malloc, memset, memmove, free, calloc and ralloc (video)

Memory can be allocated from two pools:  stack and Heap.  When we  declare variables in a function,  we allocate memory from the stack space.

 say a variable declared as

int x ;

The four bytes is allocated from the stack , predetermined during compile time.

But,  programmers may not know how much memory the program requires when they are writing the program.  So ,  during runtime,  we need memory to be allocated.  Also, Self referential structure require to be allocated dynamically.  malloc is one system function that you would call to allocate memory. This memory is allocated from heap space.

malloc  ( ) :

Step 1:

The sysetm function to allocate memory during runtime is malloc

int x= 8;

malloc ( x ) ;  // this allocates 8 bytes

malloc ( x+4 ) ; // allocates 12 bytes

malloc ( 1024 ) ; // allocates 1024 bytes

STEP 2 :

When the malloc returns, it returns a void pointer to the memory allocated.  We cannot traverse the memory using a void pointer.   So we need to cast the memory allocated to one of the known types: int, char, short, or any user defined structure.

Say we need to traverse the memory using a int.

 We do this by casting the void pointer to a int pointer like shown.

Example 1:

int \*ptr ;

ptr = ( int  \* ) malloc ( 2\* sizeof ( int)  ) ;

In the above statement, the void pointer that is returned by malloc,  and the void pointer is type casted to ( int \* ) pointer.   Then, we assign it to ptr which is int \*.

In short, the above line does three things, all at once:

1. allocate a memory of 8 bytes = 2 times sizeof  (int) ,

2. cast the memory to int \*

3. make a int pointer ptr point to that memory

Because we allocated 2 int ( = 8 bytes) , we can say ptr is now pointing to the first int ( 4 bytes) .

Now,  we can assign a value to the ptr.

\*ptr = 0x0a0b0c0d ;

which assigns 0x0a0b0c0d to the first int.

When we do ++ptr,  we have advanced the pointer to the second int.

Now we assign

\*ptr = 0xd0c0b0a ;

this means we have assigned 0xd0c0b0a to the \*ptr which in this case it is pointing to the second int.

Though we explained with just two ints,  in real life, the number of memory allocated is vastly huge.

Just to summarize:   When we allocate memory using malloc, we get the memory from the heap, not from stack.

[malloc int pointer (Links to an external site.)](https://www.youtube.com/watch?v=896nrvtA_xE&feature=youtu.be)[](https://www.youtube.com/watch?v=896nrvtA_xE&feature=youtu.be)

The next example uses structures:

Example 2:

Let us take another example.

struct \_user {

    char name [ 12 ] ;

    int age ;

} ;

let us declare a structure pointer of type

struct    \_user     \*sPtr ;

Now we will allocate memory whose size is twice the size of a structure \_user.

sPtr =  ( struct   \_user   \*  )  malloc  ( 2 \* sizeof (  struct    \_user  ) ) ;

now sPtr is now pointing to the dynamic memory whose size is twice the size of the structure.

sPtr is pointing to the first structure and we could assign values such as

(\*sPtr).age = 10 ;

strcpy ( ( \*sPtr ) . name, "John" );

How do I advance the pointer to the next structure?

sPtr++ would make it point to the next cell,  we could assign the values

(\*sPtr).age = 20

strcpy ( (\*sPtr).name, "Jenny" );

If we do sPtr++  one more time, the sPtr would be pointing beyond the allocated memory.  When you do this,  you may end up in a crash.  So, it is programmers responsibility to make sure the pointers are within the allocated memory size.

memove ( )  :

#include <string.h>

void \*memmove(void \*dest, const void \*src, size\_t n);

The memmove() function copies n bytes from memory area src to memory area dest. The memory areas may overlap: copying takes place as though the bytes in src are first copied into a temporary array that does not overlap src or dest, and the bytes are then copied from the temporary array to dest.

calloc ( ) :

 void \*calloc(size\_t nmemb, size\_t size);

The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero. If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value that can later be successfully  passed to free().

realloc ( ) :

void \*realloc(void \*ptr, size\_t size);

The realloc() function changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be  
initialized. If ptr is NULL, then the call is equivalent to malloc(size), for all values of size; if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

memset (  ) -  This function fills the first n bytes of the memory area pointed to by s with the constant byte c.

syntax: void \*memset(void \*s, int c, size\_t n) ;

The memset() function returns a pointer to the memory area s