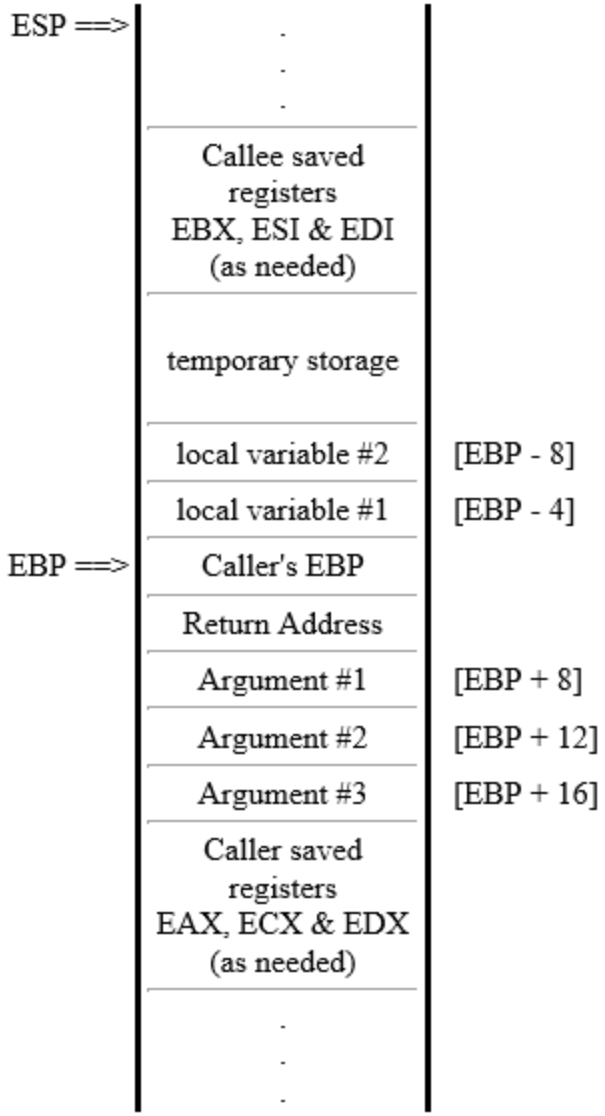
Register Usage in Function Calls



This would be the contents of the stack if we have a function **Myfunction** with the prototype:
int **Myfunction** (int arg1, int arg2, int arg3) ;

and **Myfunction** has two local integer variables.. The stack would look like this if say the main function called **Myfunction** and control of the program is still inside the function **Myfunction** . In this situation, **main** is the "**caller**" and **Myfunction** is the "**callee**".

The ESP register is being used by **Myfunction** to point to the top of the stack. The EBP register is acting as a "base pointer".

The arguments passed by main to **Myfunction** and the local variables in **Myfunction** can all be referenced as an offset from the base pointer.

The convention used here is that the **callee** is allowed to mess up the values of the **EAX, ECX and EDX** registers before returning. So, if the caller wants to preserve the values of EAX, ECX and EDX, the caller must explicitly save them on the stack before making the subroutine call
The **callee** must restore the values of the EBX, ESI and EDI registers. If the **callee** makes changes to these registers, the **callee** must save the affected registers on the stack and restore the original values before returning.

Parameters passed to Myfunction are pushed on the stack. The last argument is pushed first so in the end the first argument is on top. Local variables declared in Myfunction as well as temporary variables are all stored on the stack.

Return values of 4 bytes or less are stored in the EAX register. If a return value with more than 4 bytes is needed, then the caller passes an "extra" first argument to the callee. This extra argument is address of the location where the return value should be stored. I.e., in C parlance the function call:

x = Myfunction (a, b, c) ;

is transformed into the call:

Myfunction (&x, a, b, c) ;

Note that this only happens for function calls that return more than 4 bytes.

The caller's actions before the function call

The caller is the main function and is about to call a function **Myfunction** .

Before the function call, main is using the ESP and EBP registers for its own stack frame.

First, main pushes the contents of the registers EAX, ECX and EDX onto the stack.

This is an optional step and is taken only if the contents of these 3 registers need to be preserved.

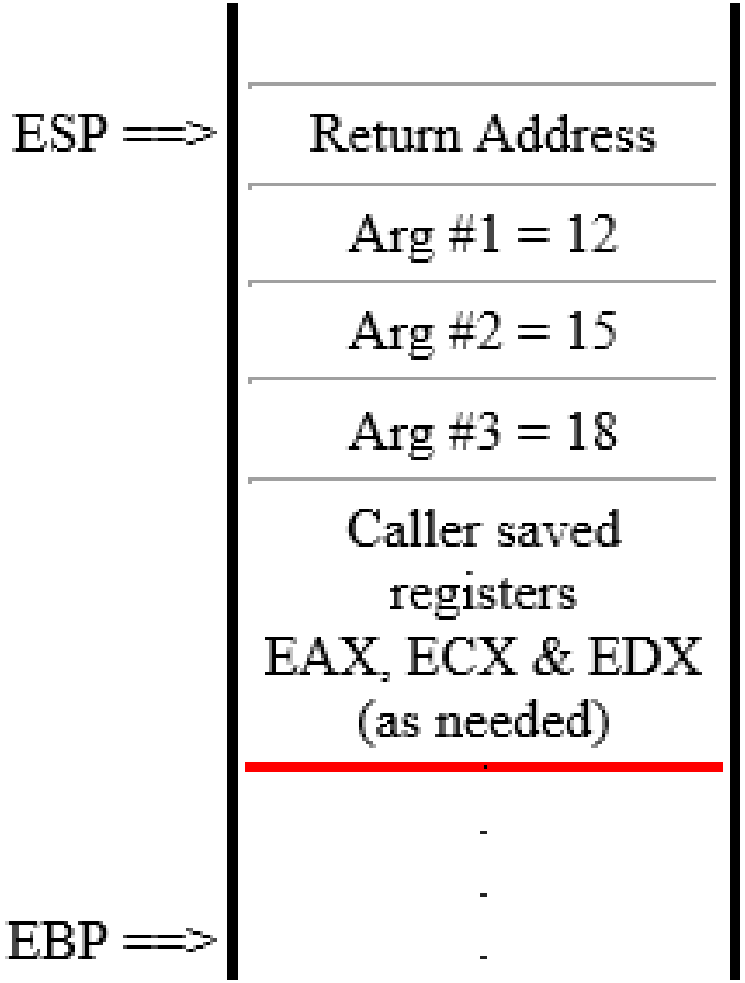
Next, main pushes the arguments for Myfunction one at a time, last argument first onto the stack.

For example, if the function call is: a = Myfunction (12, 15, 18) ; The assembly language instructions might be(what we expect are):

```
push dword 18
push dword 15
push dword 12
```

Finally, main can issue the subroutine call instruction: call Myfunction

When the call instruction is executed, the contents of the EIP register is pushed onto the stack.
Since the EIP register is pointing to the next instruction in main, the effect is that the return address is now at the top of the stack. After the call instruction, the next execution cycle begins at the label named Myfunction .



This shows the contents of the stack after the call instruction. The red line here indicates the top of the stack prior to the instructions that initiated the function call process. We will see that after the entire function call has finished, the top of the stack will be restored to this position.

The callee's actions after function call

When the function **Myfunction** , the callee, gets control of the program, it must do 3 things:

1) set up its own stack frame,
2) allocate space for local storage and
3) save the contents of the registers EBX, ESI and EDI as needed.

So, first Myfunction must set up its own stack frame.

The EBP register is currently pointing at a location in main's stack frame.

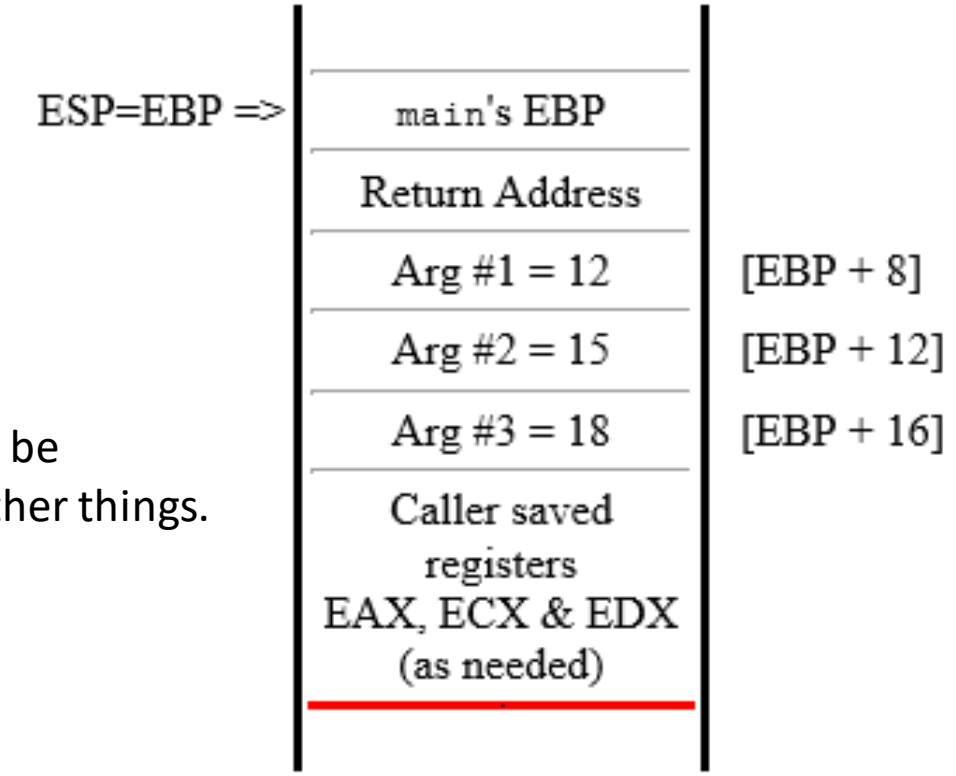
This value must be preserved. EBP is pushed onto the stack.

Then the contents of ESP is transferred to EBP. This allows the arguments to be referenced as an offset from EBP and frees up the stack register ESP to do other things.

Thus, just about all C functions begin with the two instructions:

```
push ebp
mov ebp, esp
```

The resulting stack is shown here.



Notice that in this scheme the address of the first argument is 8 plus EBP, since main's EBP and the return address each takes 4 bytes on the stack.

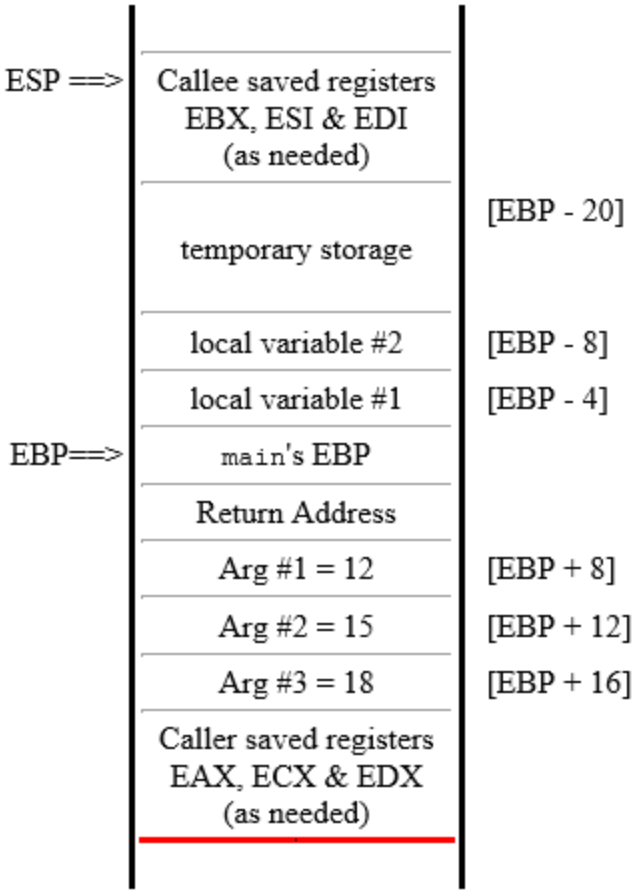
In the next step, **Myfunction** must allocate space for its local variables.

It must also allocate space for any temporary storage it might need. Here we are assuming that Myfunction has 2 local variables of type int (4 bytes each) and needs an additional 12 bytes of temporary storage. The 20 bytes needed can be allocated simply by subtracting 20 from the stack pointer: sub esp, 20

The local variables and temporary storage can now be referenced as an offset from the base pointer EBP.

Finally, Myfunction must preserve the contents of the EBX, ESI and EDI registers if it uses these.

The body of the function Myfunction can now be executed.



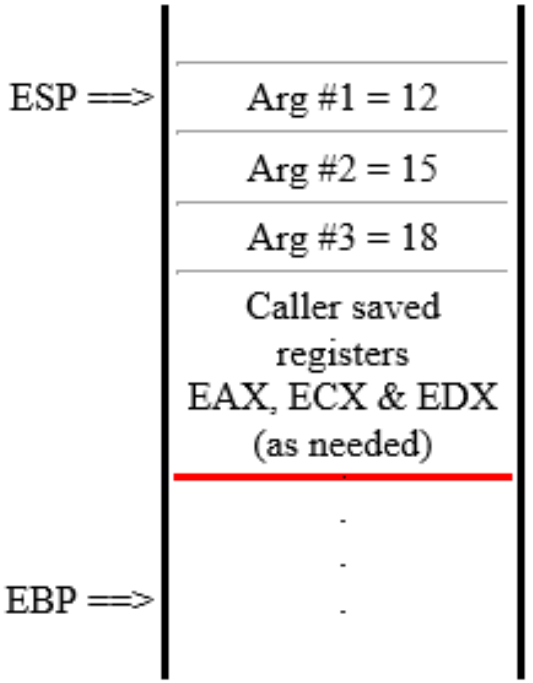
The callee's actions before returning

Before returning control to the caller, the callee **Myfunction** must first make arrangements for the return value to be stored in the EAX register.

Secondly, **Myfunction** must restore the values of the EBX, ESI and EDI registers. If these registers were modified, we pushed their original values onto the stack at the beginning of **Myfunction** .

The original values can be popped off the stack, if the ESP register is pointing to the correct location. So, it is important that we do not lose track of the stack pointer ESP during the execution of the body of **Myfunction** ---
i.e., the number of pushes and pops must be balanced. (otherwise **SEGMENTATION FAULT**)
After these two steps we no longer need the local variables and temporary storage for **Myfunction** . We can take down the stack frame with these instructions:

mov esp, ebp
pop ebp



The return instruction can now be executed. This pops the return address off the stack and stores it in the EIP register. The i386 instruction set has an instruction "leave" which does exactly the same thing as the mov and pop instructions above. Thus, it is very typical for C functions to end with the instructions:

leave
ret

The caller's actions after returning

After control of the program returns to the caller (which is main in our example), the stack is as shown in previous slide.

In this situation, the arguments passed to **Myfunction** is usually not needed anymore. We can pop all 3 arguments off the stack simultaneously by adding 12 (= 3 times 4 bytes) to the stack pointer:

add esp, 12

The caller main should then save the return value which was placed in EAX in some appropriate location.

For example if the return value is to be assigned to a variable, then the contents of EAX could be moved to the variable's memory location now.

Finally, the main function can pop the values of the EAX, ECX and EDX registers if their values were preserved on the stack prior to the function call.

This puts the top of the stack at the exact same position as before we started this entire function call process.