Dynamic memory Allocation in C

To learn from simple videos, you can always look at our **C++ video (https://pro.codesdope.com/courses/cpp)** course on **CodesDope Pro (https://pro.codesdope.com)**. It has over 750 practice questions and over 200 solved examples.

Suppose you want to put a toy in a box, but you only have an approximate idea of its size. For that, you would require a box whose size is equal to the approximate size of the toy.

We face a similar situation in C also when we want to input a sentence as an array of characters but are not sure about the number of characters in the array.

Now, while declaring the character array, if we specify its size smaller than the size of the input string, then we will get an error because the space in the memory allocated to the array is lesser than the size of the input string. This is the same case as trying to fit a big toy in a smaller box. If we specify its size much larger than the size of the input string, then the array will be allocated a space in the memory which is much larger than the size of the input string, thus unnecessarily consuming more memory even when it is not required. This is like putting a small toy in a large box.

Insufficient space **Wastage of space**

In the above case, we don't have idea about the size of the array until the compile time (when computer compiles the code and the string is input by the user). In such cases, we use **malloc()** function

malloc

`malloc` function **allocates memory at runtime**. It takes the size in bytes and allocates that much space in the memory. It means that `malloc(50)` will allocate 50 byte in the memory. It returns a void pointer and is defined in **stdlib.h**.

Let's understand it with the help of an example.

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

int main()
{
   char name[20];
   char *address;

   strcpy(name, "Harry Lee");
   address = (char*)malloc( 50 * sizeof(char) );  /* allocating memory dynamically  */
   strcpy( address, "Lee Fort, 11-B Sans Street");

   printf("Name = %s\n", name );
   printf("Address: %s\n", address );
   return 0;
}
```

Output

```
Name = Harry Lee
Address: Lee Fort, 11-B Sans Street
```

In the above example, we assigned and printed the name as we used to do till now. For address, we estimated that the number of characters should be around 50. So, the **size of address** will be **50 * sizeof(char)**.

`char *address` → Here we declared a pointer to character for address without specifying how much memory is required.

`address = (char*)malloc( 50 * sizeof(char) )` → By writing this, we assigned a memory of **'50 * sizeof(char)' bytes** for address. We used `(char*)` to typecast the pointer returned by malloc to character.

`strcpy( address, "Lee Fort, 11-B Sans Street")` → By writing this, finally we assigned the address.

</> By default, malloc returns a pointer of type void but we can **typecast** it into a pointer of any other form (as we converted it into character type in the above example).

</> If the space in memory allocated by malloc is insufficient, then the allocation fails and malloc returns NULL pointer

Let's see another example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
     int n,i,*p;
     printf("Enter number of elements: ");
     scanf("%d",&n);
     p=(int*)malloc(n * sizeof(int));  //memory allocated using malloc
     if(p == NULL)
     {
         printf("memory cannot be allocated\n");
     }
     else
     {
       printf("Enter elements of array:\n");
       for(i=0;i<n;++i)
       {
         scanf("%d",&*(p+i));
       }
       printf("Elements of array are\n");
       for(i=0;i<n;i++)
       {
         printf("%d\n",*(p+i));
       }
     }
     return 0;
}
```

Output

```
Enter number of elements:
5
Enter elements of array:
1
2
3
4
5
Elements of array are
1
2
3
4
5
```

In this example, firstly, we declared a pointer 'p' of type int which contains n elements. Thus, 'p' is an integer array containing n elements. So, we assigned a memory of size `n * sizeof(int)` to the array which the pointer 'p' is pointing to. Thus, we now have a memory space allocated to the integer array consisting of 'n' elements. Further, we ask the user to input the values of the elements of the array and display those values.

calloc

Now, suppose you want to put more than one toy in a box and you have only an approximate idea of the number of toys and the size of each. For that, you need a box the size of which is equal to the sum of the sizes of all the toys.

In such cases, we use **calloc** function. Like malloc, calloc also **allocates memory at runtime** and is defined in **stdlib.h**. It takes the number of elements and the size of each element(in bytes), initializes each element to zero and then returns a void pointer to the memory.

Its syntax is

```
void *calloc(n, element-size);
```

Here, 'n' is the number of elements and 'element-size' is the size of each element.

Let's see the last example of malloc again, but this time with `calloc`.

New Questions
Difficult subject - Java (/discussion/difficult-subject)
JAVA CLASSES - Java (/discussion/java-classes)
Java - Classes & object Level 2 - Java (/discussion/java-classes-object-level-2)
\n in print - Python (/discussion/n-in-print)
Difficult subject code - Java (/discussion/difficult-subject-code)
Why python is interpreted - Python (/discussion/why-python-is-interpreted)
nt a = 10/45*23%45/(45%4*21) b. float a = 10+45.0*23-45+(4*21.0) - C (/discussion/nt-a-1045234545421-b-float-a-1045023-454210)
Ask Yours (/add_question/)

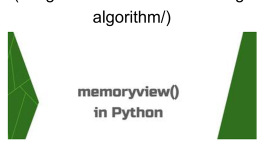
(/discussion/) (/practice/) (https://codesdope-media.nyc3.cdn.digitaloceanspaces.com/prod/media/pdf/ASCII.pdf)

(https://www.facebook.com/codesdope) (https://twitter.com/CodesDope) (https://www.linkedin.com/company/codesdope)

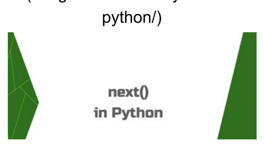
Recent Posts

pow() in Python (/blog/article/pow-in-python/)

Dutch National Flag problem - Sort 0, 1, 2 in an array (/blog/article/dutch-national-flag-algorithm/)



memoryview() in Python (/blog/article/memoryview-in-python/)



next() in Python (/blog/article/next-in-python/)
Post Yours (/blog/submit-article/)

?

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n, sizeof(int));  //memory allocated using malloc
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
      printf("Enter elements of array:\n");
      for(i=0;i<n;++i)
      {
        scanf("%d",&*(p+i));
      }
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    }
    return 0;
}
```

Output

```
Enter number of elements:
5
Enter elements of array:
1
2
3
4
5
Elements of array are
1
2
3
4
5
```

So, this is same as the example of `malloc`, with a difference in the syntax of calloc. Here we wrote `(int*)calloc(n, sizeof(int))`, where n is the number of elements in the integer array (5 in this case) and `sizeof(int)` is the size of each of that element. So the total size of the array is `n * sizeof(int)`.

</> `calloc` initializes the allocated memory to zero value whereas `malloc` doesn't.

To prove it, let's see two examples.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)calloc(n, sizeof(int));
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else{
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    }
    return 0;
}
```

Output

```
Enter number of elements:
5
Elements of array are
0
0
0
0
0
```

Here, we specified the number of elements i.e. 5 but did not initialize any. So, `calloc` initialized each of the elements to 0 and thus the value of each element got printed as 0.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    p=(int*)malloc(n, sizeof(int));
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    }
    return 0;
}
```

Output

```
prog.c: In function 'main':
prog.c:7:13: error: too many arguments to function 'malloc'
p=(int*)malloc(n, sizeof(int));
^
```

Since `malloc` doesn't initialize the elements to 0, therefore we got an error in this case.

</> calloc is used to allocate memory to mostly arrays and structures

Realloc

If suppose we allocated more or less memory than required, then we can change the size of the previously allocated memory space using **realloc**. Its syntax is as follows.

void *realloc(pointer, new-size);

Let's see an example on realloc.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char *p1;
    int m1, m2;
    m1 = 10;
    m2 = 20;
    p1 = (char*)malloc(m1);
    strcpy(p1, "Codesdope");
    p1 = (char*)realloc(p1, m2);
    strcat(p1, "Practice");
    printf("%s\n", p1);
    return 0;
}
```

Output

```
CodesdopePractice
```

In the above example, we declared a pointer 'p1' which will be used to dynamically allocate a memory space.

`p1 = (char*)malloc(m1)` → By writing this, we assigned a memory space of 10 bytes which the pointer 'p1' is pointing to. We used `(char*)` to typecast the pointer returned by malloc to character.

`strcpy(p1, "Codesdope")` → This assigns a string value "Codesdope" to the memory which the pointer p1 is pointing to. Currently, the memory space is 10 bytes which can easily store the string "Codesdope", but what if now we want that memory space to store the string "CodesdopePractice"? For this, we need to expand the size of our memory space which we can easily do with **realloc**.

`p1 = (char*)realloc(p1, m2)` → This increases the size of the memory space (whose address is stored in p1) to 20 bytes(since the value of m2 is 20) which can easily store the string "CodesdopePractice".

strcat(p1, "Practice") - This adds the string "Practice" at the end of the string stored in the memory pointed by p1 i.e. "Codesdope". So now, the memory pointed by p1, now stores the string value "CodesdopePractice".

free

free function is used to deallocate or free the memory after the program finishes which was dynamically allocated in the program. It is adviced to free the dynamically allocated memory after the program finishes so that it becomes available for future use.

void free(pointer);

This was the syntax of free function whose return type is void. Now, let's see an example where we released the dynamically allocated memory at the end of the program using free.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*p;
    printf("Enter number of elements:\n");
    scanf("%d",&n);
    p=(int*)malloc(n * sizeof(int));
    if(p == NULL)
    {
        printf("memory cannot be allocated\n");
    }
    else
    {
      printf("Enter elements of array:\n");
      for(i=0;i<n;++i)
      {
        scanf("%d",&*(p+i));
      }
      printf("Elements of array are\n");
      for(i=0;i<n;i++)
      {
        printf("%d\n",*(p+i));
      }
    }
    free(p);
    return 0;
}
```

Output

```
Enter number of elements:
5
Enter elements of array:
1
2
3
4
5
Elements of array are
1
2
3
4
5
```

So here by writing `free(p)`, we released the memory which was dynamically allocated using malloc.

To learn from simple videos, you can always look at our **C++ video (https://pro.codesdope.com/courses/cpp)** course on **CodesDope Pro (https://pro.codesdope.com)**. It has over 750 practice questions and over 200 solved examples.

PREV (/c-enjoy-with-files/) (/c-enum/) NEXT

?