**STACK AND STACK FRAMES**

The stack grows and shrinks linearly as functions are called and return. For Linux on the x86-32 architecture (and on most other Linux and UNIX implementations), the stack resides at the high end of memory and grows downward (toward the heap). A special-purpose register, the stack pointer , tracks the current top of the stack. Each time a function is called, an additional frame is allocated on the stack, the stack pointer is updated to the address of the top. The program couter is also updated to the first line in the function. When the function returns, the stack frame is removed. NOTE: Even though the stack grows downward, we still call the growing end of the stack the top We will show you soon how this works.

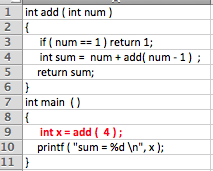
What is a stack frame :

Generally, a stack frame is a data structure that contains information about function that are called and currently executing. Information such as:

* Function arguments and local variables : In C these are referred to as automatic variables, since they are automatically created when a function is called. These variables also automatically disappear when the function returns (since the stack frame disappears), and this forms the primary semantic distinction between automatic and static (and global) variables: the latter have a permanent existence independent of the execution of functions. These arguments, local variables, the return address are stored in a self-referential structure called stack frame. The other name for stack frame is activation records. The address of the top of the stack is stored in a pointer called stack pointer. Assume each stack frame is of same size.
* Call linkage information : Each function uses certain CPU registers, such as the program counter (PC), which points to the next machine-language instruction to be executed. Each time one function calls another, a copy of these registers is saved in the called function’s stack frame so that when the function returns, the appropriate register values can be restored for the calling function. In the following examples, we use line numbers , instead of machine instructions, for our discussions as it is easier to follow.

Just to summarize : stack is a different data structure we use to implement function calls . Stack Pointer always points to the top of the stack , ie always stores the address of the top of the stack. Program counter is used to store the line numbers of your program ( ie the text area ).

Now, consider this program, we use recursion only for discussions purposes. The program finds the sum of all numbers <= 4. The stack pointer is NULL initially and program counter is in line # 7.



In brief , the flow of function calls would be

main : x = add ( 4 ) ; // Program counter at line # 9

add : sum = 4 + add ( 3 ) ; // Program counter at line # 4

add : sum = 4 + 3 + add ( 2 ) ; // Program counter at line # 4

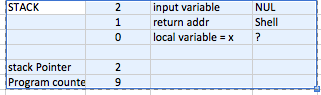
add : sum = 4 + 3 + 2 + add ( 1 ) ; // Program counter at line # 4

add : sum = 4 + 3 + 2 + 1 ; // Program counter at line # 3

add : return ( sum ) ; // Program counter at line 5

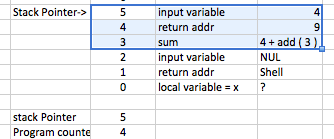
main : printf // Program counter at line 10

When the function is called at line # 9, the stack and the program counter looks like this.



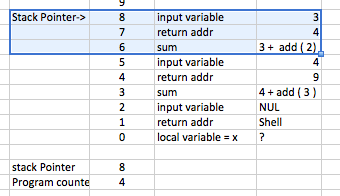
The stack pointer ( a pointer that points to the top of the stack) is at address = 2 , there is no input variable passed to main function, so it is NULL. The return address is to the shell and the value of the local variable x is unknown currently. Because the program is executing at line #9, PC value is 9.

When the execution (from line # 9) jumps to the function add, a new stack frame is allocated and placed on the top of the stack. The stack pointer now stores address= 5. Say, now the execution is at line #4 , the program counter is updated to = line 4 and stack pointer is still at address = 5 , stack looks like this.

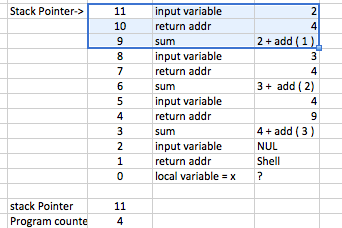


Here the return address = 9 is stored in the top stack frame. The input variable = 4 is stored in the stack frame. The local variable sum is getting updated, but it needs another call to function add ( a recursive call ) .

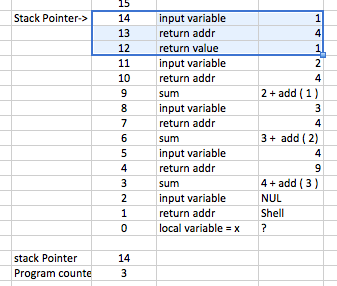
Next, add calls itself ( due to recursion ), the stack would like this



As you can see, a stack frame will be created and placed on top of the stack. In the next call to add, the stack would look like this

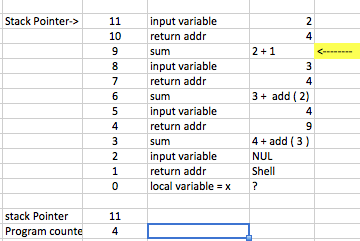


Finally, when function add ( 1 ) is called at line 4, a new stack frame is allocated and placed on top of the stack. the stack looks like this. The stack pointer is updated to 14. When the function is executing at line 3, the PC value is 3, the input variable is still 1, the return address is = 4 and the return value sum = 1.

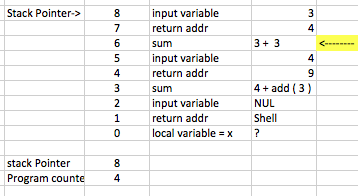


Take a note of the stack pointer and the program counter.

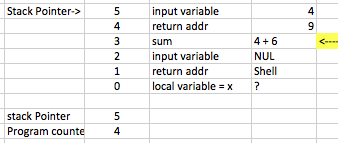
Now add ( 1 ) completes and returns the value of 1. When the add ( 1 ) returns, the stack frame is removed from the stack and the figure below shows the state of the frame.



The value of sum is now 3. Now we are in the add ( 2 ) function. This function now returns the value of sum = 3. The new stack frame looks like this:

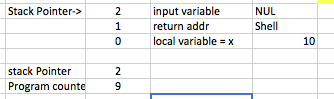


The value of sum is now 6. The function add ( 3 ) stores the value of sum = 6. it returns this value. Now the stack frame looks like this



Now we are in the function add ( 4 ) . The value of sum is now 10. This function returns this value to the caller.

Finally, the new stack frame would like this:



At last , we got the value of x to be 10 .

As you can see, the stack pointer and the program counter are used to manage how functions operate. We also showed how stack frames are used to execute functions and in particular recursion.