The Stack

Let's learn about the Stack and how it is useful in allocating memory for functions.

We'll cover the following

So far we have seen how to declare basic type variables such as `int`, `double`, etc, and complex types such as arrays and structs. The way we have been declaring them so far, with a syntax that is like other languages such as MATLAB, Python, etc, puts these variables on the **stack** in C.

What is the stack?

It's a special region of your computer's memory that stores temporary variables created by each function (including the `main()` function). 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.

Memory Management with Stack

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.

Stack variables are local in nature

A key to understanding the stack is the notion that **when a function exits**, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are **local** in nature. This is related to a concept we saw earlier known as **variable scope**, or local vs global variables. A common bug in C programming is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e. after that function has exited).

Limitation

Another feature of the stack to keep in mind, is that there is a limit (varies with OS) on the size of variables that can be stored on the stack. This is not the case for variables allocated on the **heap**.

To summarize the stack:

- the stack grows and shrinks as functions push and pop local variables
- there is no need to manage the memory yourself, variables are allocated and freed automatically
- the stack has size limits
- stack variables only exist while the function that created them, is running

Next, we'll examine the heap, which is the second type of memory structure available to us in C.