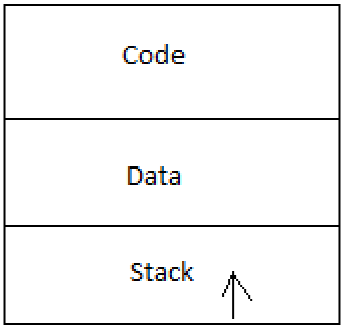
Stack and Local Variables

Let's start with this example:-

```
int globalVar;
void main()
{
    globalVar = 10;
}
```

In this program there is a global variable. As we have seen before, global variables are allocated in the data segment. What about local variables? The local variables are allocated from the stack. Once the method has completed its execution, these are de-allocated. The program structure with stack will look like -



The stack will grow from the higher address to lower address in the memory which is allocated for the stack region of the program. Let's take another example and see how the stack is used.

```
void fun()
{
    int locVar = 0;
    locVar++;
}
```

this will get translated into a code similar to this-

```
fun:
  stack_top -= 4
  stack_top[0] = 0
  stack_top[0] ++
  stack_top += 4
  return
```

Initially the stack_top was pointing to the top of the stack before the execution of the function. In the beginning of the function the stack_top is decremented by 4 bytes and space created by these 4 bytes used for the local variable locVar. And at the end of the function the stack_top again points to what it was before the execution of the function.
This example had only one local variable. If there are more local variables then the stack_top will decrease by such value so that all local variables will be accommodated.

NOTE: Number of bytes decremented may be more than actual required by sum of spaces required by all local variables. This may be done for two reasons:
(1) The architectural constraints,
(2) To optimize the code to run faster on some architecture. For example if the stack_top my be aligned to 16 bytes.

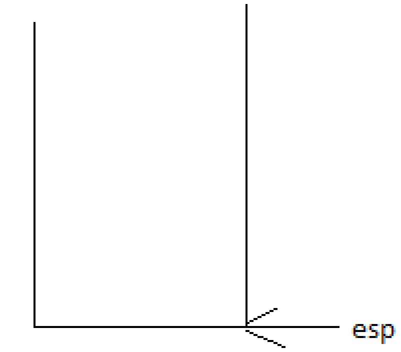
Let's see the actual assembly code which is generated by gcc:

```
fun:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $0, -4(%ebp)
    addl    $1, -4(%ebp)
    leave
    ret
```

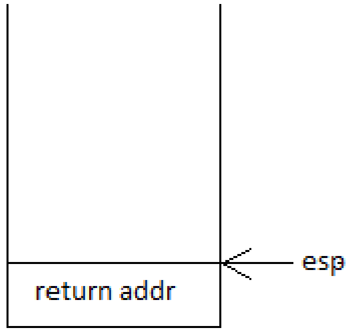
Comments on the generated code:

```
# starting of a function, called fun
fun:

# push the current ebp register. This will actually save the ebp register on the stack.
# This is done because the ebp register will be modified. When the function code has
# finished, then the value of the ebp register will be restored from the stack. So the at
# the end of the function the value of ebp register will remain as this was before
# calling this function.
    pushl   %ebp

# Now move the value of current stack pointer to ebp.
    movl    %esp, %ebp

# Decrement the current stack pointer. The space created by this will be used for
# the local variables which are declared inside function. In the above example,
# there is only one local variable which needs 4 bytes but the stack is decremented
# by 16 bytes. As stated earlier this is done to make the esp register aligned on
# 16 bytes for perf reason. The other spaces will be unused while the the function is
# executing. Suppose there are two local variables of size 4 bytes each, even in that
# case the esp will decremented by 16 bytes for the same alignment reason
# but in this there will be less unused space created.
    subl    $16, %esp

# -4(%ebp) the memory location for the local variable locVar; so set the locVar to zero
    movl    $0, -4(%ebp)

# increment is value of the locVar
    addl    $1, -4(%ebp)

# Restore the ebp register from the stack and then return. the leave
# instruction does two things
# move %ebp %esp and pop %ebp
    leave
    ret
```

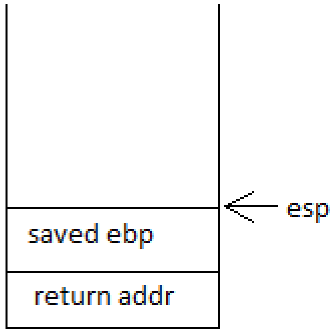
Here is a diagram describing the above code for local variables:



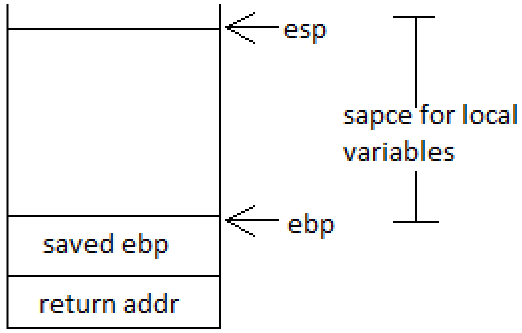
Before calling the function



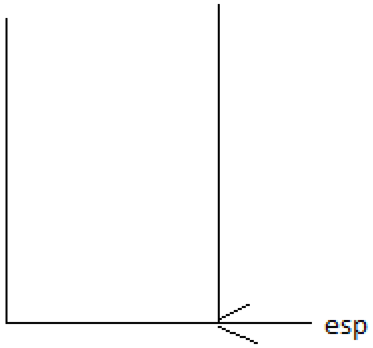
just after call instruction executed



save ebp



after creating space for local variables



after returing from function

<u>QUESTION:</u> Can you visualize this - what will happen if a pointer of local variable is returned from a function? Make a diagram to explain this to yourself.

‹ Program Structure (/cin/programstructure.html)
up (/cin/cin.html)
Translation of Arithmetic Operations › (/cin/arithmeticop.html)