

Forum:[Hack Forums](#)

Thread: [Clarification on stack/heap allocation/free](#)

SIM-3602340 - February 9, 2020, 11:49 am

Suppose that i have this struct, and that i use malloc to allocate memory for it.

Code struct test

```
{
    char *str;
    int size;
};
```

struct test *t = malloc(sizeof(test));

Then i init the values like this

Code t->str = "test 123\0";

t->size = 10;

What happens if I do this? Is it correct?

Code free(t->str);

free(t);

My fear is that "test 123\0" gets allocated onto the stack (because i am not using malloc), and so i should not use free.

If my assumption is correct, how do i free str memory then? Do i just free the structure?

I believe someone more expert can answer me, thank you in advice!

Shisho - February 9, 2020, 1:31 pm

To understand the correct use of free() you will need to come to terms with the fact that you only need a free() to match every malloc() that you do.

free() takes as an argument a pointer to memory which you have created with malloc() which is the beginning of your structure. Your string is allocated on the stack because your structure only stores the pointer to t->str not the actual string.

If you were to try to free(t->str) you would be trying to free an invalid pointer which would crash your program. You can only free(t) which is the structure you allocated memory on the stack for.

If you want to clear t->str you just need to set it to NULL or "" would make it point to nothing or be nothing. The memory is still allocated for a pointer to t->str in the structure so it's not like that memory is going anywhere until you free(t).

I hope that this clears up things a little bit.

morph - February 9, 2020, 1:31 pm

The stack is "freed" upon leaving the scope, i.e. the function

dracx - February 10, 2020, 7:13 am

Your struct has 1) a pointer and 2) an int. A pointer is simply a variable that holds an address, meaning it will NOT hold your actual string, it will only hold the address to where you're actual string is located in memory.

By initializing `t->str = "test 123\0";` the actual string is actually put inside the stack, not on the `t->str`, and the `t->str` now contains the address of that string (in the stack).

Now, freeing `t->str` would probably crash your code (or whatever) because first, the address that `t->str` holds is not pointing to the heap (not from `malloc`, `calloc`, `realloc`) and second, dang it would simply mess up ur code, hope I could explain more but my english is kinda limiting me rn, lol, sorry.

Looking at your code, you may be wanting something like this:

Code struct test

```
{
    char *str;
    int size;
};

struct test *t = malloc(sizeof(test));
t->str = malloc(sizeof("test 123\0"));
strcpy(t->str, "test 123\0");
// Or instead of doing the two lines of code before this, you could just
// t->str = strdup("test 123\0");
// still free'able
t->size = 10; // Is this supposed to be the size of the string pointed by str above?

// Then this is now valid, and is actually necessary
free(t->str);
free(t);
Ok, I may be wrong at whatever I just said, please correct me.
```

akali - February 10, 2020, 7:55 am

The job of the compiler is to manage stack memory. Compilers and functions have standards to follow in how the stack and registers should be managed. It's called calling conventions. Heap is for you to manage.

SIM-3602340 - February 11, 2020, 10:18 am

(Feb 10, 2020 08:13 AM)dracx Wrote: [\[CITE\]](#) Your struct has 1) a pointer and 2) an int. A pointer is simply a variable that holds an address, meaning it will NOT hold your actual string, it will only hold the address to where your actual string is located in memory.

By initializing `t->str = "test 123\0";` the actual string is actually put inside the stack, not on the `t->str`, and the `t->str` now contains the address of that string (in the stack).

Now, freeing `t->str` would probably crash your code (or whatever) because first, the address that `t->str` holds is not pointing to the heap (not from `malloc`, `calloc`, `realloc`) and second, dang it would simply mess up ur code, hope I could explain more but my english is kinda limiting me rn, lol, sorry.

Looking at your code, you may be wanting something like this:

Code struct test

```
{
    char *str;
    int size;
};
```

```
struct test *t = malloc(sizeof(test));
t->str = malloc(sizeof("test 123\0"));
strcpy(t->str, "test 123\0");
// Or instead of doing the two lines of code before this, you could just
// t->str = strdup("test 123\0");
// still free'able
t->size = 10; // Is this supposed to be the size of the string pointed by str above?
```

```
// Then this is now valid, and is actually neccessary
free(t->str);
free(t);
```

Ok, I may be wrong at whatever I just said, please correct me.

Thanks for your answer, what you sad is correct I think. The problem is that I have a function that is taking a char pointer (to set the str pointer) to initialize and return a struct pointer. I also would like to have a function that clears from memory that struct. This is the crucial part (and why I asked the first question) how can I Be sure that the pointer I am receiving is pointing to heap memory and not stack? Because like you already sad, I can't just free a pointer that is addressing the stack

dracx - February 12, 2020, 3:40 am

(Feb 11, 2020 11:18 AM)Cereal000 Wrote: [\[CITE\]](#) Thanks for your answer, what you sad is correct I think. The problem is that I have a function that is taking a char pointer (to set the str pointer) to initialize and return a struct pointer. I also would like to have a function that clears from memory that struct. This is the crucial part (and why I asked the first question) how can I Be sure that the pointer I am receiving is pointing to heap memory and not stack? Because like you already sad, I can't just free a pointer that is addressing the stack

Ok, so bout my previous comment, am not really sure if the string will be placed on the stack or some read-only memory in some data segment or whatever, it varies in compiler implementation, but the point is, it won't be stored on the pointer variable you had.

Now, you have a function that takes a char pointer as an argument and uses that to initialize and return a struct pointer. What is your goal exactly? Not to leave any traces of the string in memory (well, after your clear function)?

SIM-3602340 - February 12, 2020, 4:18 am

My point is that i don't want to leak memory ahah, I am worried that after i clear the struct, that string is still in memory, taking space.

akali - February 12, 2020, 5:00 am

(Feb 12, 2020 05:18 AM)Cereal000 Wrote: [\[CITE\]](#) My point is that i don't want to leak memory ahah, I am worried that after i clear the struct, that string is still in memory, taking space.

malloc Will never return a pointer to the stack. Only to the heap. Unless you use some weird compiler settings which forces it to allocate on the stack. But calling free on memory doesn't set that memory values to 0. That's something you have to do manually.

```
Code char* str = malloc(6);
strcpy(str,"hello");
```

```
memset(str,0,6);//clear region of memory
free(str);
```

And if you really want to be certain if it's on the heap or the stack , just print the addresses and check if they are within the stack base and cieling.

```
Code int x = 10;
int y = 20;
int* z = malloc(sizeof(int));
```

```
printf("%p %p %p",&x,&y,z);
```

//x and y should be very close to each other such as x4B8, x4B4, while z should be higher than the stack ceiling

SIM-3602340 - February 12, 2020, 6:14 am

Oh true I can just check the addresses and try to assume if an address is owned by the stack or by the heap, thank you i didn't think about it