What and where are the stack and heap?

Asked 15 years, 6 months ago Modified 3 months ago Viewed 1.9m times

9401

- What are the stack and heap?
- Where are they located physically in a computer's memory?
- To what extent are they controlled by the OS or language run-time?
- What is their scope?
- What determines their sizes?
- What makes one faster?

data-structures memory-management heap-memory dynamic-memory-allocation stack-memory

Share Edit Follow Flag

edited Dec 1, 2023 at 11:15
trincot
330k 35 259 296

asked Sep 17, 2008 at 4:18
mattshane
94.1k 3 18 5

248 a really good explanation can be found here What's the difference between a stack and a heap? – Songo Dec 16, 2013 at 11:32

19 Also (really) good: codeproject.com/Articles/76153/... (the stack/heap part) – Ben Feb 15, 2014 at 5:50

21 youtube.com/watch?v=clOUdVDDzIM&spfreload=5 – Selvamani Jun 11, 2016 at 5:42

5 Related, see Stack Clash. The Stack Clash remediations affected some aspects of system variables and behaviors like `rlimit_stack` . Also see Red Hat Issue 1463241 – jww Jun 21, 2017 at 16:23

5 @mattshane The definitions of stack and heap don't depend on value and reference types whatsoever. In other words, the stack and heap can be fully defined even if value and reference types never existed. Further, when understanding value and reference types, the stack is just an implementation detail. Per Eric Lippert: The Stack Is An Implementation Detail, Part One. – Matthew Nov 12, 2017 at 22:38

31 Answers

Sorted by: Highest score (default)

1 2 Next

The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data.

When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.

The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.

Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

To answer your questions directly:

> *To what extent are they controlled by the OS or language runtime?*

The OS allocates the stack for each system-level thread when the thread is created. Typically the OS is called by the language runtime to allocate the heap for the application.

> *What is their scope?*

The stack is attached to a thread, so when the thread exits the stack is reclaimed. The heap is typically allocated at application startup by the runtime, and is reclaimed when the application (technically process) exits.

> *What determines the size of each of them?*

The size of the stack is set when a thread is created. The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system).

> *What makes one faster?*

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 deallocation. 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.

A clear demonstration:

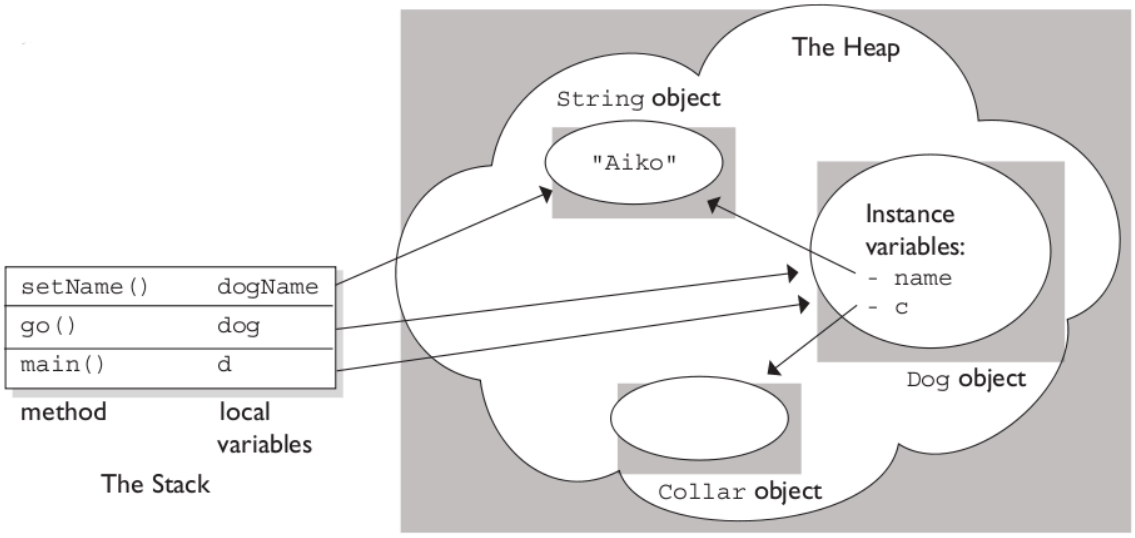


Image source: vikashazrati.wordpress.com

Share Edit Follow Flag

edited Nov 28, 2021 at 14:10
user1666620
4,800 19 28

answered Sep 17, 2008 at 4:52
Jeff Hill
70.5k 3 19 7

140 Good answer - but I think you should add that while the stack is allocated by the OS when the process starts (assuming the existence of an OS), it is maintained inline by the program. This is another reason the stack is faster, as well - push and pop operations are typically one machine instruction, and modern machines can do at least 3 of them in one cycle, whereas allocating or freeing heap involves calling into OS code. – sqykly Oct 8, 2013 at 8:31

488 I'm really confused by the diagram at the end. I thought I got it until I saw that image. – Sina Madani Aug 15, 2016 at 19:06

16 @Anarelle the processor runs instructions with or without an os. An example close to my heart is the SNES, which had no API calls, no OS as we know it today - but it had a stack. Allocating on a stack is addition and subtraction on these systems and that is fine for variables destroyed when they are popped by returning from the function that created them, but constrast that to, say, a constructor, of which the result can't just be thrown away. For that we need the heap, which is not tied to call and return. Most OS have APIs a heap, no reason to do it on your own – sqykly Oct 13, 2016 at 15:06

8 "stack is the memory set aside as scratch space". Cool. But where is it actually "set aside" in terms of Java memory structure?? Is it Heap memory/Non-heap memory/Other (Java memory structure as per betsol.com/2017/06/...) – chepaiytrath Jul 22, 2018 at 6:22

8 @JatinShashoo Java runtime, as bytecode interpreter, adds one more level of virtualization, so what you referred to is just Java application point of view. From operating system point of view all that is just a heap, where Java runtime process allocates some of its space as "non-heap" memory for processed bytecode. Rest of that OS-level heap is used as application-level heap, where object's data are stored. – kbec Sep 6, 2018 at 15:41

2689

Stack:

- Stored in computer RAM just like the heap.
- Variables created on the stack will go out of scope and are automatically deallocated.
- Much faster to allocate in comparison to variables on the heap.
- Implemented with an actual stack data structure.
- Stores local data, return addresses, used for parameter passing.
- Can have a stack overflow when too much of the stack is used (mostly from infinite or too deep recursion, very large allocations).
- Data created on the stack can be used without pointers.
- You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
- Usually has a maximum size already determined when your program starts.

Heap:

- Stored in computer RAM just like the stack.
- In C++, variables on the heap must be destroyed manually and never fall out of scope. The data is freed with `delete`, `delete[]`, or `free`.
- Slower to allocate in comparison to variables on the stack.
- Used on demand to allocate a block of data for use by the program.
- Can have fragmentation when there are a lot of allocations and deallocations.
- In C++ or C, data created on the heap will be pointed to by pointers and allocated with `new` or `malloc` respectively.
- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at run time or if you need to allocate a lot of data.
- Responsible for memory leaks.

Example:

```
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)
  {
    //Create 500 bytes on the stack
    char buffer[500];
```

```
    //Create 500 bytes on the heap
    pBuffer = new char[500];

  }//<-- buffer is deallocated here, pBuffer is not
}//<--- oops there's a memory leak, I should have called delete[] pBuffer;
```

Share Edit Follow Flag

edited Jul 28, 2017 at 0:38
Rob ♦
27.1k 16 86 101

answered Sep 17, 2008 at 4:20

Brian R. Bondy
343k 125 597 638

39 The pointer pBuffer and the value of b are located on the stack, and are mostly likely allocated at the entrance to the function. Depending on the compiler, buffer may be allocated at the function entrance, as well. – Andy Mar 18, 2009 at 22:48

47 It is a common misconception that the `C` language, as defined by the `C99` language standard (available at open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf), requires a "stack". In fact, the word 'stack' does not even appear in the standard. This answers statements wrt/ to `C` 's stack usage are true in general, but is in no way required by the language. See knosof.co.uk/cbook/cbook.html for more info, and in particular how `C` is implemented on odd-ball architectures such as en.wikipedia.org/wiki/Burroughs_large_systems – johne Sep 1, 2009 at 4:37

63 @Brian You should explain *why* buffer[] and the pBuffer pointer are created on the stack and why pBuffer's data is created on the heap. I think some ppl might be confused by your answer as they might think the program is specifically instructing that memory be allocated on the stack vs heap but this is not the case. Is it because Buffer is a value type whereas pBuffer is a reference type? – Howiecamp Feb 8, 2010 at 4:56

53 "Responsible for memory leaks" - Heaps are not responsible for memory leaks! Lazy/Forgetful/ex-java coders/coders who dont give a crap are! – Laz Mar 25, 2013 at 8:22

8 Also the comments about scope and allocation are wrong - Scope is not connected to the stack or the heap at all. *Variables on the heap must be destroyed manually and never fall out of scope.* isn't correct; it would be more correct to say "Data on the heap isn't freed when variables that reference them go out of scope. It's up to you (or the garbage collector) to free them. – Orion Edwards Sep 1, 2014 at 2:12

1514

The most important point is that heap and stack are generic terms for ways in which memory can be allocated. They can be implemented in many different ways, and the terms apply to the basic concepts.

- In a stack of items, items sit one on top of the other in the order they were placed there, and you can only remove the top one (without toppling the whole thing over).

The simplicity of a stack is that you do not need to maintain a table containing a record of each section of allocated memory; the only state information you need is a single pointer to the end of the stack. To allocate and de-allocate, you just increment and decrement that single pointer. Note: a stack can sometimes be implemented to start at the top of a section of memory and extend downwards rather than growing upwards.

- In a heap, there is no particular order to the way items are placed. You can reach in and remove items in any order because there is no clear 'top' item.

Heap allocation requires maintaining a full record of what memory is allocated and what isn't, as well as some overhead maintenance to reduce fragmentation, find contiguous memory segments big enough to fit the requested size, and so on. Memory can be deallocated at any time leaving free space. Sometimes a memory allocator will perform maintenance tasks such as defragmenting memory by moving allocated memory around, or garbage collecting - identifying at runtime when memory is no longer in scope and deallocating it.

These images should do a fairly good job of describing the two ways of allocating and freeing memory in a stack and a heap. Yum!

- To what extent are they controlled by the OS or language runtime?

  As mentioned, heap and stack are general terms, and can be implemented in many ways. Computer programs typically have a stack called a call stack which stores information relevant to the current function such as a pointer to whichever function it was called from, and any local variables. Because functions call other functions and then return, the stack grows and shrinks to hold information from the functions further down the call stack. A

program doesn't really have runtime control over it; it's determined by the programming language, OS and even the system architecture.

A heap is a general term used for any memory that is allocated dynamically and randomly; i.e. out of order. The memory is typically allocated by the OS, with the application calling API functions to do this allocation. There is a fair bit of overhead required in managing dynamically allocated memory, which is usually handled by the runtime code of the programming language or environment used.

- What is their scope?

  The call stack is such a low level concept that it doesn't relate to 'scope' in the sense of programming. If you disassemble some code you'll see relative pointer style references to portions of the stack, but as far as a higher level language is concerned, the language imposes its own rules of scope. One important aspect of a stack, however, is that once a function returns, anything local to that function is immediately freed from the stack. That works the way you'd expect it to work given how your programming languages work. In a heap, it's also difficult to define. The scope is whatever is exposed by the OS, but your programming language probably adds its rules about what a "scope" is in your application. The processor architecture and the OS use virtual addressing, which the processor translates to physical addresses and there are page faults, etc. They keep track of what pages belong to which applications. You never really need to worry about this, though, because you just use whatever method your programming language uses to allocate and free memory, and check for errors (if the allocation/freeing fails for any reason).

- What determines the size of each of them?

  Again, it depends on the language, compiler, operating system and architecture. A stack is usually pre-allocated, because by definition it must be contiguous memory. The language compiler or the OS determine its size. You don't store huge chunks of data on the stack, so it'll be big enough that it should never be fully used, except in cases of unwanted endless recursion (hence, "stack overflow") or other unusual programming decisions.

  A heap is a general term for anything that can be dynamically allocated. Depending on which way you look at it, it is constantly changing size. In modern processors and operating systems the exact way it works is very abstracted anyway, so you don't normally need to worry much about how it works deep down, except that (in languages where it lets you) you mustn't use memory that you haven't allocated yet or memory that you have freed.

- What makes one faster?

  The stack is faster because all free memory is always contiguous. No list needs to be maintained of all the segments of free memory, just a single pointer to the current top of the stack. Compilers usually store this pointer in a special, fast register for this purpose. What's more, subsequent operations on a stack are usually concentrated within very nearby areas of memory, which at a very low level is good for optimization by the processor on-die caches.

Share Edit Follow Flag

edited Mar 12, 2020 at 0:37

answered Mar 19, 2009 at 14:38

thomasrutter
116k 31 151 168

27 David I don't agree that that is a good image or that "push-down stack" is a good term to illustrate the concept. When you add something to a stack, the other contents of the stack *aren't* pushed down, they remain where they are. – thomasrutter Aug 13, 2012 at 3:40

16 This answer includes a big mistake. Static variables are not allocated on the stack. See my answer [link] stackoverflow.com/a/13326916/1763801 for clarification. you are equating "automatic" variables with "static" variables, but they are not at all the same – davec Nov 10, 2012 at 23:07

19 Specifically, you say "statically allocated local variables" are allocated on the stack. Actually they are allocated in the data segment. Only automatically allocated variables (which includes most but not all local variables and also things like function parameters passed in by value rather than by reference) are allocated on the stack. – davec Nov 11, 2012 at 1:44

15 I've just realised you're right - in C, *static allocation* is its own separate thing rather than a term for anything that's not *dynamic*. I've edited my answer, thanks. – thomasrutter Nov 12, 2012 at 0:29

9 It's not just C. Java, Pascal, Python and many others all have the notions of static versus automatic versus dynamic allocation. Saying "static allocation" means the same thing just about everywhere. In no language does static allocation mean "not dynamic". You want the term "automatic" allocation for what you are describing (i.e. the things on the stack). – davec Nov 12, 2012 at 17:16

823

(I have moved this answer from another question that was more or less a dupe of this one.)

The answer to your question is implementation specific and may vary across compilers and processor architectures. However, here is a simplified explanation.

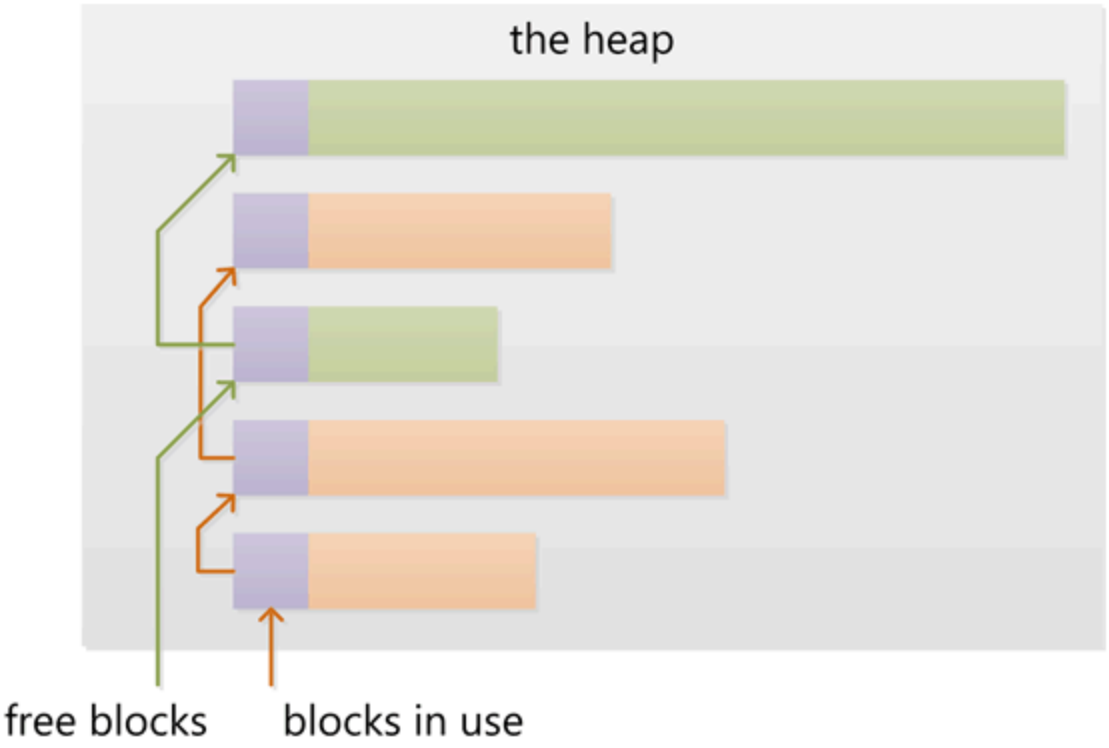
- Both the stack and the heap are memory areas allocated from the underlying operating system (often virtual memory that is mapped to physical memory on demand).
- In a multi-threaded environment each thread will have its own completely independent stack but they will share the heap. Concurrent access has to be controlled on the heap and is not possible on the stack.

The heap

- The heap contains a linked list of used and free blocks. New allocations on the heap (by `new` or `malloc`) are satisfied by creating a suitable block from one of the free blocks. This requires updating the list of blocks on the heap. This *meta information* about the blocks on the heap is also stored on the heap often in a small area just in front of every block.
- As the heap grows new blocks are often allocated from lower addresses towards higher addresses. Thus you can think of the heap as a *heap* of memory blocks that grows in size as

memory is allocated. If the heap is too small for an allocation the size can often be increased by acquiring more memory from the underlying operating system.

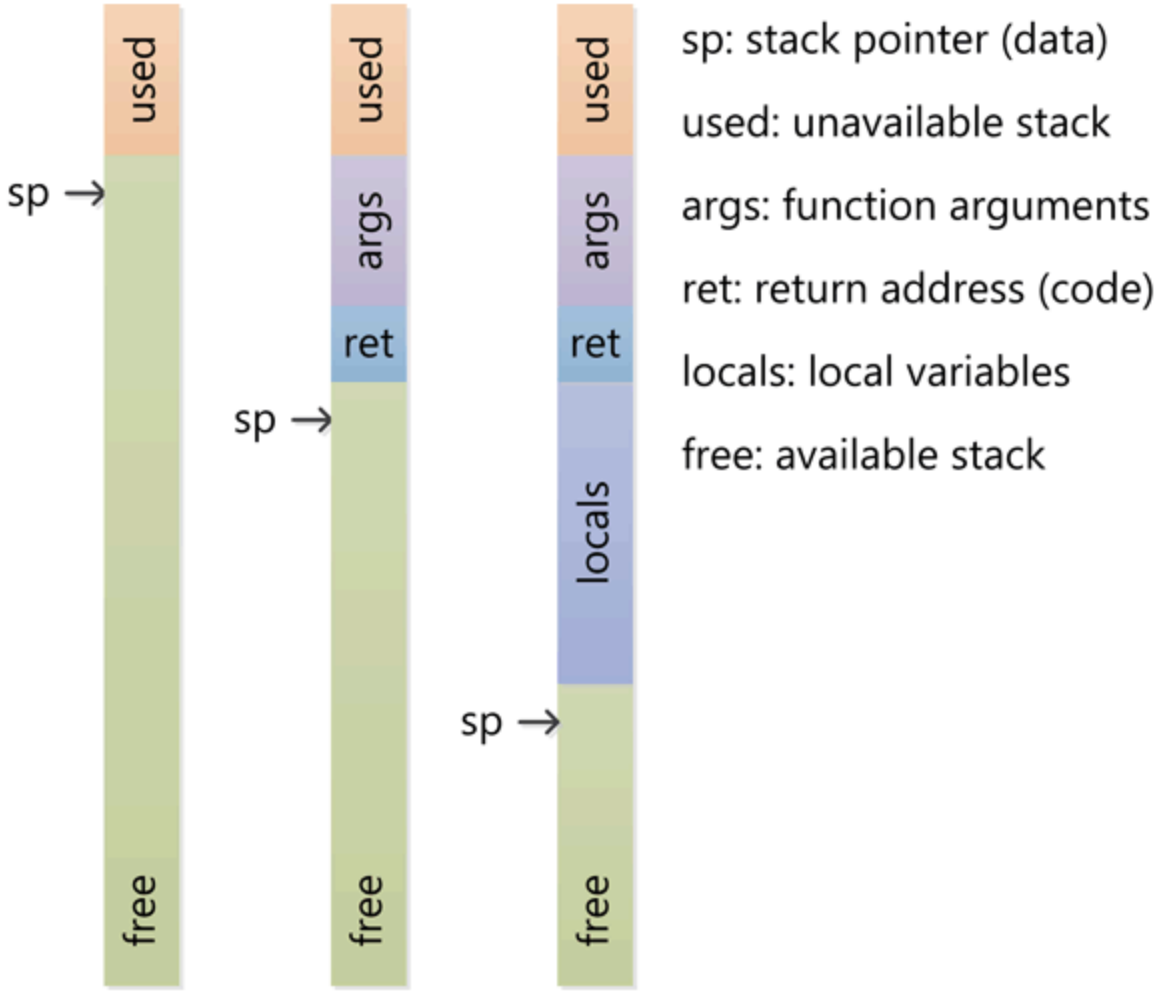
- Allocating and deallocating many small blocks may leave the heap in a state where there are a lot of small free blocks interspersed between the used blocks. A request to allocate a large block may fail because none of the free blocks are large enough to satisfy the allocation request even though the combined size of the free blocks may be large enough. This is called *heap fragmentation*.
- When a used block that is adjacent to a free block is deallocated the new free block may be merged with the adjacent free block to create a larger free block effectively reducing the fragmentation of the heap.



The stack

- The stack often works in close tandem with a special register on the CPU named the *stack pointer*. Initially the stack pointer points to the top of the stack (the highest address on the stack).
- The CPU has special instructions for *pushing* values onto the stack and *popping* them off the stack. Each *push* stores the value at the current location of the stack pointer and decreases the stack pointer. A *pop* retrieves the value pointed to by the stack pointer and then increases the stack pointer (don't be confused by the fact that *adding* a value to the stack *decreases* the stack pointer and *removing* a value *increases* it. Remember that the stack grows to the bottom). The values stored and retrieved are the values of the CPU registers.

- If a function has parameters, these are pushed onto the stack before the call to the function. The code in the function is then able to navigate up the stack from the current stack pointer to locate these values.
- When a function is called the CPU uses special instructions that push the current *instruction pointer* onto the stack, i.e. the address of the code executing on the stack. The CPU then jumps to the function by setting the instruction pointer to the address of the function called. Later, when the function returns, the old instruction pointer is popped off the stack and execution resumes at the code just after the call to the function.
- When a function is entered, the stack pointer is decreased to allocate more space on the stack for local (automatic) variables. If the function has one local 32 bit variable four bytes are set aside on the stack. When the function returns, the stack pointer is moved back to free the allocated area.
- Nesting function calls work like a charm. Each new call will allocate function parameters, the return address and space for local variables and these *activation records* can be stacked for nested calls and will unwind in the correct way when the functions return.
- As the stack is a limited block of memory, you can cause a *stack overflow* by calling too many nested functions and/or allocating too much space for local variables. Often the memory area used for the stack is set up in such a way that writing below the bottom (the lowest address) of the stack will trigger a trap or exception in the CPU. This exceptional condition can then be caught by the runtime and converted into some kind of stack overflow exception.



Can a function be allocated on the heap instead of a stack?

No, activation records for functions (i.e. local or automatic variables) are allocated on the stack that is used not only to store these variables, but also to keep track of nested function calls.

How the heap is managed is really up to the runtime environment. C uses `malloc` and C++ uses `new`, but many other languages have garbage collection.

However, the stack is a more low-level feature closely tied to the processor architecture. Growing the heap when there is not enough space isn't too hard since it can be implemented in the library call that handles the heap. However, growing the stack is often impossible as the stack overflow only is discovered when it is too late; and shutting down the thread of execution is the only viable option.

Share Edit Follow Flag

edited Sep 1, 2021 at 15:07

Géry Ogam
6,966 5 38 75

answered Jul 31, 2009 at 15:54

Martin Liversage
106k 22 211 259

42 @Martin - A very good answer/explanation than the more abstract accepted answer. A sample assembly program showing stack pointers/registers being used vis a vis function calls would be more

illustrative. – Bikal Lem Apr 25, 2012 at 16:42

5 Every reference type is composition of value types(int, string etc). As it is said, that value types are stored in stack than how does it work when they are part of reference type. – Nps Feb 15, 2014 at 10:26

23 This answer was the best in my opinion, because it helped me understand what a return statement really is and how it relates to this "return address" that I come across every now and then, what it means to push a function onto the stack, and why functions are pushed onto stacks. Great answer! – Alex Mar 19, 2014 at 19:59

6 This is the best in my opinion, namely for mentioning that the heap/stack are *very* implementation specific. The other answers assume a *lot* of things about the language and the environment/OS. +1 – Qix - MONICA WAS MISTREATED Nov 29, 2014 at 2:00

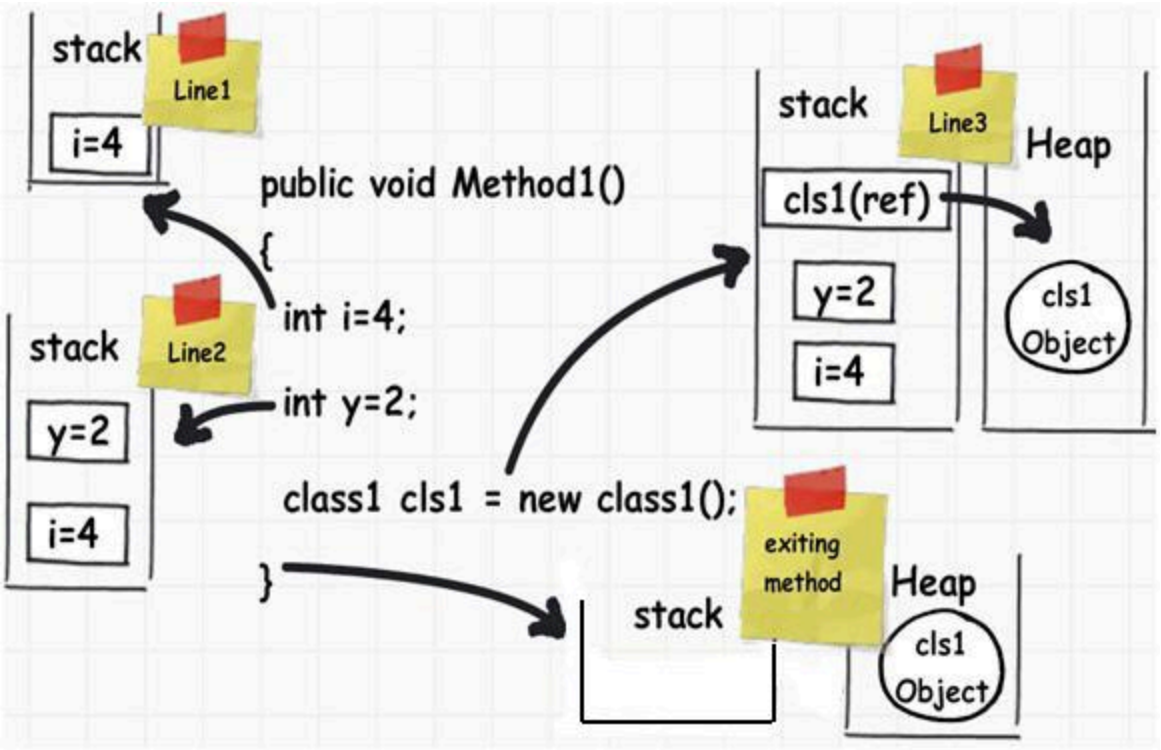
3 What do you mean "The code in the function is then able to navigate up the stack from the current stack pointer to locate these values." ? Can you elaborate on this please? – Koray Tugay May 3, 2015 at 9:36

448

In the following C# code

```
public void Method1()
{
    int i = 4;
    int y = 2;
    class1 cls1 = new class1();
}
```

Here's how the memory is managed



`Local Variables` that only need to last as long as the function invocation go in the stack. The heap is used for variables whose lifetime we don't really know up front but we expect them to last a while. In most languages it's critical that we know at compile time how large a variable is if we want to store it on the stack.

Objects (which vary in size as we update them) go on the heap because we don't know at creation time how long they are going to last. In many languages the heap is garbage collected to find objects (such as the cls1 object) that no longer have any references.

In Java, most objects go directly into the heap. In languages like C / C++, structs and classes can often remain on the stack when you're not dealing with pointers.

More information can be found here:

The difference between stack and heap memory allocation « timmurphy.org

and here:

Creating Objects on the Stack and Heap

This article is the source of picture above: Six important .NET concepts: Stack, heap, value types, reference types, boxing, and unboxing - CodeProject

but be aware it may contain some inaccuracies.

Share Edit Follow Flag

edited Apr 19, 2022 at 11:53
tagurit
515 5 15

answered Nov 9, 2012 at 12:28
Snowcrash
82.9k 93 273 390

17 This is incorrect. i and cls are not "static" variables. they are called "local" or "automatic" variables. It is a very important distinction. See [link] stackoverflow.com/a/13326916/1763801 for clarification – davec Nov 10, 2012 at 23:05

10 I did not say they were static *variables*. I said that int and cls1 are static *items*. Their memory is statically allocated and therefore they go on the stack. This is in contrast to an object which requires dynamic memory allocation which therefore goes on the heap. – Snowcrash Nov 20, 2012 at 14:38

13 I quote "Static items... go on the stack". This is just flat out wrong. Static items go in the data segment, automatic items go on the stack. – davec Nov 21, 2012 at 16:55

16 Also whoever wrote that codeproject article doesn't know what he is talking about. For instance, he says "primitive ones needs static type memory" which is completely untrue. Nothing stops you from allocating primitives in the heap dynamically, just write something like "int array[] = new int[num]" and voila, primitives allocated dynamically in .NET. That is just one of several inaccuracies. – davec Nov 21, 2012 at 17:02

1 @SnowCrash one question about your picture - how do I access `i` after allocating `y`? Do I have to pop up `y`? Swap them? What if there are a lot of local variables separating them? – confused00 Jan 16, 2015 at 12:53

234

Other answers just avoid explaining what static allocation means. So I will explain the three main forms of allocation and how they usually relate to the heap, stack, and data segment below. I also will show some examples in both C/C++ and Python to help people understand.

"Static" (AKA statically allocated) variables are not allocated on the stack. Do not assume so - many people do only because "static" sounds a lot like "stack". They actually exist in neither the stack nor the heap. They are part of what's called the data segment.

However, it is generally better to consider "**scope**" and "**lifetime**" rather than "stack" and "heap".

Scope refers to what parts of the code can access a variable. Generally we think of **local scope** (can only be accessed by the current function) versus **global scope** (can be accessed anywhere) although scope can get much more complex.

Lifetime refers to when a variable is allocated and deallocated during program execution. Usually we think of **static allocation** (variable will persist through the entire duration of the program, making it useful for storing the same information across several function calls) versus **automatic allocation** (variable only persists during a single call to a function, making it useful for storing information that is only used during your function and can be discarded once you are done) versus **dynamic allocation** (variables whose duration is defined at runtime, instead of compile time like static or automatic).

Although most compilers and interpreters implement this behavior similarly in terms of using stacks, heaps, etc, a compiler may sometimes break these conventions if it wants as long as behavior is correct. For instance, due to optimization a local variable may only exist in a register or be removed entirely, even though most local variables exist in the stack. As has been pointed out in a few comments, you are free to implement a compiler that doesn't even use a stack or a heap, but instead some other storage mechanisms (rarely done, since stacks and heaps are great for this).

I will provide some simple annotated C code to illustrate all of this. The best way to learn is to run a program under a debugger and watch the behavior. If you prefer to read python, skip to the end of the answer :)

```
// Statically allocated in the data segment when the program/DLL is first loaded
// Deallocated when the program/DLL exits
// scope - can be accessed from anywhere in the code
int someGlobalVariable;

// Statically allocated in the data segment when the program is first loaded
// Deallocated when the program/DLL exits
// scope - can be accessed from anywhere in this particular code file
static int someStaticVariable;

// "someArgument" is allocated on the stack each time MyFunction is called
// "someArgument" is deallocated when MyFunction returns
// scope - can be accessed only within MyFunction()
```

```
void MyFunction(int someArgument) {

    // Statically allocated in the data segment when the program is first loaded
    // Deallocated when the program/DLL exits
    // scope - can be accessed only within MyFunction()
    static int someLocalStaticVariable;

    // Allocated on the stack each time MyFunction is called
    // Deallocated when MyFunction returns
    // scope - can be accessed only within MyFunction()
    int someLocalVariable;

    // A *pointer* is allocated on the stack each time MyFunction is called
    // This pointer is deallocated when MyFunction returns
    // scope - the pointer can be accessed only within MyFunction()
    int* someDynamicVariable;

    // This line causes space for an integer to be allocated in the heap
    // when this line is executed. Note this is not at the beginning of
    // the call to MyFunction(), like the automatic variables
    // scope - only code within MyFunction() can access this space
    // *through this particular variable*.
    // However, if you pass the address somewhere else, that code
    // can access it too
    someDynamicVariable = new int;


    // This line deallocates the space for the integer in the heap.
    // If we did not write it, the memory would be "leaked".
    // Note a fundamental difference between the stack and heap
    // the heap must be managed. The stack is managed for us.
    delete someDynamicVariable;

    // In other cases, instead of deallocating this heap space you
    // might store the address somewhere more permanent to use later.
    // Some languages even take care of deallocation for you... but
    // always it needs to be taken care of at runtime by some mechanism.

    // When the function returns, someArgument, someLocalVariable
    // and the pointer someDynamicVariable are deallocated.
    // The space pointed to by someDynamicVariable was already
    // deallocated prior to returning.
    return;
}

// Note that someGlobalVariable, someStaticVariable and
// someLocalStaticVariable continue to exist, and are not
// deallocated until the program exits.
```

A particularly poignant example of why it's important to distinguish between lifetime and scope is that a variable can have local scope but static lifetime - for instance, "someLocalStaticVariable" in the code sample above. Such variables can make our common but informal naming habits very confusing. For instance when we say "*local*" we usually mean "*locally scoped automatically allocated variable*" and when we say global we usually mean "*globally scoped statically allocated variable*". Unfortunately when it comes to things like "*file scoped statically allocated variables*" many people just say... "*huh???*".

Some of the syntax choices in C/C++ exacerbate this problem - for instance many people think global variables are not "static" because of the syntax shown below.

```
int var1; // Has global scope and static allocation
static int var2; // Has file scope and static allocation

int main() {return 0;}
```

Note that putting the keyword "static" in the declaration above prevents var2 from having global scope. Nevertheless, the global var1 has static allocation. This is not intuitive! For this reason, I try to never use the word "static" when describing scope, and instead say something like "file" or "file limited" scope. However many people use the phrase "static" or "static scope" to describe a variable that can only be accessed from one code file. In the context of lifetime, "static" *always* means the variable is allocated at program start and deallocated when program exits.

Some people think of these concepts as C/C++ specific. They are not. For instance, the Python sample below illustrates all three types of allocation (there are some subtle differences possible in interpreted languages that I won't get into here).

```
from datetime import datetime

class Animal:
    _FavoriteFood = 'Undefined' # _FavoriteFood is statically allocated

    def PetAnimal(self):
        curTime = datetime.time(datetime.now()) # curTime is automatically allocatedion
        print("Thank you for petting me. But it's " + str(curTime) + ", you should feed
me. My favorite food is " + self._FavoriteFood)

class Cat(Animal):
    _FavoriteFood = 'tuna' # Note since we override, Cat class has its own statically
allocated _FavoriteFood variable, different from Animal's

class Dog(Animal):
    _FavoriteFood = 'steak' # Likewise, the Dog class gets its own static variable.
Important to note - this one static variable is shared among all instances of Dog,
hence it is not dynamic!


if __name__ == "__main__":
    whiskers = Cat() # Dynamically allocated
    fido = Dog() # Dynamically allocated
    rinTinTin = Dog() # Dynamically allocated

    whiskers.PetAnimal()
    fido.PetAnimal()
    rinTinTin.PetAnimal()

    Dog._FavoriteFood = 'milkbones'
    whiskers.PetAnimal()
    fido.PetAnimal()
    rinTinTin.PetAnimal()

# Output is:
# Thank you for petting me. But it's 13:05:02.255000, you should feed me. My favorite
```

```
food is tuna
# Thank you for petting me. But it's 13:05:02.255000, you should feed me. My favorite
food is steak
# Thank you for petting me. But it's 13:05:02.255000, you should feed me. My favorite
food is steak
# Thank you for petting me. But it's 13:05:02.255000, you should feed me. My favorite
food is tuna
# Thank you for petting me. But it's 13:05:02.255000, you should feed me. My favorite
food is milkbones
# Thank you for petting me. But it's 13:05:02.256000, you should feed me. My favorite
food is milkbones
```

Share Edit Follow Flag edited Mar 5, 2022 at 9:31 community wiki 22 revs, 7 users 72% davec

2 I would refer to a static variable declared within a function as having only local *accessibility*, but would generally not use the term "scope" with it. Also, it may be worth noting that the one stack/heap aspect with which languages have essentially zero flexibility: a language which saves execution context on a stack cannot use that same stack to hold things which will need to outlive the contexts wherein they are created. Some languages like `PostScript` have multiple stacks, but have a "heap" that behaves more like a stack. – supercat Dec 9, 2013 at 21:53

2 you must be kidding. can you really define static variable inside a function ? – Zaeem Sattar May 16, 2017 at 9:57

2 @zaeemsattar absolutely and this is not ususual to see in C code – davec Jun 22, 2017 at 22:41

1 @ZaeemSattar Think of the static function variable like a hidden global or like a private static member variable. – Tom Leys Jul 26, 2018 at 22:42

229

The Stack When you call a function the arguments to that function plus some other overhead is put on the stack. Some info (such as where to go on return) is also stored there. When you declare a variable inside your function, that variable is also allocated on the stack.

Deallocating the stack is pretty simple because you always deallocate in the reverse order in which you allocate. Stack stuff is added as you enter functions, the corresponding data is removed as you exit them. This means that you tend to stay within a small region of the stack unless you call lots of functions that call lots of other functions (or create a recursive solution).

The Heap The heap is a generic name for where you put the data that you create on the fly. If you don't know how many spaceships your program is going to create, you are likely to use the new (or malloc or equivalent) operator to create each spaceship. This allocation is going to stick around for a while, so it is likely we will free things in a different order than we created them.

Thus, the heap is far more complex, because there end up being regions of memory that are unused interleaved with chunks that are - memory gets fragmented. Finding free memory of the

size you need is a difficult problem. This is why the heap should be avoided (though it is still often used).

Implementation Implementation of both the stack and heap is usually down to the runtime / OS. Often games and other applications that are performance critical create their own memory solutions that grab a large chunk of memory from the heap and then dish it out internally to avoid relying on the OS for memory.

This is only practical if your memory usage is quite different from the norm - i.e for games where you load a level in one huge operation and can chuck the whole lot away in another huge operation.

Physical location in memory This is less relevant than you think because of a technology called Virtual Memory which makes your program think that you have access to a certain address where the physical data is somewhere else (even on the hard disc!). The addresses you get for the stack are in increasing order as your call tree gets deeper. The addresses for the heap are un-predictable (i.e implimentation specific) and frankly not important.

Share Edit Follow Flag

edited Sep 17, 2008 at 4:34

answered Sep 17, 2008 at 4:27

Tom Leys
18.7k 7 41 63

18 A recommendation to avoid using the heap is pretty strong. Modern systems have good heap managers, and modern dynamic languages use the heap extensively (without the programmer really worrying about it). I'd say use the heap, but with a manual allocator, don't forget to free!
– Greg Hewgill Sep 17, 2008 at 4:31

3 If you can use the stack or the heap, use the stack. If you can't use the stack, really no choice. I use both a lot, and of course using std::vector or similar hits the heap. For a novice, you avoid the heap because the stack is simply so easy!! – Tom Leys Sep 17, 2008 at 4:35

1 "This is why the heap should be avoided (though it is still often used)." I'm not sure what this practically means, especially as memory is managed differently in many high level languages. As this question is tagged language-agnostic, I'd say this particular comment/line is ill-placed and not applicable.
– LintfordPickle Jul 25, 2018 at 9:34

3 Good point @JonnoHampson - While you make a valid point, I'd argue that if you're working in a "high level language" with a GC you probably don't care about memory allocation mechanisms at all - and so don't even care what the stack and heap are. – Tom Leys Jul 26, 2018 at 22:22

178

Others have answered the broad strokes pretty well, so I'll throw in a few details.

1. Stack and heap need not be singular. A common situation in which you have more than one stack is if you have more than one thread in a process. In this case each thread has its own stack. You can also have more than one heap, for example some DLL configurations can

result in different DLLs allocating from different heaps, which is why it's generally a bad idea to release memory allocated by a different library.

2. In C you can get the benefit of variable length allocation through the use of alloca, which allocates on the stack, as opposed to alloc, which allocates on the heap. This memory won't survive your return statement, but it's useful for a scratch buffer.

3. Making a huge temporary buffer on Windows that you don't use much of is not free. This is because the compiler will generate a stack probe loop that is called every time your function is entered to make sure the stack exists (because Windows uses a single guard page at the end of your stack to detect when it needs to grow the stack. If you access memory more than one page off the end of the stack you will crash). Example:

```
void myfunction()
{
   char big[10000000];
   // Do something that only uses for first 1K of big 99% of the time.
}
```

Share Edit Follow Flag

edited Jul 30, 2017 at 11:18

Peter Mortensen
31k 22 108 132

answered Sep 17, 2008 at 4:48

Don Neufeld
23k 11 52 50

151

Others have directly answered your question, but when trying to understand the stack and the heap, I think it is helpful to consider the memory layout of a traditional UNIX process (without threads and `mmap()` -based allocators). The Memory Management Glossary web page has a diagram of this memory layout.

The stack and heap are traditionally located at opposite ends of the process's virtual address space. The stack grows automatically when accessed, up to a size set by the kernel (which can be adjusted with `setrlimit(RLIMIT_STACK, ...)`). The heap grows when the memory allocator invokes the `brk()` or `sbrk()` system call, mapping more pages of physical memory into the process's virtual address space.

In systems without virtual memory, such as some embedded systems, the same basic layout often applies, except the stack and heap are fixed in size. However, in other embedded systems (such as those based on Microchip PIC microcontrollers), the program stack is a separate block of memory that is not addressable by data movement instructions, and can only be modified or read indirectly through program flow instructions (call, return, etc.). Other architectures, such as Intel Itanium processors, have multiple stacks. In this sense, the stack is an element of the CPU architecture.

Share Edit Follow Flag

edited Apr 17, 2022 at 23:08

tagurit
515 5 15

answered Sep 17, 2008 at 7:16

bk1e
24k 6 55 65

128

What is a stack?

A stack is a pile of objects, typically one that is neatly arranged.

Stacks in computing architectures are regions of memory where data is added or removed in a last-in-first-out manner.
In a multi-threaded application, each thread will have its own stack.

What is a heap?

A heap is an untidy collection of things piled up haphazardly.

In computing architectures the heap is an area of dynamically-allocated memory that is managed automatically by the operating system or the memory manager library.
Memory on the heap is allocated, deallocated, and resized regularly during program execution, and this can lead to a problem called fragmentation.
Fragmentation occurs when memory objects are allocated with small spaces in between that are too small to hold additional memory objects.
The net result is a percentage of the heap space that is not usable for further memory allocations.

Both together

In a multi-threaded application, each thread will have its own stack. But, all the different threads will share the heap.
Because the different threads share the heap in a multi-threaded application, this also means that there has to be some coordination between the threads so that

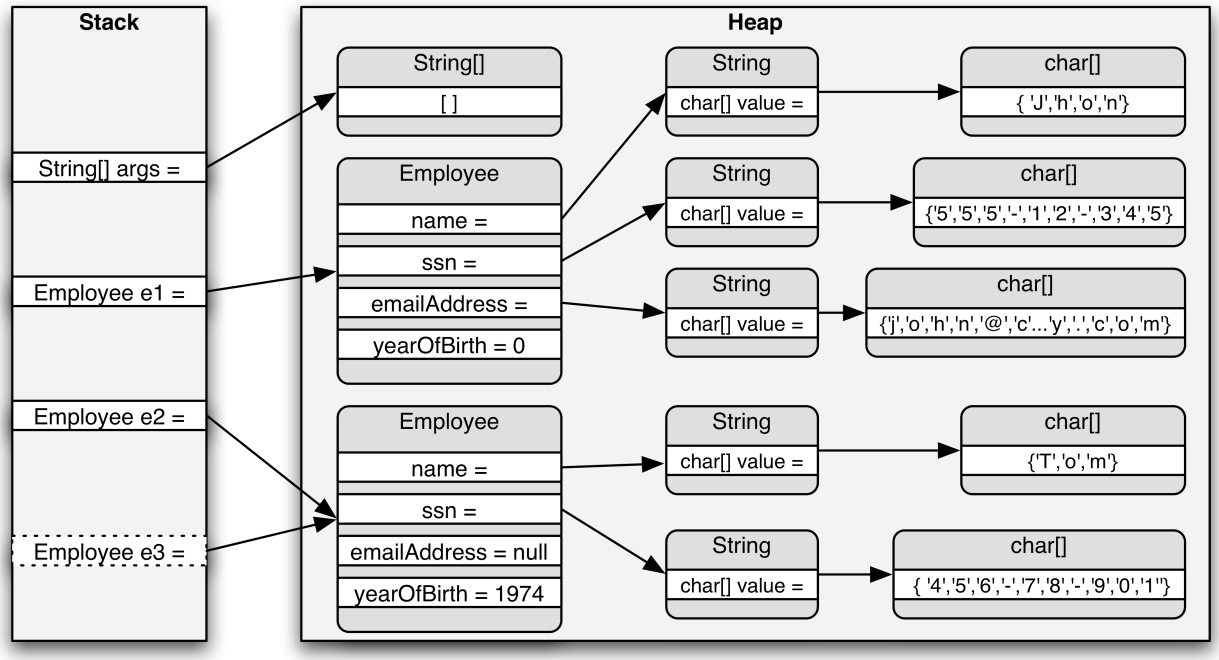
they don't try to access and manipulate the same piece(s) of memory in the heap at the same time.

Which is faster – the stack or the heap? And why?

The stack is much faster than the heap.
This is because of the way that memory is allocated on the stack.
Allocating memory on the stack is as simple as moving the stack pointer up.

For people new to programming, it's probably a good idea to use the stack since it's easier.
Because the stack is small, you would want to use it when you know exactly how much memory you will need for your data, or if you know the size of your data is very small.
It's better to use the heap when you know that you will need a lot of memory for your data, or you just are not sure how much memory you will need (like with a dynamic array).

Java Memory Model



The stack is the area of memory where local variables (including method parameters) are stored.
When it comes to object variables, these are merely references (pointers) to the actual objects on the heap.
Every time an object is instantiated, a chunk of heap memory is set aside to hold the data (state) of that object. Since objects can contain other objects, some of this data can in fact hold references to those nested objects.

Share Edit Follow Flag

edited Jun 20, 2020 at 9:12

Community Bot

1 1

answered Jun 11, 2014 at 19:42

Shreyos Adikari

12.5k 19 74 82

124

The stack is a portion of memory that can be manipulated via several key assembly language instructions, such as 'pop' (remove and return a value from the stack) and 'push' (push a value to the stack), but also call (call a subroutine - this pushes the address to return to the stack) and return (return from a subroutine - this pops the address off of the stack and jumps to it). It's the region of memory below the stack pointer register, which can be set as needed. The stack is also used for passing arguments to subroutines, and also for preserving the values in registers before calling subroutines.

The heap is a portion of memory that is given to an application by the operating system, typically through a syscall like malloc. On modern OSes this memory is a set of pages that only the calling process has access to.

The size of the stack is determined at runtime, and generally does not grow after the program launches. In a C program, the stack needs to be large enough to hold every variable declared within each function. The heap will grow dynamically as needed, but the OS is ultimately making the call (it will often grow the heap by more than the value requested by malloc, so that at least some future mallocs won't need to go back to the kernel to get more memory. This behavior is often customizable)

Because you've allocated the stack before launching the program, you never need to malloc before you can use the stack, so that's a slight advantage there. In practice, it's very hard to predict what will be fast and what will be slow in modern operating systems that have virtual memory subsystems, because how the pages are implemented and where they are stored is an implementation detail.

Share Edit Follow Flag

answered Sep 17, 2008 at 4:29

Daniel Papasian

16.3k 6 30 32

2 Also worth mentioning here that intel heavily optimizes stack accesses, especially things such as predicting where you return from a function. – Tom Leys Sep 17, 2008 at 4:37

122

I think many other people have given you mostly correct answers on this matter.

One detail that has been missed, however, is that the "heap" should in fact probably be called the "free store". The reason for this distinction is that the original free store was implemented with a data structure known as a "binomial heap." For that reason, allocating from early implementations of malloc()/free() was allocation from a heap. However, in this modern day,

most free stores are implemented with very elaborate data structures that are not binomial heaps.

Share Edit Follow Flag

answered Sep 17, 2008 at 4:57
Heath

8 Another nitpick- most of the answers (lightly) imply that the use of a "stack" is required by the `C` language. This is a common misconception, though it is the (by far) dominate paradigm for implementing `C99 6.2.4 automatic storage duration objects` (variables). In fact, the word "stack" does not even appear in the `C99` language standard: open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf – johne Sep 1, 2009 at 5:03

97

You can do some interesting things with the stack. For instance, you have functions like alloca (assuming you can get past the copious warnings concerning its use), which is a form of malloc that specifically uses the stack, not the heap, for memory.

That said, stack-based memory errors are some of the worst I've experienced. If you use heap memory, and you overstep the bounds of your allocated block, you have a decent chance of triggering a segment fault. (Not 100%: your block may be incidentally contiguous with another that you have previously allocated.) But since variables created on the stack are always contiguous with each other, writing out of bounds can change the value of another variable. I have learned that whenever I feel that my program has stopped obeying the laws of logic, it is probably buffer overflow.

Share Edit Follow Flag

answered Mar 19, 2009 at 15:55

Peter
1,316 8 4

95

Simply, the stack is where local variables get created. Also, every time you call a subroutine the program counter (pointer to the next machine instruction) and any important registers, and sometimes the parameters get pushed on the stack. Then any local variables inside the subroutine are pushed onto the stack (and used from there). When the subroutine finishes, that stuff all gets popped back off the stack. The PC and register data gets and put back where it was as it is popped, so your program can go on its merry way.

The heap is the area of memory dynamic memory allocations are made out of (explicit "new" or "allocate" calls). It is a special data structure that can keep track of blocks of memory of varying sizes and their allocation status.

In "classic" systems RAM was laid out such that the stack pointer started out at the bottom of memory, the heap pointer started out at the top, and they grew towards each other. If they

overlap, you are out of RAM. That doesn't work with modern multi-threaded OSes though. Every thread has to have its own stack, and those can get created dynamicly.

Share Edit Follow Flag

edited Mar 19, 2009 at 15:19

answered Mar 19, 2009 at 15:13

T.E.D.
44.4k 10 75 136

From WikiAnwser.

87

Stack

When a function or a method calls another function which in turns calls another function, etc., the execution of all those functions remains suspended until the very last function returns its value.

This chain of suspended function calls is the stack, because elements in the stack (function calls) depend on each other.

The stack is important to consider in exception handling and thread executions.

Heap

The heap is simply the memory used by programs to store variables. Element of the heap (variables) have no dependencies with each other and can always be accessed randomly at any time.

Share Edit Follow Flag

edited Jul 30, 2017 at 12:01

Peter Mortensen
31k 22 108 132

answered Apr 2, 2009 at 1:25

devXen
3,062 3 36 44

58

Stack

- Very fast access
- Don't have to explicitly de-allocate variables
- Space is managed efficiently by CPU, memory will not become fragmented
- Local variables only
- Limit on stack size (OS-dependent)
- Variables cannot be resized

Heap

- Variables can be accessed globally

- No limit on memory size
- (Relatively) slower access
- No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- You must manage memory (you're in charge of allocating and freeing variables)
- Variables can be resized using realloc()

Share Edit Follow Flag

edited Jul 30, 2017 at 12:14

Peter Mortensen
31k 22 108 132

answered Jan 30, 2014 at 6:33

unknown
4,917 10 45 62

56

In Short

A stack is used for static memory allocation and a heap for dynamic memory allocation, both stored in the computer's RAM.

In Detail

The Stack

The stack is a "LIFO" (last in, first out) data structure, that is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.

The advantage of using the stack to store variables, is that memory is managed for you. You don't have to allocate memory by hand, or free it once you don't need it any more. What's more, because the CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

More can be found **here**.

The Heap

The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use malloc() or calloc(), which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using free() to deallocate that memory once you don't need it any more.

If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called Valgrind that can help you detect memory leaks.

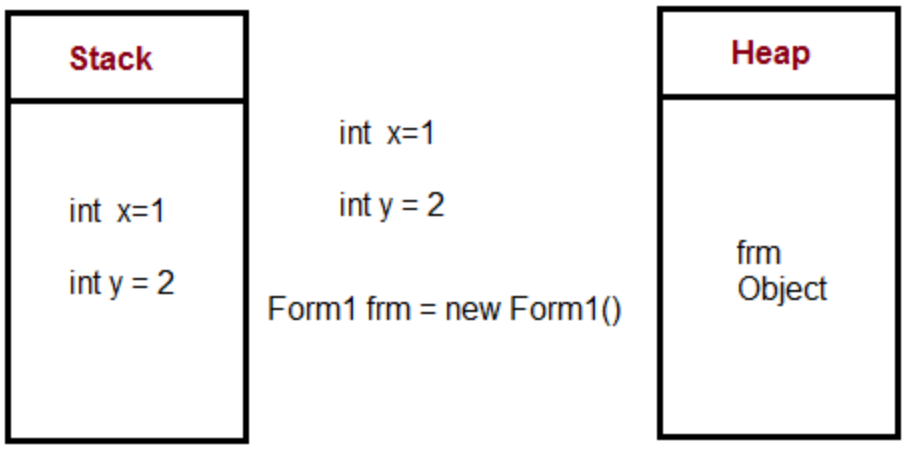
Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap. We will talk about pointers shortly.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

More can be found **here**.

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and its allocation is dealt with when the program is compiled. When a function or a method calls another function which in turns calls another function, etc., the execution of all those functions remains suspended until the very last function returns its value. The stack is always reserved in a LIFO order, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer.

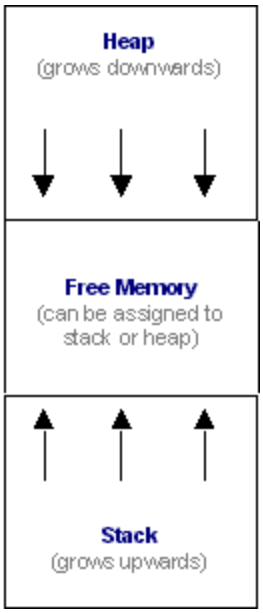
Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory. Elements of the heap have no dependencies with each other and can always be accessed randomly at any time. You can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.



You can use the stack if you know exactly how much data you need to allocate before compile time, and it is not too big. You can use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

In a multi-threaded situation each thread will have its own completely independent stack, but they will share the heap. The stack is thread specific and the heap is application specific. The stack is important to consider in exception handling and thread executions.

Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).



At run-time, if the application needs more heap, it can allocate memory from free memory and if the stack needs memory, it can allocate memory from free memory allocated memory for the application.

Even, more detail is given **here** and **here**.

Now come to **your question's answers**.

> **To what extent are they controlled by the OS or language runtime?**

The OS allocates the stack for each system-level thread when the thread is created. Typically the OS is called by the language runtime to allocate the heap for the application.

More can be found **here**.

> **What is their scope?**

Already given in top.

> "You can use the stack if you know exactly how much data you need to allocate before compile time, and it is not too big. You can use the heap if you don't know exactly how

much data you will need at runtime or if you need to allocate a lot of data."

More can be found in here.

What determines the size of each of them?

The size of the stack is set by OS when a thread is created. The size of the heap is set on application startup, but it can grow as space is needed (the allocator requests more memory from the operating system).

What makes one faster?

Stack allocation is much faster since all it really does is move the stack pointer. Using memory pools, you can get comparable performance out of heap allocation, but that comes with a slight added complexity and its own headaches.

Also, stack vs. heap is not only a performance consideration; it also tells you a lot about the expected lifetime of objects.

Details can be found from **here**.

Share Edit Follow Flag

edited Dec 9, 2018 at 15:12
zgue
3,813 9 35 40

answered May 2, 2016 at 12:16

Abrar Jahin
14.2k 24 112 164

55

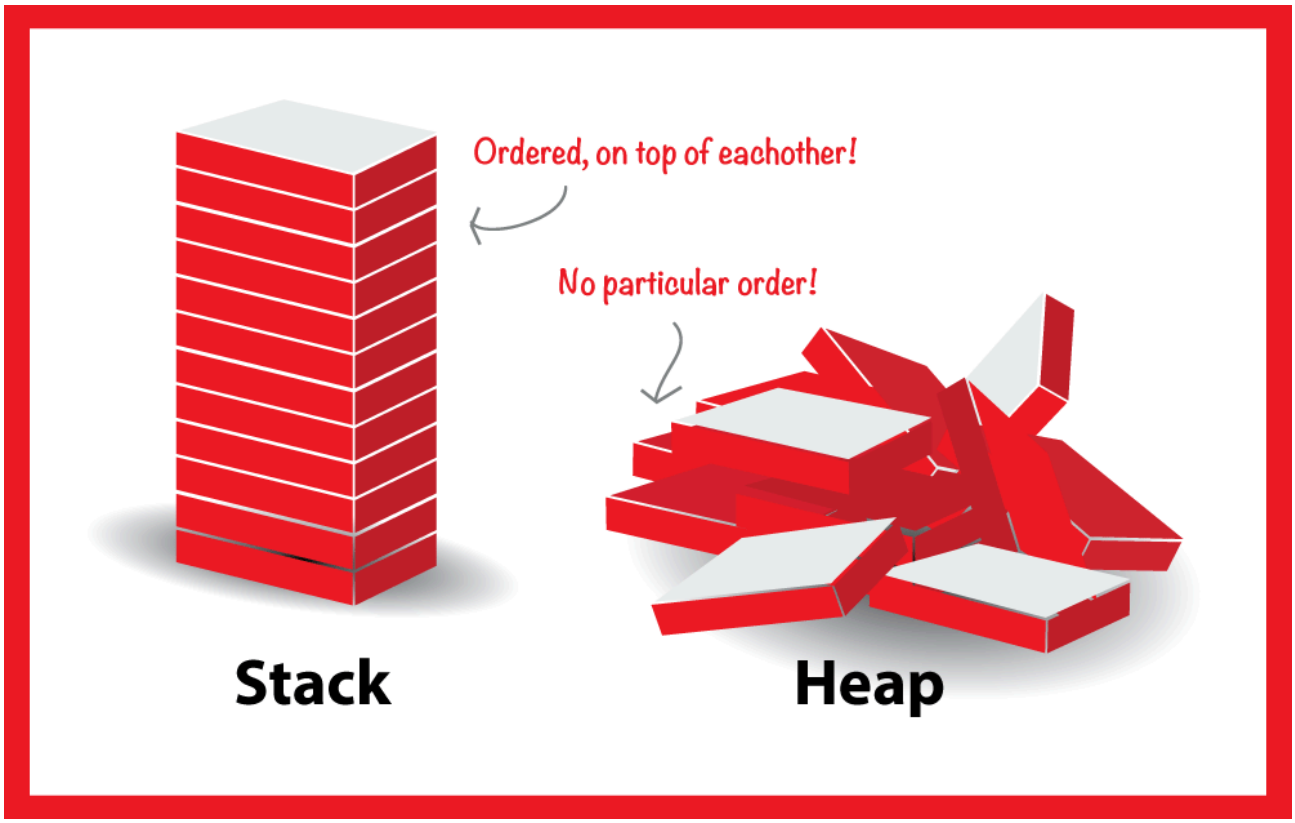
OK, simply and in short words, they mean **ordered** and **not ordered**...!

Stack: In stack items, things get on the top of each-other, means gonna be faster and more efficient to be processed!...

So there is always an index to point the specific item, also processing gonna be faster, there is relationship between the items as well!...

Heap: No order, processing gonna be slower and values are messed up together with no specific order or index... there are random and there is no relationship between them... so execution and usage time could be vary...

I also create the image below to show how they may look like:



Share Edit Follow Flag

edited Apr 23, 2018 at 2:06

answered Jul 18, 2017 at 15:04

Alireza
102k 27 275 173

|

47

stack, **heap** and **data** of each process in virtual memory:


stack, heap and static data

Share Edit Follow Flag

edited Mar 12, 2018 at 18:00

answered Sep 14, 2017 at 17:32

Yousha Aleayoub
5,140 4 54 65

37

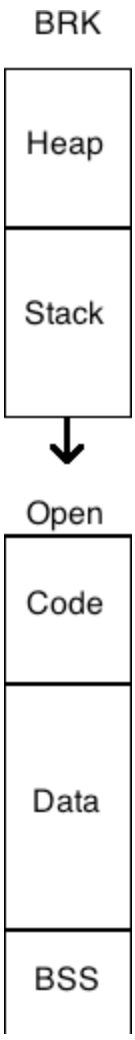
In the 1980s, UNIX propagated like bunnies with big companies rolling their own. Exxon had one as did dozens of brand names lost to history. How memory was laid out was at the discretion of the many implementors.

A typical C program was laid out flat in memory with an opportunity to increase by changing the brk() value. Typically, the HEAP was just below this brk value and increasing brk increased the amount of available heap.

The single STACK was typically an area below HEAP which was a tract of memory containing nothing of value until the top of the next fixed block of memory. This next block was often CODE which could be overwritten by stack data in one of the famous hacks of its era.

One typical memory block was BSS (a block of zero values) which was accidentally not zeroed in one manufacturer's offering. Another was DATA containing initialized values, including strings and numbers. A third was CODE containing CRT (C runtime), main, functions, and libraries.

The advent of virtual memory in UNIX changes many of the constraints. There is no objective reason why these blocks need be contiguous, or fixed in size, or ordered a particular way now. Of course, before UNIX was Multics which didn't suffer from these constraints. Here is a schematic showing one of the memory layouts of that era.



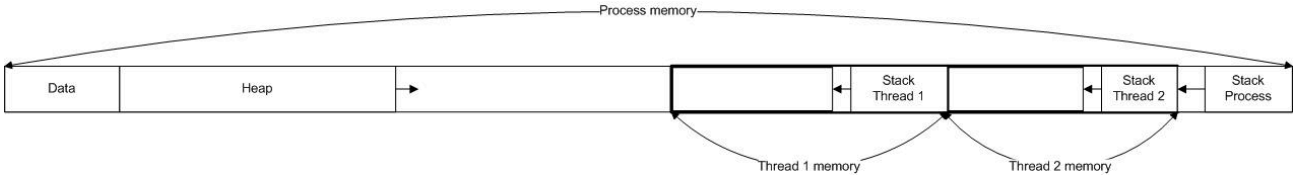
Share Edit Follow Flag

answered Mar 27, 2015 at 19:55

jlettvin
1,163 8 14

A couple of cents: I think, it will be good to draw memory graphical and more simple:

28



Arrows - show where grow stack and heap, process stack size have limit, defined in OS, thread stack size limits by parameters in thread create API usually. Heap usually limiting by process maximum virtual memory size, for 32 bit 2-4 GB for example.

So simple way: process heap is general for process and all threads inside, using for memory allocation in common case with something like **malloc()**.

Stack is quick memory for store in common case function return pointers and variables, processed as parameters in function call, local function variables.

Share Edit Follow Flag

edited Jul 30, 2017 at 12:22

Peter Mortensen
31k 22 108 132

answered Dec 17, 2015 at 15:08

Maxim Akristiniy
2,147 2 16 20

25

I have something to share, although the major points are already covered.

Stack

- Very fast access.
- Stored in RAM.
- Function calls are loaded here along with the local variables and function parameters passed.
- Space is freed automatically when program goes out of a scope.
- Stored in sequential memory.

Heap

- Slow access comparatively to Stack.
- Stored in RAM.
- Dynamically created variables are stored here, which later requires freeing the allocated memory after use.
- Stored wherever memory allocation is done, accessed by pointer always.

Interesting note:

- Should the function calls had been stored in heap, it would had resulted in 2 messy points:

1. Due to sequential storage in stack, execution is faster. Storage in heap would have resulted in huge time consumption thus making the whole program execute slower.
2. If functions were stored in heap (messy storage pointed by pointer), there would have been no way to return to the caller address back (which stack gives due to sequential storage in memory).

Share Edit Follow Flag

edited Dec 10, 2018 at 8:23

Pang
9,731 146 82 123

answered Nov 15, 2017 at 18:27

pkthapa
1,049 2 17 28

24

Since some answers went nitpicking, I'm going to contribute my mite.

Surprisingly, no one has mentioned that multiple (i.e. not related to the number of running OS-level threads) call stacks are to be found not only in exotic languages (PostScript) or platforms (Intel Itanium), but also in *fibers*, *green threads* and some implementations of *coroutines*.

Fibers, green threads and coroutines are in many ways similar, which leads to much confusion. The difference between fibers and green threads is that the former use cooperative multitasking, while the latter may feature either cooperative or preemptive one (or even both). For the distinction between fibers and coroutines, see here.

In any case, the purpose of both fibers, green threads and coroutines is having multiple functions executing concurrently, but **not** in parallel (see this SO question for the distinction) within a single OS-level thread, transferring control back and forth from one another in an organized fashion.

When using fibers, green threads or coroutines, you *usually* have a separate stack per function. (Technically, not just a stack but a whole context of execution is per function. Most importantly, CPU registers.) For every thread there're as many stacks as there're concurrently running functions, and the thread is switching between executing each function according to the logic of your program. When a function runs to its end, its stack is destroyed. So, **the number and lifetimes of stacks** are dynamic and **are not determined by the number of OS-level threads!**

Note that I said "*usually* have a separate stack per function". There're both *stackful* and *stackless* implementations of couroutines. Most notable stackful C++ implementations are Boost.Coroutine and Microsoft PPL's `async/await`. (However, C++'s resumable functions (a.k.a. "`async` and `await`"), which were proposed to C++17, are likely to use stackless coroutines.)

Fibers proposal to the C++ standard library is forthcoming. Also, there're some third-party libraries. Green threads are extremely popular in languages like Python and Ruby.

Share Edit Follow Flag

edited May 23, 2017 at 11:55
Community Bot
1 1

answered Mar 2, 2015 at 1:29
shakurov
2,388 25 29

17

Wow! So many answers and I don't think one of them got it right...

1) Where and what are they (physically in a real computer's memory)?

The stack is memory that begins as the highest memory address allocated to your program image, and it then decrease in value from there. It is reserved for called function parameters and for all temporary variables used in functions.

There are two heaps: public and private.

The private heap begins on a 16-byte boundary (for 64-bit programs) or a 8-byte boundary (for 32-bit programs) after the last byte of code in your program, and then increases in value from there. It is also called the default heap.

If the private heap gets too large it will overlap the stack area, as will the stack overlap the heap if it gets too big. Because the stack starts at a higher address and works its way down to lower address, with proper hacking you can get make the stack so large that it will overrun the private heap area and overlap the code area. The trick then is to overlap enough of the code area that you can hook into the code. It's a little tricky to do and you risk a program crash, but it's easy and very effective.

The public heap resides in it's own memory space outside of your program image space. It is this memory that will be siphoned off onto the hard disk if memory resources get scarce.

2) To what extent are they controlled by the OS or language runtime?

The stack is controlled by the programmer, the private heap is managed by the OS, and the public heap is not controlled by anyone because it is an OS service -- you make requests and either they are granted or denied.

2b) What is their scope?

They are all global to the program, but their contents can be private, public, or global.

2c) What determines the size of each of them?

The size of the stack and the private heap are determined by your compiler runtime options. The public heap is initialized at runtime using a size parameter.

2d) What makes one faster?

They are not designed to be fast, they are designed to be useful. How the programmer utilizes them determines whether they are "fast" or "slow"

REF:

https://norasandler.com/2019/02/18/Write-a-Compiler-10.html

https://learn.microsoft.com/en-us/windows/desktop/api/heapapi/nf-heapapi-getprocessheap

https://learn.microsoft.com/en-us/windows/desktop/api/heapapi/nf-heapapi-heapcreate

Share Edit Follow Flag

answered Feb 20, 2019 at 2:04

ar18
335 2 5

13

> Where and what are they (physically in a real computer's memory)?

ANSWER: Both are in RAM.

ASIDE:

RAM is like a desk and HDDs/SSDs (permanent storage) are like bookshelves. To read anything, you must have a book open on your desk, and you can only have as many books open as fit on your desk. To get a book, you pull it from your bookshelf and open it on your desk. To return a book, you close the book on your desk and return it to its bookshelf.

Stack and heap are names we give to two ways compilers store different kinds of data in the same place (i.e. in RAM).

> What is their scope?
> What determines the size of each of them?
> What makes one faster?

ANSWER:

1. **The stack is for static (fixed size) data**

   a. *At compile time, the compiler reads the variable types used in your code.*

   i. It allocates a fixed amount of memory for these variables.
   ii. This size of this memory cannot grow.

   b. *The memory is contiguous (a single block), so access is* **sometimes** *faster than the heap*

   c. *An object placed on the stack that grows in memory during runtime beyond the size of the stack causes a* **stack overflow error**

2. **The heap is for dynamic (changing size) data**

   a. *The amount of memory is limited only by the amount of empty space available in RAM*
   i. The amount used can grow or shrink as needed at runtime

   b. *Since items are allocated on the heap by finding empty space wherever it exists in RAM, data is not always in a contiguous section, which* **sometimes** *makes access slower than the stack*

   c. *Programmers manually put items on the heap with the* `new` *keyword and MUST manually deallocate this memory when they are finished using it.*
   i. Code that repeatedly allocates new memory without deallocating it when it is no longer needed leads to a **memory leak.**

ASIDE:

The stack and heap were not primarily introduced to improve speed; they were introduced to handle memory overflow. The first concern regarding use of the stack vs. the heap should be whether memory overflow will occur. If an object is intended to grow in size to an unknown amount (like a linked list or an object whose members can hold an arbitrary amount of data), place it on the heap. As far as possible, use the C++ standard library (STL) containers **vector**, **map**, and **list** as they are memory and speed efficient and added to make your life easier (you don't need to worry about memory allocation/deallocation).

After getting your code to run, if you find it is running unacceptably slow, then go back and refactor your code and see if it can be programmed more efficiently. It may turn out the problem has nothing to do with the stack or heap directly at all (e.g. use an iterative algorithm instead of a recursive one, look at I/O vs. CPU-bound tasks, perhaps add multithreading or multiprocessing).

I say *sometimes* slower/faster above because the speed of the program might not have anything to do with items being allocated on the stack or heap.

> To what extent are they controlled by the OS or language run-time?

ANSWER:

- **The stack size is determined at compile time by the compiler.**
- **The heap size varies during runtime.** (*The heap works with the OS during runtime to allocate memory.*)

ASIDE:

Below is a little more about control and compile-time vs. runtime operations.

Each computer has a unique **instruction set architecture (ISA)**, which are its hardware commands (e.g. "MOVE", "JUMP", "ADD", etc.).

- An OS is nothing more than a resource manager (controls how/when/ and where to use memory, processors, devices, and information).
- The ISA of the OS is called the **bare machine** and the remaining commands are called the **extended machine**. The **kernel** is the first layer of the extended machine. It controls things like
  - determining what tasks get to use a processor (the scheduler),
  - how much memory or how many hardware registers to allocate to a task (the dispatcher), and
  - the order in which tasks should be performed (the traffic controller).
- When we say "compiler", we generally mean the compiler, assembler, and linker together
  - The compiler turns source code into assembly language and passes it to the assembler,
  - The assembler turns the assembly language into machine code (ISA commands), and passes it to the linker
  - The linker takes all machine code (possibly generated from multiple source files) and combines it into one program.
- The machine code gets passed to the kernel when executed, which determines when it should run and take control, but the machine code itself contains ISA commands for requesting files, requesting memory, etc. So the code issues ISA commands, but everything has to pass by the kernel.

Share Edit Follow Flag

edited Jun 28, 2022 at 18:25

answered Nov 20, 2021 at 22:17

adam.hendry
4,994 6 30 53

10

A lot of answers are correct as concepts, but we must note that a stack is needed by the hardware (i.e. microprocessor) to allow calling subroutines (CALL in assembly language..). (OOP guys will call it *methods*)

On the stack you save return addresses and call → push / ret → pop is managed directly in hardware.

You can use the stack to pass parameters.. even if it is slower than using registers (would a microprocessor guru say or a good 1980s BIOS book...)

- Without stack **no** microprocessor can work. (we can't imagine a program, even in assembly language, without subroutines/functions)
- Without the heap it can. (An assembly language program can work without, as the heap is a OS concept, as malloc, that is a OS/Lib call.

Stack usage is faster as:

- Is hardware, and even push/pop are very efficient.
- malloc requires entering kernel mode, use lock/semaphore (or other synchronization primitives) executing some code and manage some structures needed to keep track of allocation.

Share Edit Follow Flag

edited Aug 7, 2017 at 8:27

answered Jul 27, 2017 at 22:14

ingconti
11.2k 3 63 49

|

8

I feel most answers are very convoluted and technical, while I didn't find one that could explain simply the reasoning behind those two concepts (i.e. why people created them in the first place?) and why you should care. Here is my attempt at one:

Data on the Stack is temporary and auto-cleaning

Data on the Heap is permanent until manually deleted

That's it.

Still, for more explanations :

The **stack** is meant to be used as the ephemeral or **working memory**, a memory space that we know will be entirely deleted regularly no matter what mess we put in there during the lifetime of our program. That's like the ***memo*** on your desk that you scribble on with anything going through your mind that you barely feel may be important, which you know you will just throw away at the end of the day because you will have filtered and organized the actual important notes in another medium, like a document or a book. We don't care for presentation, crossing-outs or unintelligible text, this is just for our work of the day and will remember what we meant an hour or two ago, it's just our quick and dirty way to store ideas we want to remember later without hurting our current stream of thoughts. That's what people mean by "the stack is the *scratchpad*".

The **heap** however is the **long-term memory**, the actual ***important document*** that will we stored, consulted and depended on for a very long time after its creation. It consequently needs to have perfect form and strictly contain the important data. That why it costs a lot to make and can't be used for the use-case of our precedent memo. It wouldn't be worthwhile, or even simply useless, to take all my notes in an academic paper presentation, writing the text as calligraphy. However this presentation is extremely useful for well curated data. That's what the heap is

meant to be. Well known data, important for the lifetime application, which is well controlled and needed at many places in your code. The system will thus never delete this precious data without you explicitly asking for it, because it knows "that's where the important data is!".

This is why you need to manage and take care of memory allocation on the heap, but don't need to bother with it for the stack.

Most top answers are merely technical details of the actual implementations of that concept in real computers.

So what to take away from this is that:

Unimportant, working, temporary, data just needed to make our functions and objects work is (generally) more relevant to be stored on the stack.

Important, permanent and foundational application data is (generally) more relevant to be stored on the heap.

This of course needs to be thought of only in the context of the lifetime of your program. Actual humanly important data generated by your program will need to be stored on an external file evidently. (Since whether it is the heap or the stack, they are both cleared entirely when your program terminates.)

PS: Those are just general rules, you can always find edge cases and each language comes with its own implementation and resulting quirks, this is meant to be taken as a guidance to the concept and a rule of thumb.

Share Edit Follow Flag

edited Feb 9, 2023 at 14:44

answered Feb 9, 2023 at 1:32

adamency
1,022 9 16

2

The stack is essentially an easy-to-access memory that simply manages its items as a - well - stack. Only **items for which the size is known in advance can go onto the stack**. This is the case for numbers, strings, booleans.

The **heap is a memory for items of which you can't predetermine the exact size and structure**. Since objects and arrays can be mutated and change at runtime, they have to go into the heap.

Source: Academind

Share Edit Follow Flag

edited Apr 10, 2020 at 5:23

answered Mar 22, 2020 at 3:16

nCardot

6,338 7 55 87

1

CPU stack and heap are physically related to how CPU and registers works with memory, how machine-assembly language works, not high-level languages themselves, even if these languages can decide little things.

All modern CPUs work with the "same" microprocessor theory: they are all based on what's called "registers" and some are for "stack" to gain performance. All CPUs have stack registers since the beginning and they had been always here, way of talking, as I know. Assembly languages are the same since the beginning, despite variations... up to Microsoft and its Intermediate Language (IL) that changed the paradigm to have a OO virtual machine assembly language. So we'll be able to have some CLI/CIL CPU in the future (one project of MS).

CPUs have stack registers to speed up memories access, but they are limited compared to the use of others registers to get full access to all the available memory for the processus. It why we talked about stack and heap allocations.

In summary, and in general, the heap is hudge and slow and is for "global" instances and objects content, as the stack is little and fast and for "local" variables and references (hidden pointers to forget to manage them).

So when we use the new keyword in a method, the reference (an int) is created in the stack, but the object and all its content (value-types as well as objects) is created in the heap, if I remember. But local elementary value-types and arrays are created in the stack.

The difference in memory access is at the cells referencing level: addressing the heap, the overall memory of the process, requires more complexity in terms of handling CPU registers, than the stack which is "more" locally in terms of addressing because the CPU stack register is used as base address, if I remember.

It is why when we have very long or infinite recurse calls or loops, we got stack overflow quickly, without freezing the system on modern computers...

C# Heap(ing) Vs Stack(ing) In .NET

Stack vs Heap: Know the Difference

Static class memory allocation where it is stored C#

What and where are the stack and heap?

https://en.wikipedia.org/wiki/Memory_management

https://en.wikipedia.org/wiki/Stack_register

Assembly language resources:

Assembly Programming Tutorial

Intel® 64 and IA-32 Architectures Software Developer Manuals

Share Edit Follow Flag

edited Jul 5, 2020 at 11:00

answered Jul 5, 2020 at 10:50

user12031933

0

Thank you for a really good discussion but as a real noob I wonder where instructions are kept? In the BEGINNING scientists were deciding between two architectures (von NEUMANN where everything is considered DATA and HARVARD where an area of memory was reserved for instructions and another for data). Ultimately, we went with the von Neumann design and now everything is considered 'the same'. This made it hard for me when I was learning assembly https://www.cs.virginia.edu/~evans/cs216/guides/x86.html because they talk about registers and stack pointers.

Everything above talks about DATA. My guess is that since an instruction is a defined thing with a specific memory footprint, it would go on the stack and so all 'those' registers discussed in assembly are on the stack. Of course then came object oriented programming with instructions and data comingled into a structure that was dynamic so now instructions would be kept on the heap as well?

Share Edit Follow Flag

answered Apr 9, 2020 at 15:29

aquagremlin

3,533 3 30 53