Comparison to NULL pointer in c

Asked 4 years, 6 months ago Modified 4 years, 6 months ago Viewed 2k times

2

This is a code from a tutorial in which user enters the size of the string and string itself. The code should uses memory allocation to reproduce the same string. I have few doubts in the code-

1. Why is the `*text` pointer initialized to NULL at the beginning? Was this initialization useful in later part of the program or it is good practice to initialize to NULL.
2. Why is it comparing the pointer to NULL. Won't the address change once we allocate a string to the pointer? At the end of the string will pointer point to NULL (no address)?
3. What is the use of `scanf(" ")`?
4. After freeing `*text` pointer, it was again allocated to NULL. So did it effectively free up memory?

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

int main()
{
   int size;
   char *text = NULL; //-------------------------------------------------------
--->1

   printf("Enter limit of the text: \n");
   scanf("%d", &size);

   text = (char *) malloc(size * sizeof(char));

   if (text != NULL) //-------------------------------------------------------
-->2
   {
       printf("Enter some text: \n");
       scanf(" "); //-------------------------------------------------------
>3
       gets(text);

       printf("Inputted text is: %s\n", text);
   }

   free(text);
   text = NULL;//-------------------------------------------------------->4
   return 0;
}
```

c pointers

Share Improve this question Follow

asked Jun 13, 2020 at 19:14

Roshan
43 • 5

Question 1 and 4: Some people may consider it good practice to set the value of a pointer to NULL, so that you never have a wild or dangling pointer. However, most of the time, I do not do this, except when I have a special reason to explicitly mark the pointer as invalid. Setting the pointer to NULL does not actually free the memory, this is only done by `free(text);`. – Andreas Wenzel Jun 13, 2020 at 19:25

@AndreasWenzel so after freeing memory one should not initialize pointer to NULL? As done in #4 – Roshan Jun 13, 2020 at 19:27

4 The source has several minor flaws, I would recommend to drop this tutorial. Especially, since it makes you ask the valid questions. – the busybee Jun 13, 2020 at 19:31

2 Question 2: If the function `malloc` fails for some reason (for example out of memory), it will return `NULL`. The `if` statement checks whether `malloc` returned `NULL`. This has nothing to do with the pointer being initialized to `NULL` before the call to `malloc`. – Andreas Wenzel Jun 13, 2020 at 19:32

3 You didn't flag `gets` which is now obsolete. Please read Why is the gets function so dangerous that it should not be used? – Weather Vane Jun 13, 2020 at 19:41

3 Answers

Sorted by: Highest score (default)

3

> Why is the *text pointer initialized to NULL at the beginning?

To protect you from your own humanity, mainly. As the code evolves, it's often easy to forget to initialize the pointer in one or more branches of code and then you're dereferencing an uninitialized pointer - it is undefined behavior, and as such it's not guaranteed to crash. In worst case, if you don't use proper tools such as Valgrind (it'd point it out right away), you can spend hours or days finding such a problem because of how unpredictable it is, and because the behavior changes based on what else was on the stack before the call - so you might see a "bug" in a completely unreleated and perfectly not-buggy code.

> Why is it comparing the pointer to NULL.

Because `malloc` can return a NULL and just because it returns it doesn't mean you can dereference it. The null pointer value is special: it means "hey, this pointer is not

valid, don't use it for anything". So before you dereference anything returned from `malloc`, you have to check that it's not null. To do otherwise is undefined behavior, and modern compilers may do quite unexpected things to your code when such behavior is present. But before asking such a question I'd advise to always check what is the function you're wondering about actually designed to do. Google `cppref malloc` and the first hit is: https://en.cppreference.com/w/c/memory/malloc. There, under the heading of *Return value*, we read:

> On failure, returns a null pointer.

That's why it's comparing the pointer to NULL!

> What is the use of scanf(" ")?

That one is easy: you could have looked it up yourself. The C standard library is *well documented*: https://en.cppreference.com/w/c/io/fscanf

When you read it, the relevant part is:

> - format: pointer to a null-terminated character string specifying how to read the input. The format string consists of [...] whitespace characters: any single whitespace character in the format string consumes all available consecutive whitespace characters from the input (determined as if by calling isspace in a loop). Note that there is no difference between "\n", " ", "\t\t", or other whitespace in the format string.

And there's your answer: `scanf(" ")` will consume any whitespace characters in the input, until it reaches either EOF or the first non-whitespace character.

> After freeing *text pointer, it was again allocated to NULL. So did it effectively free up memory?

No. First of all, the language used here is wrong: the pointer was **assigned a value of NULL**. Nothing was allocated! Pointer is like a postal address. You can replace it with the word "NOWHERE", and that's what NULL is. But putting something like "this person has no address" in your address book you have not "allocated" anything.

Yes - `free` did free the memory. Then you can set it to NULL because you're human, so that you won't forget so easily that the pointer's value is not valid anymore. It's in this case a "note to self". Humans tend to forget that a pointer is null and then will use it. Such use is undefined behavior (your program can do anything, for example erase

your hard drive). So the `text = NULL` assignment has nothing to do with the machine. It has everything to do with you: humans are not perfect, and it's best to program defensively so that you give yourself less chances to introduce a bug as you change the code, or as you work under deadline pressure, etc.

Generally speaking, the `NULL` assignment at the end of `main` is not necessary in such a simple short program. But you have to recognize the fact that `text` cannot be dereferenced after it has been `free`-d.

Personally, I find it best to leverage the property of C language that gives variables lexical scope. Once the scope ends, the variable is not accessible, so you can't write a bug that would use `text` - it won't compile. This is called "correctness by design": you design the software in such a way that some bugs are impossible by construction, and if you code the bug then the code won't compile. That's a million times better than catching the bug at runtime, or worse - having to debug it, potentially in unrelated code (remember: undefined behavior is nasty - it often manifests as problems thousands of lines away from the source).

So here's how I'd rewrite it just to address this one issue (there are others still left there):

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

void process_text(int size)
{
    char *const text = malloc(size * sizeof(char));
    if (!text) return;

    printf("Enter some text: \n");
    scanf(" ");
    gets(text);
    printf("Inputted text is: %s\n", text);

    free(text);
}

int main()
{
   int size;

   printf("Enter limit of the text: \n");
   scanf("%d", &size);

   process_text(size);
}
```

The scope of `text` is limited to the block of `process_text`. You initialize it immediately at the point of declaration: that's always preferred. There's no need to set it to NULL first, since you assign the desired value right away. You check if maybe

`malloc` has returned NULL (i.e. it failed to allocate memory), and if so you immediately return from the function. A NULL check is idiomatically written as `if (pointer) /* do something if the pointer is non-null */` or as `if (!pointer) /* do something if the pointer IS NULL */`. It's less verbose that way, and anyone reading such code is supposed to know what it means if they have any sort of experience. Now you know too what such code means. It's not a big hurdle to be aware of this idiom. It's less typing and less distraction.

Generally speaking, code that returns early should be preferred to nested `if` blocks and unending levels of indentation. When there are multiple checks before a function can do its job, they often end up in nested `if` statements, making the function much harder to read.

There's a flip side to that: in C++ the code is supposed to leverage C++ (i.e. it's not just C compiled with a C++ compiler), and the resources that have to be released when returning from a function should be automatically released by the compiler generated code that invokes destructors. But in C no such automatic destructor calls are made. So if you return from a function early, you have to make sure that you've released any resources that were allocated earlier on. Sometimes the nested `if` statements help with that, so you shouldn't be blindly following some advice without understanding the context and assumptions the advice makes :)

Although it's truly a matter of preference - and I have C++ background where the code written as above is way more natural - in C probably it'd be better not to return early:

```
void process_text_alternative_version(int size)
{
    char *text = malloc(size * sizeof(char));
    if (text) {
        printf("Enter some text: \n");
        scanf(" ");
        gets(text);
        printf("Inputted text is: %s\n", text);
    }
    free(text);
}
```

The value of `text` is only used if it's not null, but we don't return from the function early. This ensures that **in all cases will the memory block pointed to by `text` - if any - gets freed**! This is very important: it's yet another way to write code that's correct by design, i.e. in a way that makes certain mistakes either impossible or much harder to commit. Written as above, you have no way of forgetting to free the memory (unless you add a return statement somewhere inside).

It must be said that even though some decisions made in the design of the C language library have been atrocious, the interface to `free` has been thoughtfully made in a way that makes the above code valid. `free` is **explicitly allowed to be**

passed a null pointer. When you pass it a null pointer - e.g. when `malloc` above failed to allocate the memory - it will do nothing. That is: "freeing" a null pointer is a perfectly valid thing to do. It doesn't do anything, but it's not a bug. It enables writing code like above, where it's easy to see that in all cases `text` will be freed.

A VERY IMPORTANT COROLLARY: null pointer checks before `free` (in C) or `delete` (in C++) indicate that the author of the code doesn't have a clue about the most basic behavior of `free` and `delete`: it's usually an indicator that the code will be written as if it was a black magical art that no mere mortal understands. If the author doesn't understand it, that is. But we can and *must* do better: we can *educate ourselves* about what the functions/operators that we use do. It's documented. It costs no money to look that documentation up. People have spent long hours making sure the documentation is there for anyone so inclined to see. Ignoring it is IMHO the very definition of insanity. It's sheer irrationality on a wild rollercoaster ride. For the sane among us: all it takes is a google search that includes the word *cppref* somewhere. You'll get cppreference links up top, and that's a reliable resource - and collaboratively written, so you can fix any shortcomings you note, since it's a wiki. It's called "cpp"reference, but it really is two references in one: a C++ Reference as well as a C Reference.

Back to the code in question, though: someone could have written it as follows:

```
void process_text_alternative_version_not_recommended(int size)
{
    char *text = malloc(size * sizeof(char));
    if (text) {
        printf("Enter some text: \n");
        scanf(" ");
        gets(text);
        printf("Inputted text is: %s\n", text);
        free(text);
    }
}
```

It's just as valid, but such form defeats the purpose: it's not clear at a glance that `text` is always freed. You have to inspect the condition of the `if` block to convince yourself that indeed it will get freed. This code will be OK for a while, and then years later someone will change it to have a bit fancier `if` condition. And now you got yourself a memory leak, since in some cases `malloc` will succeed, but `free` won't be called. You're now hoping that some future programmer, working under pressure and stress (almost invariably!) will notice and catch the problem. Defensive programming means that we protect ourselves not only from bad inputs (whether errant or malicious), but also from our own inherent human fallibility. Thus it makes most sense in my opinion to use the first alternative version: it won't turn into a memory leak no matter how you modify the `if` condition. But beware: messing up the `if` condition may turn it into undefined behavior if the test becomes broken such that the body of

`if` executes in spite of the pointer being null. It's not possible to completely protect ourselves from us, sometimes.

As far as constness is concerned, there are 4 ways of declaring the `text` pointer. I'll explain what they all mean:

1. `char *text` - a non-const pointer to non-const character(s): the pointer can be changed later to point to something else, and the characters it points to can be changed as well (or at least the compiler won't prevent you from doing it).
2. `char *const text` - a const pointer to non-const character(s) - the pointer itself cannot be changed past this point (the code won't compile if you try), but the characters will be allowed to be changed (the compiler won't complain but that doesn't mean that it's valid to do it - it's up to you the programmer to understand what the circumstances are).
3. `const char *text` - a non-const pointer to const character(s): the pointer can be changed later to point somewhere else, but the characters it points to cannot be changed using that pointer - if you try, the code won't compiler.
4. `const char *const text` - a const pointer to const character(s): the pointer cannot be changed after its definition, and it cannot be used to change the character(s) it points to - an attempt to do either will prevent the code from compiling.

We chose variant #2: the pointed-to characters can't be constant since `gets` will definitely be altering them. If you used the variant #4, the code wouldn't compile, since `gets` expects a pointer to non-const characters.

Choosing #2 we're less likely to mess it up, and we're explicit: this pointer here will remain the same for the duration of the rest of this function.

We also `free` the pointer immediately before leaving the function: there's no chance we'll inadvertently use it after it was freed, because there's literally nothing done after `free`.

This coding style protects you from your own humanity. Remember that a lot of software engineering has nothing whatsoever to do with machines. The machine doesn't care much about how comprehensible the code is: it will do what it's told - the code can be completely impenetrable to any human being. The machine doesn't care one bit. The only entitities that are affected - positively or negatively - by the design of the code are the human developers, maintainers, and users. Their humanity is an inseparable aspect of their being, and that implies that they are imperfect (as opposed to the machine which normally is completely dependable).

Finally, this code has a big problem - it again has to do with humans. Indeed you ask the user to enter the size limit for the text. But the assumption must be that humans - being humans - will invariably mess it up. And you'll be absolutely in the wrong if you blame them for messing it up: to err is human, and if you pretend otherwise then you're just an ostrich sticking your head in the sand and pretending there's no problem.

The user can easily make a mistake and enter text longer than the size they declared. That's undefined behavior: the program at this point can do anything, up to and including erasing your hard drive. Here it's not even a joke: in some circumstances it's possible to artificially create an input to this program that would cause the hard drive to indeed be wiped. You may think that it's a far-off possibility, but that's not the case. If you wrote this sort of a program on an Arduino, with an SD card attached, I could create input for both size and text that would cause the contents of the SD card to be zeroed - possibly even an input that can all be typed on a keyboard without use of special control characters. I'm 100% serious here.

Yes, typically this "undefined behavior means you'll format your hard drive" is said tongue-in-cheek, but that doesn't mean preclude it from being a true statement in the right circumstances (usually the more expensive the circumstances, the truer it becomes - such is life). Of course in most cases the user is not malicious - merely error-prone: they'll burn your house down because they were drunk, not because they tried to kill you - that's an awesome consolation I'm sure! But if you get a user that's an adversary - oh boy, they absolutely will leverage all such buffer overrun bugs to take over your system, and soon make you think hard about your choice of career. Maybe landscaping doesn't look all that bad in retrospect when the alternative is to face a massive lawsuit over loss of data (whether disclosure of data or a true loss when the data is wiped and lost).

To this effect, `gets()` is an absolutely forbidden sort of an interface: it's not possible to make it safe, that is: to make it work when faced with users that are either human, drunk and just error-prone, or worse - an adversary determined to create yet another "Data leak from Bobby Tables' Bank, Inc." headline in the newspaper.

In the second round of fixes, we need to get rid of the `gets` call: it's basically a big, absurdly bad mistake that the authors of the original C standard library have committed. I am not joking when I say that millions if not billions of dollars have been lost over decades because `gets` and similarly unsafe interfaces should never ever have been born, and because programmers have been unwittingly using them in spite of their inherently broken, dangerous and unsafe design. What's the problem: well, how on Earth can you tell `gets` to limit the length of input to actually fit in however much memory you have provided? Sadly, you can't. `gets` assumes that you-the-programmer have made no mistakes, and that wherever the input's coming from will

fit into the space available. Ergo `gets` is totally utterly broken and any reasonable C coding standard will simply state "Calls to `gets` are not allowed".

Yes. Forget about `gets`. Forget about any examples you saw of people calling `gets`. They are all wrong. Every single one of them. I'm serious. All code using `gets` is broken, there's no qualification here. If you use `gets`, you're basically saying "Hey, I've got nothing to lose. If some big institution exposes millions of their users' data, I'm fine with getting sued and having to live under a bridge thereafter". I bet you'd be *not* so happy about getting sued by a million angry users, so that's where the tale of `gets` ends. From now on it doesn't exist, and if someone tell you about using `gets`, you need to look at them weird and tell them "WTF are you talking about? Have you lost your mind?!". That's the only proper response. It's that bad of a problem. No exaggeration here, I'm not trying to scare you.

As for what to do instead of `gets`? Of course it's a well solved problem. See this question to learn everything you should know about it!

Share
Improve this answer
Follow

edited Jun 13, 2020 at 22:52

answered Jun 13, 2020 at 22:03

Kuba hasn't forgotten Monica

98.2k ● 17 ● 164 ● 327

2

In this function:

1. It is not needed at all as there is no danger that the automatic variable will be used not initialized in this function
2. This test checks if `malloc` was successful or not. If `malloc` fails it returns `NULL`
3. a bit weird way to skip blanks
4. This statement is not needed at all. The function terminates and variable stops to exists.

The conclusion: I would not rather recommend this kind of code to be used as an example when you learn programming. The authors C knowledge is IMO very limited

Share Improve this answer Follow

answered Jun 13, 2020 at 21:12

0

66.9k ● 4 ● 37 ● 81

-1

Whenever we declare a variable, it is a good practice to initialize it with some value. As you are declaring a dynamic array here, you are initializing it with `NULL`.

It is set to NULL so that it can be helpful to check if the text is valid or not. If somehow the malloc failed, the text will be still `NULL`. So you can check whether the malloc failed or not to allocate the memory. Try to put an invalid number for size like `-1`. You will see that the program won't prompt for the text input, as malloc failed and text is still `NULL`. I think this answer your query 1, 2, and 4 about why the text is being set to NULL and why it is checking whether the text is NULL or not.

For the 3rd query, After you get the input of size using `scanf("%d", &size);`, you are pressing `Enter`. If you don't use the `scanf(" ")` the pressed `Enter` will be taken as the end of gets(text) and text would be always empty. So to ignore the `Enter` pressed after `scanf("%d", &size);`, `scanf(" ")` is being used.

Share

Improve this answer

Follow

edited Jun 13, 2020 at 19:47

answered Jun 13, 2020 at 19:38

ksohan

1,203 ● 2 ● 10 ● 24

Please don't use C++ to answer C questions. They are two different languages. Further, there is no guarantee that `NULL` equates to an integral value of `0` as far as I'm aware. – Qix - MONICA WAS MISTREATED Jun 13, 2020 at 19:47

Thanks, I removed that part. – ksohan Jun 13, 2020 at 19:49

4 Tutorials Point is **not** a standards body, and even worse, their usage of `%u` is undefined behavior. – Qix - MONICA WAS MISTREATED Jun 13, 2020 at 20:03

1 @SeanTashlik Sooo wrong: "The macros are `NULL` which expands to an implementation-defined null pointer constant" Note the C standard does ***NOT*** specify that `NULL` must be zero. See stackoverflow.com/questions/2597142/... for many examples of systems where `NULL` is **not** zero. – Andrew Henle Jun 13, 2020 at 21:43

1 @P__J__ I don't care about the Linux Kernel. I care about the C standard. – Qix - MONICA WAS MISTREATED Jun 13, 2020 at 22:41