**[What is the difference of run-time stack, free storage (heap), static storage and code storage in C++?](https://www.quora.com/What-is-the-difference-of-run-time-stack-free-storage-heap-static-storage-and-code-storage-in-C++" \t "_blank)**

Originally Answered: What is the difference of run-time stac, free storage (heap), static storage and code storage in C++?

**Code Storage:**

1. It is also known as the *text section*. All the code that you write in your .cpp is stored here in the form of executable instructions.
2. The text segment is *sharable*and placed in memory most of the time.
3. It is *read-only* so that no program could modify its instructions. It is placed below stack/heap so that their overflow could not overwrite it.

**Static Storage:**

1. It is divided into two parts:
   1. *Initialized Data Segment:* It contains the *global and static variables*that are initialized explicitly by the programmer.
   2. *Uninitialized Data segment:* It is popularly known as *bss*(block started by symbol) segment. It contains all the *global and static variables* that are *not initialized* or are initialized to 0.
2. Static and global variables are program specific.

**Heap(Free Store):**

1. The memory which is allocated dynamically at run-time is assigned address in heap.
2. It starts from the end of BSS and grows up towards the stack.
3. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size. In C++, memory is assigned dynamically by the use of new operator and dereferenced using delete operator.
4. new is an operator while malloc is a function.

**Stack:**

1. Heap and stack grow in opposite directions. Free memory exhausts when the heap pointer meets the stack pointer.
2. A stack pointer register tracks the top of the stack. It is adjusted each time a value is pushed onto the stack. The set of values pushed for one function call is termed a stack frame. A stack frame consists at minimum of a return address.
3. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables.
4. This is how recursive functions in C++ can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

**[Related](https://www.quora.com/Is-stack-memory-faster-than-heap?no_redirect=1" \t "_blank)**

**[Is stack memory faster than heap?](https://www.quora.com/Is-stack-memory-faster-than-heap?no_redirect=1" \t "_blank)**

I’m really *stunned* to see that all the other (so far 4) answers are totally **wrong by omission**: they muse about memory caching and such… but do not even mention the *real huge elephant* in the room, namely the memory manager and the **heap allocator**!

The right answer is: **stack is *catastrophically* faster** than heap! Here’s why:

When you allocate a new memory block on the heap (with *malloc/calloc/realloc* or *new* in C/C++), here is what *really happens*:

* The OS kernel is asked whether it actually has that space available (yep: you have to ask the OS, a huge expense!!)…. for *all programs* it runs at that moment. Why? Because you can easily ask for a block of 32Gigs (in a 64bit OS) even if your machine only has 8Gigs of physical RAM!
* If the answer is yes, then the OS needs to reserve that block in the RAM for you… but that may take a *very long time* if your machine is heavily loaded, as it may involve *virtual memory* — in which case
  + The OS must go over all the RAM used by all the programs it runs at that moment, and decide that some of it may be reasonably removed from the RAM to make place for what you’re asking for… and put it **on the disk**(!) instead
  + Needless to say, it must then *physically write it on the disk* in order not to lose the contents of it, before it can return you that freed RAM block
* With the resulting memory block to be given to you, there’s still much to do, as the OS must
  + Possibly (and quite often, actually!) reorganize the pieces of free space to “coagulate” them into a *contiguous block* not smaller than the size you’re asking for
  + Reserve that block for you, which means marking it in use
* Conversely, when you *free* a block allocated on the heap (with *free/delete/delete[]* in C/C++), it must mark that block as being again available for reuse
* If your program is heavily multi-threaded (as are mine), then spare for the typically tiny TLS (Thread Local Storage), allocating on the heap must be atomic (because there’s *just one RAM*, whatever number of threads you may be running), which means putting all other threads that may want to allocate at the same time *on hold* until all the jazz above is done.

And now you compare all of the above *huge stuff* with a single thing: just incrementing the stack pointer!!!!

How come? Simple: the stack is allocated *at the start* of the run of your program. There is **no call to a memory manager** that would call the heap allocator — the stack pointer (a super-fast CPU register!) is simply incremented, in a **single CPU cycle**.

Ah… and what if you run out of that stack space? Simple: your program will “simply” crash with stack overflow — that’s why you should never use large arrays as local variables (which are all put on the stack): they may well cause a stack overflow.

Stack is *very slightly* riskier (only if you’re not careful!)… but is *catastrophically faster* than the heap.

UPDATE 17-Oct-21:

Thank you all for your appreciation. But apologies, for the first time in my several years on Quora, I’m disabling further Comments on this Answer — the simple reason being that they keep repeating themselves over and over again, and I’m just tired of debunking them one by one with the same arguments… over and over again.  
I have no idea why people eager to comment don’t make the obvious effort of at least a cursory look at the (already addressed) previous comments.

All in all, I come to the conclusion that the few negative (and duly debunked!) Comments are due to their authors’ unwillingness (or incapacity?) to **think things through**down to **the detail** — where the proverbial Devil always is.

Most of them fall in the following categories — which I summarize below with my earlier replies, so you don’t need to go through the many Comments yourself:

* **Claim**: *Both* the stack and the heap are subject to RAM **paging**, so they are necessarily *equally* fast.  
  **Refutation**: Well, this is a actually a **non-argument**, because all it shows is that paging affects both stack and heap to the same extent — which means paging can be simply ignored in comparing their *relative speed*. What it does *not* show is whether there are *other factors* in play — and there  
  That pseudo-argument is akin to claiming that a crummy Chevy must necessarily be as fast as a F1 car, because they *both* run on gasoline. Smart, heh?
* **Claim**: Memory managers typically *don’t call the kernel*: RAM is allocated in **big chunks** from the kernel, and is dispensed from there without a need to a kernel call.  
  **Refutation**: Question is (as per the OP’s *actual Question*): **is it as fast** as incrementing the stack pointer in a *single CPU cycle*?  
  You seem to believe that “when you call malloc for 100 bytes, it first checks if it has 100 bytes free in the memory allocated by the kernel” is basically *for free*. Oh, really? Please do think it through *to the detail*: what would that entail, that “checking”?  
  If it is simply checking that the last allocation still left at least 100 bytes *at the end* in whatever chunk was allocated, then it would be pretty fast indeed. But, lo and behold, it would *also* ignore all the space that was *freed* in the meantime, leading to a horrendous misuse of memory, leaving those chunks in a Gruyère state (i.e. full of holes).  
  So it turns out it’s a bit more complicated: a list of holes (left by freeing — because *you can free stuff* allocated on the heap in whatever order!!) needs to be kept up to date, to see whether any of them can be *reused* in the new allocation without the expensive call for a new chunk.  
  Dunno whether you’ve ever done such bookkeeping, but let me assure you it’s not trivial (insofar the allocated bits are of arbitrary sizes… as they *can* be on the heap)… and **it *is* expensive**. Do you think you could manage as fast as a *single CPU cycle*, like the stack counter does? Yep, didn’t think so - yet *that* was the OP’s *actual Question*.  
  And of course, as soon as the manager runs out of the chunks, it **does** need to call the kernel to request more.
* **Claim**: Physical memory **doesn’t need to be contiguous** (that’s a good one, IMHO).  
  **Refutation**: Oh, really? Given that a page is typically a *measly 4k,* when I allocate a Meg, it looks like the OS **does** need to find 256 *contiguous* pages. Granted, it’s a different granularity from bytes — but it’s the same jazz of keeping track of used/reserved pieces (pages) of RAM, and *reorganizing*when necessary.  
  Unless, of course, you have a magical way of hopping from one page to another *non-contiguous* one (e.g. in a *memcpy* or *memset* across the chunk boundaries) — with exactly zero overhead. Ah, you don’t… I thought so.  
  Please do realize that in 32GBs of RAM (a mundane amount these days), there are about *8 million pages* of 4k each — and the OS needs to keep track of availability of each of them, and “coagulate” them if necessary into contiguous chunks large enough for what was requested. That does cost *a lot of time* — especially when compared to a *single CPU cycle* on stack.
* **Claim**: Many allocators **support multi-threading**. Even standard glibc malloc supports it, so you can allocate from multiple threads at the same time.  
  **Refutation:** Well, that’s just **plain false**, except for the tiny TLS which I did mention. You should have a look at my Comments exchange with [Jonas Oberhauser](https://www.quora.com/profile/Jonas-Oberhauser), where I report a *very steep* gain in efficiency just by eliminating heap fiddling — involving *very modest* memory sizes — in a heavily multi-threaded algorithm.  
  Just to assure you: that one was built with GNU’s g++.

Short answer: no; in most OS and architectures, both heap and stack are allocated from RAM. There are exceptions at the low and high ends of the computer processor spectrum.

**[Related](https://www.quora.com/Why-are-heaps-used-for-memory-allocation-Why-arent-stacks-or-any-others-used" \t "_blank)**

**[Why are heaps used for memory allocation? Why aren't stacks or any others used?](https://www.quora.com/Why-are-heaps-used-for-memory-allocation-Why-arent-stacks-or-any-others-used" \t "_blank)**

Originally Answered: Why heaps used for memory allocation, why not stack or any other?

You're getting a bit confused. The portion of memory used for dynamic memory allocation is just called a "heap". It has no relation to the data structure heap. It's just a coincidence they're called the same thing.

**[Related](https://www.quora.com/Are-classes-and-structures-in-C-on-the-stack-or-in-the-heap" \t "_blank)**

**[Are classes and structures in C++ on the stack or in the heap?](https://www.quora.com/Are-classes-and-structures-in-C-on-the-stack-or-in-the-heap" \t "_blank)**

It depends. If you allocate them inside a function as local variables, for example

1. void somefunction(){
2. MyClass foo;
3. …
4. }

they’ll be allocated on the stack. If you allocate them as global variables, such as

1. MyClass foo;
2. void somefunction(){
3. }

they’ll be in the static area - neither the heap, nor the stack, but an area of fixed size allocated for this purpose when the program is loaded. If you allocate them inside a function using new, like this:

1. void somefunction(){
2. MyClass \*foo = new MyClass();
3. …
4. delete foo;
5. }

they’ll be on the heap.

**[Related](https://www.quora.com/Is-the-stack-faster-than-the-heap-If-so-why-Why-isn%E2%80%99t-memory-allocation-needed-for-stack-Why-can%E2%80%99t-a-compiler-know-where-an-object-is-in-heap" \t "_blank)**

**[Is the stack faster than the heap? If so, why? Why isn’t memory allocation needed for stack? Why can’t a compiler know where an object is in heap?](https://www.quora.com/Is-the-stack-faster-than-the-heap-If-so-why-Why-isn%E2%80%99t-memory-allocation-needed-for-stack-Why-can%E2%80%99t-a-compiler-know-where-an-object-is-in-heap" \t "_blank)**

Originally Answered: Is the stack faster than the heap? If so, why?

The main reason is that the stack fits the local CPU cache.

Imagine declaring an int i; variable; say, in a simple for-loop, such as for (int i = 0; i < n; ++i) {}.

Even a hardcore Java person would realize that by allocating i "externally" (effectively, by means of making some Int i = new Int(); call), you're wasting resources.

Now, say you have a class that is a few -- or even just one -- integer large. Why on Earth would you want to allocate it using new, if there's an easy way?

This easy way is possible -- and widely encouraged -- in C++ because instead of reference-counting-based garbage collection C++ uses scope-based object lifetime management. The compiler *knows* that as the current scope ends, objects automatically allocated in it should be disposed. Thus, the memory layout for locally allocated objects *conveniently* matches how the stack works for nested function calls.

This convenience is, of course, not a coincidence. It's the philosophy of how C++ deals with object lifetimes.

Downside: When the "flexibility" of object lifetime management is needed, plain auto-allocated objects won't work -- since they are allocated on stack, they will no longer be valid as the called function leaves its scope. Thus, the programmer has to explicitly declare those objects "persistent" on larger scale -- most often, by using smart pointers, although effectively this is the field where bare pointers and raw new/delete become necessary.

(This, by the way, is one common pitfall of C++: passing a pointer/reference to a stack-allocated object into an async call without ensuring that the stack frame allocated it will live long enough. Futures and promises FTW in this case.)

Upside: Performance, engineering beauty, and remarkable predictability of object ownership management.

As for memory allocation itself, it's a relatively minor gain in performance compared to data locality for better caching. It's true that memory itself does not have to be "allocated" for stack-allocated objects, however, user-defined constructors and destructors for stack-allocated objects, of course, do get called.

At the same time, memory fragmentation, that is such a pain for garbage-collected languages, is a problem that is completely nonexistent for stack-based allocation and lifetime management.

Oh, and did I mention this scope-based allocation renders the finally construct absolutely unnecessary? Well, it does.

**[Related](https://www.quora.com/Is-the-call-stack-in-RAM-Wheres-the-heap-free-store" \t "_blank)**

**[Is the call-stack in RAM? Where's the heap (free store)?](https://www.quora.com/Is-the-call-stack-in-RAM-Wheres-the-heap-free-store" \t "_blank)**

Originally Answered: Is the stack in RAM? Where's the heap?

* stack and heap are just structures made in memory (ram) for proper organisation of data.
* heaps are generally allocated once for each application while stacks are allocated for each and every thread of the application.
* heaps are used for storing global variables and thread safe data which belongs to the entire application while stacks are used for dealing with functions and local variables delcared within the functions and for threads
* Hence essentially they are just a collection of memory addresses with certain rules regarding how to access and manipulate them.
* hence while you have a LIFO(last in first out ) policy for stacks which have predetermined fixed size, heaps are allocated dynamically as and when needed by the OS. new memory is freed /allocated by function calls like free() or malloc();
* stacks are way faster than heaps cause stack pointers are generally stored in registers by the compilers . also for accesing elements off the stack u simply need to increment/decrement the stack pointer while in case of heap u have complex and diverse traversal mechanisms like linked lists for storing free and used blocks which again further leads to problems of fragmentation etc.

A picture containing chart

Description automatically generated

* [code]int foo()  
  { char \*pBuffer; //<--nothing allocated yet (excluding the pointer itself, which is allocated here on the stack).
* bool b = true; // Allocated on the stack.
* if(b)
* { char buffer[500]; //Create 500 bytes on the stack
* pBuffer = new char[500]; //Create 500 bytes on the heap
* }
* //<-- buffer is deallocated here, pBuffer is not}
* //<--- o
* [/code]

**[Related](https://www.quora.com/Is-heap-memory-part-of-RAM" \t "_blank)**

**[Is heap memory part of RAM?](https://www.quora.com/Is-heap-memory-part-of-RAM" \t "_blank)**

Yes, the entire anatomy of a running process exists in RAM (except for the instructions running on the CPU, the variables in CPU registers and the cache, etc.). The stack is also in RAM, and one of the most important behaviors of the heap and stack are that they grow towards one another as they get larger. Hence, if you use too much stack memory (perhaps by excessive recursion), the stack will actually collide with the address space of the heap, resulting in a

1. Exception: stack overflow

or something similar. In C++ this would be a segfault.

In reality, the heap, stack, program data, etc. may not actually exist near one another on the physical RAM though. Memory addresses in a program are virtual, and are mapped to physical addresses in memory by the OS. In the following picture, the **data** segment refers to the heap.

Diagram

Description automatically generated

**[Related](https://www.quora.com/C-memory-is-organized-into-the-stack-and-heap-Do-different-programming-languages-organize-memory-differently-or-are-they-all-the-same" \t "_blank)**

**[C++ memory is organized into the stack and heap. Do different programming languages organize memory differently, or are they all the same?](https://www.quora.com/C-memory-is-organized-into-the-stack-and-heap-Do-different-programming-languages-organize-memory-differently-or-are-they-all-the-same" \t "_blank)**

The terminology of stack and heap predates C++ by more than 10 years and C++ standard doesn't use these terms but automatic and dynamic allocation. It's by convention that compilers put automatic variables on the stack, and by using malloc() call dynamic allocations by default on system or tasks’ (threads’ or processes’) heaps.

In general all natively compilable languages use such model under the hood. Most dynamic languages I know don't distinguish where variables are kept. Languages used in JVM, including Java itself, use only dynamic allocation for every variable, and memory maintenance is purely job of JVM where freeing is handled by highly tunable garbage collectior. So running instance of JVM grabs huge slice of system memory, and usually doesn't free it until the application runs.

In modern computing there can be more stacks even for a single program and heap is usually something purely virtual, including memory swapped into disk.

**[Related](https://www.quora.com/Are-the-concepts-of-stack-and-heap-only-applicable-to-Java-or-to-other-languages-as-well" \t "_blank)**

**[Are the concepts of stack and heap only applicable to Java or to other languages as well?](https://www.quora.com/Are-the-concepts-of-stack-and-heap-only-applicable-to-Java-or-to-other-languages-as-well" \t "_blank)**

Most programming languages treat memory as four different regions:

1. Code: The app (machine code) runs from here
2. Data: Globals, statics, constants (BSS) (lifetime=run-time)
3. Heap: Most structs/objects (lifetime mn’gd by programmer or GC)
4. Stack: Call hierarchy of locals, args, and return addresses (LIFO)

Of course the usage of these areas varies between languages. For instance, in Java ALL objects reside on the heap. In C++ the programmer may place objects on the heap, on the stack or in global/static data. In Javascript, locals and parameters are not placed on the stack but, instead, on the heap. The same is true for Lambdas in all languages. In Haskell and other functional languages almost all of the data resides on the heap until the underlying execution engine evaluates it, primarily using stack.

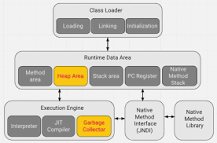
Various languages manage the heap in different ways. In Java, the programmer allocates any needed objects in heap space, but the run-time frees them when there are no longer any references, via garbage collection. In Objective-C, reference-counting is used but it is not automatic - it is up to the programmer to leverage the reference counts or to use compile-time auto-generated code to do so. C heap management is completely manual - all done by the programmer. C++ heap management can be all manual, like C, but most often is handled by using various kinds of auto-pointers that use references on the stack to control the lifetime of objects on the heap.

If one gets familiar with assembly language it is easy to see why these four areas make sense. And it is clear when looking at the architecture of a CPU that the hardware was designed to leverage these four areas.

**TL;DR;**

The underlying approach is indeed very similar across languages, but the way that is presented to and managed by the programmer varies.

What is off-heap memory?



Off-heap memory refers to the **memory allocated directly to the operative system**, it can be part of the same physical memory or/and disk access based such as memory mapped-files.Apr 8, 2021

**[Related](https://www.quora.com/What-is-off-heap-memory-storage" \t "_blank)**

**[What is off-heap memory storage?](https://www.quora.com/What-is-off-heap-memory-storage" \t "_blank)**

Originally Answered: what is off-heap memory storage?

Off-heap memory is the use of memory outside of a garbage collected heap in Java, Scala, Python, or other languages. This is commonly used to for native interfaces written in C/C++. Apache SPARK RDDs are a very common use case of this. Another exampled is a ByteBuffer in Java that can act like a native array from C/C++.

**[Related](https://www.quora.com/What-is-the-difference-between-stack-and-heap?no_redirect=1" \t "_blank)**

**[What is the difference between stack and heap?](https://www.quora.com/What-is-the-difference-between-stack-and-heap?no_redirect=1" \t "_blank)**

In the context of Operating Systems, stack and heap are the two sections of the memory layout of a process. The stack is used to keep track of variables/parameters local to a function in a program. Whenever you call a new function, a new stack frame is pushed to the stack with parameters and variables local to that function. When that function returns, the stack frame is popped out and the context switches back to the previous function (the caller).

A heap is a kind of a global memory pool. A function can allocate memory on the heap if it wants the data to live longer than the function itself. Objects allocated on the heap are accessible to all the functions, given they have the reference/address of the object to access it. In C, you can allocate memory on the heap using the malloc(3) family of functions.

**[Related](https://www.quora.com/With-the-increasing-size-of-computer-ram-will-storage-drives-ever-become-defunct-allowing-all-data-to-be-kept-within-the-ram-thus-creating-a-hybrid-of-sorts" \t "_blank)**

**[With the increasing size of computer ram, will storage drives ever become defunct allowing all data to be kept within the ram, thus creating a hybrid of sorts?](https://www.quora.com/With-the-increasing-size-of-computer-ram-will-storage-drives-ever-become-defunct-allowing-all-data-to-be-kept-within-the-ram-thus-creating-a-hybrid-of-sorts" \t "_blank)**

Not as RAM is currently implemented.

What we commonly refer to as “RAM” is typically implemented as “DRAM” or “Dynamic Random Access Memory.” Due to the design of DRAM, it is very fast compared to SSD’s or HDD’s, but relatively sensitive to electrical disturbances. It is also **volatile**, meaning that it loses its contents almost immediately upon losing power. And the contents must be refreshed every 10–100 ms in order to retain its contents.

That last part is important, and what really sets RAM apart from storage drives. Storage drives are relatively resilient to outside forces, and can retain data for years without any power required. This property of DRAM is not a problem while the system is running, but it becomes a big problem when the system is powered off.

It is possible to use some system memory as a storage drive (look up “ram disk”), but it’s only used rarely, because it (a) requires a battery system to retain contents when the system is off, (b) will lose its contents if the battery dies, (c) is quite expensive per gigabyte, and (d) provides speed benefits that only matter in rare situations.

Some systems keep data in RAM for long periods of time. For example, when was the last time you restarted your phone? Most people don’t do it very often, and usually when they do, it’s only because the battery died and it shut itself off. Most people leave their phone in a sleep state so it can still receive notifications and perform various background tasks. During this time, it’s still providing power to keep the contents of RAM active.

But even in this situation, it is important to have a nonvolatile storage system as well. Otherwise you would need to carefully conserve your battery such that you never run it all the way down. Doing so would mean you lose everything. Including the operating system!

Perhaps the memory technology that we use for system memory will change in the future to a nonvolatile design that would allow long-term and low-latency data storage. But I certainly won’t hold my breath on that, as that would involve a dramatic shift in the way we use and program computers.

**[Related](https://www.quora.com/What-is-heap-memory-allocation" \t "_blank)**

**[What is heap memory allocation?](https://www.quora.com/What-is-heap-memory-allocation" \t "_blank)**

Originally Answered: What is Heap memory allocation?

Greetings….!!!

Presumably you mean *heap* from a memory allocation point of view, not from a data structure point of view (the term has multiple meanings).

A very simple explanation is that the **heap** is the portion of memory where *dynamically allocated* memory resides (i.e. memory allocated via malloc). Memory allocated from the heap will remain allocated until one of the following occurs:

1. The memory is free'd
2. The program terminates

If all references to allocated memory are lost (e.g. you don't store a pointer to it anymore), you have what is called a *memory leak*. This is where the memory has still been allocated, but you have no easy way of accessing it anymore. Leaked memory cannot be reclaimed for future memory allocations, but when the program ends the memory will be free'd up by the operating system.

Contrast this with **stack** memory which is where local variables (those defined within a method) live. Memory allocated on the stack generally only lives until the function returns (there are some exceptions to this, e.g. static local variables).

With Best Wishes…!!!  
Great Day..!!!

**[Related](https://www.quora.com/What-is-the-meaning-of-the-acronym-%E2%80%9CBIOS%E2%80%9D?no_redirect=1" \t "_blank)**

**[What is the meaning of the acronym “BIOS”?](https://www.quora.com/What-is-the-meaning-of-the-acronym-%E2%80%9CBIOS%E2%80%9D?no_redirect=1" \t "_blank)**

Originally Answered: [What is the full form of the word “BIOS”?](https://www.quora.com/What-is-the-meaning-of-the-acronym-%E2%80%9CBIOS%E2%80%9D?no_redirect=1)

**“What is the full form of the word ‘BIOS’?”**

Not to be pedantic (he said, ironically) but BIOS isn’t a word - it’s an acronym that someone at IBM probably worked very, very hard at making into a true acronym.

It expands into “Basic Input/Output System”. It’s the technology in computers that checks hardware on power-up, and enables the various hardware components (processor, memory, disk adapters, video cards, network interfaces, serial busses, etc.) to work together to let you do what you need to do.

It’s largely been replaced in recent years with EFI (Extensible Firmware Interface) systems that act more like mini-operating systems than monolithic, near-ROM systems that BIOSes are.

**[Related](https://www.quora.com/The-Zig-programming-language-has-coroutines-that-don-t-require-heap-allocation-How-is-that-possible-Where-are-suspended-coroutines-saved-if-no-heap-allocation-involved" \t "_blank)**

**[The Zig programming language has coroutines that don’t require heap allocation. How is that possible? Where are suspended coroutines saved if no heap allocation involved?](https://www.quora.com/The-Zig-programming-language-has-coroutines-that-don-t-require-heap-allocation-How-is-that-possible-Where-are-suspended-coroutines-saved-if-no-heap-allocation-involved" \t "_blank)**

My understanding [from this](https://ziglang.org/learn/why_zig_rust_d_cpp/) is that you still have to allocate a frame for the async function. (Apparently, that’s the [preferred term](https://ziglang.org/download/0.5.0/release-notes.html#Async-Functions) now in Zig.) You’re not *forced* into heap allocation.

“Doesn’t require *heap* allocation” is not the same as “doesn’t require allocation.” If you have a place for the invocation frame whose lifetime is a superset of the invocation itself, then you can put the frame there, as I understand it.

As the second link above states:

In Zig 0.5.0, calling an async function can no longer fail. The async function frame is provided by the caller via [Result Location Semantics](https://ziglang.org/download/0.5.0/release-notes.html#Result-Location-Semantics), and can be in the caller's stack frame. Async functions are no longer generic, and do not require the async keyword. Zig infers that a function is async when it observes that the function contains a **suspension point**. Async functions can be called the same as normal functions. A function call of an async function is a suspend point.

When a regular function is called, a frame is pushed to the stack, the function runs until it reaches a return statement, and then the frame is popped from the stack. At the callsite, the following code does not run until the function returns.

An async function is a function whose callsite is split into an async initiation, followed by an await completion. **Its frame is provided explicitly by the caller,** and it can be suspended and resumed any number of times.

I bolded the punchline in the final paragraph. The caller provides the frame. It can come from the heap, the stack, or anywhere else, provided its lifetime equals or exceeds the async function invocation’s lifetime.

**[Related](https://www.quora.com/Why-is-memory-size-RAM-always-in-power-of-2" \t "_blank)**

**[Why is memory size (RAM) always in power of 2?](https://www.quora.com/Why-is-memory-size-RAM-always-in-power-of-2" \t "_blank)**

First thing here is, digital circuits can understand and work on 2 values only. Either 0 or 1.

So all architecture is essentially designed around this.

If I had just 1 address line, how many bits of memory could I access? 2! Address bit 0 and Address bit 1.

Now moving on, to represent, say 3 bits of memory, how many bits of address would you need? 2 (00, 01, 10).

But using 2 bits of address you can actually address not just 3 bits, but 4(00, 01, 10, 11). So why limit it to 3 bits?

If you extrapolate this to any number there’s not a power of 2, you’ll have address lines that will always be able to address the nearest power of 2.

Not just memory, but everything the processor uses is a power of 2. Quite literally everything.

* Caches are powers of 2.
* Address and data lines are powers of 2(16bit, 32bit, 64bit etc)

All because the CPU can understand only 2. 0 and 1.

**[Related](https://www.quora.com/Why-does-C-show-a-garbage-value-for-uninitialized-variables-instead-of-null-as-Java-does" \t "_blank)**

**[Why does C show a garbage value for uninitialized variables, instead of null as Java does?](https://www.quora.com/Why-does-C-show-a-garbage-value-for-uninitialized-variables-instead-of-null-as-Java-does" \t "_blank)**

I feel like I've answered this before, but I couldn't find it so I'll hit it up again.

C is not an intelligent programming language. It is a very straight forward, no nonsense type of language. It doesn't assume things. It doesn't guess. It simply does what it is told. That's it.

What you are referring to is going a little beyond what the user is doing. I say "hey I want 7 bytes of memory” and C says “"ok here”, but Java says “"ok I cleaned them out for you too!”

The reason this isn't necessarily standard is because it is an extra step to null those values. If I am guaranteed to write over them, zeroing them in the first place is a waste of time.

[sarcasm] Plus, there's an added functionality of being able to get garbage values whenever I want! [/sarcasm]

Each has its uses, but the differences in C and Java actually have reasons behind them.

**[Related](https://www.quora.com/Do-C-programmers-intentionally-avoid-dynamic-memory-allocation-when-coding-in-C" \t "_blank)**

**[Do C programmers intentionally avoid dynamic memory allocation when coding in C?](https://www.quora.com/Do-C-programmers-intentionally-avoid-dynamic-memory-allocation-when-coding-in-C" \t "_blank)**

Only lazy incompetent ones. Using dynamic memory allocation avoids putting hard limits into programs, the program can allocate it’s data structures according to the size of the data the program has to deal with.

Many data structures are more easily handled with dynamic memory such as linked lists, trees, heaps.

There are occasions when dynamic memory should be avoid in time critical systems where the allocation might fail or take longer than allowable, but that is the exception. Very small memory systems also don’t support dynamic memory allocation or not very well, but these are specialist systems and not general purpose computers.

**[Related](https://www.quora.com/Why-does-register-memory-exist-when-it-has-such-small-storage-compared-to-RAM" \t "_blank)**

**[Why does register memory exist when it has such small storage compared to RAM?](https://www.quora.com/Why-does-register-memory-exist-when-it-has-such-small-storage-compared-to-RAM" \t "_blank)**

Because of two things.

1. It’s way way way way way way way way way faster than RAM. This is essential.
2. It’s actually where the data the CPU is actively working on goes. The CPU never actually acts directly on memory. Components within the CPU only work within the CPU itself and only on registers.

Fun fact: Almost all your hardware has registers, because to work hardware needs data in specific places to function. System memory is ENTIRELY just for data that is ABOUT to be worked on, not data that is ACTIVELY being operated. The CPU, like all other components, has to make requests for data from memory.

The ISA of CPUs also depends on registers. When working on assembly, registers are more or less analogous to the more immediate variables in higher-level languages, barring actual memory addresses themselves.

**[Related](https://www.quora.com/Stack-memory-is-allocated-in-any-random-order-whereas-Heap-memory-is-allocated-in-a-contiguous-block-do-you-agree" \t "_blank)**

**[“Stack memory is allocated in any random order whereas Heap memory is allocated in a contiguous block” - do you agree?](https://www.quora.com/Stack-memory-is-allocated-in-any-random-order-whereas-Heap-memory-is-allocated-in-a-contiguous-block-do-you-agree" \t "_blank)**

It can be true, although I’d say “random” is misleading. More like “arbitrary.”

**Heap**

With a pointer-incrementing allocator, heap allocations are always made in linear address increasing order. If you have a large enough virtual address space, you’ll never even need to rewind that pointer. Just keep mapping more physical pages as the pointer bumps upward. Let garbage collection compact the heap as necessary, and unmap pages from the tail as they empty out.

At any given time, the span of virtual addresses associated with the heap is contiguous.

Such an allocator is more feasible in a 64-bit VM than a 32-bit VM. But, even a 64-bit VM space will eventually run out of virtual addresses. So, with a slight modification we can define contiguous as contiguous *modulo the virtual address arena size.* After all, that’s how circular buffering works.

**Stack**

Meanwhile, you can construct an invocation record stack (aka. “call stack”) by allocating procedure invocation records and local variables on the previously described heap. Because each invocation record (and any local objects) will be interspersed with other, unrelated allocations, the stack itself will not necessarily reside in contiguous memory.

This structure also allows to suspend a function invocation as part of a coroutine. Thus, the stack need not operate in strict LIFO fashion, despite its name. That makes the stack even less contiguous.