Functions in C for Memory Management

Introduction

In the previous lecture video, we discussed stack and heap memory. Let us look into available functions in C programming language that can be used to create (allocate space) or destroy (deallocate space) memory in heap.

We will focussing on following functions:

1. malloc() for allocation of memory space. We will also talk about realloc() and calloc() functions for the same.
2. free() for deallocation of memory space.

malloc()

The malloc() call is simple. You have to pass a size as an argument, asking for space on the heap. If it succeeds, then it returns back the pointer to the space, otherwise it runs null.

To add on to this, the single parameter malloc() takes is of type size_b (size in bytes), which simply is the amount of bytes you need. However, it is considered a bad practice to directly type in a number (let's say 20). Instead, various inbuilt functions and macros are utilized. For example, to allocate space for a integer value, we have to simply do this:

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

The Syntax can be summarised as:
ptr = (cast *) malloc(byte-size)
```

One more thing to note here is that the invocation of malloc() uses the sizeof() operator. sizeof() is considered a compile-time operator in C, as the actual size can be known at compile time only and thus, this way a number is substituted in the argument to malloc(). Due to this reason, sizeof() is correctly considered as an operator and not a function call (a function call would take place at runtime).

For more details, we recommend you to go through the man pages for malloc() call. You can do that by typing the command: man malloc.

Before ending our discussion on malloc, let us write a program in C, where we allocate space for 10 integers.

```
#include<stdio.h>
int main() {
    int *x = (int *) malloc(10 * sizeof(int));
    return 0;
}
```

calloc()

calloc() is another call that the memory-allocation library supports. One point of difference between malloc() and calloc() is that malloc() doesn't initialize the allocated memory, whereas calloc() allocates the memory and initializes the block of allocated memory to zero.

Another point of difference is that unlike malloc(), calloc() receives two arguments:

1. Number of blocks to be allocated
2. Size of each block

Let us look at a program in C, where we are first allocating space using malloc, then deallocating it and then allocating it again using calloc() and finally, deallocating the created memory space for the last time using free().

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

int main()
{
    int* ptr;
    ptr = (int*)malloc(10 * sizeof(int));
    free(ptr);

    ptr = (int*)calloc(10, sizeof(int));
    free(ptr);

    return (0);
}
```

realloc()

The function realloc() can be handy, when you've already allocated space for something (say, an array), and then you need to add more space to it.

realloc() creates a new bigger area of memory, brings or copies the old area's contents into it, and returns the pointer to the new bigger area.

When you type "**man malloc**" on terminal, then it defines the declaration of realloc() in the synopsis as:

`void *realloc(void * ptr, size_t size),` here ptr denotes the location of the old area and size is the new size of the new bigger area.

Let us now use realloc() to increase the size of an array from 5 to 10. We will just focus on realloc parts of this snippet of code.

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

int main()
{
    /*
        Allocation of memory space for an array of size 5
    */
    int n = 10;

    // using realloc(), we are increasing the size of array to 10
    ptr = realloc(ptr, n * sizeof(int));

    /*
        free up the allocated memory
    */

    return 0;

}
```

free()

To free up the allocated memory, you have to simply call the free(). Let me demonstrate the usage of free() using an example code:

```
#include<stdio.h>
int main() {
    int *ptr = malloc(10 * sizeof(int));
    /*...*/
    free(ptr);
    return 0;
}
```

The routine takes one argument, a pointer returned by malloc(). Thus, you might notice, the size of the allocated region is not passed in by the user, and must be tracked by the memory-allocation library itself.