Tutorial Link https://codequotient.com/tutorials/Dynamic Memory Allocation/5c3ddf475eaff7081ba8737c

TUTORIAL

Dynamic Memory Allocation

Chapter

1. Dynamic Memory Allocation

Topics

Memory to program variables can be allocated in two manners either in static/Compile-time or dynamic/Run-time. Compile time variables are allocated in stack and come and go as functions are called and return. For these variables, the size of the allocation must be compile-time constant. 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. statically allocated memory is not 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. All these problems can be solved by using dynamic memory allocation. Usually we need dynamic memory allocation in following cases: -

- When you need a lot of memory. Typical stack size is small in most systems, so anything bigger than 50-100KB should better be dynamically allocated, or you're risking crash.
- When the memory must live after the function returns. Stack memory gets destroyed when function ends, dynamic memory is freed when you want.
- When you're building a structure (like array, or graph) of size that is unknown (i.e. may get big), dynamically changes or is too hard to precalculate.
- Dynamic allocation allows your code to naturally request memory piece by piece at any moment and only when you need it. As, It is not possible to repeatedly request more and more stack space in a for loop.

In general we can distinguish between static allocation and dynamic allocation as below: -

In dynamic memory allocation, memory can be increased or decreased at run time. C allows following four functions in stdlib.h file for dynamic memory management: -

```
malloc():   allocates single block of requested memory.
calloc():    allocates multiple block of requested memory.
realloc():    reallocates the memory occupied by malloc() or
calloc() functions.
free():      frees the dynamically allocated memory.
```

Let’s examine these function one by one: -

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory in heap area of specified size and return a pointer of type void which can be casted into pointer of any form. It returns NULL if sufficient memory is not available. The prototype is: -

```
ptr = (cast_type*) malloc(size_in_bytes_required);
```

For example, in following program we request a memory block for holding an integer using malloc: -

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

int main()
{
  int *p;
  p=(int*)malloc(sizeof(int));        // use of sizeof operator for platform independence
  printf("Initial value at memory allocated by malloc is = %d\n",*p);
  *p=10;
  printf("Value at memory allocated by malloc after initialization = %d\n",*p);
  return 0;
}
```

It will produce the following output: -

```
Initial value at memory allocated by malloc is = 3127564
Value at memory allocated by malloc after initialization = 10
```

Because malloc might not be able to service the request, it might return a null pointer and it is good programming practice to check for this:

```
int *var1 = malloc(sizeof(int));
if (var1 == NULL)
{
        printf("malloc failed\n");
    return(-1);
}
```

In following program, memory is allocated using malloc for character variables: -

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

```
#include<stdlib.h>

int main()
{
  int size = 7;
  char *str = (char *)malloc(sizeof(char)*size);  // Ask for storage of 7 characters in Heap
  if (str == NULL)
  {
    fprintf(stderr, "malloc failed\n");
    return(-1);
  }
  *(str+0) = 'C';        // Pointer to the memory allocated is in str
  *(str+1) = 'O';      // we can use   str = "CODING"; also directly
  *(str+2) = 'D';
  *(str+3) = 'I';
  *(str+4) = 'N';
  *(str+5) = 'G';
  *(str+6) = '\0';      // Null character is stored at the end.

  printf("%s",str);     // will print the character string
  free(str);        // to free the memory asked
  return 0;
}
```

C

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

int main()
{
  int tot,size,i;
  char** Coll=NULL;    // An array of character pointers. Always remember,
  //    char* is a string.
  //     char** is an array of strings
  tot = 5;        // How many names you want to store ?
```

```
11
12    Coll = (char**)malloc(tot * sizeof(char**));
13    // Allocate memory for total number of names
14    size = 10;    // What is length of each name ?
15    for (i=0;i<tot;i++)
16      Coll[i] = (char*)malloc(size * sizeof(char*));
17    // Allocate memory for each name in the array
18
19    // Enter the names :
20    strcpy(Coll[0],"Ram");          // Input all the names
21    strcpy(Coll[1],"Shyam");          // Input all the names
22    strcpy(Coll[2],"Mohan");          // Input all the names
23    strcpy(Coll[3],"Sohan");          // Input all the names
24    strcpy(Coll[4],"Murari");          // Input all the
  names
25    printf("You entered the following names : \n");
26    for (i=0;i<tot;i++)
27      printf("%s\n",Coll[i]);        // Names are accessed
28    return 0;
29  }
```

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero. The prototype is: -

```
ptr = (cast-type*) calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array of n elements. For example:

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

This statement allocates contiguous space in memory for an array of 10 elements each of size of int. We can achieve same functionality as

calloc() by using malloc() followed by memset(),

```
ptr = malloc(size);      // allocate the size bytes in ptr
memset(ptr, 0, size);      // initialize the size bytes to 0.
```

The above program can be written using calloc() as: -

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

int main()
{
  int *p;
  p=(int*)calloc(1,sizeof(int));        // use of sizeof operator for platform independence
  printf("Initial value at memory allocated by calloc is = %d\n",*p);
  *p=10;
  printf("Value at memory allocated by calloc after initialization = %d\n",*p);
  free(p);
  return 0;
}
```

For allocating arrays using calloc() we can write as below:

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

int main()
{
  int num, i, *ptr, sum = 0;
  num = 6;              // Enter number of elements
  ptr = (int*) calloc(num, sizeof(int));
  // Allocate storage for num integers and initialize all bytes to 0
  if(ptr == NULL)
  {
    printf("Error! memory not allocated.");
```

```
13      exit(-1);
14    }
15    printf("Value at memory returned by calloc()
    initially.\n");
16    for(i = 0; i < num; ++i)
17      printf("%d ", *(ptr + i));
18    //Enter elements of array: -
19    ptr[0] = 1;
20    ptr[1] = 4;
21    ptr[2] = 7;
22    ptr[3] = 2;
23    ptr[4] = 6;
24    ptr[5] = 3;
25    printf("\nValue Entered by user :\n");
26    for(i = 0; i < num; ++i)
27      printf("%d ", *(ptr + i));
28    free(ptr);
29    return 0;
30  }
```

Similarly, we can use calloc() to alolocate space for character arrays or strings too.

realloc()

If memory allocated by malloc() or calloc() is not sufficient at some stage, you can reallocate the memory by realloc() function. In short, it changes the memory size. The syntax of realloc() function: -

```
void *realloc(void *ptr, size_t size);
```

If “size” is zero, then call to realloc is equivalent to “free(ptr)”. And if “ptr” is NULL and size is non-zero then call to realloc is equivalent to “malloc(size)”. realloc() deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size. The contents of the new object is identical to that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any

bytes in the new object beyond the size of the old object have indeterminate values. For example, the following first request space for 2 integers and later it needs one more integer so space extended by realloc(): -

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

int main()
{
  int *ptr = (int *)malloc(sizeof(int)*2);
  int i;
  int *ptr_new;
  *ptr = 10;         // first element.
  *(ptr + 1) = 20;    // second element through pointer arithmetic
  printf("Firstly, two elements are there in array as below: \n");
  for(i = 0; i < 2; i++)
    printf("%d \t", *(ptr + i));
  ptr_new = (int *)realloc(ptr, sizeof(int)*3);    // reallocation to extend size.
  *(ptr_new + 2) = 30;        // 3rd element by pointer arithmetic
  printf("\nAfter realloc() there are three elements in array as below: \n");
  for(i = 0; i < 3; i++)
    printf("%d \t", *(ptr_new + i));
  printf("\nFirst two elements are same as above and third element is changed \n");
  free(ptr);
  return 0;
}
```

realloc() will create a new space in memory for newly required number of elements, and then will copy all the old elements in this new space. This whole process is very time consuming, so use of realloc() will degrade the performance sometimes. So, you have to use this function carefully.

free()

The memory occupied by malloc(), calloc() or realloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit. The syntax of free() function:

```
free(ptr);
```

Note that the free function does not accept size as a parameter. How does free() function know how much memory to free given just a pointer? The most common way is to store size of memory so that free() knows the size of memory to be deallocated. When memory allocation is done, the actual heap space allocated is one word larger than the requested memory. The extra word is used to store the size of the allocation and is later used by free( ).