Popular software interfaces

by Yannis Kiourktsoglou

April 2013

1. Calling conventions

1.1. Introduction

When calling a function that accepts passing parameters (which is the case in most function calls) some questions regarding the techniques used for passing the parameters must have answers available to the caller of each function. And because these techniques which are called **calling conventions** are the same throughout a programming language, this means that the developer must know how the language used treats the passing parameters and how the functions called probably part of a library expect to receive these parameters.

Such questions are :

- where are the parameters copied by the caller so that the called function finds them and uses them properly
- what is the format (mainly the size) of these parameters
- in what order are the parameters passed from the caller to the callee
- are there other values that must be also passed by the caller or returned by the callee and therefore are there registers in the CPU that must be protected
- who will clean up the memory area used for this exchange of information

In this presentation we will deal with the two most popular calling conventions:

a. **cdecl** : meaning c declaration and obviously originating from C language
b. **stdcall** : meaning standard call , first used by Pascal

Having completed the study of these two traditional conventions, the reader can easily deal with other – newer for instance x64 conventions if needed.

From the above 5 questions, the first 3 have exactly the same answer for both calling conventions examined here.

Before describing these answers first a little terminology :

- A caller calls a function - the called function is the callee
- The callee expects to receive arguments (passing parameters) and returns values using a calling convention.
- The caller must know what calling convention the callee uses and prepare the values properly for the called function to use. Calling a function with the wrong calling convention is a disaster.

Regarding the two presented conventions here:

1.1.1. they both refer to the order of scalar parameters (single values) that have a size equal to the size of each value on the stack, which logically is equal to the address bus since the stack is a place where primarily addresses are temporarily stored. (read below about the Stack)

1.1.2. the parameters are copied on the stack (pushed) and the order that this is done is from right to left, for example the function call

```
foo (a,b,c)
```

will first push the value of c , then the value of b and finally the value of a.

Reviewing the above : The answers to the first 3 questions are for both conventions:

- passing parameters are pushed on the stack
- the size of each parameter is equal to one pushed value but in case a complex argument must be pushed every individual part of such an argument is treated as every other parameter
- the order that the parameters or individual parts of larger parameters are passed to the stack is from right to left

What's left to distinguish between the two calling conventions is :

- preserving registers that are not guaranteed to remain unchanged after a call
- who does the clean-up ? the caller or the callee

It is time to describe the stack so that all the above are made clearer to the reader.

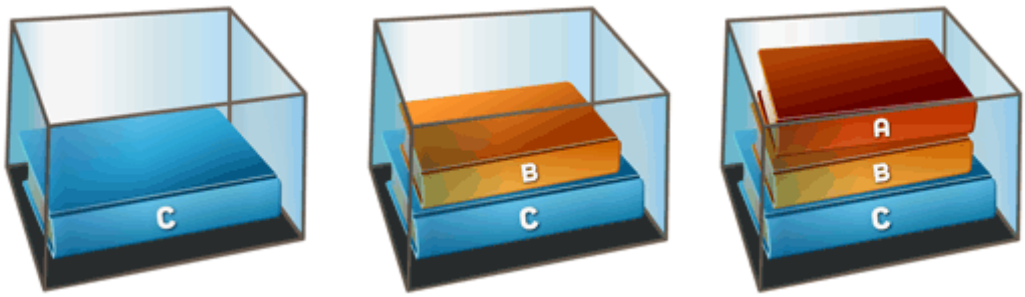
1.2 The Stack

1.2.1 The concept

The Stack is an idea - a mechanism better that applies the idea based on the method called LIFO (Last In First Out) for storing values. Some people think that the Stack is a specific memory area pre-defined by the hardware architecture. No ! this is wrong.

What does LIFO mean? It means that if I put items in a basket one over the other then when the time comes to take out values from the basket, the value I will pick first will be the last one sent to the basket : **Last In - First Out** .

STEP 1 STEP 2 STEP 3



The above pictures demonstrate the LIFO principle combined with the previously mentioned expression "right to left". Let's introduce the assembly language keywords **push** and **pop** at this point. If A, B and C are the arguments then passing them from right to left means:

First push C (Step 1)
Then push B (Step 2)
Finally push A (Step 3).

This is what the caller must do. Now the called function "knows" that on the top of the stack is A. The first item that will be **popped** is A

1.2.2 The mechanism

So the question is how can the LIFO logic be supported by a mechanism of the CPU. The answer is not very difficult but it doesn't look like the concept described before.
Here is an attempt to explain the mechanism of the stack in simple statements.

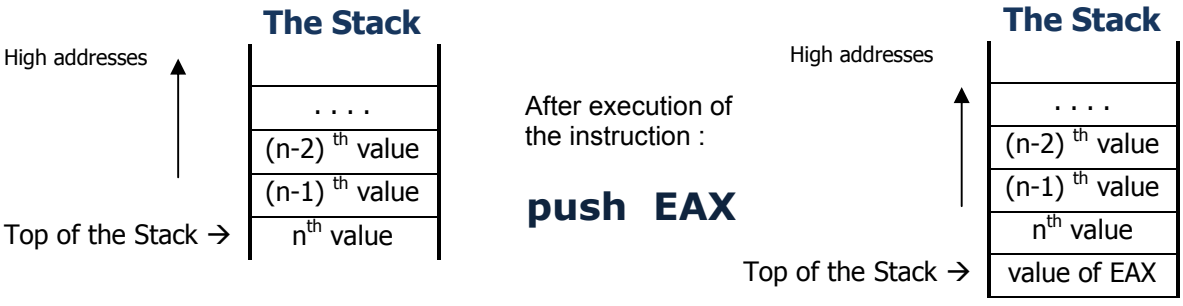
a. The Stack is NOT a component of the processor nor a special memory area inside the RAM or anywhere else in the hardware. Instead the **Stack Pointer** is. The Stack Pointer is a register in the CPU that has the size of the address bus, because it always contains an address. What address ? The Top of the Stack.
b. The program "sets up a stack" : it allocates a memory area from the memory available for the program for example 50 K of memory (sometimes much less – don't have the wrong impression that the stack is a huge memory area) and assigns the address of this area to the Stack Pointer register (SP).
c. using the CPU instructions PUSH and POP the programs add values on the stack or extract values from it : they push and pop values.
The PUSH and POP instructions have an operand which is the source or destination respectively of the value transferred to/from the Stack

PUSH reg	
a.	update the SP to point at the next available location over the stack and then
b.	copy the value of the reg to the (new) Top of the Stack using the SP value

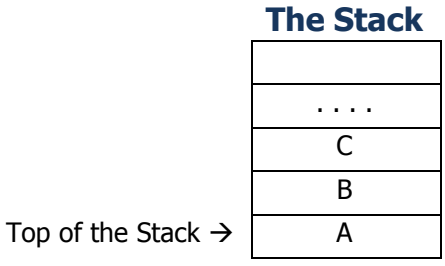
POP reg	
a.	copy the value at the Top of the Stack to the reg using the SP value
b.	update the SP in order to "lower" the stack by one value

d. there is however a very important "secret" in the implementation of the stack mechanism with the use of the SP :
The Top of the Stack is at the . . . bottom !!

(there are very good reasons why this happens but they are beyond the scope of this section here).



Going back to the A,B,C parameters example, the called function "knows" that when called the stack will look something like this :

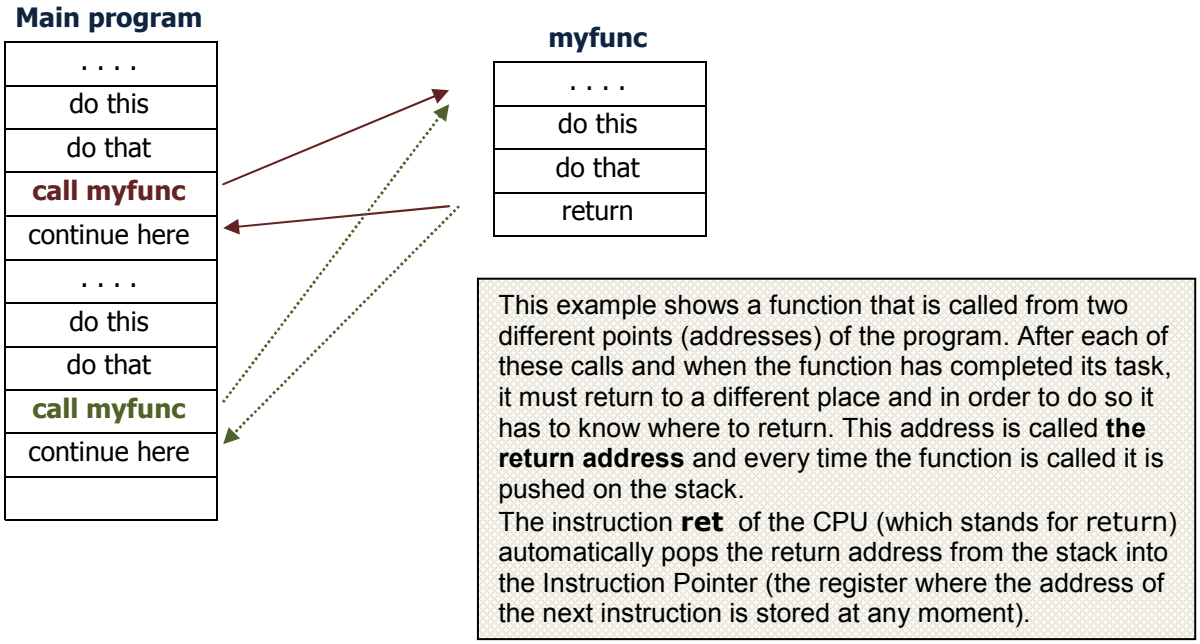


and this means that if we POP a value from the stack then this will be the values of A.

1.2.3 Return addresses

Unfortunately, this is not all. The callee must also take into consideration that ***when a function is called then the address where it must return when the call terminates, is pushed on the stack***.

Note: It is understood that functions are called from several points inside a program and therefore each function must be able to return back to where it was called from to resume execution at the instruction next to the "call" instruction.



This example shows a function that is called from two different points (addresses) of the program. After each of these calls and when the function has completed its task, it must return to a different place and in order to do so it has to know where to return. This address is called **the return address** and every time the function is called it is pushed on the stack.
The instruction **ret** of the CPU (which stands for return) automatically pops the return address from the stack into the Instruction Pointer (the register where the address of the next instruction is stored at any moment).

1.3 stdcall vs. cdecl

1.3.1 Function calls

In the previous section we saw how the stack operates for the hardware point of view.
Reviewing the stack mechanisms , there are two main situations where the processor uses the Stack :

Instructions	Purpose
PUSH / POP	to temporarily store a value on the stack and later take it back
CALL / RET	to remember the return address when branching to a functions

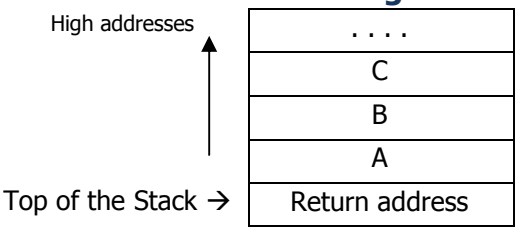
Now we will examine the methods that the various languages use in order to make possible the transfer of values from a calling module to a function that it calls, knowing that the function expects to find data somehow, somewhere.
In other words the previous section 1.2 discussed the hardware specs (and these cannot be violated) whereas in this section we talk about software techniques.

The calling conventions which are presented in this chapter both "agree" in how they pass the parameters when they call a function :

- Passing parameters are passed from right to left.

Going back to the example of 1.1.2 , now that we know how the Stack behaves, we can tell that after calling the function of the example the stack will look like this :

The Stack after calling foo



Two comments :

- The Top of stack is continuously stored in the Stack Pointer register (ESP for Intel notation – historically SP was the 16-bit Stack Pointer and ESP is the 32-bit SP)
- The size of every entry in the stack is NOT one byte but equal to the size of the address bus. Let's assume for this presentation that this size is 32 bits = 4 bytes

Conclusion

- the return address is @ mem.address SP , this is written in assembly language as [ESP]
- value of A is at [ESP+4]
- value of B is at [ESP+8]
- value of C is at [ESP+12]

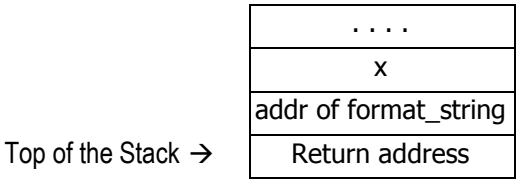
Under these conventions therefore, the functions "know" where the passing parameters are.

An advantage of this "right to left" method is that functions that receive variable number of parameters can manipulate them easily because always the first passing parameter is at [ESP+4] for instance in C language :

```
printf("x is %d\n" , x);
```

has two passing parameters

The Stack after calling this printf

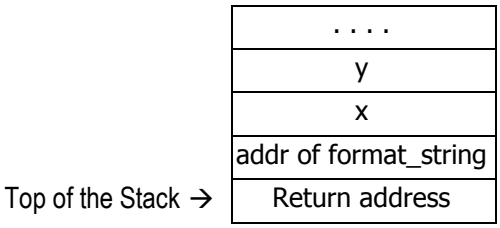


whereas

```
printf("x is %d and y is %d\n" , x, y);
```

has three . However the *format_string* (first passing parameter) is always pushed last on the stack and the rest is obvious:

The Stack after calling this printf



- The called function knows where to find the formatstring
  - right over the return address !

1.3.2 A nice practice : Taking precautions for a "happy" return

In 1.2.3 it was mentioned that the function in order to pass the control back to the next instruction after the call instruction of the caller (how to return) will use the return address on the stack. To make this clearer, when the instruction **RET** (return) is encountered in the code of the function, it instructs the program to resume execution after the call instruction.
Then the function uses the top value of the stack as the return address.
This means that if the ESP has somehow changed and does NOT point to the return address then a disaster expects us.
Conclusion :

> ***It is the programmer's duty to see that when the RET statement in a function is to be executed, then the top of the stack must contain the return address.***

One practice could be *"as many POPs , that many PUSHes"* .

But Intel (and Intel compatible) processors, have provided a better solution in their design.
The Base Pointer is a special register abbreviated as EBP and is described as "the redundant stack pointer" . It is as if we have a second Stack Pointer. And how do we use it ?

As soon as a function is called the status in the Stack is as shown in the diagrams of 1.3.1.
Knowing that inside the function there will be **pushes** and **pops** and the ESP value may be lost causing the disaster mentioned right before, we wish to save the ESP value in the EBP register. But if we do so then we will lose the EBP value !!!!

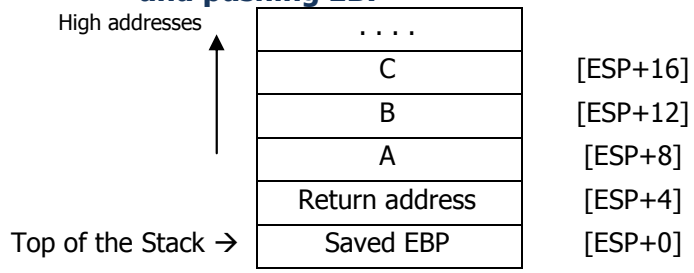
What can we do ?

PUSH the current value of EBP on the stack and then save the ESP in the EBP register.
In this manner when the time comes (to terminate the function)

- we will put back the value from EBP to ESP
- we will then POP the EBP to restore the original value of EBP
- and we ll be ready to return

Right after pushing EBP , the picture will be the following :

The Stack after calling foo and pushing EBP



and of course EBP = ESP ➔ EBP points to the top of the Stack

The only thing that changed is that due to pushing EBP, now the address of A (which is the first passing parameter) will be [ESP+8] (=[EBP+8]) and not [ESP+4] (=[EBP+4]).

It is not in the scope of this discussion but let's mention that EBP is now the base reference for a. the ESP value, b. the passing parameters and c. the local variables that all compilers but programmers as well follow the practice to store under the stack (EBP-4,EBP-8 etc.)

1.3.3 The return procedure – before the RET

We are not going to describe what exactly happens to the various registers of our processor under the two conventions under discussion, because this would lead us to a specialized and somehow complicated analysis of the registers and the instruction set of a processor and this is NOT our intention.

We will only mention that the two conventions include also those register that can be modified and mainly the registers where the return value will be placed. Keep in mind that although a function can take many parameters it always returns one value. Usually this value is in register EAX (32 bit) and if it happens that the return value must be a 64-bit value then two registers are used → EAX and EDX .
Note : in C language we call **void functions** those functions that do not return a result. It would be more precise to say that :

void functions are those functions that do not return a meaningful result

because there will always be a value in the register(s) mentioned before.

That's all. More details will not be presented here.

1.3.4 The return procedure – after the RET

It was already mentioned that both conventions pass the parameters from right to left.
So up to here there is no difference between them.

And because there has to be some difference (otherwise there would only be one calling convention) here it is :

After POPing the EBP (see previous paragraph) we are ready to **return** because the Stack pointer contains the return address.
And that what the program must do at this point: Return !

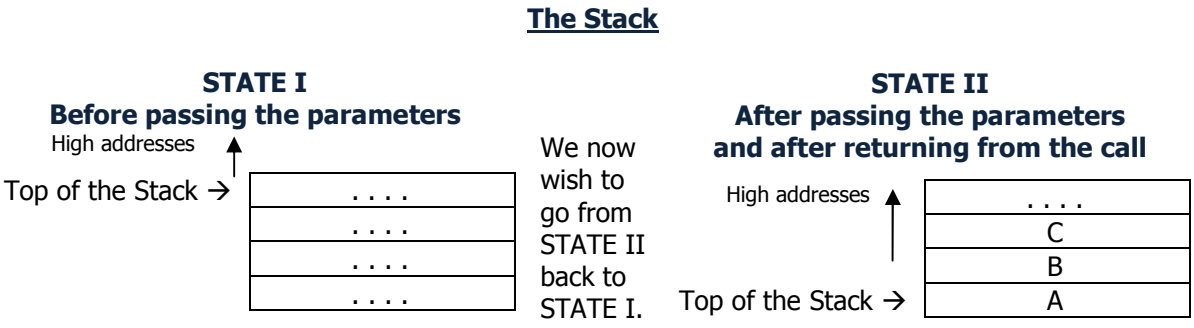
Let's now try to see the situation from the point of view of the caller and not of the callee.
The caller

- pushed the passing parameters and
- called the function

but now that the control has returned back to the instruction following the **CALL** instruction (see 1.2.3) the passing parameters remain on the stack !

Someone has to clean up the stack !

Let's try and illustrate this, using the foo example :



All that must be done is (in this example)

- Add 12 to the value of ESP since there are 3 parameters and each one occupies 4 bytes (3*4=12). To generalize this, if there are n parameters then we must add n*4 to ESP.

That's all ! the answer to the question *"WHAT must be done ?"* is easy.

Things however become interesting and a little complicated when it comes to the question *"WHO is going to do it ?"* . Because there are two candidates to undertake this duty :

a. the caller
b. the callee

and this is exactly the **difference between stdcall and cdecl**.

Inter supports a variation of the return instruction RET which takes an integer constant as an operand:

RET n

This instruction "says" : *Return and immediately add n to the ESP.*

In other words using this facility the function can adjust the ESP and therefore cleanup the Stack.

For example, using this method the last (machine language) instruction of the **foo(...)** function that takes three passing parameters would be

RET 12

This is the stdcall calling convention :

stdcall	The stack is cleaned up by the callee (the function).

Using the stdcall convention, right after the return instruction, the stack is in STATE I and there is nothing that the caller must do.

And to make a long story short under the cdecl calling convention :

cdecl	The stack is cleaned up by the caller

How is this done under cdecl ?

After returning from the function call an instruction like this is executed

ADD ESP,12

which means *"Add 12 to the value of ESP"* . In this manner we will go back to STATE I from STATE II (see above) and so the Stack Pointer will point exactly where it should.

1.3.5 Pro's and con's

- An advantage of stdcall is that if a program calls the same function many times, then it does not have to adjust tha stack every time that the function returns, because the adjustment is done by the function.
- A serious disadvantage of stdcall however is that if a function is called with variable number of passing parameters (line the printf example of 1.3.1) then the function will do an incorrect stack adjustment which of course will be fatal. This means that we cannot call functions with different numbers of parameters. But additionally this will not prevent an accident in case a function for some reason is called with an incorrect number of parameters.

A "philosophical" discussion regarding which one is better has no meaning however, because every language operates under the one or the other calling convention and our job is not to criticize but to be aware of which language does what and take our measures.
For example in C / C++ there is a keyword __stdcall that allows us to write functions that despite the cdecl nature of the language will use the stdcall technique. As long as we know it and we take the right measures, everything will go fine.

1.4 An example of cdecl

1.4.1. The code

Here is a very simple C program. You can ignore the details in the what_time function.
This function puts the time in hh:mm:ss format into the string passed as passing parameter.
In the example the main code calls the what_time with a pasing parameter expecting that on return string s will contain the current time in the mentioned format .

```
#include <stdio.h>
#include <time.h>

void what_time(char strTime[])
{ struct tm *tp;
  time_t t;
    time(&t);
    tp=localtime(&t);
    sprintf(strTime,"%02.2d:%02.2d:%02.2d",tp->tm_hour,tp->tm_min,tp->tm_sec);
}
void main()
{ static char s[100];
       what_time(s);
       printf("The time is %s\n",s);

}
```

The following code is a combination of the c code and assembly while the program was debugged. After each C statement is the code in assembly (machine language) and shows us how the compiler manages the calls to functions using the cdecl convention.

1.4.2 Call with one passing parameter

Let's first examine the simple case : call the function what_time with one passing parameter. There are only there machine language instruction there.

a. `011D1E1E  push        offset s (11D74C0h)`
Push the address of s onto the stack

b. `011D1E23  call        what_time (11D11D6h)`
Call the function

c. `add         esp,4`
Adjust the stack by adding 4 (size of one passing parameter in bytes). Under the cdecl convention the caller must adjust the stack ; the function won't do it (see 1.3.4) .

```
 . . .
      what_time(s);
011D1E1E  push        offset s (11D74C0h)
011D1E23  call        what_time (11D11D6h)
011D1E28  add         esp,4     <-------   4 bytes x 1 value on the stack
      printf("The time is %s\n",s);
011D1E2B  mov         esi,esp
011D1E2D  push        offset s (11D74C0h)
011D1E32  push        offset string "The time is %s\n" (11D5810h)
011D1E37  call        dword ptr [__imp__printf (11D82B0h)]
011D1E3D  add         esp,8     <-------   4 bytes x 2 values on the stack
. . .
```

References

1. MSDN Argument passing and naming conventions
http://msdn.microsoft.com/en-us/library/984x0h58.aspx

2. Agner Fog. Technical University of Denmark : Calling conventions for different C++ compilers and operating systems http://www.agner.org/optimize/calling_conventions.pdf