ESP_LEC_4

Stack Segment.

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.

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.

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).

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's 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.

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

example

```
#include <stdio.h>

double multiplyByTwo (double input) {
  double twice = input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int age = 30;
  double salary = 12345.67;
  double myList[3] = {1.2, 2.3, 3.4};

  printf("double    your    salary    is    %.3f\n",
multiplyByTwo(salary));

  return 0;
}
```