DEFINITION OF DYNAMIC MEMORY ALLOCATION

Dynamic memory allocation is necessary to manage available memory. For example, during compile time, we may not know the exact memory needs to run the program. So, for the most part, memory allocation decisions are made during the run time. C also does not have automatic garbage collection like Java does. Therefore, a C programmer must manage all dynamic memory used during the program execution.

Definition: "C dynamic memory allocation refers to performing manual memory management for dynamic memory allocation in the C programming language via a group of functions in the C standard library, namely malloc, realloc, calloc and free."

The C programming language manages memory statically, automatically, or dynamically. Static-duration variables are allocated in main memory, usually along with the executable code of the program, and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and automatic-duration variables, the size of the allocation must be compile-time constant (except for the case of variable-length automatic arrays. If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

The lifetime of allocated memory can also cause concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by

allocating it from the free store (informally called the "heap"), an area of memory structured for this purpose.

Learning DMA is important for developing algorithm-based applications like Stack and Queue Implementation.

A primitive variable's memory is allocated at compile time called as static memory allocation/compile time memory allocation. Static memory is of fixed size and it is for primitive variables.

Ex.: int a;

Int arr[5];

Struct Emp e[5];

Dynamic memory allocation means for ex. Size of an array can be increased or decreased based on the elements we are adding or deleting. i.e input is unknown. It is used when the amount(size) of memory is variable and is known only during run-time. Dynamic allocation is achieved using certain functions like malloc(), calloc(), realloc(), free in C and "new", "delete" in C++. To allocate memory dynamically we need to take the help of certain predefined functions in stdlib.h library. stdlib.h provides 4 different functions that are used for allocating and deallocating the memory they are: Malloc(), calloc(), realloc(), free()

ACCESSING THE ADDRESS OF A VARIABLE

A variable's address is accessed using Pointer's. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined

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

```
{
  int var1;
  char var2[10];
  printf("Address of var1 variable: %x\n", &var1  );
  printf("Address of var2 variable: %x\n", &var2  );
  return 0;
}
```

OUTPUT

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

DECLARING AND INITIALIZING POINTERS

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

datatype *pointer–name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations.

int *ip; /*pointer to an integer */

double*dp; /*pointer to an double */

float *fp; /*pointer to an float */

char*ch; /*pointer to an 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.

After declaring a pointer variable, we can store the memory address into it. This process is known as initialization.

Once the pointer variable has been declared, we can use the assignment operator to initialize the variable. Consider the following example:

```
int  var = 20;
int *p;
p = &var;
```

In the above example, we are assigning the address of variable var to the pointer variable p by using the assignment operator.

```
int  var = 20;
int *p = &var;
```

There are a few important operations, which we will do with the help of pointers very frequently.

(a) We define a pointer variable

(b) assign the address of a variable to a pointer and

(c) finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations

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

  int  var = 20;  /* actual variable declaration */
  int  *ip;       /* pointer variable declaration */

ip = &var;  /* store address of var in pointer variable*/
printf("Address of var variable: %x\n", &var  );
/* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip );
```

```
/* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );
return 0;
}
```

OUTPUT

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>
int main () {
int  *ptr = NULL;
printf("The value of ptr is : %x\n", ptr  );
return 0;
}
```

OUTPUT

```
The value of ptr is 0
```

ACCESSING A VARIABLE USING POINTER

After declaring and initializing a pointer variable, we can access the value of the variable pointed by the pointer variable using the * operator. The * operator is also known as indirection operator or dereferencing operator. Consider the following example:

```
int var = 20;
int *p = &var;
```

printf(”Var = %d”, *p);

In the above example, we are printing the value of the variable var by using the pointer variable p along with the * operator.

STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION

Static Memory Allocation: Memory is allocated for the declared variable by the compiler. The address can be obtained by using ‘address of’ operator and can be assigned to a pointer. The memory is allocated during compile time. Since most of the declared variables have static memory, this kind of assigning the address of a variable to a pointer is known as static memory allocation.

Static memory allocation process is done at compile time; in static allocation memory is allocated at the beginning of execution to applications, variable. In static memory allocation even if we do not need most of the memory at particular instance of program or variable still we could not use allocated memory for any other purpose.

Every day large amount of data is generated termed as Big data, this big data need not only store but also maintain replication of it for fast accessing purpose and to avoid data loss because of system crash or natural disaster. Memory management issue comes because of inefficient way of allocating memory and de-allocating memory.

For example, we are declaring static array to stored integer data, when we declare array of 5000, Required memory = 5000*2 = 10000 bytes on windows (32 bit) and Required memory = 5000*4 = 20000 on Linux or 64-bit windows. 10000 (on Windows) and 20000 (on Linux) has reserved for above declared array and we could not use that memory, even that array contains only one integer or no data. Suppose if array has contained 5000 integers, we have deleted 4999 still we could not use that memory for other purpose.

Above strategy is called static memory allocation. Above problem with memory management arises because of static memory allocation, in such

situation static memory allocation fails to handle memory management efficiently. Still static memory allocation has few advantages over dynamic memory allocation like allocating fast speed, mostly not faced fragmentation problem, no extra algorithms required to achieve allocation task. Even these benefits with static memory allocation still mostly we prefer dynamic allocation because of its way of allocating memory. Now we will discuss about dynamic memory allocation and also discuss why dynamic memory allocation is better than static memory allocation technique.

Dynamic Memory Allocation: Allocation of memory at the time of execution (run time) is known as dynamic memory allocation. The functions calloc() and malloc() support allocating of dynamic memory. Dynamic allocation of memory space is done by using these functions when value is returned by functions and assigned to pointer variables.

Dynamic Memory Allocation is also known as Manual Memory Management. In DMA the memory is allocated at run time. It is allocated whenever program, application, data, variable demands with required amount of bytes. The C programming language manages memory statically, automatically, or dynamically. Static-duration variables are allocated in main memory, usually along with the executable code of the program, and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and automatic-duration variables, the size of the allocation must be compile-time constant (except for the case of variable-length automatic arrays. If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

The lifetime of allocated memory can also cause concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations

the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the free store (informally called the "heap"), an area of memory structured for this purpose.

MEMORY ALLOCATION FUNCTIONS

C language offers 4 dynamic memory allocation functions. They are,

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

Function	Syntax
malloc()	malloc(number *sizeof(int));
calloc()	calloc(number, sizeof(int));
realloc()	realloc(pointer_name, number*sizeof(int));
free()	free(pointer_name);

THE MALLOC() FUNCTION:

malloc() function is used to allocate space in memory during the execution of the program.

malloc() does not initialize the memory allocated during execution. It carries garbage value.

malloc() function returns null pointer if it couldn’t able to allocate requested amount of memory.

General syntax: malloc(number *sizeof(int));

The malloc function allocates a memory block of size n bytes (size_t is equivalent to an unsigned integer) The malloc function returns a pointer (void*)

to the block of memory. That void* pointer can be used for any pointer type. malloc allocates a contiguous block of memory. If enough contiguous memory is not available, then malloc returns NULL. Therefore, always check to make sure memory allocation was successful by using,

```
void* p; if ((p=malloc(n)) == NULL)
 return 1;
 else
{ /* memory is allocated */}
```

Example: if we need an array of n ints, then we can do int* A = malloc(n*sizeof(int));

A holds the address of the first element of this block of 4n bytes, and A can be used as an array. For example,

```
if (A != NULL)
 for (i=0;i<n;i++)
 A[i] = 0;
```

will initialize all elements in the array to 0. We note that A[i] is the content at address (A+i). Therefore we can also write

```
for (i=0;i<n;i++)
 *(A+i) = 0;
```

Recall that A points to the first byte in the block and A+I points to the address of the ith element in the list. That is &A[i]. We can also see the operator [] is equivalent to doing pointer arithmetic to obtain the content of the address. A dynamically allocated memory can be freed using free function.

For example free(A); will cause the program to give back the block to the heap (or free memory). The argument to free is any address that was returned by a prior call to malloc. If free is applied to a location that has been freed before, a double free memory error may occur. We note that malloc returns a block of void* and therefore can be assigned to any type.

```
double* A = (double*)malloc(n);
```

```
int* B = (int*)malloc(n);
char* C = (char*)malloc(n);
```

In each case however, the addresses A+i, B+i, C+i are calculated differently.

- A + i is calculated by adding 8i bytes to the address of A (assuming sizeof(double) = 8)
- B + i is calculated by adding 4i bytes to the address of B
- C + i is calculated by adding i bytes to the address of C

SAMPLE PROGRAM FOR MALLOC():

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(){
char *mem_allocation;
/* memory is allocated dynamically */
mem_allocation = malloc( 20 * sizeof(char) );
if( mem_allocation== NULL )
{
printf("Couldn't able to allocate requested memory\n");
}
else
{
strcpy( mem_allocation,"fresh2refresh.com");
}
printf("Dynamically allocated memory content : " \"%s\n",
mem_allocation ); free(mem_allocation);
}
```

OUTPUT

```
Dynamically allocated memory content : fresh2refresh.com
```

THE CALLOC() FUNCTION

The C library function calloc(number, sizeof(int)) allocates the requested memory and returns a pointer to it. The difference in malloc and calloc is that malloc does not set the memory to zero where as calloc sets allocated memory to zero.

General Syntax: Calloc(number, sizeof(int));

Parameters:

nitems − This is the number of elements to be allocated.

size − This is the size of elements.

```
int main()
{
    char *mem_allocation;
   /* memory is allocated dynamically */
    mem_allocation = calloc( 20, sizeof(char) );
    if( mem_allocation== NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
        strcpy( mem_allocation,"fresh2refresh.com");
    }
    printf("Dynamically allocated memory content : "
\"%s\n",mem_allocation );
    free(mem_allocation);
    }
```

OUTPUT

Dynamically allocated memory content: fresh2refresh.com

THE REALLOC() FUNCTION

realloc() function modifies the allocated memory size by malloc() and calloc() functions to new size. If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

General Syntax: realloc(pointer_name,number *sizeof(int));

FREE() FUNCTION IN C:

free() function frees the allocated memory by malloc(), calloc(), realloc() functions and returns the memory to the system.

General Syntax: free(pointer_name);

```
int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 * sizeof(char) );
   if( mem_allocation == NULL )
  {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation,"fresh2refresh.com");
   }
   printf("Dynamically allocated memory content : "
   \"%s\n",mem_allocation);
   mem_allocation=realloc(mem_allocation,100*sizeof(char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
```

```
    }
    else
    {
       strcpy( mem_allocation,"space is extended upto " \ "100 characters");
    }
    printf("Resized memory:%s\n",mem_allocation);
    free(mem_allocation);
}
```

OUTPUT:

Dynamically allocated memory content: fresh2refresh.com

Resized memory : space is extended upto 100 characters

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS

MALLOC()	CALLOC()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int)); For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	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 = 1600 bytes
malloc() doesn't initializes the allocated memory. It contains garbage values.	calloc() initializes the allocated memory to zero
e 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));