Dynamic memory

Dynamic memory allocation is the process of allocation of memory space at the run time. We use the concept of dynamic memory allocation to reduce the wastage of memory, and it is the optimal way of memory allocation. To grasp the concept completely, we will see the memory layout of C programming.

Memory Allocation in C:

Memory allocation in C can be divided into four segments. We will discuss each of them in detail because their understanding is crucial for us to understand about memory layout.

Code:

Code composes of all the text segment of our program. We will not go into a more detailed description of this segment as we are already familiar with it.

Variables:

By variables, we mean both global and static variables. Global variables can be used anywhere in the program, while static has its limitations inside the function. The variable segment is further divided into two segments, depending on the data they can store.

- Data segment: stores initialized data i.e., data whose value is already given.

For example:

```
int i = 0;
```

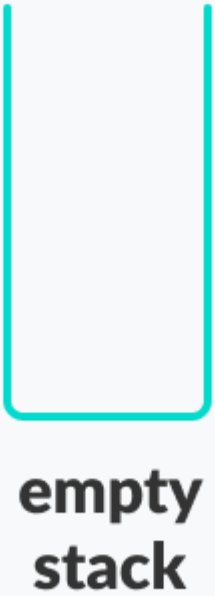
- bss segment: stores uninitialized data i.e., data whose variable is initialized only.

For example:

```
int i = 0;
```

Stack:

The stack is a LIFO data structure. It's size increases when the program moves forward. We will cover the stack in detail in this tutorial. Let us start with the actual description with the help of a diagram.



Initially, the stack looks like a bucket in which the last entry to be inserted will be the first one to get out. That is why it is known as LIFO data structure i.e., last in first out.

Suppose that we push a function A into the stack. Function A will start executing. Now the function A is calling another Function B during its execution. The Function B will be pushed into the stack, and the program will start executing B. Now, if B is calling another function C, then the program will push C into the stack and will start its execution. Now, after C has been executed completely, the program will pop C from the stack as it was the last one in and start executing B. When B has been executed completely, it will be popped out, and A will start executing until the stack becomes empty.

Stack Overflow:

When a stack gets exhausted due to bad programming skills or some logical error, the phenomenon is known as Stack Overflow.

Heap:

Heap is a tree-based data structure. It’s size increases when we allocate memory dynamically. To use the heap data structure, we have to create a pointer in our main function that will point to some memory block in a heap. The disadvantage of using heap is that the memory will not get freed automatically when the pointer gets overwritten.

Static	Dynamic
Allocation before execution	Allocation at run time
Non-reusable memory	Reusable memory
Less optimal way	More optimal way

For the allocation of memory using the heap, we have four functions:

- Malloc
- Calloc
- Realloc
- Free

malloc():

malloc stands for memory allocation. As can be guessed by its name, it requests memory from the heap and returns a pointer to the memory. The pointer is of the void type, so that we can typecast it for any variables. All the values at the allocation time are initialized to garbage values. Its syntax is simple as we have to provide the memory space along with the size we want in bytes.

Syntax:

```
ptr = (ptr-type*) malloc(size_in_bytes)
```

For example:

```
int *ptr;
ptr = (int*) malloc (3* sizeof(int))
```

Note: We are using the sizeof() function here because the size of int may differ in different systems, so to be on the safe side.

calloc():

calloc stands for contiguous allocation. It also requests memory from the heap and returns a pointer to the memory and has the same functionality as malloc(), two main differences, though. We have to send as parameters the number of blocks needed along with their size. The second difference is a major one. That is, in calloc(), the values at the allocation time are initialized to 0 instead of garbage value.

Syntax:

```
ptr = (ptr-type*) calloc(n,size_in_bytes)
```

For example:

```
int *ptr;
ptr = (int*) malloc (10, sizeof(int))
```

realloc():

realloc stands for reallocation. It is used in cases where the dynamic memory is insufficient or wants to increase the already allocated memory to store more data. Its syntax is simple as we just have to overwrite the memory already allocated as a parameter in the function while providing the data related to the pointer.

Syntax:

```
ptr = (ptr-type*) realloc(ptr,new_size_in_bytes)
```

For example:

```
ptr = (int*) realloc (ptr, 5* sizeof(int))
```

free():

As we discussed earlier, while discussing the disadvantages of dynamic memory allocation that we have to free up the allocated memory space manually as there is no automatic procedure for that. So free is used to free up the space occupied by the allocated memory. Its syntax is the easiest of all, as we have to send the pointer as a parameter inside the function.

Syntax:

```
free(ptr)
```