Difference between malloc and calloc?

What is the difference between doing:

```
ptr = (char **) malloc (MAXELEMS * sizeof(char *));
```

or:

```
ptr = (char **) calloc (MAXELEMS, sizeof(char*));
```

When is it a good idea to use calloc over malloc or vice versa?

c malloc calloc

edited Aug 25 '13 at 17:40
Mat
154k ● 27 ● 285 ● 317

asked Oct 8 '09 at 15:04
user105033
6,200 ● 15 ● 44 ● 60

9 In C you don't cast the result of `malloc` family – Lưu Vĩnh Phúc Feb 17 '16 at 7:55

4 In C, you could write the above more generically as: `ptr = calloc(MAXELEMS, sizeof(*ptr));` – chqrlie Oct 26 '16 at 7:19

1 An interesting post about the difference between calloc and malloc+memset vorpus.org/blog/why-does-calloc-exist – ddddavidee Dec 13 '16 at 16:06

17 Answers

`calloc()` zero-initializes the buffer, while `malloc()` leaves the memory uninitialized.

EDIT:

Zeroing out the memory may take a little time, so you probably want to use `malloc()` if that performance is an issue. If initializing the memory is more important, use `calloc()`. For example, `calloc()` might save you a call to `memset()`.

edited Sep 14 '12 at 9:53
Jaguar
16.6k ● 23 ● 98 ● 147

answered Oct 8 '09 at 15:05
Fred Larson
43.6k ● 9 ● 88 ● 132

172 The *alloc variants are pretty mnemonic - clear-alloc, memory-alloc, re-alloc. – Cascabel Oct 8 '09 at 15:07

35 Use malloc() if you are going to set everything that you use in the allocated space. Use calloc() if you're going to leave parts of the data uninitialized - and it would be beneficial to have the unset parts zeroed. – Jonathan Leffler Oct 8 '09 at 15:16

220 `calloc` is not necessarily more expensive, since OS can do some tricks to speed it up. I know that FreeBSD, when it gets any idle CPU time, uses that to run a simple process that just goes around and zeroes out deallocated blocks of memory, and marks blocks thus processes with a flag. So when you do `calloc`, it first tries to find one of such pre-zeroed blocks and just give it to you - and most likely it will find one. – Pavel Minaev Oct 8 '09 at 15:18

24 I tend to feel that if your code becomes "safer" as a result of zero-initing allocations by default, then your code is insufficiently safe whether you use malloc or calloc. Using malloc is a good indicator that the data needs initialisation - I only use calloc in cases where those 0 bytes are actually meaningful. Also note that calloc doesn't necessarily do what you think for non-char types. Nobody really uses trap representations any more, or non-IEEE floats, but that's no excuse for thinking your code is truly portable when it isn't. – Steve Jessop Oct 8 '09 at 15:51

10 @SteveJessop "Safer" isn't the correct word. I think "Deterministic" is the better term. Code that is more deterministic rather than having failures that are dependent on timing and data sequences, will be easier to isolate failures. Calloc is sometimes an easy way to get that determinism, versus explicit initialization. – dennis May 4 '14 at 13:57

A less known difference is that in operating systems with optimistic memory allocation, like Linux, the pointer returned by `malloc` isn't backed by real memory until the program actually touches it.

`calloc` does indeed touch the memory (it writes zeroes on it) and thus you'll be sure the OS is

See for instance this SO question for further discussion about the behavior of malloc

edited May 23 '17 at 11:33 — Community ♦ — 1 ● 1

answered Oct 18 '09 at 20:44 — Isak Savo — 23.5k ● 9 ● 45 ● 83

32 `calloc` need not write zeros. If the allocated block consists mostly of new zero pages provided by the operating system, it can leave those untouched. This of course requires `calloc` to be tuned to the operating system rather than a generic library function on top of `malloc`. Or, an implementor could make `calloc` compare each word against zero before zeroing it. This would not save any time, but it would avoid dirtying the new pages. – R.. Jan 4 '11 at 13:46

3 @R.. interesting note. But in practice, does such implementations exist in the wild? – Isak Savo Jan 4 '11 at 14:00

9 All `dlmalloc`-like implementations skip the `memset` if the chunk was obtained via `mmap`ing new anonymous pages (or equivalent). Usually this kind of allocation is used for larger chunks, starting at 256k or so. I don't know of any implementations that do the comparison against zero before writing zero aside from my own. – R.. Jan 5 '11 at 15:57

1 `omalloc` also skips the `memset`; `calloc` does not need to touch any pages that are not already used by the application (page cache), ever. Though, extremely primitive `calloc` implementations differ. – mirabilos Mar 31 '14 at 21:05

8 glibc's calloc checks if it's getting fresh memory from the OS. If so, it knows it DOESN'T need to write it, because mmap(..., MAP_ANONYMOUS) returns memory that's already zeroed. – Peter Cordes Dec 7 '14 at 22:45

One often-overlooked advantage of `calloc` is that (conformant implementations of) it will help protect you against integer overflow vulnerabilities. Compare:

```
size_t count = get_int32(file);
struct foo *bar = malloc(count * sizeof *bar);
```

vs.

```
size_t count = get_int32(file);
struct foo *bar = calloc(count, sizeof *bar);
```

The former could result in a tiny allocation and subsequent buffer overflows, if `count` is greater than `SIZE_MAX/sizeof *bar`. The latter will automatically fail in this case since an object that large cannot be created.

Of course you may have to be on the lookout for non-conformant implementations which simply ignore the possibility of overflow... If this is a concern on platforms you target, you'll have to do a manual test for overflow anyway.

edited Dec 3 '13 at 10:22 — Daren Thomas — 39.3k ● 35 ● 122 ● 180

answered Aug 13 '10 at 16:53 — R.. — 145k ● 19 ● 233 ● 511

10 Apparently the arithmetic overflow was what caused OpenSSH hole in 2002. Good article from OpenBSD on the perils of this with memory-related functions: undeadly.org/cgi?action=article&sid=20060330071917 – Komrade P. Apr 24 '14 at 14:35

2 @KomradeP.: Interesting. Sadly the article you linked has misinformation right at the beginning. The example with `char` is **not** an overflow but rather an implementation-defined conversion when assigning the result back into a `char` object. – R.. Apr 24 '14 at 15:32

It's there probably for illustration purpose only. Because compiler is likely to optimise that away anyway. Mine compiles into this asm: push 1. – Komrade P. Apr 24 '14 at 17:22

1 @tristopia: The point is not that the code is exploitable on all implementations, but that it's incorrect without additional assumptions and thus not correct/portable usage. – R.. Sep 20 '14 at 21:38

3 @tristopia: If your mode of thinking is "`size_t` is 64-bit so that's no problem", that's a flawed way of thinking that's going to lead to security bugs. `size_t` is an abstract type that represents sizes, and there's no reason to think the arbitrary product of a 32-bit number and a `size_t` (note: `sizeof *bar` could in principle be greater than 2^32 on a 64-bit C implementation!) fits in `size_t`. – R.. Sep 20 '14 at 23:40

There's no difference in the size of the memory block allocated. `calloc` just fills the memory block with physical all-zero-bits pattern. In practice it is often assumed that the objects located in the memory block allocated with `calloc` have initilial value as if they were initialized with literal `0`, i.e. integers should have value of `0`, floating-point variables - value of `0.0`, pointers - the appropriate null-pointer value, and so on.

From the pedantic point of view though, `calloc` (as well as `memset(..., 0, ...)`) is only guaranteed to properly initialize (with zeroes) objects of type `unsigned char`. Everything else is not guaranteed to be properly initialized and may contain so called *trap representation*, which causes undefined behavior.

In other words, for any type other than `unsigned char` the aforementioned all-zero-bits patterm might represent an illegal value, trap representation.

Later, in one of the Technical Corrigenda to C99 standard, the behavior was defined for all integer types (which makes sense). I.e. formally, in the current C language you can initialize only integer types with `calloc` (and `memset(..., 0, ...)`). Using it to initialize anything else in general case leads to undefined behavior, from the point of view of C language.

In practice, `calloc` works, as we all know :), but whether you'd want to use it (considering the above) is up to you. I personally prefer to avoid it completely, use `malloc` instead and perform my own initialization.

Finally, another important detail is that `calloc` is required to calculate the final block size *internally*, by multiplying element size by number of elements. While doing that, `calloc` must watch for possible arithmetic overflow. It will result in unsuccessful allocation (null pointer) if the requested block size cannot be correctly calculated. Meanwhile, your `malloc` version makes no attempt to watch for overflow. It will allocate some "unpredictable" amount of memory in case overflow happens.

edited Sep 4 '13 at 4:49

answered Oct 18 '09 at 20:59

AnT

241k ● 30 ● 376 ● 622

Per the "another important detail" paragraph: that seems make to `memset(p, v, n * sizeof type);` a problem because `n * sizeof type` may overflow. Guess I'll need to use a `for(i=0;i<n;i++) p[i]=v;` loop for robust code. – chux Feb 14 '15 at 13:18

It would be helpful if there were a standard means by which code could assert that an implementation must use all-bits-zero as a null pointer (refusing compilation otherwise), since there exist implementations that use other null-pointer representations, but they are comparatively rare; code that doesn't have to run on such implementations can be faster if it can use calloc() or memset to initialize arrays of pointers. – supercat Aug 13 '16 at 21:03

1 vorpus.org/blog/why-does-calloc-exist – ddddavidee Dec 13 '16 at 16:07

@chux No, if an array with `n` elements exist where a element have the size `sizeof type`, then `n*sizeof type` can not overflow, because the maximum size of any object must be less than `SIZE_MAX`. – 12431234123412341234123 Jul 26 '17 at 13:49

@12431234123412341234123 True about an *array* size <= `SIZE_MAX`, yet there are no *arrays* here. The pointer returned from `calloc()` can point to allocated memory than exceeds `SIZE_MAX`. Many implementations do limit the product of the 2 args to `calloc()` to `SIZE_MAX`, yet the C spec does not impose that limit. – chux Jul 26 '17 at 14:01

The documentation makes the calloc look like malloc, which just does zero-initialize the memory; this is not the primary difference! The idea of calloc is to abstact copy-on-write semantics for memory allocation. When you allocate memory with calloc it all maps to same physical page which is initialized to zero. When any of the pages of the allocated memory is written into a physical page is allocated. This is often used to make HUGE hash tables, for example since the parts of hash which are empty aren't backed by any extra memory (pages); they happily point to the single zero-initialized page, which can be even shared between processes.

Any write to virtual address is mapped to a page, if that page is the zero-page, another physical page is allocated, the zero page is copied there and the control flow is returned to the client process. This works same way memory mapped files, virtual memory, etc. work.. it uses paging.

Here is one optimization story about the topic: http://blogs.fau.de/hager/2007/05/08/benchmarking-fun-with-calloc-and-zero-pages/

edited Aug 16 '13 at 8:08

answered Aug 15 '13 at 11:16

SnappleLVR

383 ● 3 ● 8

---

from an article Benchmarking fun with calloc() and zero pages on Georg Hager's Blog

> When allocating memory using calloc(), the amount of memory requested is not allocated right away. Instead, all pages that belong to the memory block are connected to a single page containing all zeroes by some MMU magic (links below). If such pages are only read (which was true for arrays b, c and d in the original version of the benchmark), the data is provided from the single zero page, which – of course – fits into cache. So much for memory-bound loop kernels. If a page gets written to (no matter how), a fault occurs, the “real” page is mapped and the zero page is copied to memory. This is called copy-on-write, a well-known optimization approach (that I even have taught multiple times in my C++ lectures). After that, the zero-read trick does not work any more for that page and this is why performance was so much lower after inserting the – supposedly redundant – init loop.

answered Aug 28 '13 at 5:51

Ashish Chavan
301 ● 3 ● 12

where is the link? – Rupesh Yadav. Feb 18 '17 at 10:09

1 first line of answer contains link to Georg Hager's Blog. – Ashish Chavan Feb 22 '17 at 9:20

`calloc` is generally `malloc+memset` to 0

It is generally slightly better to use `malloc+memset` explicitly, especially when you are doing something like:

```
ptr=malloc(sizeof(Item));
memset(ptr, 0, sizeof(Item));
```

That is better because `sizeof(Item)` is know to the compiler at compile time and the compiler will in most cases replace it with the best possible instructions to zero memory. On the other hand if `memset` is happening in `calloc`, the parameter size of the allocation is not compiled in in the `calloc` code and real `memset` is often called, which would typically contain code to do byte-by-byte fill up until long boundary, than cycle to fill up memory in `sizeof(long)` chunks and finally byte-by-byte fill up of the remaining space. Even if the allocator is smart enough to call some `aligned_memset` it will still be a generic loop.

One notable exception would be when you are doing malloc/calloc of a very large chunk of memory (some power_of_two kilobytes) in which case allocation may be done directly from kernel. As OS kernels will typically zero out all memory they give away for security reasons, smart enough calloc might just return it withoud additional zeroing. Again - if you are just allocating something you know is small, you may be better off with malloc+memset performance-wise.

edited Sep 18 '14 at 18:27

AGS
12.1k ● 5 ● 37 ● 57

answered Aug 16 '14 at 21:46

virco
141 ● 1 ● 4

+1 for the reminder that a generic implementation of a functionality in a system library is not necessarily faster than the same operation in the user code. – Patrick Schlüter Sep 20 '14 at 7:30

There is also a second point that make `calloc()` slower than `malloc()`: the multiplication for the size. `calloc()` is required to use a generic multiplication (if `size_t` is 64 bits even the very costly 64 bits*64 bits=64 bits operation) while the malloc() will often have a compile time constant. – Patrick Schlüter Sep 20 '14 at 7:37

3 glibc calloc has some smarts to decide how to most efficiently clear the returned chunk, e.g. sometimes only part of it needs clearing, and also an unrolled clear up to 9*sizeof(size_t). Memory is memory, clearing it 3 bytes at a time isn't going to be faster just because you're then going to use it to hold `struct foo { char a,b,c; };`. `calloc` is always better than `malloc` + `memset`, if you're always going to clear the whole `malloc`ed region. `calloc` has a careful but efficient check for int overflow in size * elements, too. – Peter Cordes Dec 7 '14 at 22:58

`malloc()` allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block.

```
void *malloc(size_t size);
```

`malloc()` doesn't initialize the allocated memory.

`calloc()` allocates the memory and also initializes the allocates memory to all bits zero.

```
void *calloc(size_t num, size_t size);
```

edited Oct 26 '16 at 7:15

chqrlie
50.2k ● 7 ● 40 ● 95

answered Oct 26 '16 at 7:01

geetha
95 ● 2 ● 5

There are two differences.
First, is in the number of arguments. `malloc()` takes a single argument (memory required in bytes), while `calloc()` needs two arguments.
Secondly, `malloc()` does not initialize the memory allocated, while `calloc()` initializes the allocated memory to ZERO.

- `calloc()` allocates a memory area, the length will be the product of its parameters. `calloc` fills the memory with ZERO's and returns a pointer to first byte. If it fails to locate enough space it returns a `NULL` pointer.

Syntax: `ptr_var=(cast_type *)calloc(no_of_blocks , size_of_each_block);` i.e. `ptr_var=(type *)calloc(n,s);`

- `malloc()` allocates a single block of memory of REQUSTED SIZE and returns a pointer to first byte. If it fails to locate requsted amount of memory it returns a null pointer.

Syntax: `ptr_var=(cast_type *)malloc(Size_in_bytes);` The `malloc()` function take one argument, which is the number of bytes to allocate, while the `calloc()` function takes two arguments, one being the number of elements, and the other being the number of bytes to allocate for each of those elements. Also, `calloc()` initializes the allocated space to zeroes, while `malloc()` does not.

answered Sep 14 '12 at 9:58

Jaguar
16.6k ● 23 ● 98 ● 147

Difference 1: malloc() usually allocates the memory block and it is initialized memory segment. calloc() allocates the memory block and initialize all the memory block to 0.

Difference 2: If you consider malloc() syntax, it will take only 1 argument. Consider the following example below:

data_type ptr = (cast_type *)malloc(sizeof(data_type)*no_of_blocks);

Ex: If you want to allocate 10 block of memory for int type,

```
int *ptr = (int *) malloc(sizeof(int) * 10 );
```

If you consider calloc() syntax, it will take 2 arguments. Consider the following example below:

data_type ptr = (cast_type *)calloc(no_of_blocks, (sizeof(data_type)));

Ex: if you want to allocate 10 blocks of memory for int type and Initialize all that to ZERO,

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

Similarity:

Both malloc() and calloc() will return void* by default if they are not type casted .!

edited 2 days ago

Sold Out
458 ● 4 ● 17

answered Jan 12 '15 at 7:52

Shivaraj Bhat
569 ● 6 ● 16

And why do you keep data_type and cast_type different ? – Sold Out 2 days ago

---

The `calloc()` function that is declared in the `<stdlib.h>` header offers a couple of advantages over the `malloc()` function.

1. It allocates memory as a number of elements of a given size, and
2. It initializes the memory that is allocated so that all bits are zero.

edited Aug 25 '13 at 17:38

answered Jul 30 '13 at 16:18

Vipin Diwakar
75 ● 3

A difference not yet mentioned: **size limit**

`void *malloc(size_t size)` can only allocate up to `SIZE_MAX`.

`void *calloc(size_t nmemb, size_t size);` can allocate up about `SIZE_MAX*SIZE_MAX`.

This ability is not often used in many platforms with linear addressing. Such systems limit `calloc()` with `nmemb * size <= SIZE_MAX`.

Consider a type of 512 bytes called `disk_sector` and code wants to use *lots* of sectors. Here, code can only use up to `SIZE_MAX/sizeof disk_sector` sectors.

```
size_t count = SIZE_MAX/sizeof disk_sector;
disk_sector *p = malloc(count * sizeof *p);
```

Consider the following which allows an even larger allocation.

```
size_t count = something_in_the_range(SIZE_MAX/sizeof disk_sector + 1, SIZE_MAX)
disk_sector *p = calloc(count, sizeof *p);
```

Now if such a system can supply such a large allocation is another matter. Most today will not. Yet it has occurred for many years when `SIZE_MAX` was 65535. Given Moore's law, suspect this will be occurring about 2030 with certain memory models with `SIZE_MAX == 4294967295` and memory pools in the 100 of GBytes.

edited Oct 5 '17 at 13:37

answered Aug 29 '15 at 22:12

chux

68.8k ● 7 ● 55 ● 120

1 Generally, size_t will be capable of holding size of the largest kind of object a program could handle. A system where size_t is 32 bits is unlikely to be able to handle an allocation larger than 4294967295 bytes, and a system that would be able to handle allocations that size would almost certainly make `size_t` larger than 32 bits. The only question is whether using `calloc` with values whose product exceeds `SIZE_MAX` could be relied upon to yield zero rather than returning a pointer to a smaller allocation. – supercat Aug 13 '16 at 21:00

Agree about your *generalization*, yet the C spec allows for `calloc()` allocations exceeding `SIZE_MAX`. It has happened in the past with 16-bit `size_t` and as memory continues to cheapen, I see no reason it will cannot happen going forward even if it not *common*. – chux Aug 15 '16 at 1:48

1 The C Standard makes it possible for code to *request* an allocation whose size exceeds `SIZE_MAX`. It certainly does not require that there be any circumstance under which such an allocation might succeed; I'm not sure there's any particular benefit from mandating that implementations that cannot handle such allocations must return `NULL` (especially given that it's common for some implementations to have `malloc` return pointers to space that's not yet committed and might not be available when code actually tries to use it). – supercat Aug 15 '16 at 15:14

Further, where there may have been systems in the past whose available addressing range exceeded the largest representable integer, I do not see any realistic possibility of that ever again occurring, since that would require a storage capacity of billions of gigabytes. Even if Moore's Law continued to hold, going from the point where 32 bits ceases to be enough to the point where 64 bits ceased to be enough would take twice as long as getting from the point where 16 bits was enough to the point where 32 wasn't. – supercat Aug 15 '16 at 15:20

1 Why would an implementation which can accommodate a single allocation in excess of 4G not define `size_t` to `uint64_t`? – supercat Aug 15 '16 at 17:00

malloc(): Allocates requested size of bytes and returns a pointer first byte of allocated space

calloc(): Allocates space for an array elements, initializes to zero and then returns a pointer to memory

answered Jun 7 '14 at 14:47

suresh pareek

90 ● 1 ● 11

The major differences between malloc and calloc are:

1. malloc stands for **memory allocation** whereas calloc stands for **contiguous allocation**.
2. malloc takes only **one argument**, the size of the block whereas calloc takes **two arguments**, number of blocks to be allocated and size of each block.

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

   ptr = (cast-type*)calloc(no of blocks, block-size); // calloc
3. malloc doesn't perform memory initialisation and all the addresses store **garbage value** whereas calloc performs memory initialisation and addresses are initialised to either **Zero or Null values**.

answered Jul 20 '17 at 18:01

Lov Verma

397 ● 3 ● 10

"... *whereas calloc stands for* ***contiguous allocation***." this is arguable. – alk Sep 18 '17 at 12:12

by contiguous allocation, it is meant that the number of blocks which will be allocated dynamically are contiguous in memory. – Lov Verma Sep 20 '17 at 14:52

The name malloc and calloc() are library functions that allocate memory dynamically.
It means that memory is allocated during runtime(execution of the program) from heap segment.

Initialization: malloc() allocates a memory block of given size (in bytes) and returns a pointer to the beginning of the block.

```
>  malloc() doesn't initialize the allocated memory. If we try to access
 the content of memory block then we'll get garbage values. void *
> malloc( size_t size );

> calloc() allocates the memory and also initializes the allocates
 memory block to zero. If we try to access the content of these blocks
 then we'll get 0.

> void * calloc( size_t num, size_t size );
```

> A number of arguments: Unlike malloc(), calloc() takes two arguments: 1) A number of blocks to be allocated. 2) Size of each block.

Most Important :

> **It would be better to use malloc over calloc, unless we want the zero-initialization because malloc is faster than calloc. So if we just want to copy some stuff or do something that doesn't require filling of the blocks with zeros, then malloc would be a better choice.**

answered Aug 24 '17 at 11:37

jsroyal
395 ● 5 ● 17

```
char *ptr = (char *) malloc (n * sizeof(char));
```

just allocates `n bytes` of memory without any initialization (ie; those memory bytes will contain any garbage values).

However, `calloc()` method in c does the initialization to `0` for all the occupied memory bytes in addition to the function that `malloc()` does.

edited Sep 18 '17 at 17:11

answered May 14 '15 at 7:57

Rahul Raina
1,375 ● 10 ● 19

The malloc() takes a single argument, while calloc() takess two.

Second, malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO. Both malloc and calloc are used in C language for dynamic memory allocation they obtain blocks of memory dynamically.

edited Oct 6 '17 at 10:29

answered Aug 24 '17 at 11:51

Sanjay Kumaar
268 ● 2 ● 11

---

protected by Community ♦ Nov 6 '16 at 16:15

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?