dynamic allocation to n integers and storing n integers

```
#include<stdio.h>  
#include<stdlib.h>  
int main() {  
    int *x; int n, i;  
    printf("Enter the number of elements\n");  
    scanf("%d",&n);  
    x = (int*)malloc(n * sizeof(int)); //memory will be allocated for 5 integer numbers  
    if (x == NULL) //to check if memory is allocated or not by malloc  
        printf("Memory not allocated.\n");  
    else  
    {
```

```

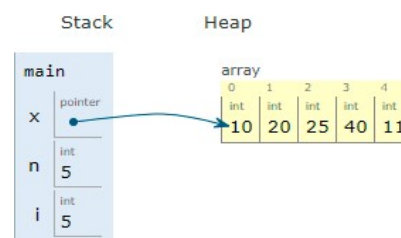
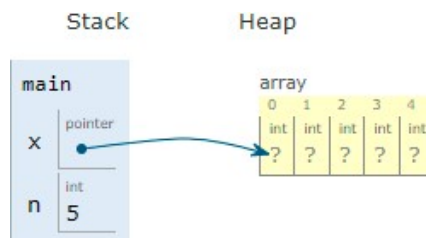
printf("Memory successfully allocated using malloc.\n");
printf("Enter the integer values:\n");
for (i = 0; i < n; i++) // To read inputs from user
{
    scanf("%d",&x[i]);
}
printf("Output: "); // Printing
for (i = 0; i < n; i++)
    printf("%d\t", *(x+i));
}
return 0;
}

```

```

C:\Users\Dell>a
Enter the number of elements
5
Memory successfully allocated using malloc.
Enter the integer values:
10
20
25
40
11
Output: 10    20    25    40    11

```



2. calloc():

It allocates space for elements, **initialize them to zero** and then return a void pointer to the memory.

Prototype: **void *calloc(size_t nmemb, size_t size);**

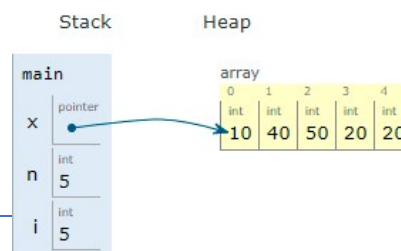
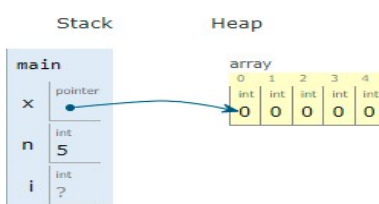
Allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. If calloc fails, it returns NULL. NULL may also be returned by a successful call to calloc() with nmemb or size equal to zero.

In the above Coding Example_2, if malloc is replaces with calloc as below,

```

x = (int*)calloc(n,sizeof(int));

```



3. realloc():

This function is used **to resize a previously allocated memory block** (via malloc() or calloc()). It adjusts the size of the memory block **without losing existing data** (up to the new size).

Prototype: **void *realloc(void *ptr, size_t size);**

It returns a pointer to the newly allocated memory block if the reallocation is successful. This pointer may be the same as the original or a new one, depending on whether the existing block could be resized in place. If the function fails to allocate the requested memory, it returns NULL, and the original memory block remains unchanged. To prevent memory leaks, it is recommended to assign the return value to a temporary pointer before updating the original one.

Note:

If ptr is NULL, then the call is equivalent to malloc(size), for all values of size.

If size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr).

Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done. The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. **If realloc() fails the original block is left untouched, and it is not freed or moved.**

Coding Exampe_3: Demo of realloc if we modify the number of memory blocks allocated using DMA function

```
#include<stdio.h>
#include<stdlib.h>
int new_size;
int main()
{
    int* p = (int*)malloc(3*sizeof(int));
    if(p == NULL)
        printf("memory not allocated\n");
    else
    {
```



```

    int i;
    printf("initial address is %p\n",p);
    printf("enter three elements\n");
    for(i = 0;i<3;i++)
    {
        scanf("%d",&p[i]);
    }
    printf("entered elements are\n");
    for(i = 0;i<3;i++)
    {
        printf("%d\t",p[i]);
    }
}

printf("\nenter the new size: ");
scanf("%d",&new_size);
p = (int*)realloc(p,new_size*sizeof(int));
if(p==NULL)
    printf("not allocated\n");
else
{
    int i;
    printf("new address is %p\n",p);
    for(i = 0;i<new_size;i++)
    {
        printf("%d\t",p[i]);
    }
}

return 0;
}

```

```

C:\Users\Dell>a
initial address is 006813E8
enter three elements
33
22
15
entered elements are
33    22    15
enter the new size: 10

new address is 006813E8
33    22    15    1547322173    150994953    3194    6819304 682079
2     977485170    1852397404

```

New size > initial size

```

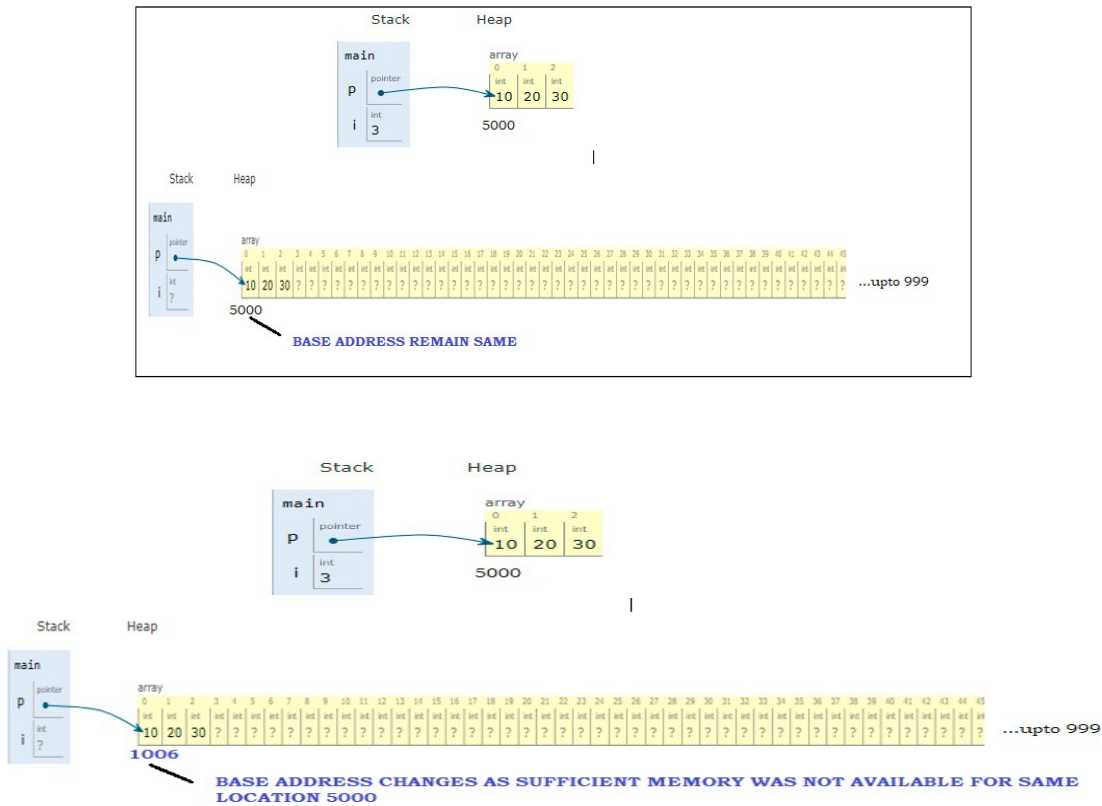
C:\Users\Dell>a
initial address is 00CF13E8
enter three elements
34
45
12
entered elements are
34    45    12
enter the new size: 2

new address is 00CF13E8
34    45

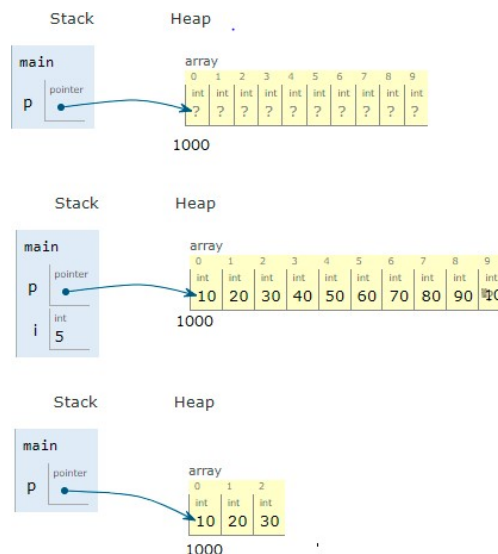
```

New size <= initial size

In Coding Example_3, when new size is greater than 3, here are the pictorial representations.



In Coding Example_3, when new size is less than 3, here is the pictorial representation.

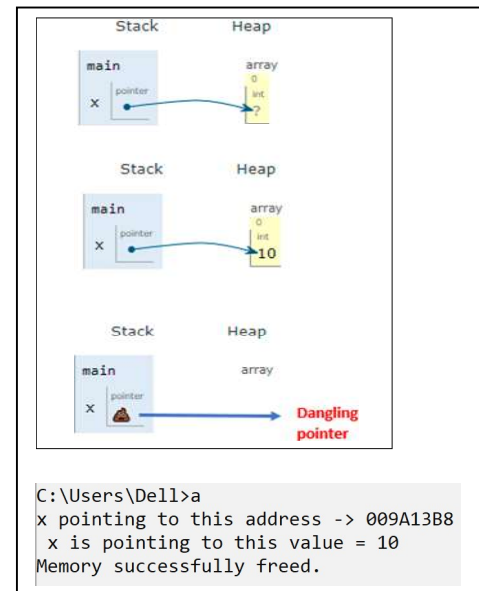


4. free()

It is used to **deallocate memory** that was previously allocated using malloc(), calloc(), or realloc(). Helps preventing **memory leaks** by returning the memory back to the system once it is no longer needed.

Coding Exampe_5: Usage of free

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *x;
    x = (int*)malloc(sizeof(int));
    *x = 10;
    printf("x pointing to this address -> %p\n", x);
    printf("x is pointing to this value = %d", *x);
    free(x); // x is dangling pointer after this statement
    printf("\nMemory successfully freed.\n");
    return 0;
}
```



How does free function knows how much memory must be returned/released?

On allocation of Memory using memory management functions, it stores somewhere in memory the # of bytes allocated. This is known as **book keeping information**. We cannot access this info. But implementation has access to it. The function free() finds this size given the pointer returned by allocation functions and de-allocates the required amount of memory. Observing the diagrams, we can understand that once the allocated memory block is deallocated using the function, the pointer still holds the old address and becomes a **dangling pointer**.

Keep coding smartly with Dynamic Memory Allocation!