</> ALX LESSON 0x0B C - malloc, free, calloc, realloc

C - Programming

TABLE OF CONTENTS

01

Overview topics

02

Learning Objectives

03

Quiz questions

04

hands on lab practice

01

OVERVIEW topics

Topics

C
Programming
Topics

what is the difference between automatic and dynamic allocation

what is malloc and free and how to use them

why and when use malloc

how to use valgrind to check for memory leak

Slides On Telegram

https://t.me/alx_2023

C Programming

Topics

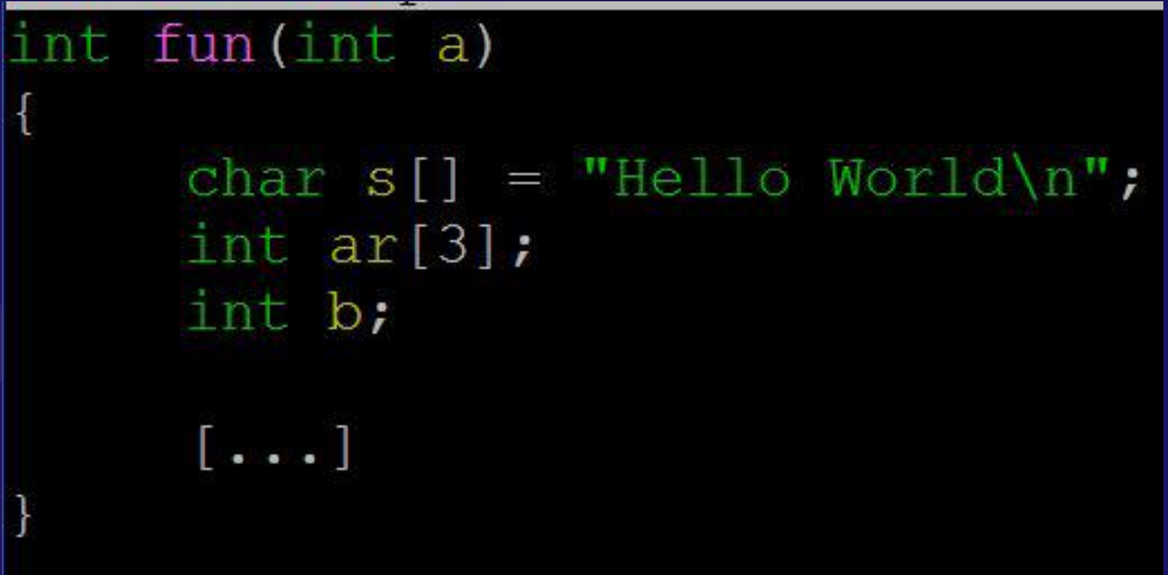
02

Learning Objectives

Automatic allocation

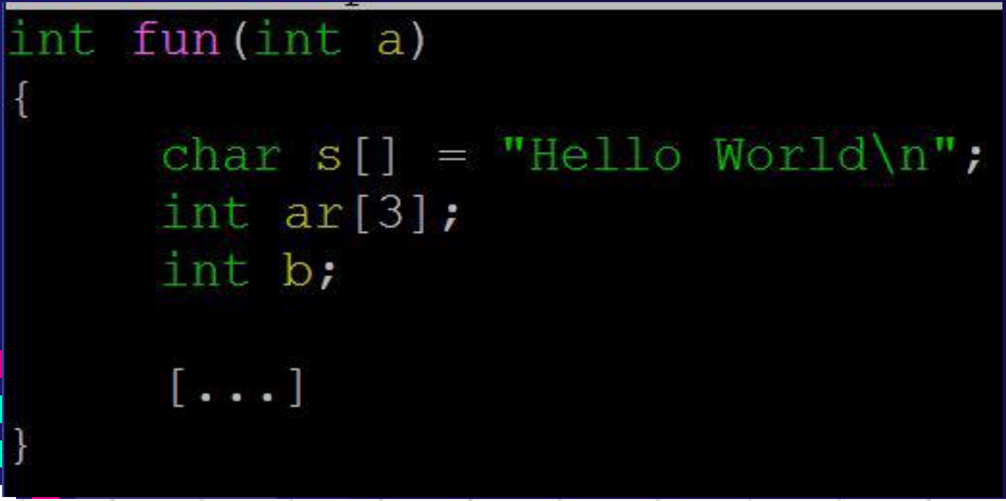
When you declare variables or when you use strings within double quotes, the program is taking care of all the memory allocation. You do not have to think about it.

Ex :

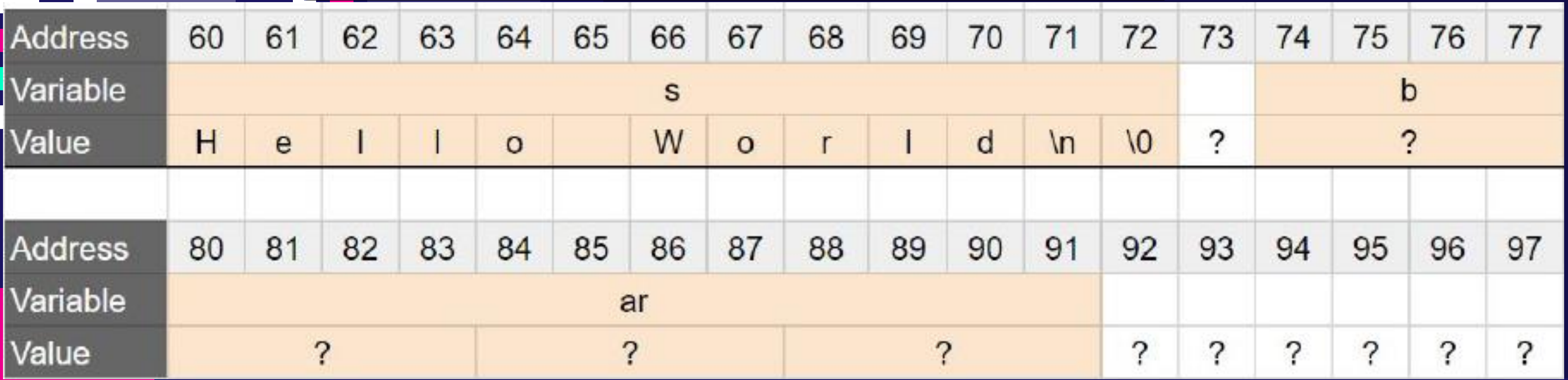


```
int  fun(int  a)
{
        char s[]  =  "Hello World\n";
        int ar[3];
        int b;

         [...]
}
```

Arguments to main



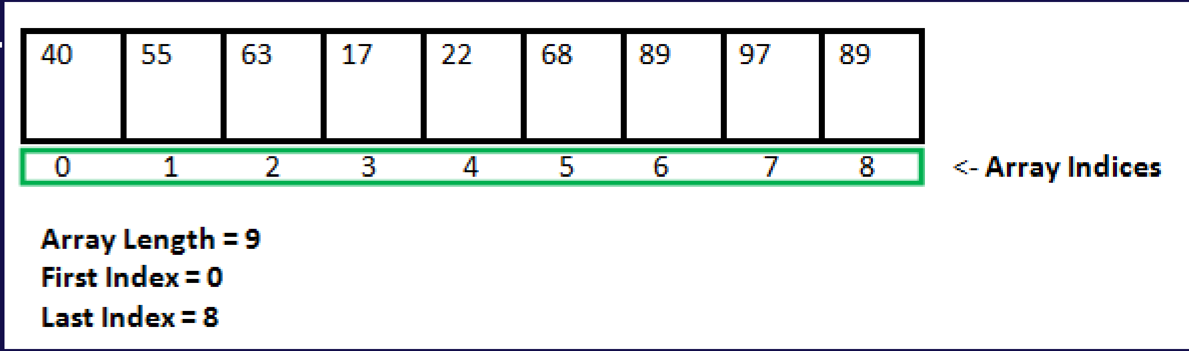
```
int fun(int a)
{
        char s[] = "Hello World\n";
        int ar[3];
        int b;

         [...]
}
```



Address	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77
Variable	s														b			
Value	H	e	l	l	o		W	o	r	l	d	\n	\0	?	?			
Address	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97
Variable	ar																	
Value	?				?				?				?	?	?	?	?	?

Dynamic allocation

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



40	55	63	17	22	68	89	97	89	
0	1	2	3	4	5	6	7	8	<- Array Indices

Array Length = 9
First Index = 0
Last Index = 8

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

Dynamic allocation

This procedure is referred to as Dynamic Memory Allocation in C.
Therefore, C Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:

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

malloc() method

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax:

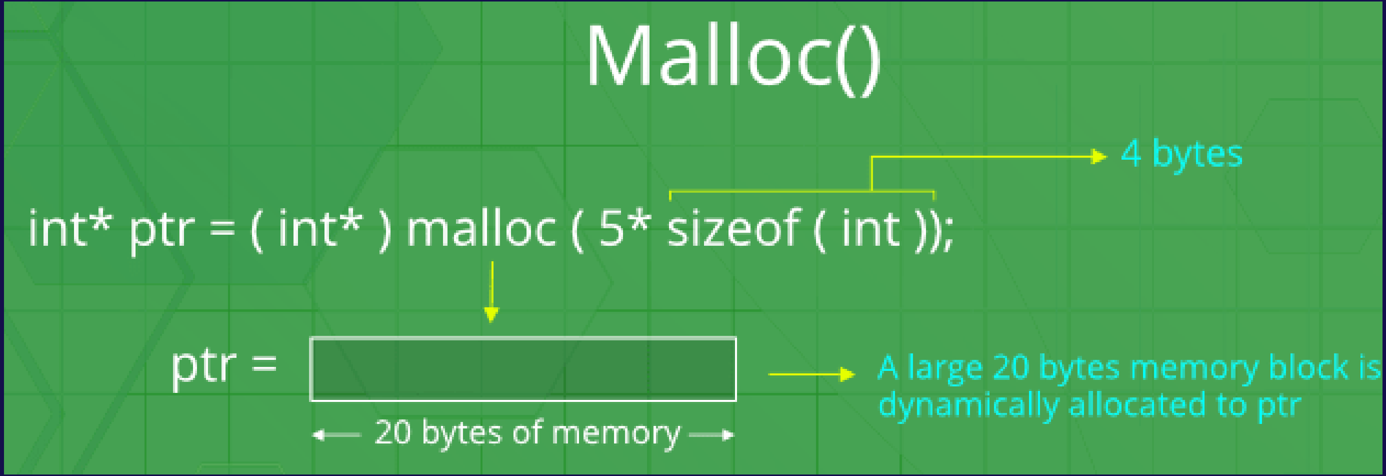
```
ptr = (cast-type*) malloc(byte-size)
```

malloc() method

For Example:

```
int* ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

If space is insufficient, allocation fails and returns a NULL pointer.

malloc() method

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

int main()
{

    // This pointer will hold the // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

calloc() method

“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

1. It initializes each block with a default value ‘0’.
2. It has two parameters or arguments as compare to malloc().

Syntax:

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

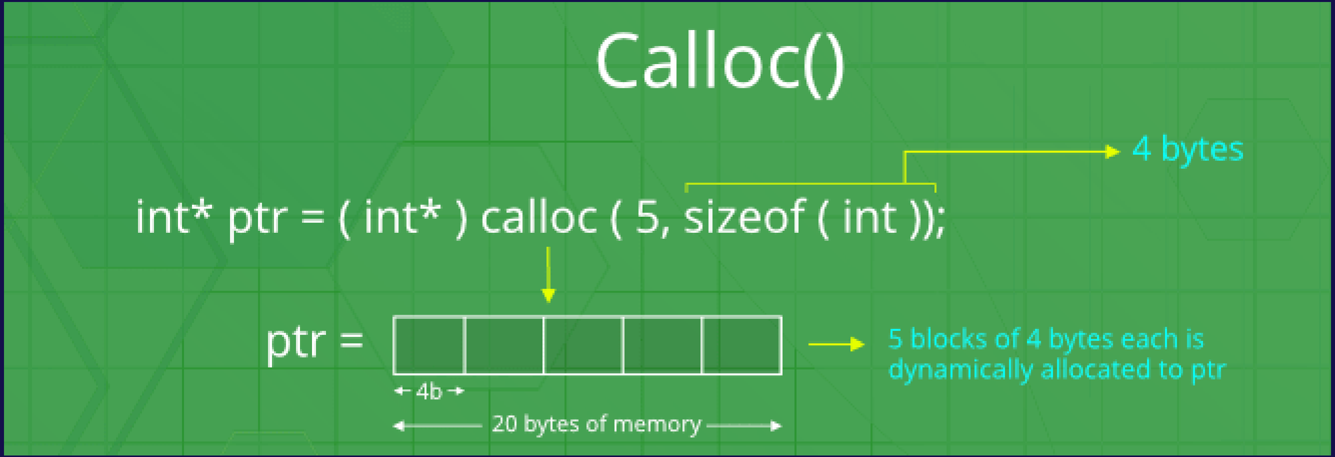
here, n is the no. of elements and element-size is the size of each element.

calloc() method

For Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr

If space is insufficient, allocation fails and returns a NULL pointer.

calloc() method

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

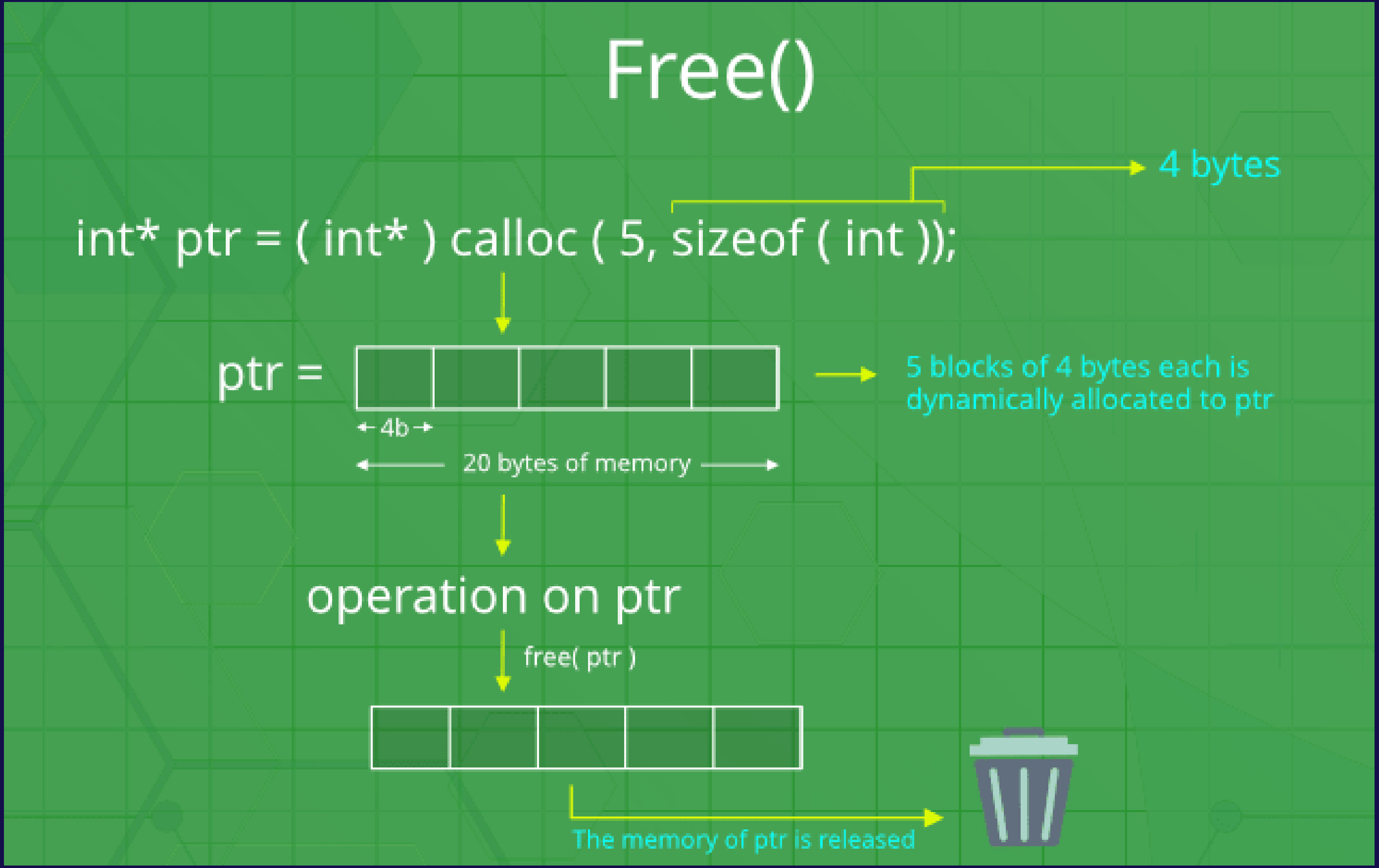
free() method

“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```

free() method



Free()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

4 bytes

ptr =

5 blocks of 4 bytes each is dynamically allocated to ptr

operation on ptr

free(ptr)

The memory of ptr is released

free() method

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

int main()
{
        // This pointer will hold the // base address of the block created
        int *ptr, *ptr1;
        int n, i;

        // Get the number of elements for the array
        n = 5;
        printf("Enter number of elements: %d\n", n);

        // Dynamically allocate memory using malloc()
        ptr = (int*)malloc(n * sizeof(int));

        // Dynamically allocate memory using calloc()
        ptr1 = (int*)calloc(n, sizeof(int));

        // Check if the memory has been successfully  allocated by malloc or not
        if (ptr == NULL || ptr1 == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else {

                // Memory has been successfully allocated
                printf("Memory successfully allocated using malloc.\n");

                // Free the memory
                free(ptr);
                printf("Malloc Memory successfully freed.\n");

                // Memory has been successfully allocated
                printf("\nMemory successfully allocated using calloc.\n");

                // Free the memory
                free(ptr1);
                printf("Calloc Memory successfully freed.\n");
        }

        return 0;
}
```

realloc() method

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

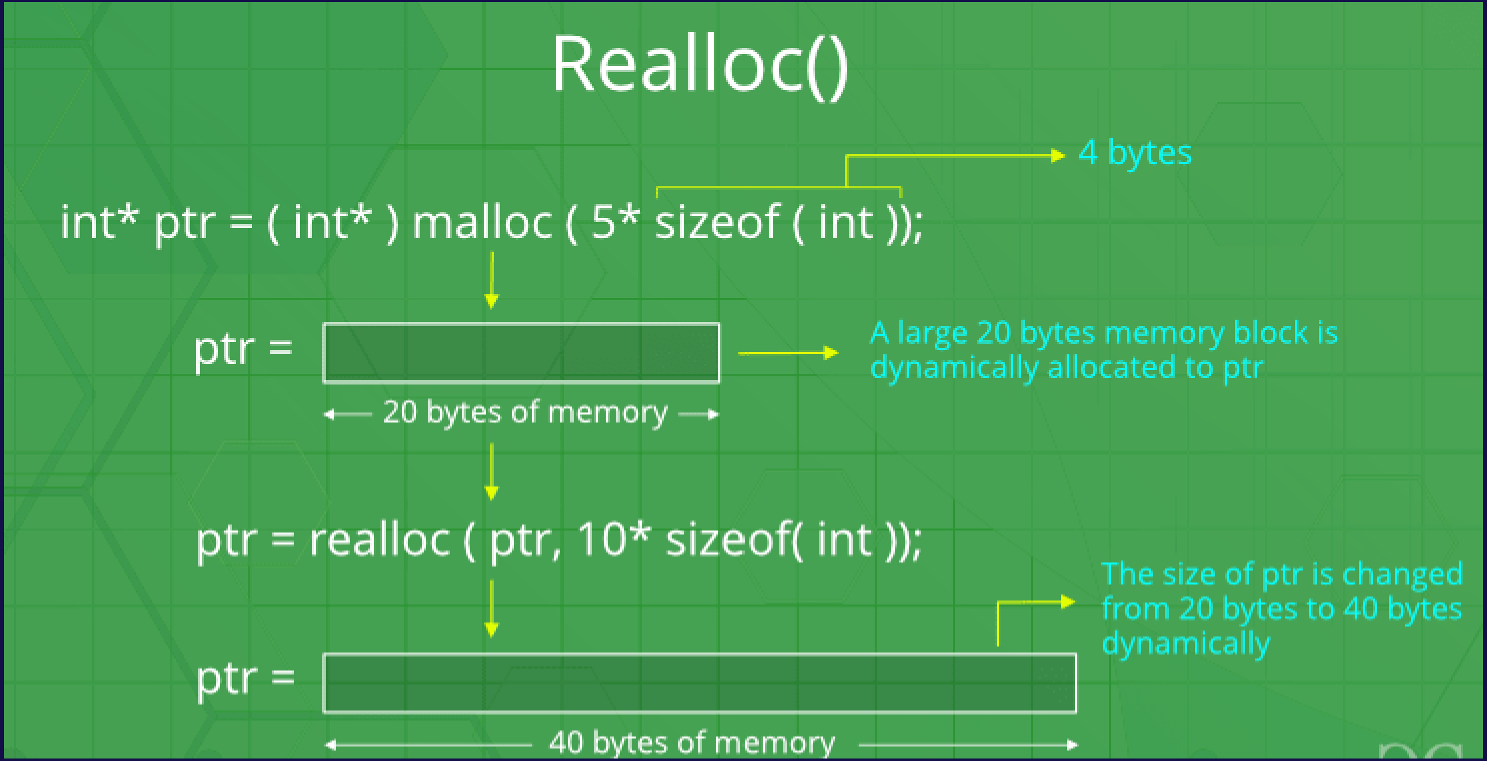
Syntax:

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.

realloc() method

For Example:



```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

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

Realloc()
4 bytes
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr =
The size of ptr is changed from 20 bytes to 40 bytes dynamically
40 bytes of memory

If space is insufficient, allocation fails and returns a NULL pointer.

realloc() method

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

realloc() method

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int index = 0, i = 0, n,
        *marks; // this marks pointer hold the base address
                // of the block created
    int ans;
    marks = (int*)malloc(sizeof(
        int)); // dynamically allocate memory using malloc
    // check if the memory is successfully allocated by
    // malloc or not?
    if (marks == NULL) {
        printf("memory cannot be allocated");
    }
    else {
        // memory has successfully allocated
        printf("Memory has been successfully allocated by "
            "using malloc\n");
        printf("\n marks = %pc\n",
            marks); // print the base or beginning
                    // address of allocated memory

        do {
            printf("\n Enter Marks\n");
            scanf("%d", &marks[index]); // Get the marks
            printf("would you like to add more(1/0): ");
            scanf("%d", &ans);

            if (ans == 1) {
                index++;
                marks = (int*)realloc(
                    marks,
                    (index + 1)
                        * sizeof(
                            int)); // Dynamically reallocate
                                // memory by using realloc

                // check if the memory is successfully
                // allocated by realloc or not?
                if (marks == NULL) {
                    printf("memory cannot be allocated");
                }
                else {
                    printf("Memory has been successfully "
                        "reallocated using realloc:\n");
                    printf(
                        "\n base address of marks are:%pc",
                        marks); ////print the base or
                                ///beginning address of
                                ///allocated memory
                }
            }
        } while (ans == 1);
        // print the marks of the students
        for (i = 0; i <= index; i++) {
            printf("marks of students %d are: %d\n ", i,
                marks[i]);
        }
        free(marks);
    }
    return 0;
}
```

how to use valgrind to check for memory leak

Valgrind is a tool for detecting memory leaks and other memory-related errors in C/C++ programs. Here are the steps to use Valgrind to check for memory leaks:

Install Valgrind: If you don't already have Valgrind installed on your system, you can download and install it from the Valgrind website or using your package manager.

Compile your program with debug symbols: In order for Valgrind to provide detailed information about memory errors, you need to compile your program with debug symbols. For example, if you are using GCC, you can use the -g option to enable debug symbols.

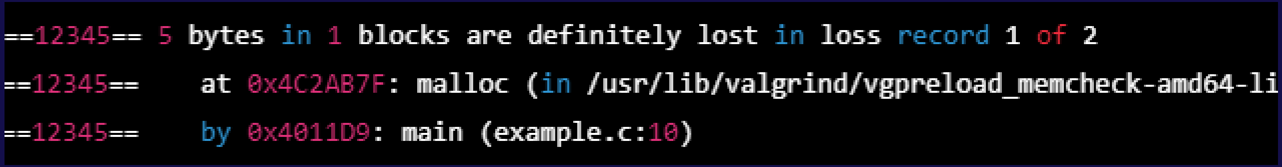
Run your program with Valgrind: To check for memory leaks, you need to run your program with Valgrind. Here is an example command:

```
valgrind --leak-check=full ./your_program
```

The --leak-check=full option tells Valgrind to perform a detailed leak check.

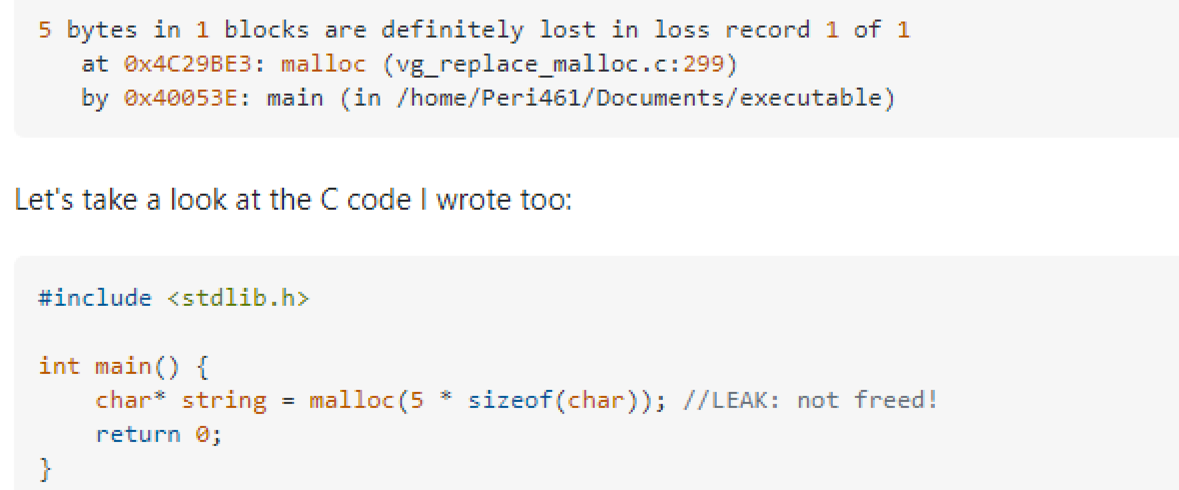
how to use valgrind to check for memory leak

If leak found



```
==12345== 5 bytes in 1 blocks are definitely lost in loss record 1 of 2
==12345==    at 0x4C2AB7F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-li
==12345==    by 0x4011D9: main (example.c:10)
```

This output indicates that there is a memory leak of 5 bytes in the program. The leak occurred at line 10 of example.c, where malloc() was called to allocate memory that was not freed.

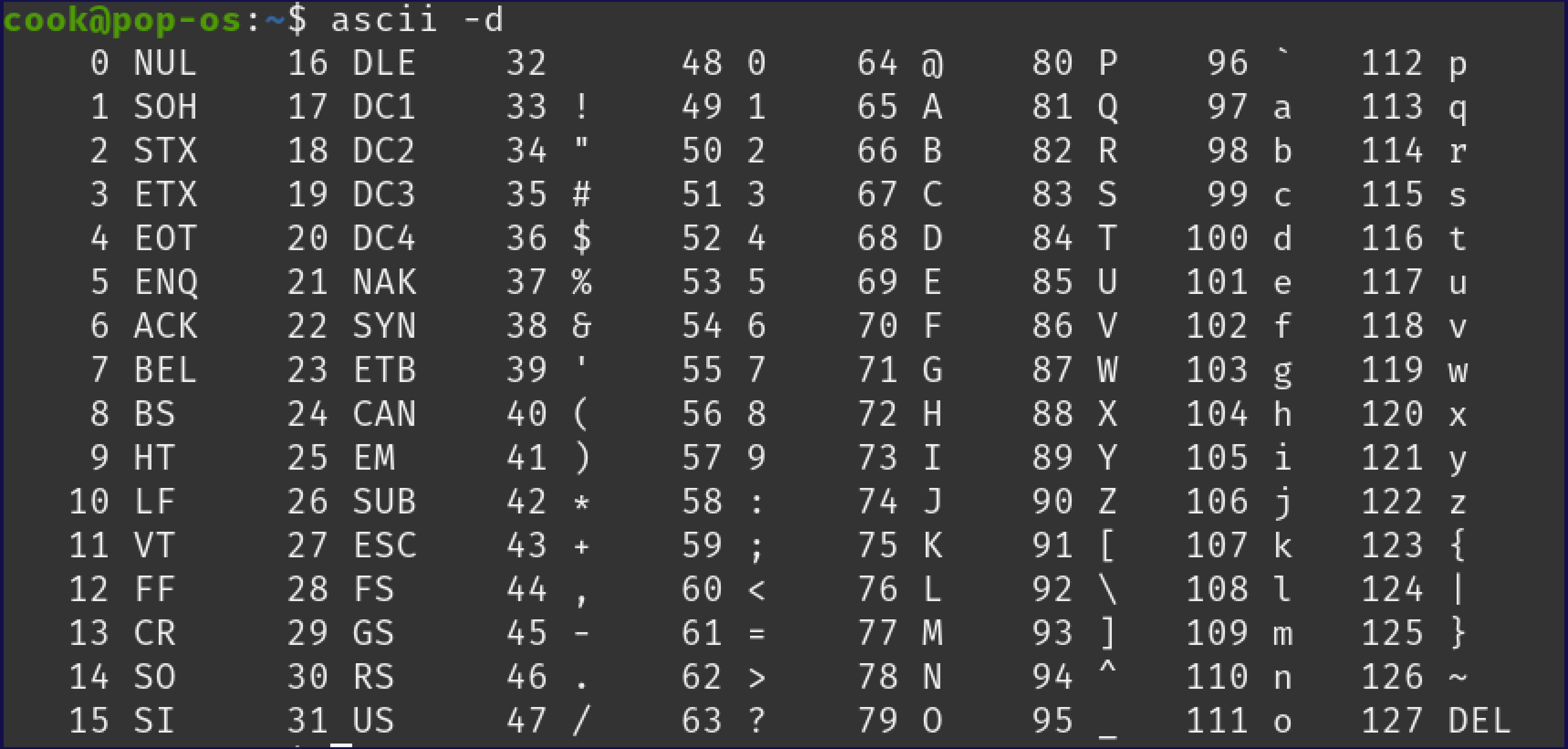


```
5 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C29BE3: malloc (vg_replace_malloc.c:299)
   by 0x40053E: main (in /home/Peri461/Documents/executable)
```

Let's take a look at the C code I wrote too:

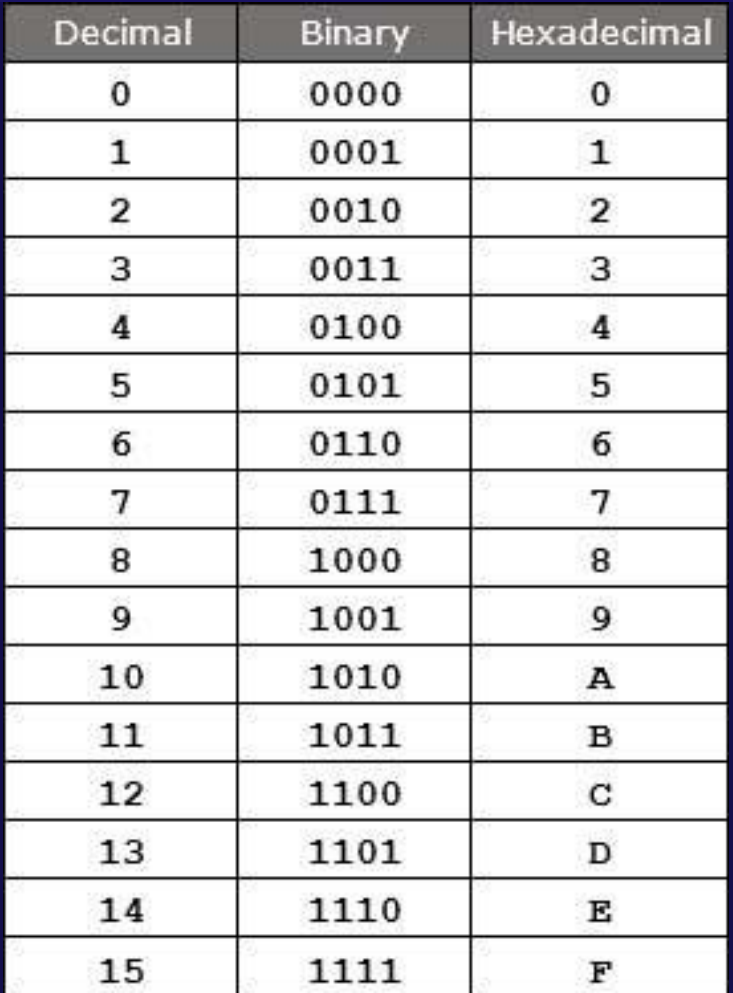
```
#include <stdlib.h>

int main() {
    char* string = malloc(5 * sizeof(char)); //LEAK: not freed!
    return 0;
}
```

What is the ASCII character set



```
cook@pop-os:~$ ascii -d
    0 NUL    16 DLE    32        48 0      64 @      80 P      96 `     112 p
    1 SOH    17 DC1    33 !      49 1      65 A      81 Q      97 a     113 q
    2 STX    18 DC2    34 "      50 2      66 B      82 R      98 b     114 r
    3 ETX    19 DC3    35 #      51 3      67 C      83 S      99 c     115 s
    4 EOT    20 DC4    36 $      52 4      68 D      84 T     100 d     116 t
    5 ENQ    21 NAK    37 %      53 5      69 E      85 U     101 e     117 u
    6 ACK    22 SYN    38 &      54 6      70 F      86 V     102 f     118 v
    7 BEL    23 ETB    39 '      55 7      71 G      87 W     103 g     119 w
    8 BS     24 CAN    40 (      56 8      72 H      88 X     104 h     120 x
    9 HT     25 EM     41 )      57 9      73 I      89 Y     105 i     121 y
   10 LF     26 SUB    42 *      58 :      74 J      90 Z     106 j     122 z
   11 VT     27 ESC    43 +      59 ;      75 K      91 [     107 k     123 {
   12 FF     28 FS     44 ,      60 <      76 L      92 \     108 l     124 |
   13 CR     29 GS     45 -      61 =      77 M      93 ]     109 m     125 }
   14 SO     30 RS     46 .      62 >      78 N      94 ^     110 n     126 ~
   15 SI     31 US     47 /      63 ?      79 O      95 _     111 o     127 DEL
```

Hexadecimal Numbering System



Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

04

Hands on lab Practice

Have a Question
Leave a Comment!

Subscribe

To stay updated with latest videos

Share

To let the others know more

Thanks