2.3 DYNAMIC MEMORY ALLOCATION

Objectives

- Learn how to allocate and free memory, and to control dynamic arrays of any type of data in general and structures in particular.

- Practice and train with dynamic memory in the world of work oriented applications.
- To know about the pointer arithmetic
- How to create and use array of pointers.

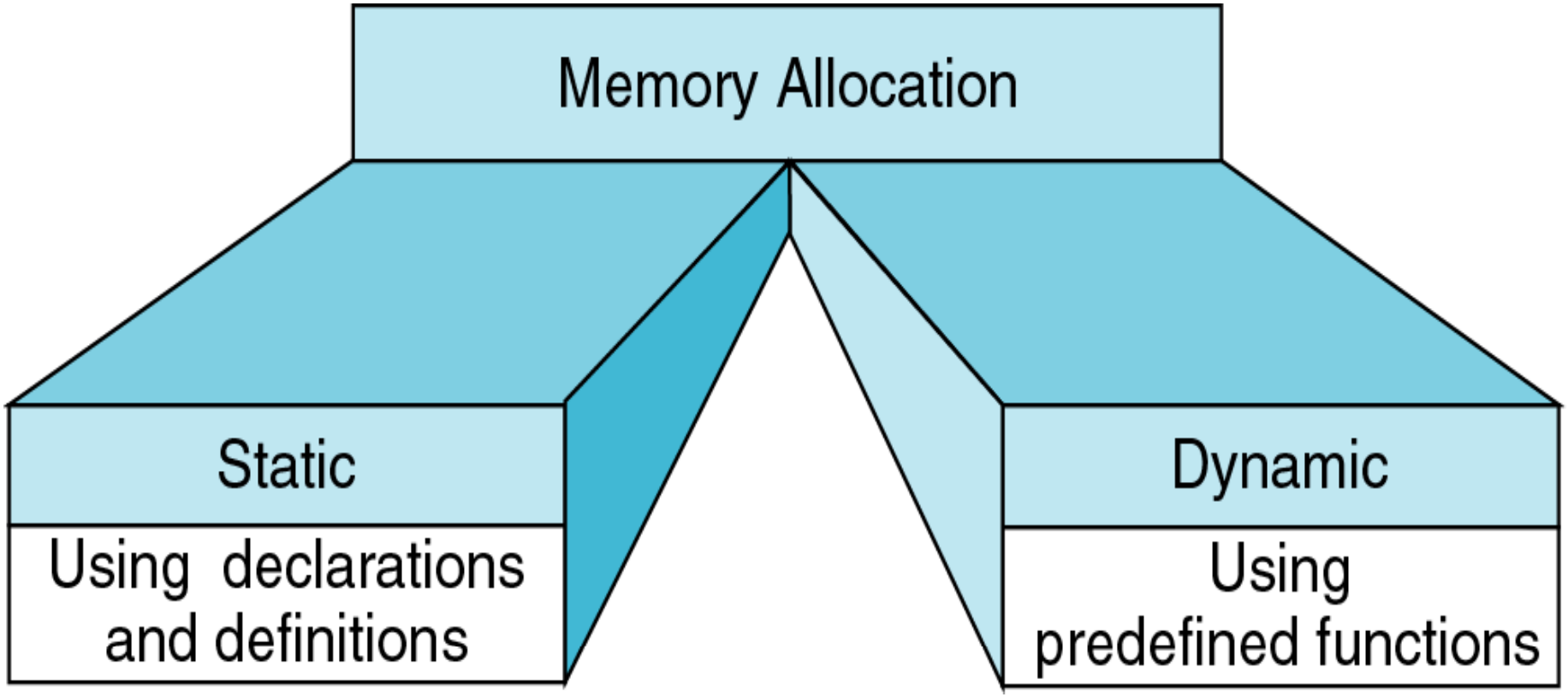
Agenda

- Dynamic memory allocation
  - Malloc
  - Calloc
  - Realloc
  - free
- Pointer Arithmetic
- Array of pointers

Introduction

- While doing programming, if you are aware about the size of an array, then it is easy and you can define it as an array.
- For example to store a name of any person, it can go max 100 characters so you can define something as follows:
  - char name[100]
  - But now let us consider a situation where you have no idea about the length of the text you need to store, for example you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later based on requirement we can allocate memory .

Memory Allocation Function



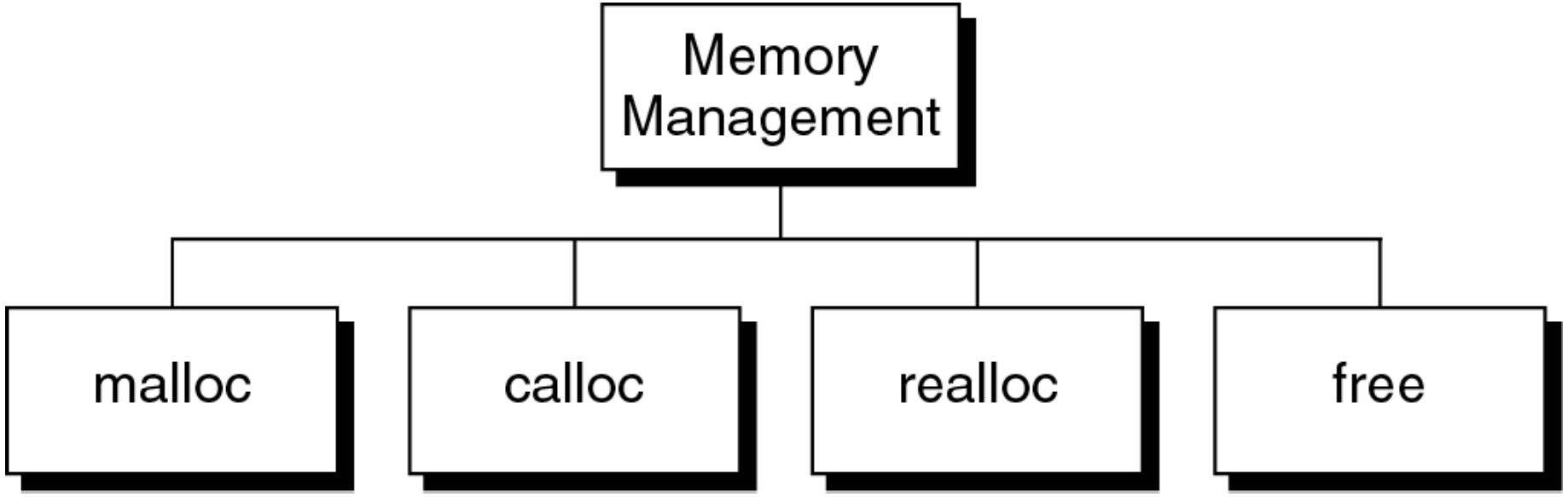
Difference between Static and Dynamic memory allocation

S.no	Static memory allocation	Dynamic memory allocation
1	In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
2	Memory size can't be modified while execution.	Memory size can be modified while execution.
	Example: array	Example: Linked list

Introduction

- Creating and maintaining dynamic structures requires **dynamic memory allocation—** the ability for a program to *obtain more memory space at execution time to hold new values*, and to *release space no longer needed*.

Memory Allocation Functions



Syntax

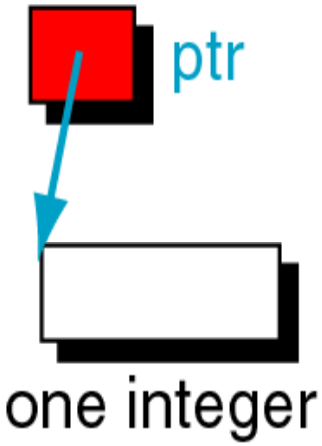
- The following are the function used for dynamic memory allocation
- **void *malloc(int num);**
  - This function allocates an array of **num** bytes and leave them uninitialized.
- **void *calloc(int num, int size);**
  - This function allocates an array of **num** elements each of which size in bytes will be **size**.
- **void *realloc(void *address, int newsize);**
  - This function re-allocates memory extending it upto **newsize**.
- **void free(void *address);**
  - This function releases a block of memory block specified by address.

Block Memory Allocation (malloc)

- Malloc function allocates a block of memory that contains the number of bytes specified in its parameter.
- It returns a void pointer to the first byte of the allocated memory
- The allocated memory is not initialized . We should therefore assume that it will contain unknown values and initialize it as required by our program.
- The function declaration is as follows
  - ***void* malloc (size_t size)***
- If it is not successful malloc return NULL pointer.
- An attempt to allocate memory from heap when memory is insufficient is known as **overflow.**

malloc

- It is up to the program to check the memory overflow
- If it doesn't the program produces invalid results or aborts with an invalid address the first time the pointer is used.



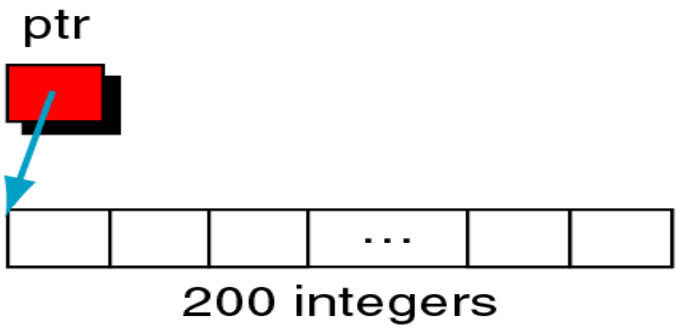

```
if (!(ptr = (int *)malloc(sizeof(int))))
   /* No memory available */
   exit (100) ;
/* Memory available */
...
```

malloc

- Malloc function has one more potential error.
- If we call malloc function with zero size , the results are unpredictable.
- It may return a NULL pointer
- ***Never call malloc with a zero size!!!!!***

Contiguous Memory Allocation (calloc)

- Calloc is primarily used to allocate memory for arrays.
- It differs from malloc only in that it sets memory to null characters.
  - ***void *calloc (size_t element-count, size_t element-size)***



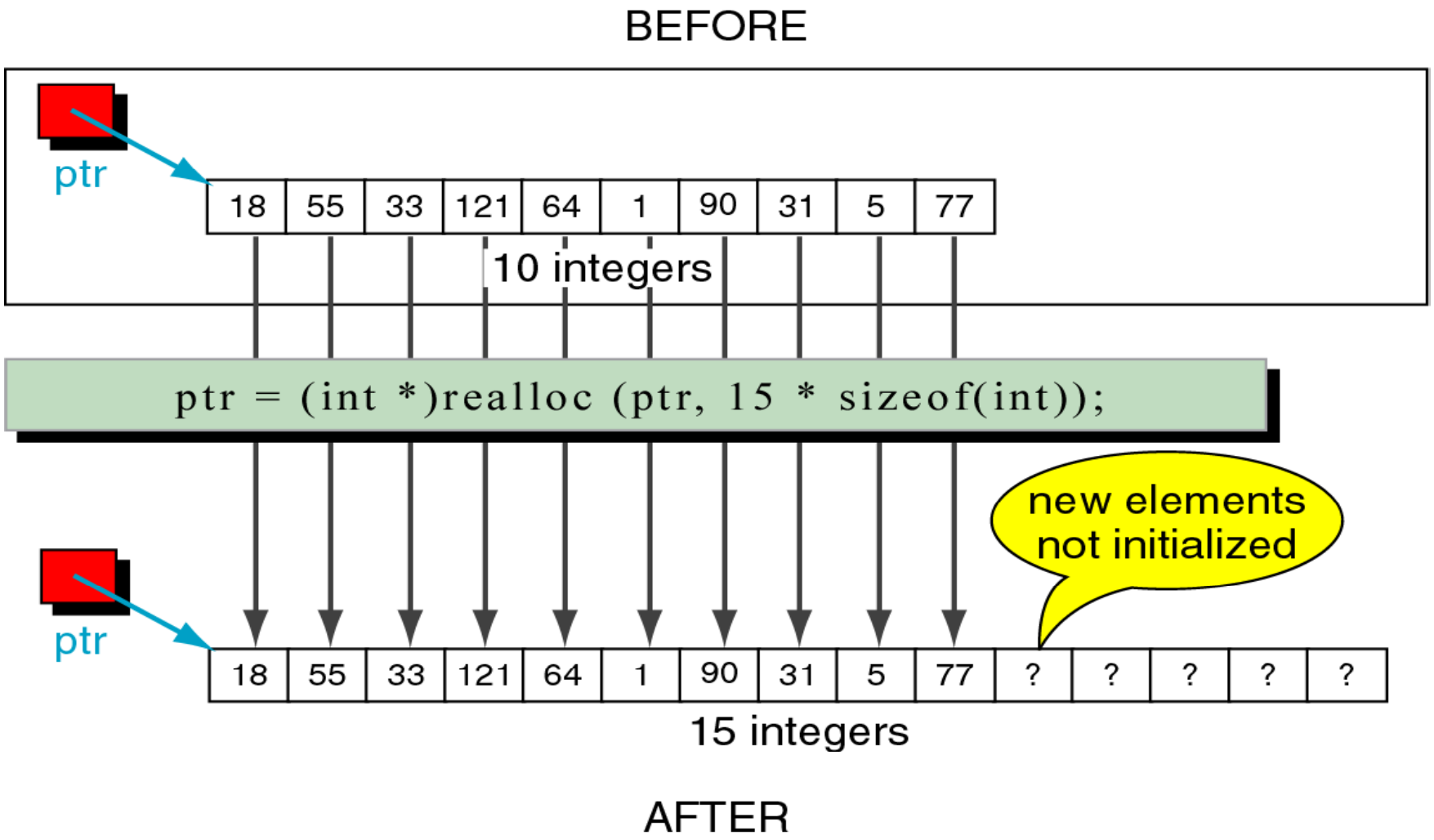

```
if (!(ptr = (int *)calloc (200, sizeof(int))))
   /* No memory available */
   exit (100) ;

/* Memory available */
...
```

Reallocation of memory(realloc)

- The realloc function can be highly inefficient and should be used advisedly.
- When given a pointer to the previously allocated block of memory, realloc changes the size of the block by deleting or extending the memory at the end of the block.
- If memory cannot be extended because of other allocations, realloc allocates a completely new block and copies the existing memory allocation to new allocation, and deletes the old allocation.
  - ***void *realloc (void* ptr, size_t newSize)***

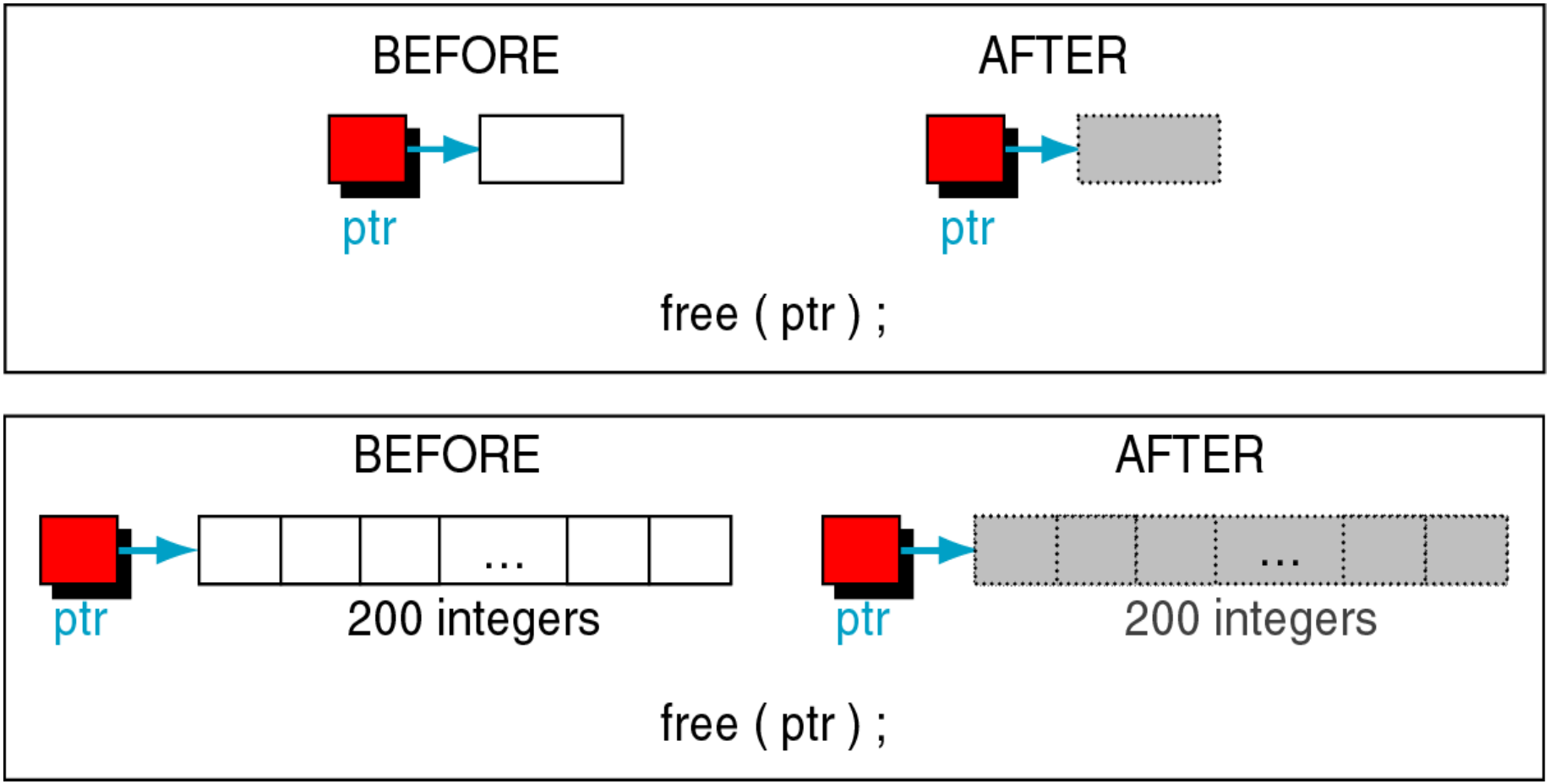
realloc


BEFORE
ptr
18
55
33
121
64
1
90
31
5
77
10 integers
ptr = (int *)realloc (ptr, 15 * sizeof(int));
new elements
not initialized
ptr
18
55
33
121
64
1
90
31
5
77
?
?
?
?
?
15 integers
AFTER

Releasing Memory (free)

- When memory locations allocated by *malloc, calloc or realloc* are no longer needed, they should be freed using the predefined function ***free***.
  - ***void free(void* ptr)***
- Below shows the example where first one releases a single element allocated with malloc
- Second example shows 200 elements were allocated with calloc . When free the pointer 200 elements are returned to the heap.

free


BEFORE
AFTER
ptr
ptr
free (ptr) ;
BEFORE
AFTER
...
...
ptr
200 integers
ptr
200 integers
free (ptr) ;

Difference between malloc and calloc

S.no	malloc()	calloc()
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block.	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory.
	Total = 80 bytes	Total = 1600 bytes
3	malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
4	type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

Resizing and Releasing Memory

- When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

- Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**.

Example-1

```
#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("Error! memory not allocated.");
      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;
}
```

Enter number of elements: 3
Enter elements of array: 2 7 1
Sum=10

Dynamic-ex1.c

Example - 2

```
#include <stdio.h>
#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));
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      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;
}
```

Enter number of elements: 3
Enter elements of array: 2 1 3
Sum=6

Dynamic-ex2.c

Example - 3

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int *ptr,i,n1,n2;
   printf("Enter size of array: ");
   scanf("%d",&n1);
   ptr=(int*)malloc(n1*sizeof(int));
   printf("Address of previously allocated memory: ");
   for(i=0;i<n1;++i)
       printf("%u\t",ptr+i);
   printf("\nEnter new size of array: ");
   scanf("%d",&n2);
   ptr=realloc(ptr,n2);
   for(i=0;i<n2;++i)
       printf("%u\t",ptr+i);
   return 0;
}
```

```
Enter size of array: 3
Address of previously allocated memory: 7474944
7474948 7474952
Enter new size of array: 5
7474944 7474948 7474952 7474956 7474960
```

Example - 4

```
#include <stdio.h>
#include <string.h>
int main()
{
    char name[100];
    char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 200 * sizeof(char) );
    if( description == NULL )
    {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
        {
        strcpy( description, "Zara ali a DPS student in class 10th");
    }
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
}
```

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```


Dynamic-ex4.c

Example - 5

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char name[100];
    char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 30 * sizeof(char) );
    if( description == NULL )
    {
            fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
            strcpy( description, "Zara ali a DPS student.");
    {
```

Example – 5 Cont---

```
    /* suppose you want to store bigger description */
    description = realloc( description, 100 * sizeof(char) );
    if( description == NULL )
    {
            fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
            strcat( description, "She is in class 10th");
    }
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
    /* release memory using free() function */
    free(description);
}
```

Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th

Dynamic-ex5.c

Memory Leaks

- A memory leak occurs when allocated memory is never used again but is not freed.
- I can happen when
  - The memory's address is lost
  - The free function is never invoked though it should be
- The problem with memory leak is that the memory cannot be reclaimed and use later. The amount of memory available t the heap manager will be decreased.
- If the memory is repeatedly allocated and then lost, then the program may terminate when more memory is needed but malloc cannot allocate it because it ran out of memory.

Example

```
char *chunk;
while(10
{
        chunk=(char*) malloc (1000000);
        printf("allocating\n");
        }
```

- The variable chunk is assigned memory from heap. However this memory is not freed before another block of memory is assigned to it.
- Eventfully the application will run out of memory and terminate abnormally.