# What REALLY happens when you don't free after malloc before program termination?

Asked 15 years, 9 months ago  Modified 11 months ago  Viewed 179k times

665

We are all taught that you MUST free every pointer that is allocated. I'm a bit curious, though, about the real cost of not freeing memory. In some obvious cases, like when `malloc()` is called inside a loop or part of a thread execution, it's very important to free so there are no memory leaks. But consider the following two examples:

First, if I have code that's something like this:

```
int main()
{
    char *a = malloc(1024);
    /* Do some arbitrary stuff with 'a' (no alloc functions) */
    return 0;
}
```

What's the real result here? My thinking is that the process dies and then the heap space is gone anyway so there's no harm in missing the call to `free` (however, I do recognize the importance of having it anyway for closure, maintainability, and good practice). Am I right in this thinking?

Second, let's say I have a program that acts a bit like a shell. Users can declare variables like `aaa = 123` and those are stored in some dynamic data structure for later use. Clearly, it seems obvious that you'd use some solution that will calls some *alloc function (hashmap, linked list, something like that). For this kind of program, it doesn't make sense to ever free after calling `malloc` because these variables must be present at all times during the program's execution and there's no good way (that I can see) to implement this with statically allocated space. Is it bad design to have a bunch of memory that's allocated but only freed as part of the process ending? If so, what's the alternative?

c malloc free

Share
Improve this question
Follow

edited Jan 12, 2022 at 10:48

user438383
6,196 ● 10 ● 29 ● 47

asked Mar 17, 2009 at 15:29

Scott
10.7k ● 13 ● 40 ● 37

36 People below keep saying that a good modern OS does cleanup but what if the code is running in kernel mode (e.g., for performance reasons)? Are kernel mode programs (in Linux for example) sandboxed? If not, I believe so you would need to manually free everything, I suppose, even before any abnormal terminations like with abort(). – SO Stinks Mar 3, 2015 at 23:48

12 @Dr.PersonPersonII Yes, code running in kernel mode typically has to manually free everything. – zwol Dec 7, 2016 at 14:39

7 I would like to add that `free(a)` doesn't really do anything to actually free memory! It merely resets some pointers in the libc implementation of malloc which keep track of available chunks of memory inside a big mmapped memory page (commonly called the "heap"). That page is still going to only be freed when your program terminates, not before. – Marco Bonelli Aug 16, 2019 at 1:37

3 @MarcoBonelli Partially true. If the `malloc()`ed memory came from the "normal" sbrk heap, and was on its end, `sbrk()` is called to reduce the memory image. And if `malloc()` allocated the memory via `mmap()`, it is unmapped in `free()`. – glglgl Aug 28, 2019 at 21:01

5 @SOStinks A kernel "program" doesn't exit like a userspace program. The equivalent of kernel code "existing" is the system shutting down, in which case you don't need to free it. It's just that in the kernel, you're typically running for so long that there are very few cases when you *don't* need to free memory. – forest May 22, 2021 at 3:08

## 20 Answers

Sorted by: Highest score (default)

462

Just about every modern operating system will recover all the allocated memory space after a program exits. The only exception I can think of might be something like Palm OS where the program's static storage and runtime memory are pretty much the same thing, so not freeing might cause the program to take up more storage. (I'm only speculating here.)

So generally, there's no harm in it, except the runtime cost of having more storage than you need. Certainly in the example you give, you want to keep the memory for a variable that might be used until it's cleared.

However, it's considered good style to free memory as soon as you don't need it any more, and to free anything you still have around on program exit. It's more of an exercise in knowing what memory you're using, and thinking about whether you still need it. If you don't keep track, you might have memory leaks.

On the other hand, the similar admonition to close your files on exit has a much more concrete result - if you don't, the data you wrote to them might not get flushed, or if they're a temp file, they might not get deleted when you're done. Also, database handles should have their transactions committed and then closed when you're done with them. Similarly, if you're using an object oriented language like C++ or Objective

C, not freeing an object when you're done with it will mean the destructor will never get called, and any resources the class is responsible might not get cleaned up.

Share

Improve this answer

Follow

edited Mar 24, 2009 at 15:41

answered Mar 17, 2009 at 15:32

Paul Tomblin

183k ● 59 ● 323 ● 410

24 It's probably would be also good to mention that not everyone is using a modern operating system, if someone takes your program (and it still runs on an OS that does not recover memory) runs it then GG. – user105033 Nov 12, 2009 at 19:48

154 Which part of "However, it's considered good style to free memory as soon as you don't need it any more, and to free anything you still have around on program exit." do you consider wrong, then? – Paul Tomblin Dec 31, 2009 at 22:37

34 If you have a memory store that you need right up until the moment the program exits, and you're not running on a primitive OS, then freeing the memory just before you exit is a stylistic choice, not a defect. – Paul Tomblin Jan 2, 2010 at 12:49

41 @Paul -- Just agreeing with EvilTeach, is not considered good style to free memory, it's incorrect not to free memory. Your wording makes this seem about as important as wearing a handkerchief which matches your tie. Actually, it's on the level of wearing pants. – Heath Hunnicutt Jun 24, 2011 at 18:17

15 Final paragraph is plain wrong. In the C language, normal program termination via `exit` or the implicit `exit` when returning from `main` closes and flushes all open files. – R.. GitHub STOP HELPING ICE Jan 29, 2012 at 4:53

130

Yes you are right, your example doesn't do any harm (at least not on most modern operating systems). All the memory allocated by your process will be recovered by the operating system once the process exits.

Source: Allocation and GC Myths (PostScript alert!)

> Allocation Myth 4: Non-garbage-collected programs should always deallocate all memory they allocate.
>
> The Truth: Omitted deallocations in frequently executed code cause growing leaks. They are rarely acceptable. but Programs that retain most allocated memory until program exit often perform better without any intervening deallocation. Malloc is much easier to implement if there is no free.
>
> In most cases, **deallocating memory just before program exit is pointless.** The OS will reclaim it anyway. Free will touch and page in the dead objects; the OS won't.

> Consequence: Be careful with "leak detectors" that count allocations. Some "leaks" are good!

That said, you should really try to avoid all memory leaks!

Second question: your design is ok. If you need to store something until your application exits then its ok to do this with dynamic memory allocation. If you don't know the required size upfront, you can't use statically allocated memory.

Share

Improve this answer

Follow

edited Aug 23, 2014 at 11:34

Csq

5,845 ● 6 ● 27 ● 39

answered Mar 17, 2009 at 15:32

compie

10.5k ● 15 ● 57 ● 79

3 Might be because the question, as I read it, is what is actually happening to the leaked memory, not whether this specific example is okay. I wouldn't vote it down though, because it's a still a good answer. – DevinB Mar 17, 2009 at 15:49

3 Probably that there were (early Windows, early Mac OS), and maybe still are, OSes which require processes to free memory before exit otherwise the space isn't reclaimed. – Pete Kirkham Mar 17, 2009 at 15:54

14 I think it is wrong to explain the need for free()'ing the memory by saying "because of the leak detector". This is like saying "you have to drive slowly in a play street because police men could be waiting for you with a speed camera". – Sebastian Mach Mar 24, 2009 at 15:23

6 Indeed, a one-time modest-sized leak is a non-problem, even in a long-running program. (Emphasis on the "one-time" part.) However, it's still best-practice to clean it up so that the validator doesn't whine - not so much because shutting the validator up is useful on its own, but because if you have a bunch of "acceptable" failures in your validation output then it is much harder to find the *unacceptable* ones. – Jordan Brown Nov 25, 2019 at 23:16

1 The quoted paper was written with UNIX systems on mind. The claims made about OS reclamation of memory in the paper are NOT portable! ISO/IEC 9899, which defines the `malloc()` function, makes no requirement that program termination (as by `exit()`, `abort()`, `_Exit()`) implicitly frees memory. Telling people that it's ok to not call `free()` is inconsequential on Linux, Windows, and most (all?) Unix systems. But it is still bad advice: There are myriads of other hosted environments out there in the embedded/IoT and retro space, where not calling `free()` has bad consequences. – Christian Hujer Apr 25, 2022 at 2:47

76

=== What about **future proofing** and **code reuse**? ===

If you **don't** write the code to free the objects, then you are limiting the code to only being safe to use when you can depend on the memory being free'd by the process being closed ... i.e. small one-time use projects or "throw-away"[1] projects)... where you know when the process will end.

If you **do** write the code that free()s all your dynamically allocated memory, then you are future proofing the code and letting others use it in a larger project.

---

[1] regarding "throw-away" projects. Code used in "Throw-away" projects has a way of not being thrown away. Next thing you know ten years have passed and your "throw-away" code is still being used).

I heard a story about some guy who wrote some code just for fun to make his hardware work better. He said "just a hobby, won't be big and professional". Years later lots of people are using his "hobby" code.

Share

Improve this answer

Follow

edited Mar 15, 2018 at 13:42

answered Mar 17, 2009 at 16:43

Trevor Boyd Smith

**19.2k** ● 35 ● 129 ● 190

22 Downvoted for "small projects". There are many large projects which very intentionally *do not* free memory on exit because it's a waste of time if you know your target platforms. IMO, a more accurate example would have been "isolated projects". E.g. if you're making a reusable library that will be included in other applications, there is no well-defined exit point so you shouldn't be leaking memory. For a standalone application, you will always know exactly when the process is ending and can make a conscious decision to offload cleanup to the OS (which has to do the checks either way). – Dan Bechard Jul 14, 2017 at 16:10

8 Yesterday's application is today's library function, and tomorrow it'll be linked into a long-lived server that calls it thousands of times. – Adrian McCarthy Mar 14, 2018 at 22:01

2 @AdrianMcCarthy: If a function checks whether a static pointer is null, initializes it with `malloc()` if it is, and terminates if the pointer is still null, such a function may be safely used an arbitrary number of times even if `free` is never called. I think it's probably worthwhile to distinguish memory leaks which can use up an unbounded amount of storage, versus situations which can only waste a finite and predictable amount of storage. – supercat Apr 24, 2018 at 18:46

@supercat: My comment was talking about code changing over time. Sure, leaking a bounded amount of memory is not a problem. But someday, somebody's gonna want to change that function so that it's no longer using a static pointer. If the code has no provision for being able to release the pointed-to memory, that's going to be a hard change (or, worse, the change will be bad and you'll end up with an unbounded leak). – Adrian McCarthy Apr 24, 2018 at 20:57

2 @AdrianMcCarthy: Changing the code to no longer use a static pointer would likely require moving the pointer into some kind of "context" object, and adding code to create and destroy such objects. If the pointer is always `null` if no allocation exists, and non-null when an allocation does exist, having code free the allocation and set the pointer to `null` when a context is destroyed would be straightforward, especially compared with everything else that would need to be done to move static objects into a context structure. – supercat Apr 24, 2018 at 21:08

---

## You are correct, no harm is done and it's faster to just exit

61

There are various reasons for this:

- All desktop and server environments simply release the entire memory space on exit(). They are unaware of program-internal data structures such as heaps.
- Almost all `free()` implementations *do not ever* return memory to the operating system anyway.
- More importantly, it's a waste of time when done right before exit(). At exit, memory pages and swap space are simply released. By contrast, a series of free() calls will burn CPU time and can result in disk paging operations, cache misses, and cache evictions.

Regarding the *possiblility* of future code reuse justifing the *certainty* of pointless ops: that's a consideration but it's arguably not the Agile way. YAGNI!

Share

Improve this answer

Follow

edited Nov 16, 2016 at 18:08

answered Dec 28, 2009 at 6:04

DigitalRoss

146k ● 25 ● 252 ● 331

3 I once worked on a project where we spent a short amount of time trying to understand a programs memory usage (we were required to support it, we didn't write it). Based on the experience I anecdotally agree with your second bullet. However, I'd like to hear you (or someone) provide more proof that this is true. – user106740 Dec 28, 2009 at 15:53

4 Never mind, found the answer: stackoverflow.com/questions/1421491/… . Thank you SO! – user106740 Dec 28, 2009 at 16:10

8 The YAGNI principle works both ways: You're never gonna need to optimize the shutdown path. Premature optimizations and all that. – Adrian McCarthy Apr 24, 2018 at 22:02

You could mention that freeing memory before exit is not only a waste of runtime but also of developer and potentially test time. – Peter - Reinstate Monica May 12, 2021 at 11:06

1 The "Agile" argument is wrong. This is also agile: Given two or more otherwise roughly equivalent options, choose the path that makes future change easier. And that may as well be the path where `free()` is called before `exit()`: That `exit()` may not necessarily *remain* at the end of the program when the program evolves. So, the "Agile" argument is "it depends". – Christian Hujer Apr 25, 2022 at 3:05

46

I completely disagree with everyone who says OP is correct or there is no harm.

Everyone is talking about a modern and/or legacy OS's.

But what if I'm in an environment where I simply have no OS? Where there isn't anything?

Imagine now you are using thread styled interrupts and allocate memory. In the C standard ISO/IEC:9899 is the lifetime of memory stated as:

> 7.20.3 Memory management functions
>
> 1 The order and contiguity of storage allocated by successive calls to the calloc, malloc, and realloc functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation.[...]

So it has not to be given that the environment is doing the freeing job for you. Otherwise it would be added to the last sentence: "Or until the program terminates."

So in other words: Not freeing memory is not just bad practice. It produces non portable and not C conform code. Which can at least be seen as 'correct, if the following: [...], is supported by environment'.

But in cases where you have no OS at all, no one is doing the job for you (I know generally you don't allocate and reallocate memory on embedded systems, but there are cases where you may want to.)

So speaking in general plain C (as which the OP is tagged), this is simply producing erroneous and non portable code.

Share

Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 • 1

answered May 8, 2015 at 7:42

dhein

6,561 • 5 • 44 • 79

12 A counter-argument is that if you're an embedded environment, you - as the developer - would be far more fastidious in your memory management in the first place. Usually, this is to the point of actually pre-allocating static fixed memory beforehand rather than having any runtime mallocs/reallocs at all. – John Go-Soco Apr 21, 2016 at 13:45

5 @lunarplasma: While what you are saying is not incorrect, that doesn't change the fact what the languages standard is stating, and everyone who acts against/further it, may it even be by common sense, is producing limited code. I can understand if some one says "I dont have to care about it", as there are enough cases where it is ok. BUT that one should at least know WHY he doesn't have to care. and especially not omit it as long a question is not related to that special case. And since OP is asking about C in general under theoretical(school) aspects. It is not ok to say "You don't need to"! – dhein Apr 21, 2016 at 14:00

8 In most environments where there is no OS, there is no means via which programs can "terminate". – supercat Oct 14, 2016 at 18:24

2 @supercat: As I have written before: You are right about it. But if someone is asking about it in regards to teaching reasons and school aspects, it is not right to say "You don't need to think about it sicne most of the time it doesn't matter" The wording and behavior of the

language definition is given for a reason, and just because most enviroments handle it for you, you can't say there is no need to care about. Thats my point. – dhein Oct 31, 2016 at 12:42

6 -1 for quoting the C standard while most of it does NOT apply in the absence of an operating system, as there is no runtime to provide the features the standard mandates, especially regarding memory management and the standard library functions (which are also obviously absent along with the runtime/OS). – user3160514 Feb 16, 2017 at 0:58

---

26

I typically free every allocated block once I'm sure that I'm done with it. Today, my program's entry point might be `main(int argc, char *argv[])` , but tomorrow it might be `foo_entry_point(char **args, struct foo *f)` and typed as a function pointer.

So, if that happens, I now have a leak.

Regarding your second question, if my program took input like a=5, I would allocate space for a, or re-allocate the same space on a subsequent a="foo". This would remain allocated until:

1. The user typed 'unset a'
2. My cleanup function was entered, either servicing a signal or the user typed 'quit'

I can not think of any *modern* OS that does not reclaim memory after a process exits. Then again, free() is cheap, why not clean up? As others have said, tools like valgrind are great for spotting leaks that you really do need to worry about. Even though the blocks you example would be labeled as 'still reachable' , its just extra noise in the output when you're trying to ensure you have no leaks.

Another myth is "*If its in main(), I don't have to free it*", this is incorrect. Consider the following:

```
char *t;

for (i=0; i < 255; i++) {
    t = strdup(foo->name);
    let_strtok_eat_away_at(t);
}
```

If that came prior to forking / daemonizing (and in theory running forever), your program has just leaked an undetermined size of t 255 times.

A good, well written program should always clean up after itself. Free all memory, flush all files, close all descriptors, unlink all temporary files, etc. This cleanup function should be reached upon normal termination, or upon receiving various kinds of fatal signals, unless you want to leave some files laying around so you can detect a crash and resume.

Really, be kind to the poor soul who has to maintain your stuff when you move on to other things .. hand it to them 'valgrind clean' :)

Share
Improve this answer
Follow

edited Mar 3, 2016 at 15:54

answered Mar 18, 2009 at 8:36



user50049

1 `free() is cheap` unless you have a billion of data structures with complex relationship that you have to release one by one, traversing the data structure to try to release everything might end up significantly increasing your shut down time, especially if half of that data structure is already paged out to the disk, without any benefit. – Lie Ryan Mar 20, 2013 at 0:37

6 @LieRyan If you have a billion, as in *literally* a billion structures, you most decidedly have other problems that need a specialized degree of consideration - way beyond the scope of this particular answer :) – user50049 Aug 25, 2014 at 18:18

16

It is completely fine to leave memory unfreed when you exit; malloc() allocates the memory from the memory area called "the heap", and the complete heap of a process is freed when the process exits.

That being said, one reason why people still insist that it is good to free everything before exiting is that memory debuggers (e.g. valgrind on Linux) detect the unfreed blocks as memory leaks, and if you have also "real" memory leaks, it becomes more difficult to spot them if you also get "fake" results at the end.

Share Improve this answer Follow

answered Mar 17, 2009 at 15:33

Antti Huima
25.5k ● 3 ● 54 ● 71

1 Does not Valgrind do a pretty good job distinguishing between "leaked" and "still reachable"? – Christoffer Mar 17, 2009 at 15:39

13 -1 for "completely fine" It is bad coding practice to leave allocated memory without freeing it. If that code was extracted into a library, then it would cause memleaks all over the place. – DevinB Mar 17, 2009 at 15:52

7 +1 to compensate. See compie's answer. `free` at `exit` time considered harmful. – R.. GitHub STOP HELPING ICE Jan 29, 2012 at 4:55

14

This code will usually work alright, but consider the problem of code reuse.

You may have written some code snippet which doesn't free allocated memory, it is run in such a way that memory is then automatically reclaimed. Seems allright.

Then someone else copies your snippet into his project in such a way that it is executed one thousand times per second. That person now has a huge memory leak in his program. Not very good in general, usually fatal for a server application.

Code reuse is typical in enterprises. Usually the company owns all the code its employees produce and every department may reuse whatever the company owns. So by writing such "innocently-looking" code you cause potential headache to other people. This may get you fired.

Share

Improve this answer

Follow

edited May 8, 2015 at 7:08

WedaPashi

3,872 ● 1 ● 29 ● 44

answered Mar 18, 2009 at 8:00

sharptooth

170k ● 108 ● 535 ● 1k

4 It may be worth noting the possibility not just of someone copying the snippet, but also the possibility of a program that was written to do some particular action once being modified to do it repeatedly. In such a case, it would be fine to have memory get allocated *once* and then used repeatedly without ever being freed, but allocating and abandoning the memory for every action (without freeing it) could be disastrous. – supercat Nov 15, 2014 at 17:47

14

> What's the real result here?

**Your program leaked the memory. Depending on your OS, it *may* have been recovered.**

Most modern **desktop** operating systems *do* recover leaked memory at process termination, making it sadly common to ignore the problem (as can be seen by many other answers here.)

But you are relying on a safety feature not being part of the language, one you should not rely upon. Your code might run on a system where this behaviour *does* result in a "hard" memory leak, **next** time.

Your code might end up running in kernel mode, or on vintage / embedded operating systems which do not employ memory protection as a tradeoff. (MMUs take up die space, memory protection costs additional CPU cycles, and it is not too much to ask from a programmer to clean up after himself).

You can use and re-use memory (and other resources) any way you like, but make sure you deallocated all resources before exiting.

---

I thought I'd add this little historical gem, a screenshot from the Amiga's Rom Kernel Manuals, i.e. the official platform documentation.

```
In fact, Amiga programmers need to be careful with every system resource,
not just memory.  All system resources from audio channels to the floppy
disk drives are shared among tasks.  Before using a resource, you must ask
the system for access to the resource.  This may fail if the resource is
already being used by another task.

Once you have control of a resource, no other task can use it, so give it
up as soon as you are finished.  When your program exits, you must give
everything back whether it's memory, access to a file, or an I/O port. You
are responsible for this, the system will not do it for you automatically.
```

Share

Improve this answer

Follow

edited Aug 16, 2023 at 10:49

answered Jan 13, 2016 at 9:46

DevSolar

**70.1k** ● 21 ● 137 ● 215

On platforms where applications may use things like DMA without operating systems being aware of it [used to be common on the PC when using hardware the OS hadn't contemplated, or on the Amiga when using the hardware to achieve better graphics and sound than accommodated by the OS], having the OS leave memory as allocated when an application exits will cause storage to leak, which may lead to memory exhaustion, but freeing storage which is about to be written by a DMA operation is a recipe for memory corruption. – supercat Oct 25, 2020 at 16:37

@supercat That's just another resource leak, just of a different kind. Of course you have to de-register the memory with whoever might access it before freeing it. You can't free memory that is currently an IPC message to another process, either. – DevSolar Oct 26, 2020 at 6:01

2 In Amiga exec.library not calling Free() after using AllocMem() will leave the memory "lost" until reboot, malloc and free will be using these under the hood. – Richie Oct 17, 2021 at 13:10

1 @Richie Exactly what I had in mind when I wrote the answer. That some OS will clean up after you does not change the fact that YOU did leak the resource in the first place. – DevSolar Oct 17, 2021 at 17:51

1 @Arioch'The Messages, registered event handlers etc. are *dead* easy to turn into resources leaked. I've seen a co-worker write a resource-leaking Java program in a minute or two. (As in, leaking resources that weren't even recovered after the JVM shut down.) `malloc()`ed memory (that isn't registered otherwise) usually *does* get recovered by memory-protected systems. The fact remains, your program is faulty for not cleaning up after itself, either way. The only difference is whether secondary safety features of the OS did it for you. – DevSolar Jun 5 at 14:40

**11**

If you're using the memory you've allocated, then you're not doing anything wrong. It becomes a problem when you write functions (other than main) that allocate memory without freeing it, and without making it available to the rest of your program. Then your program continues running with that memory allocated to it, but no way of using it. Your program *and other running programs* are deprived of that memory.

Edit: It's not 100% accurate to say that other running programs are deprived of that memory. The operating system can always let them use it at the expense of swapping your program out to virtual memory (`</handwaving>`). The point is, though, that if your program frees memory that it isn't using then a virtual memory swap is less likely to be necessary.

Share
Improve this answer
Follow

edited Mar 17, 2009 at 15:55

answered Mar 17, 2009 at 15:34

Bill the Lizard
405k ● 211 ● 572 ● 889

---

7

There's actually a section in the OSTEP online textbook for an undergraduate course in operating systems which discusses exactly your question.

The relevant section is "Forgetting To Free Memory" in the Memory API chapter on page 6 which gives the following explanation:

> In some cases, it may seem like not calling free() is reasonable. For example, your program is short-lived, and will soon exit; ***in this case, when the process dies, the OS will clean up all of its allocated pages and thus no memory leak will take place per se.*** While this certainly "works" (see the aside on page 7), it is probably a bad habit to develop, so be wary of choosing such a strategy

This excerpt is in the context of introducing the concept of virtual memory. Basically at this point in the book, the authors explain that one of the goals of an operating system is to "virtualize memory," that is, to let every program believe that it has access to a very large memory address space.

Behind the scenes, the operating system will translate "virtual addresses" the user sees to actual addresses pointing to physical memory.

However, sharing resources such as physical memory requires the operating system to keep track of what processes are using it. So if a process terminates, then it is within the capabilities and the design goals of the operating system to reclaim the process's memory so that it can redistribute and share the memory with other processes.

---

*EDIT:* The aside mentioned in the excerpt is copied below.

> ASIDE: **WHY NO MEMORY IS LEAKED ONCE YOUR PROCESS EXITS**

> When you write a short-lived program, you might allocate some space using `malloc()`. The program runs and is about to complete: is there need to call `free()` a bunch of times just before exiting? While it seems wrong not to, no memory will be "lost" in any real sense. The reason is simple: there are really two levels of memory management in the system. The first level of memory management is performed by the OS, which hands out memory to processes when they run, and takes it back when processes exit (or otherwise die). The second level of management is within each process, for example within the heap when you call `malloc()` and `free()`. Even if you fail to call `free()` (and thus leak memory in the heap), the operating system will reclaim all the memory of the process (including those pages for code, stack, and, as relevant here, heap) when the program is finished running. No matter what the state of your heap in your address space, the OS takes back all of those pages when the process dies, thus ensuring that no memory is lost despite the fact that you didn't free it.
>
> Thus, for short-lived programs, leaking memory often does not cause any operational problems (though it may be considered poor form). When you write a long-running server (such as a web server or database management system, which never exit), leaked memory is a much bigger issue, and will eventually lead to a crash when the application runs out of memory. And of course, leaking memory is an even larger issue inside one particular program: the operating system itself. Showing us once again: those who write the kernel code have the toughest job of all...
>
> > *from Page 7 of Memory API chapter of*
> >
> > **Operating Systems: Three Easy Pieces**
> > Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau Arpaci-Dusseau Books March, 2015 (Version 0.90)

Share
Improve this answer
Follow

edited Oct 17, 2017 at 2:08

answered Oct 17, 2017 at 1:37

spenceryue
390 ● 6 ● 8

5

There's no real *danger* in not freeing your variables, but if you assign a pointer to a block of memory to a different block of memory without freeing the first block, the first block is no longer accessible but still takes up space. This is what's called a memory leak, and if you do this with regularity then your process will start to consume more and more memory, taking away system resources from other processes.

If the process is short-lived you can often get away with doing this as all allocated memory is reclaimed by the operating system when the process completes, but I would advise getting in the habit of freeing all memory you have no further use for.

Share Improve this answer Follow

answered Mar 17, 2009 at 15:33

Kyle Cronin
79k ● 45 ● 151 ● 167

2 @KyleCronin I would *much* rather have a segfault than a memory leak, because both are serious bugs and segfaults are easier to detect. All too often memory leaks go unnoticed or unresolved because they are "pretty benign". My RAM and I wholeheartedly disagree. – Dan Bechard Apr 11, 2016 at 17:49

1 @Dan As a developer, sure. As a user, I'll take the memory leak. I'd rather have software that works, albeit with leaking memory, over software that doesn't. – Kyle Cronin Apr 11, 2016 at 22:02

@KyleCronin Again, I wouldn't. As a user, I've used software with "small" memory leaks that continuously degrade performance over time. It's the most miserably frustrating experience I've ever had, up there with unexplained BSODs. I would much rather the software just crash and tell me straight up that it doesn't work, so I can move on with my life instead of clinging to a false hope that if I waste another 6 hours I might be able to get my work done. Like DevinB said, I think we'd all prefer to just eliminate both rather than argue about which ones are more annoying. ;) – Dan Bechard Apr 11, 2016 at 22:34

To summarize, "I'd rather have software that works, albeit with leaking memory, over software that doesn't." is where we fundamentally disagree. Software with memory leaks *doesn't* "work". – Dan Bechard Apr 11, 2016 at 22:37

4

It depends on the scope of the project that you're working on. In the context of your question, and I mean just your question, then it doesn't matter.

For a further explanation (optional), some scenarios I have noticed from this whole discussion are as follows:

1. If you're working in an embedded environment where you cannot rely on the main OS to reclaim the memory for you, then you should free them since memory leaks can really crash the system if unnoticed.
2. If you're working on a personal project where you won't disclose it to anyone else, then you can skip it (assuming you're using it on the main OS) or include it for "best practices" sake.
3. If you're working on a project and plan to have it open source, then you need to do more research into your audience and figure out if freeing the memory would be the better choice.
4. If you have a large library and your audience consisted of only the main OS, then you don't need to free it as their OS will help them to do so. In the meantime, by

not freeing, your libraries/program may help to make the overall performance snappier since the program does not have to close every data structure, prolonging the shutdown time (imagine a very slow excruciating wait to shut down your computer before leaving the house...)

I can go on and on specifying which course to take, but it ultimately depends on what you want to achieve with your program. Freeing memory is considered good practice in some cases and not so much in some, so it ultimately depends on the specific situation you're in and asking the right questions at the right time. Good luck!

Share
Improve this answer
Follow

edited Jan 21 at 11:48

Toby Speight
**30.6k** ● 49 ● 72 ● 113

answered Oct 21, 2020 at 7:46

ken_you_not
**93** ● 1 ● 2 ● 11

Some major open source projects (including the GCC compiler) don't bother `free`-ing *all* dynamically `malloc`ed memory. I pragmatically believe they are right to do so (since the OS kernel would recover it at process termination) – Basile Starynkevitch Aug 8, 2023 at 16:40

---

**3**

You are correct, memory is automatically freed when the process exits. Some people strive not to do extensive cleanup when the process is terminated, since it will all be relinquished to the operating system. However, while your program is running you should free unused memory. If you don't, you may eventually run out or cause excessive paging if your working set gets too big.

Share Improve this answer Follow

answered Mar 17, 2009 at 15:35

Michael
**55.3k** ● 5 ● 126 ● 145

3

You are absolutely correct in that respect. In small trivial programs where a variable must exist until the death of the program, there is no real benefit to deallocating the memory.

In fact, I had once been involved in a project where each execution of the program was very complex but relatively short-lived, and the decision was to just keep memory allocated and not destabilize the project by making mistakes deallocating it.

That being said, in most programs this is not really an option, or it can lead you to run out of memory.

Share Improve this answer Follow

answered Mar 17, 2009 at 15:35



Uri

89.6k ● 51 ● 226 ● 322

---

2

If you're developing an application from scratch, you can make some educated choices about when to call free. Your example program is fine: it allocates memory, maybe you have it work for a few seconds, and then closes, freeing all the resources it claimed.

If you're writing anything else, though -- a server/long-running application, or a library to be used by someone else, you should expect to call free on everything you malloc.

Ignoring the pragmatic side for a second, it's much safer to follow the stricter approach, and force yourself to free everything you malloc. If you're not in the habit of watching for memory leaks whenever you code, you could easily spring a few leaks. So in other words, yes -- you can get away without it; please be careful, though.

Share Improve this answer Follow

answered Mar 17, 2009 at 15:35

ojrac

13.4k ● 6 ● 39 ● 40

---

2

If a program forgets to free a few Megabytes before it exits the operating system will free them. But if your program runs for weeks at a time and a loop inside the program forgets to free a few bytes in each iteration you will have a mighty memory leak that will eat up all the available memory in your computer unless you reboot it on a regular basis => even small memory leaks might be bad if the program is used for a seriously big task even if it originally wasn't designed for one.

Share Improve this answer Follow

answered Jul 27, 2018 at 4:22

Gunter Königsmann

380 ● 4 ● 7

+100 times this. If your server program leaks every time it services something, you will eventually run out of memory. For a run once program, it is less of a big deal. For a run 24x7 it is a big deal. From my perspective, if you don't deallocate your resources, you are a no-hire. – EvilTeach Jan 4, 2023 at 18:06

1

It depends on the OS environment the program is running in, as others have already noted, and for long running processes, freeing memory and avoiding even very slow leaks is important always. But if the operating system deals with stuff, as Unix has done for example since probably forever, then you don't need to free memory, nor close files (the kernel closes all open file descriptors when a process exits.) If your program allocates a lot of memory, it may even be beneficial to exit without "hesitation". I find that when I quit Firefox, it spends several !minutes ! paging in gigabytes of memory in many processes. I guess this is due to having to call destructors on C++ objects. This is actually terrible. Some might argue, that this is necessary to save state consistently, but in my opinion, long-running interactive programs like browsers, editors and design programs, just to mention a few, should ensure that any state information, preferences, open windows/pages, documents etc is frequently written to permanent storage, to avoid loss of work in case of a crash. Then this state-saving can be performed again quickly when the user elects to quit, and when completed, the processes should just exit immediately.

Share Improve this answer Follow

answered Apr 28, 2022 at 6:25

LHP

180 ● 1 ● 10

0

All memory allocated for this process will be marked unused by OS then reused, because the memory allocation is done by user space functions.

Imagine OS is a god, and the memories is the materials for creating a wolrd of process, god use some of materials creat a world (or to say OS reserved some of memory and create a process in it). No matter what the creatures in this world have done the materials not belong to this world won't be affected. After this world expired, OS the god, can recycle materials allocated for this world.

Modern OS may have different details on releasing user space memory, but that has to be a basic duty of OS.

Share

Improve this answer

Follow

edited Aug 1, 2022 at 15:43

answered Aug 1, 2022 at 15:35



08822407d

108 ● 8

The flags of the question are "c malloc free". Hence your answer is wrong because it could be that there is no OS at all. See the correct answer of dhein: stackoverflow.com/a/30118469/1950345 There are already a lot of answers and your answer does not provide new details. See how-to-answer. – reichhart Aug 5, 2022 at 9:21

@reichhart the question stresses that memory is retained "before program termination" which means there IS some operating system. Well, technically, a hard reset or power down of bare hardware can be seen a termination too, but those surely would reclaim the memory as well. – Arioch 'The Jun 4 at 20:25

-2

I think that your two examples are actually only one: the `free()` should occur only at the end of the process, which as you point out is useless since the process is terminating.

In you second example though, the only difference is that you allow an undefined number of `malloc()`, which could lead to running out of memory. The only way to handle the situation is to check the return code of `malloc()` and act accordingly.

Share Improve this answer Follow

answered Mar 17, 2009 at 15:38

mouviciel

67.8k ● 13 ● 108 ● 142

**Highly active question**. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.