# The Craft of Coding

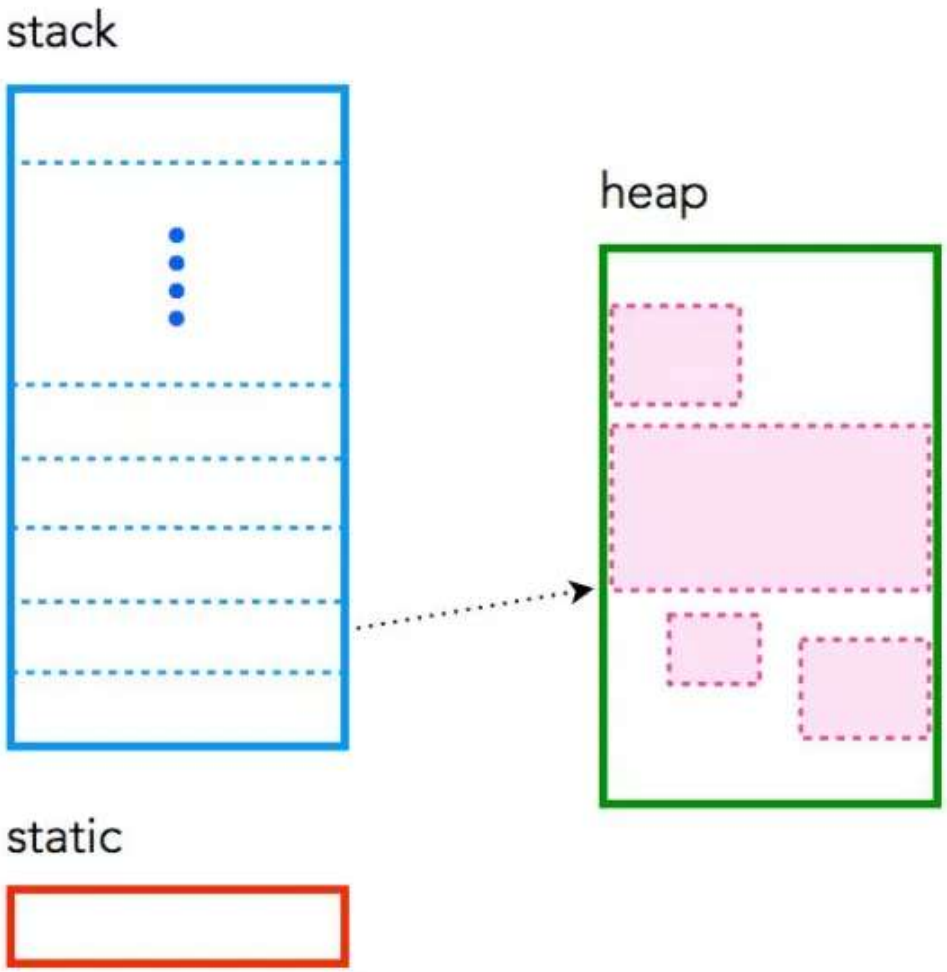
## Musings on programming

# Memory in C – the stack, the heap, and static

*07/12/201508/09/2018*

The great thing about C is that it is so intertwined with memory – and by that I mean that the programmer has quite a good understanding of "*what goes where*". C has three different pools of memory.

– **static**: global variable storage, permanent for the entire run of the program.
– **stack**: local variable storage (automatic, continuous memory).
– **heap**: dynamic storage (large pool of memory, not allocated in contiguous order).




(https://craftofcoding.files.wordpress.com/2015/12/stackmemory4.jpg)

## Static memory

Static memory persists throughout the entire life of the program, and is usually used to store things like *global* variables, or variables created with the **static** clause. For example:

```
int theforce;
```



On many systems this variable uses 4 bytes of memory. This memory can come from one of two places. If a variable is declared *outside* of a function, it is considered global, meaning it is accessible anywhere in the program. Global variables are **static,** and there is only one copy for the entire program. Inside a function the variable is allocated on the stack. It is also possible to force a variable to be static using the **static** clause. For example, the same variable created inside a function using the **static** clause would allow it to be stored in static memory.

```
static int theforce;
```

## Stack memory

The *stack* is used to store variables used on the inside of a function (including the **main()** function). It's a LIFO, "**L**ast-**I**n,-**F**irst-**O**ut", structure. Every time a function declares a new variable it is "pushed" onto the stack. Then when a function finishes running, all the variables associated with that function on the stack are deleted, and the memory they use is freed up. This leads to the "local" scope of function variables. The stack is a special region of memory, and automatically managed by the CPU – so you don't have to allocate or deallocate memory. Stack memory is divided into successive frames where each time a function is called, it allocates itself a fresh stack frame.

Note that there is generally a limit on the size of the stack – which can vary with the operating system (for example OSX currently has a default stack size of 8MB). If a program tries to put too much information on the stack, **stack overflow** will occur. Stack overflow happens when all the memory in the stack has been allocated, and further allocations begin overflowing into other sections of memory. Stack overflow also occurs in situations where recursion is incorrectly used.

A summary of the stack:

- the stack is managed by the CPU, there is no ability to modify it
- variables are allocated and freed automatically
- the stack it not limitless – most have an upper bound
- the stack grows and shrinks as variables are created and destroyed
- stack variables only exist whilst the function that created them exists

## Heap memory

The *heap* is the diametrical opposite of the stack. The *heap* is a large pool of memory that can be used dynamically – it is also known as the "free store". This is memory that is not automatically managed – you have to explicitly allocate (using functions such as malloc), and deallocate (e.g. free) the memory. Failure to free the memory when you are finished with it will result in what is known as a *memory leak* – memory that is still "being used", and not available to other processes. Unlike the stack, there are generally no restrictions on the size of the heap (or the variables it creates), other than the physical size of memory in the machine. Variables created on the heap are accessible anywhere in the program.

Oh, and heap memory requires you to use **pointers**.

A summary of the heap:

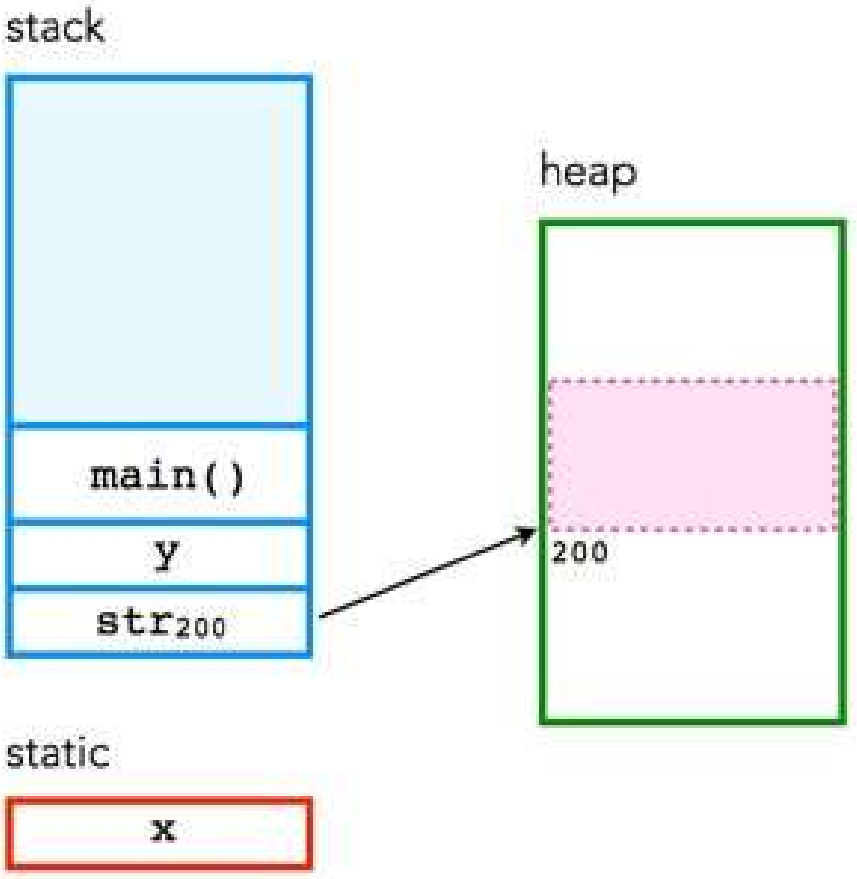
- the heap is managed by the programmer, the ability to modify it is somewhat boundless
- in C, variables are allocated and freed using functions like malloc() and free()
- the heap is large, and is usually limited by the physical memory available
- the heap requires pointers to access it

## An example of memory use

Consider the following example of a program containing all three forms of memory:

```
#include <stdio.h>
#include <stdlib.h>

int x;

int main(void)
{
    int y;
    char *str;

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```

The variable **x** is static storage, because of its global nature. Both **y** and **str** are dynamic stack storage which is deallocated when the program ends. The function **malloc()** is used to allocate 100 pieces of of dynamic heap storage, each the size of char, to **str**. Conversely, the function **free()**, deallocates the memory associated with **str**.




(https://craftofcoding.files.wordpress.com/2015/12/stackmemory31.jpg)



*Posted in: C, memory, programming* | *Tagged: automatic, heap, stack*