Is accessing data in the heap faster than from the stack?

Asked 10 years, 6 months ago Modified 2 months ago Viewed 92k times

115

I know this sounds like a general question and I've seen many similar questions (both here and on the web) but none of them are really like my dilemma.

Say I have this code:

```
void GetSomeData(char* buffer)
{
    // put some data in buffer
}

int main()
{
     char buffer[1024];
     while(1)
     {
          GetSomeData(buffer);
          // do something with the data
     }
     return 0;
}
```

Would I gain any performance if I declared buffer[1024] globally?

I ran some tests on unix via the time command and there are virtually no differences between the execution times.

But I'm not really convinced...

In theory should this change make a difference?

c++ c performance heap-memory stack-memory

Share

Improve this question

Follow

edited Jun 15, 2021 at 10:14

trincot

348k ● 35 ● 270 ● 320

asked Jun 5, 2014 at 10:09

conectionist

2,904 ● 6 ● 31 ● 51

9 It is irrelevant for the speed where the memory you access is (unless we talk about stuff like NUMA), but through how many indirections you access it. – PlasmaHH Jun 5, 2014 at 10:11

1 Accessing from heap is slightly slower as i know. However you shouldn't think about that. You should allocate everything in the stack by default, unless you need the data on the heap. – Melkon Jun 5, 2014 at 10:12

3 Accessing from heap is slightly slower cause of the indirection, look at @PlasmaHH comment. There's no difference between stack and heap memory, they are both somewhere in RAM. – Dimitrios Bouzas Jun 5, 2014 at 10:16

13 This should *not* be marked as a duplicate of a question about *allocation* performance when it is about *access* performance. – Felix Glas Jun 5, 2014 at 11:06

1 I've been surprised to find that sometime stack allocation performs substantially worse than heap allocation - especially in MT programs. I believe it may have to do with different access a patterns sometimes causing more cache misses. – Rafael Baptista Dec 4, 2015 at 16:42

8 Answers

Sorted by: Highest score (default)

139

> Is accessing data in the heap faster than from the stack?

Not inherently... on every architecture I've ever worked on, all the process "memory" can be expected to operate at the same set of speeds, based on which level of CPU cache / RAM / swap file is holding the current data, and any hardware-level synchronisation delays that operations on that memory may trigger to make it visible to other processes, incorporate other processes'/CPU (core)'s changes etc.. (With multi-CPU-socket motherboards using Non-Uniform Memory Architecture (NUMA), the time for one CPU to access memory that's "closer" to the other CPU tends to differ though, but that's a bit outside the scope of this question.)

The OS (which is responsible for page faulting / swapping), and the hardware (CPU) trapping on accesses to not-yet-accessed or swapped-out pages, would not even be tracking which pages are "global" vs "stack" vs "heap"... a memory page is a memory page.

While the global vs stack vs heap usage to which memory is put is unknown to the OS and hardware, and all are backed by the same type of memory with the same performance characteristics, there are other subtle considerations (described in detail after this list):

- ***allocation*** - time the program spends "allocating" and "deallocating" memory, including occasional `sbrk` (or similar) virtual address allocation as the heap usage grows
- ***access*** - differences in the CPU instructions used by the program to access globals vs stack vs heap, and extra ***indirection*** via a runtime pointer when using heap-based data,

- ***layout*** - certain data structures ("containers" / "collections") are more cache-friendly (hence faster), while general purpose implementations of some require heap allocations and may be less cache friendly.

Allocation and deallocation

For **global data** (including C++ namespace data members), the virtual address will typically be calculated and hardcoded at **compile time** (possibly in absolute terms, or as an offset from a segment register; occasionally it may need tweaking as the process is loaded by the OS).

For **stack**-based data, the stack-pointer-register-relative address can also be calculated and hardcoded at **compile time**. Then the stack-pointer-register may be adjusted by the total size of function arguments, local variables, return addresses and saved CPU registers as the function is entered and returns (i.e. at runtime). Adding more stack-based variables will just change the total size used to adjust the stack-pointer-register, rather than having an increasingly detrimental effect.

Both of the above are effectively free of runtime allocation/deallocation overhead, while heap based overheads are very real and may be significant for some applications...

For **heap**-based data, a **runtime** heap allocation library must consult and update its internal data structures to track which parts of the block(s) aka pool(s) of heap memory it manages are associated with specific pointers the library has provided to the application, until the application frees or deletes the memory. If there is insufficient virtual address space for heap memory, it may need to call an OS function like `sbrk` to request more memory (Linux may also call `mmap` to create backing memory for large memory requests, then unmap that memory on `free`/`delete`).

Access

Because the absolute virtual address, or a segment- or stack-pointer-register-relative address can be calculated at compile time for global and stack based data, runtime access is very fast.

With heap hosted data, the program has to access the data via a runtime-determined pointer holding the virtual memory address on the heap, sometimes with an offset from the pointer to a specific data member applied at runtime. That may take a little longer on some architectures.

For the heap access, both the pointer and the heap memory must be in registers for the data to be accessible (so there's more demand on CPU caches, and at scale - more cache misses/faulting overheads).

Note: these costs are often insignificant - not even worth a look or second thought unless you're writing something where latency or throughput are enormously important.

Layout

If successive lines of your source code list global variables, they'll be arranged in adjacent memory locations (albeit with possible padding for alignment purposes). The same is true for stack-based variables listed in the same function. This is great: if you have X bytes of data, you might well find that - for N-byte cache lines - they're packed nicely into memory that can be accessed using X/N or X/N + 1 cache lines. It's quite likely that the other nearby stack content - function arguments, return addresses etc. will be needed by your program around the same time, so the caching is very efficient.

When you use heap based memory, successive calls to the heap allocation library can easily return pointers to memory in different cache lines, especially if the allocation size differs a fair bit (e.g. a three byte allocation followed by a 13 byte allocation) or if there's already been a lot of allocation and deallocation (causing "fragmentation"). This means when you go to access a bunch of small heap-allocated memory, at worst you may need to fault in as many cache lines (in addition to needing to load the memory containing your pointers to the heap). The heap-allocated memory won't share cache lines with your stack-allocated data - no synergies there.

Additionally, the C++ Standard Library doesn't provide more complex data structures - like linked lists, balanced binary trees or hash tables - designed for use in stack-based memory. So, when using the stack programmers tend to do what they can with arrays, which are contiguous in memory, even if it means a little brute-force searching. The cache-efficiency may well make this better overall than heap based data containers where the elements are spread across more cache lines. Of course, stack usage doesn't scale to large numbers of elements, and - without at least a backup option of using heap - creates programs that stop working if given more data to process than expected.

Discussion of your example program

In your example you're contrasting a global variable with a function-local (stack/automatic) variable... there's no heap involved. Heap memory comes from `new` or `malloc`/`realloc`. For heap memory, the performance issue worth noting is that the application itself is keeping track of how much memory is in use at which addresses - the records of all that take some time to update as pointers to memory are handed out by `new`/`malloc`/`realloc`, and some more time to update as the pointers are `delete`d or `free`d.

For global variables, the allocation of memory may effectively be done at compile time, while for stack based variables there's normally a stack pointer that's incremented by the compile-time-calculated sum of the sizes of local variables (and some housekeeping data) each time a function is called. So, when `main()` is called there may be some time to modify the stack pointer, but it's probably just being modified by a different amount rather than not modified if there's no `buffer` and modified if there is, so there's no difference in runtime performance at all.

Note

I omit some boring and largely irrelevant details above. For example, some CPUs use "windows" of registers to save the state of one function as they enter a call to another function; some function state will be saved in registers rather than on the stack; some function arguments will be passed in registers rather than on the stack; not all Operating Systems use virtual addressing; some non-PC-grade hardware may have more complex memory architecture with different implications....

Share

Improve this answer

Follow

edited Jan 30, 2023 at 0:01

answered Jun 5, 2014 at 10:28

Tony Delroy

106k ● 16 ● 184 ● 263

3 Re your first sentence: I started to write the same thing, but as you point out in what follows, it *isn't* true; what is true (on most processors today) is that the speed doesn't depend on where the memory is located, per se, but rather on what was accessed previously. – James Kanze Jun 5, 2014 at 10:36

@JamesKanze "it isn't true" - well, depends on perspective - it's true that a cache miss is slower than a cached hit (at whatever level of caching), and that the same stepped performance profile applies regardless of the globals+statics/stack/heap/thread-specificity/sharing/ etc. use to which the memory may be being put by the application... that's my intended point, though I agree it could be worded better and will have a crack at it. – Tony Delroy Jun 5, 2014 at 10:43

1 @Tony D: could you clarify my confusion? So stack is roughly the same as fast as heap by accessing (writing/loading), but it should be faster in terms of allocation because it's already done at compiler time which doesn't add much overhead to running? Thanks – dragonxlwang Aug 5, 2015 at 19:08

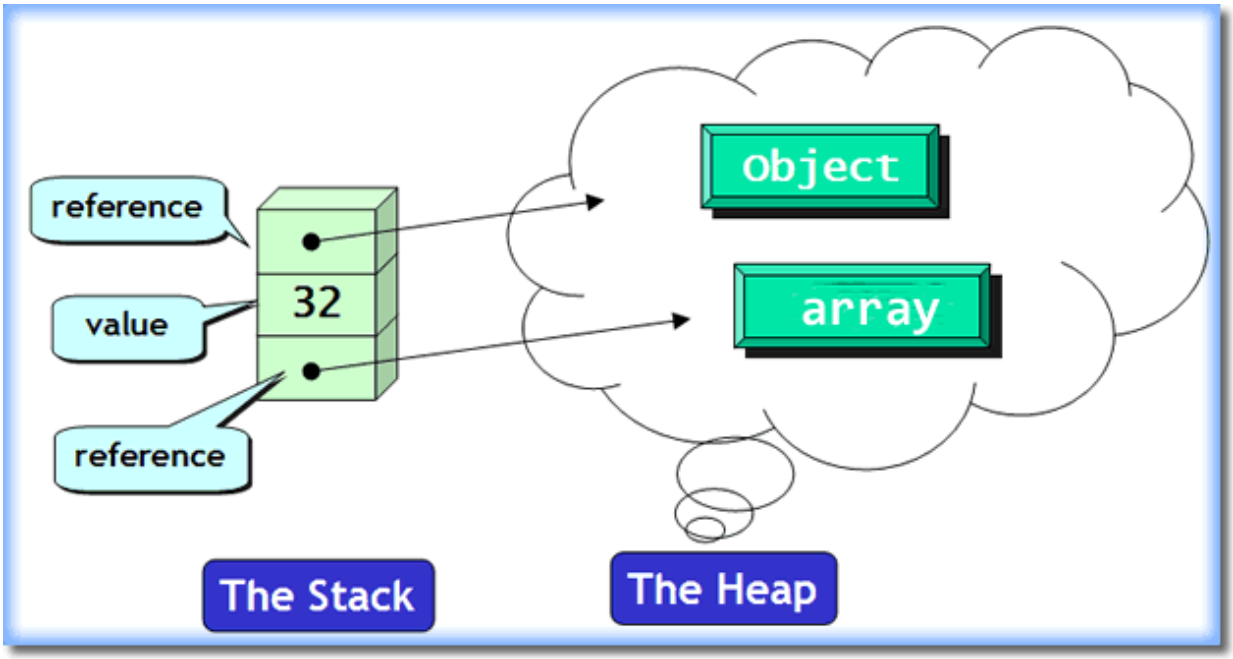
2 @dragonxlwang: that's about the size of it, yes. Cheers. – Tony Delroy Aug 6, 2015 at 3:39

7 This is such an excellent and thorough answer. Thank you so much. It really cleared up a lot of points of confusion I had around why the Stack and Heap have different performance characteristics despite both being allocated in RAM. In particular, the fact that stack pointers can be figured out at compile time was a huge insight! – Nicholas Montaño Feb 15, 2021 at 4:33

Quoting from Jeff Hill's answer:

52

> **The stack is faster** because the access pattern makes it trivial to allocate and deallocate memory from it (a pointer/integer is simply incremented or decremented), while the heap has much more complex bookkeeping involved in an allocation or free. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast. Another performance hit for the heap is that the heap, being mostly a global resource, typically has to be multi-threading safe, i.e. each allocation and deallocation needs to be - typically - synchronized with "all" other heap accesses in the program.



Share

Improve this answer

Follow

edited May 23, 2017 at 12:34

Community Bot

1 • 1

answered Jun 5, 2014 at 10:33

haccks

106k • 27 • 179 • 271

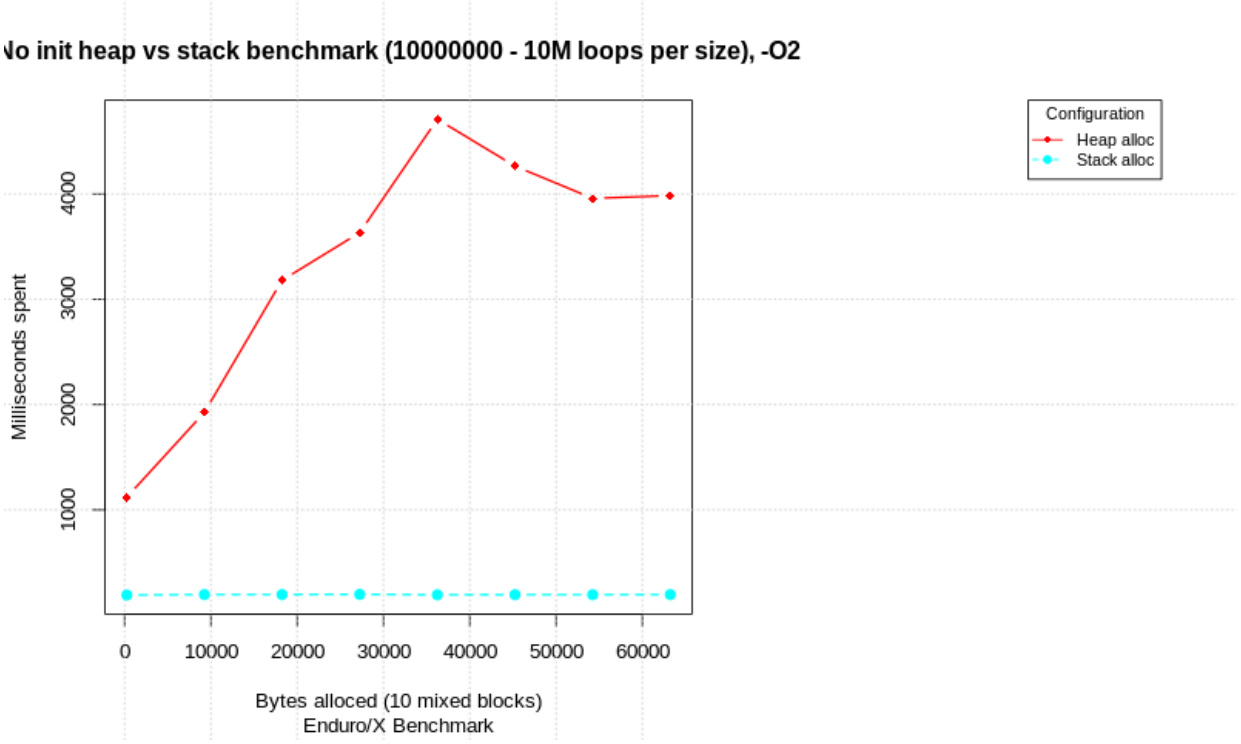
30 "Is accessing data in the heap faster than from the stack?" is the question, Your emphasis is actually wrong, if you have the same data with the same access pattern, then theoretically the heap should be just as fast as the stack. If your data is an array, *accesses* should take the same amount of time as long as the data is contiguous. The stack will have faster times if you have several small bits of data that are everywhere in ram. – Krupip Jul 12, 2017 at 20:18

18

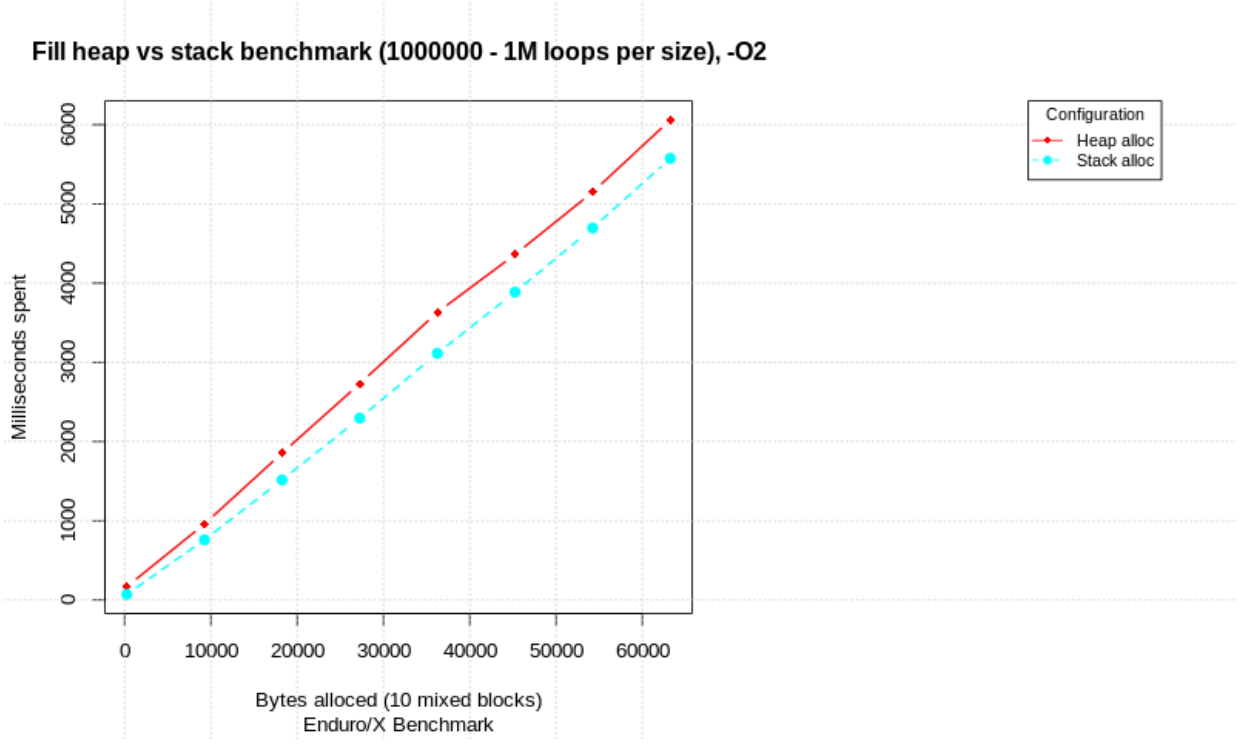
There is blog post available on this topic stack-allocation-vs-heap-allocation-performance-benchmark Which shows the allocation strategies benchmark. Test is written in C and performs compare between pure allocation attempts, and allocation with memory init. At different total data sizes, number of loops are performed and time is measured. Each allocation consists of 10 different alloc/init/free blocks with different sizes (total size shown in charts).

Test are run on Intel(R) Core(TM) i7-6600U CPU, Linux 64 bit, 4.15.0-50-generic, Spectre and Meltdown patches disabled.

With out init:



With init:



In the result we see that there is significant difference in pure allocations with out data init. The stack is faster than heap, but take a note that loop count is ultra high.

When allocated data is being processed, the gap between stack & heap performance seems to reduce. At 1M malloc/init/free (or stack alloc) loops with 10 allocation attempts at each loop, stack is only 8% ahead of heap in terms of total time.

Share

Improve this answer

Follow

edited Jul 15, 2019 at 18:54

gaganso

2,991 ● 2 ● 27 ● 45

answered Jun 27, 2019 at 21:23

Madars Vi

1,007 ● 11 ● 12

Important note: If allocations and frees are done in the right order, a lot of the malloc/heap is essentially operating like a stack with only a little overhead. If allocations are semi-random, the bookkeeping costs significantly increase. – mczarnek Aug 4, 2022 at 19:34

10

For what it's worth, the loop in the code below - which just reads from and writes to each element in a big array - consistently runs 5x faster on my machine when the array is on the stack vs when it's on the heap (GCC, Windows 10, -O3 flag), even right after a reboot (when heap fragmentation is minimized):

```
const int size = 100100100;
int vals[size]; // STACK
// int *vals = new int[size]; // HEAP
startTimer();
for (int i = 1; i < size; ++i) {
    vals[i] = vals[i - 1];
}
stopTimer();
std::cout << vals[size - 1];
// delete[] vals; // HEAP
```

Of course, I first had to increase the stack size to 400 MB. Note that the printing of the last element at the end is needed to keep the compiler from optimizing everything away.

Share

Improve this answer

Follow

edited Dec 21, 2021 at 9:41

answered May 5, 2019 at 8:55

Gumby The Green

633 ● 7 ● 12

How can we increase the stack size? – Paiman Roointan Nov 24, 2020 at 15:37

2 @PaimanRoointan Under linux, you can use `ulimit -s` – Li Chen Apr 11, 2021 at 7:47

1 interestingly, on my machine the stack version takes about 50% MORE time: 132312[µs] vs. 93081[µs]. With -Ofast the stack takes the lead sometimes, but only by a meaningless advantage, statistically speaking: e.g., 41276[µs] vs. 42756[µs]. Other times the heap is faster. – John Perry Sep 10 at 15:59

8

Your question doesn't really have an answer; it depends on what else you are doing. Generally speaking, most machines use the same "memory" structure over the entire process, so regardless of where (heap, stack or global memory) the variable resides, access time will be identical. On the other hand, most modern machines have a

hierarchial memory structure, with a memory pipeline, several levels of cache, main memory, and virtual memory. Depending on what has gone on previously on the processor, the actual access may be to any one of these (regardless of whether it is heap, stack or global), and the access times here vary enormously, from a single clock if the memory is in the right place in the pipeline, to something around 10 milliseconds if the system has to go to virtual memory on disk.

In all cases, the key is locality. If an access is "near" a previous access, you greatly improve the chance of finding it in one of the faster locations: cache, for example. In this regard, putting smaller objects on the stack may be faster, because when you access the arguments of a function, you're access on stack memory (with an Intel 32-bit processor, at least---with better designed processors, arguments are more likely to be in registers). But this will probably not be an issue when an array is involved.

Share Improve this answer Follow

answered Jun 5, 2014 at 10:32

James Kanze

154k ● 18 ● 189 ● 336

@IceCold No, because then you'd get a comparison of totally irrelevant speeds. Computers use caches, and without them they'd be really slow. – relatively Dec 5 at 18:21

6

when allocating buffers on stack the optimization scope is not the cost of accessing the memory but rather the elimination of often very expensive dynamic memory allocation on the heap (stack buffer allocation can be considered instantaneous as the stack as a whole is allocated at thread startup).

Share Improve this answer Follow

answered Jun 5, 2014 at 10:26

bobah

18.9k ● 2 ● 41 ● 74

0

It depends on how you store a pointer.

If it's on the stack, then you first need to dereference it which means you need to perform mov operation from memory to any free general-purpose register. And after that, you do the same mov but this time you read the actual value.

If it's in a register (for example, if you pass a pointer to a function, a compiler might usually (in fact, most of the times) optimizes that and stores the argument in a register), there's no difference at all

Share Improve this answer Follow

answered Sep 22 at 18:37

Heil Programmierung
187 ● 3 ● 12

-2

Giving that variables and variable arrays that are declared on the heap is slower is just a fact. Think about it this way;

Globally created variables are allocated once and deallocated once the program is closing. For a heap object your variable has to be allocated on the spot each time the function is ran, and deallocated in the end of the function..

Ever tried allocating an object pointer within a function? Well better free / delete it before the function exits, orelse you will have yourself a memoryleak giving that you are not doing this in a class object where it is free'd/deleted inside the deconstructor.

When it comes to accessing of an array they all work the same, a memory block is first allocated by sizeof(DataType) * elements. Later can be accessed by ->

```
1 2 3 4 5 6
^ entry point [0]
      ^ entry point [0]+3
```

Share Improve this answer Follow

answered Jun 5, 2014 at 10:17

SuperAgenten Johannes Schaeder
45 ● 5

heap and stack allocation are completely different beasts. stack allocation is practically free, so it doesn't matter how many times you have to do it. – Karoly Horvath Jun 5, 2014 at 10:21

1 downvoted 3 times but nobody explained what is wrong with this answer. so +1 from me. – IceCold Apr 26, 2019 at 8:07

2 *"For a heap object your variable has to be allocated on the spot each time the function is ran, and deallocated in the end of the function.."* - this just isn't true (you didn't mean "stack object" there?) Heap allocations return a pointer that can be used at any later time for deallocation - the allocation and deallocation code doesn't have any necessary connection with entry or exit from a single function; they can even happen outside all functions (i.e. a global pointer can track heap memory). Functions can return pointers for the caller to manage. Lots of misinformation here. – Tony Delroy Jun 9, 2022 at 17:27