Stack and Heap

The **code** section contains your **bytecode**.

The **Stack** section of memory contains **methods, local variables and reference variables.**

The **Heap** section contains **Objects** (may also contain reference variables).

The **Static** section contains **Static data/methods**.

Of all of the above 4 sections, you need to understand the allocation of memory in Stack & Heap the most, since it will affect your programming efforts

Points to Remember:

- When a method is called , a frame is created on the top of stack.
- Once a method has completed execution , flow of control returns to the calling method and its corresponding stack frame is flushed.
- Local variables are created in the stack
- Instance variables are created in the heap & are part of the object they belong to.
- Reference variables are created in the stack.

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** 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 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 free. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast.

Stack:

- Stored in computer **RAM** just like the heap.
- Variables created on the stack will go out of scope and **automatically deallocate**.
- 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 inifinite (or too much) 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.
- 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++ data created on the heap will be pointed to by pointers and allocated with new or malloc
- 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 runtime or if you need to allocate a lot of data.*
- Responsible for **memory leaks.**

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).

- 
- 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.
- 

It does a fairly good job of describing the two ways of allocating and freeing memory in a stack and a heap.

Process

STACK

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).

HEAP

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).