SubRoutine Calls

Introduction to Subroutine Calls

This chapter covers the fundamental structure of subroutine calls, with a focus on the runtime stack. Subroutine calls are common in C and C++ programming, and debugging these calls can require an understanding of the runtime stack.

In C and C++, subroutines are referred to as **functions**, while in Java, they are known as **methods.** In MASM, they are termed **procedures.**

Values passed to a subroutine by a calling program are termed arguments. However, once these values are received by the called **sub-routine**, they become **parameters**.

Stack frames are used to manage subroutine calls. A **stack frame** is a region of memory on the runtime stack that is used to store the subroutine's local variables and parameters.

- Subroutine calls are a fundamental part of low-level programming.
- The runtime stack is used to manage subroutine calls.
- **Arguments** passed to a subroutine **become parameters** within the subroutine.
- Stack frames are used to store local variables and parameters for subroutines.

==========

Stack Frames

==========

In this section, we'll delve into the concept of stack frames, specifically focusing on stack parameters.

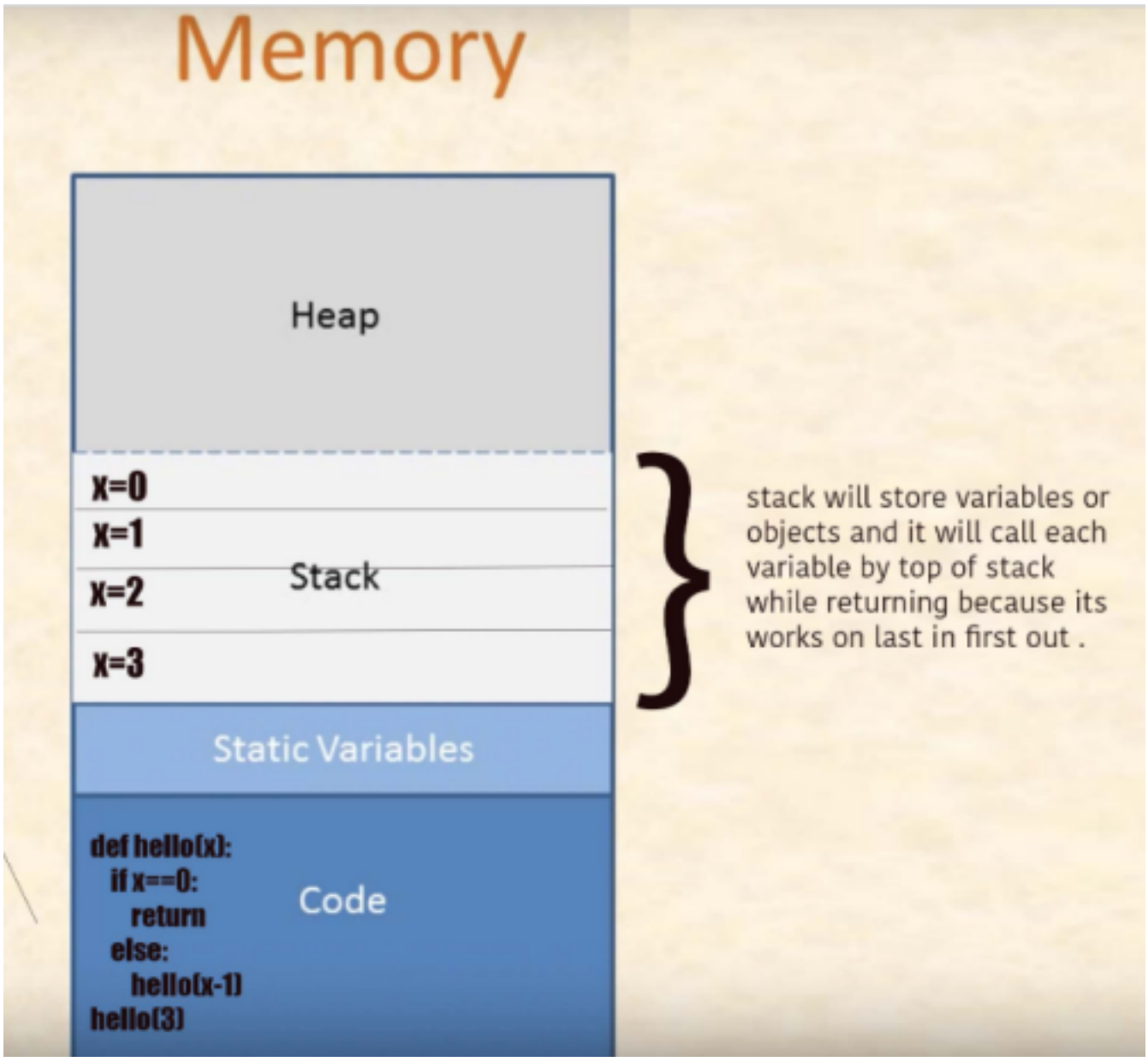
Stack Parameters

In 32-bit mode, stack parameters are the norm for Windows API functions. In 64-bit mode, Windows functions receive a combination of both register and stack parameters.

To pass a parameter to a subroutine on the stack, the **caller function pushes** the parameter onto the stack before calling the subroutine. The subroutine then accesses the parameter by using the stack pointer register.

The Anatomy of a Stack Frame

A **stack frame**, often referred to as an **activation record**, is a designated area on the stack used for various purposes.



It serves as the container for passed arguments, the subroutine return address, local variables, and saved registers. The construction of a stack frame typically involves the following sequential steps:

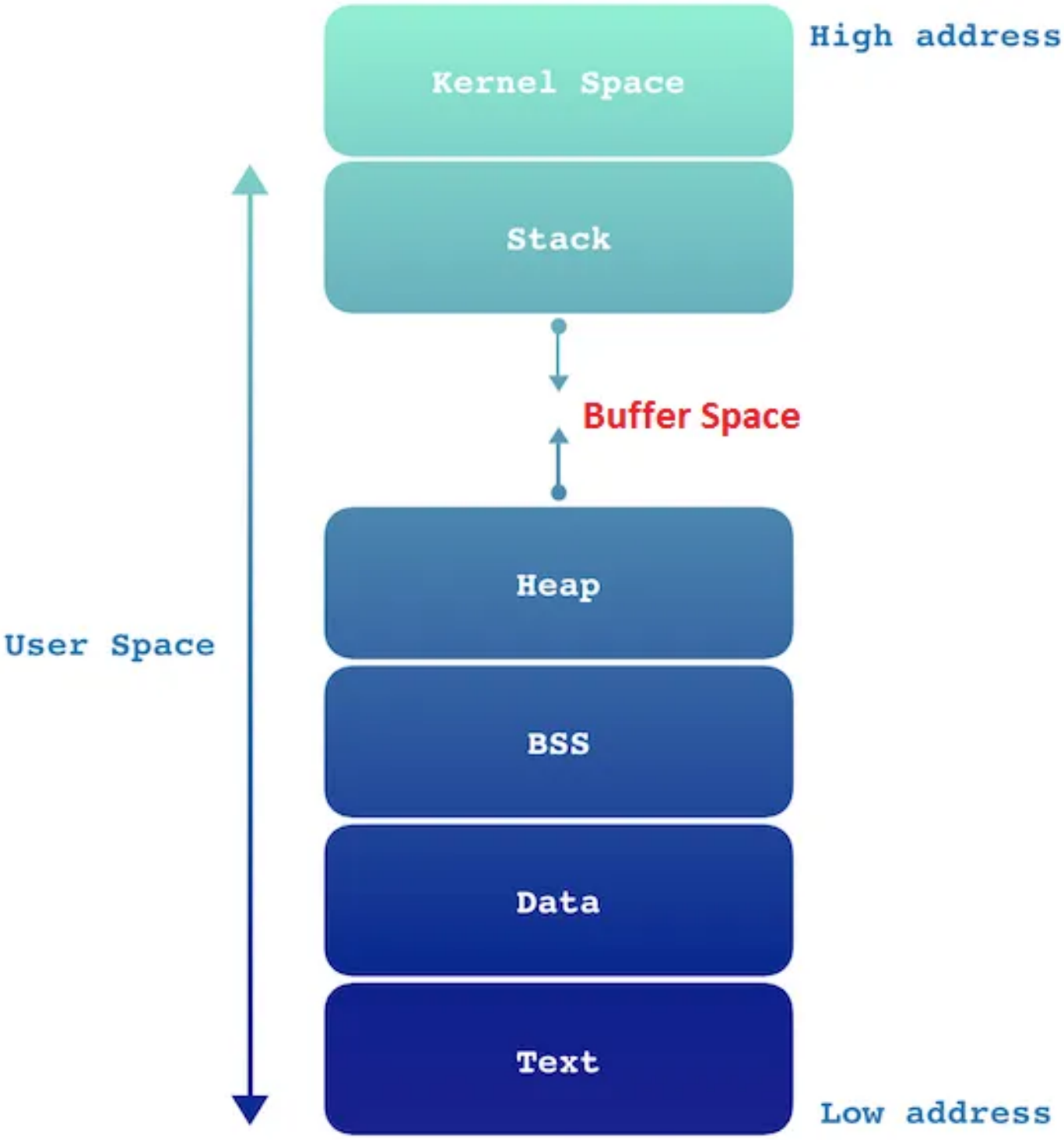
Passed arguments, if any, are pushed onto the stack.

As the subroutine begins its execution, the **Extended Base Pointer (EBP)** is pushed onto the stack.

EBP is set equal to the value of the **Stack Pointer (ESP).**

From this point onward, **EBP** acts as a fundamental **reference point** for all the subroutine parameters.

If there are local variables, the **Stack Pointer (ESP)** is **decremented** to allocate space for these variables on the stack. **We said stack pointer starts from highest memory addresses, getting decremented as long as parameters and local variables are being pushed onto the stack. Pops usually increment the stack pointer.** So, stack grows downwards.



If any registers need to be preserved, they are pushed onto the stack. The structure and organization of a stack frame can be heavily influenced by a **program's memory model** and its **chosen argument passing convention.**

Understanding the concept of passing arguments on the stack is of paramount importance. This is because nearly all high-level programming languages rely on this method.

For instance, when calling functions in the 32-bit Windows Application Programming Interface (API), you'll find it essential to

pass arguments on the stack.

However, as you delve into 64-bit programming, you'll encounter a different parameter passing convention, which we will explore in detail in Chapters ahead.

==========

Calls and stack:

When "Jackie" (an external procedure) calls "Rennex" (an internal procedure) in MASM:

• When "Jackie" calls "Rennex," it's Jackie who pushes Rennex's return address onto the stack.

• This return address points to the location in "Jackie" where execution should resume after "Rennex" completes its tasks.

• So, it is "Jackie" who takes care of preserving the return address for "Rennex."

• When "Rennex" finishes executing and reaches the point where it needs to return, it uses this **saved return address** to determine where it should return to, which, in this case, is the location within "Jackie" where the call to "Rennex" occurred.

• So, Jackie, as the calling procedure, takes responsibility for saving and restoring the return address when it calls Rennex.

• Rennex uses this saved return address to correctly return control to Jackie once its execution is complete.

• After the execution of "Rennex" is complete, it's typically the responsibility of the calling procedure, in this case, "Jackie," to manage the stack. Specifically, "Jackie" needs to issue a POP instruction to remove the return address from the stack.

Here's how it works:

• "Jackie" calls "Rennex" and pushes Rennex's return address onto the stack.

• "Rennex" executes and reaches its return point**(ret instruction)**. At this stage, Rennex uses the saved return address to determine where to transfer control back, which is the location within "Jackie" where the call to "Rennex" occurred.

• After Rennex's execution, control returns to "Jackie." Now, it's "Jackie's" responsibility to pop the return address from the stack.

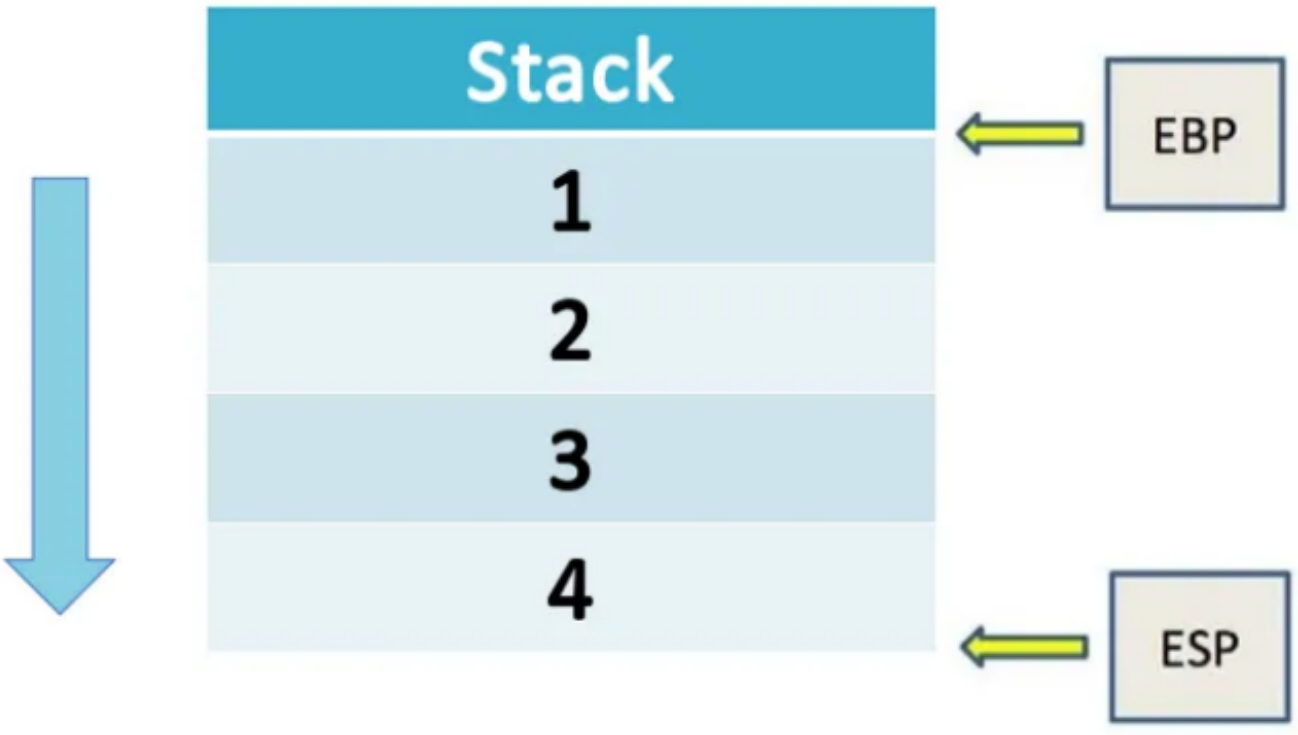
• This is done using the POP instruction, which retrieves the value from the top of the stack and adjusts the stack pointer (ESP) accordingly.

• By executing this POP instruction, "Jackie" effectively removes **Rennex's return address** from the stack, ensuring that the stack is correctly managed and that the program's flow is maintained.

==========

This image can confuse you once you meet something like it.

It is still the same, stack is still pointing to the top of the stack.



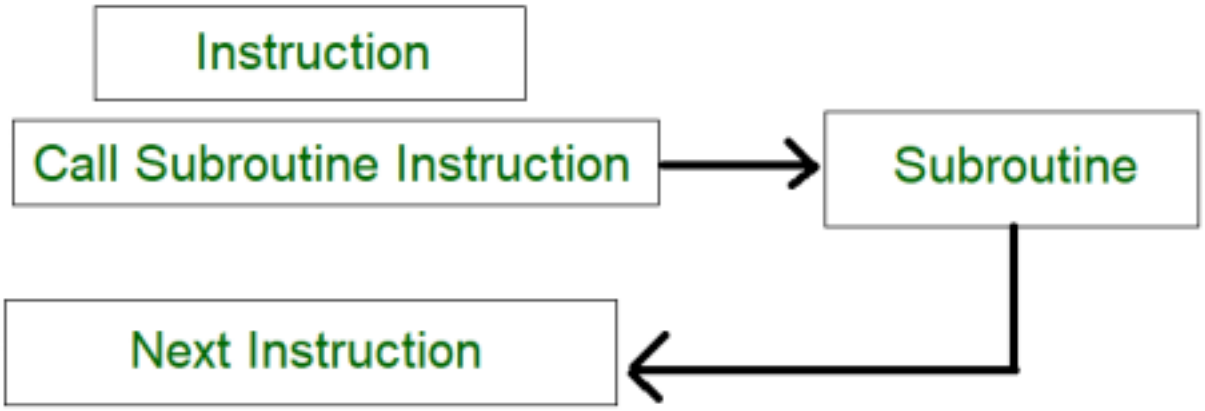
In this example, the stack pointer (ESP) points to the top of the stack frame, which contains the value 4. The extended base pointer (EBP) points to the bottom of the stack frame, which contains the value 1.

This may seem counterintuitive, since stack pointers typically start from higher memory addresses to lower memory addresses.

However, it is important to remember that the stack grows downwards. This means that when a new item is pushed onto the stack, its address is lower than the address of the previous item on the stack.

In the example image, the value 1 was pushed onto the stack first, followed by the value 4. Therefore, the value 1 is at the bottom of the stack frame, and the value 4 is at the top of the stack frame.

EBP is used to keep track of the bottom of the stack frame. This is useful for subroutines, which need to be able to access their local variables and parameters, even if the caller function has pushed new items onto the stack since the subroutine was called.



When a subroutine is called, it pushes the EBP register onto the stack. It then sets the EBP register to the current value of the ESP register. This effectively creates a new stack frame for the subroutine.

The subroutine can then access its local variables and parameters by using the EBP register as a reference point. For example, to access the first local variable, the subroutine would subtract 4 from the EBP register.

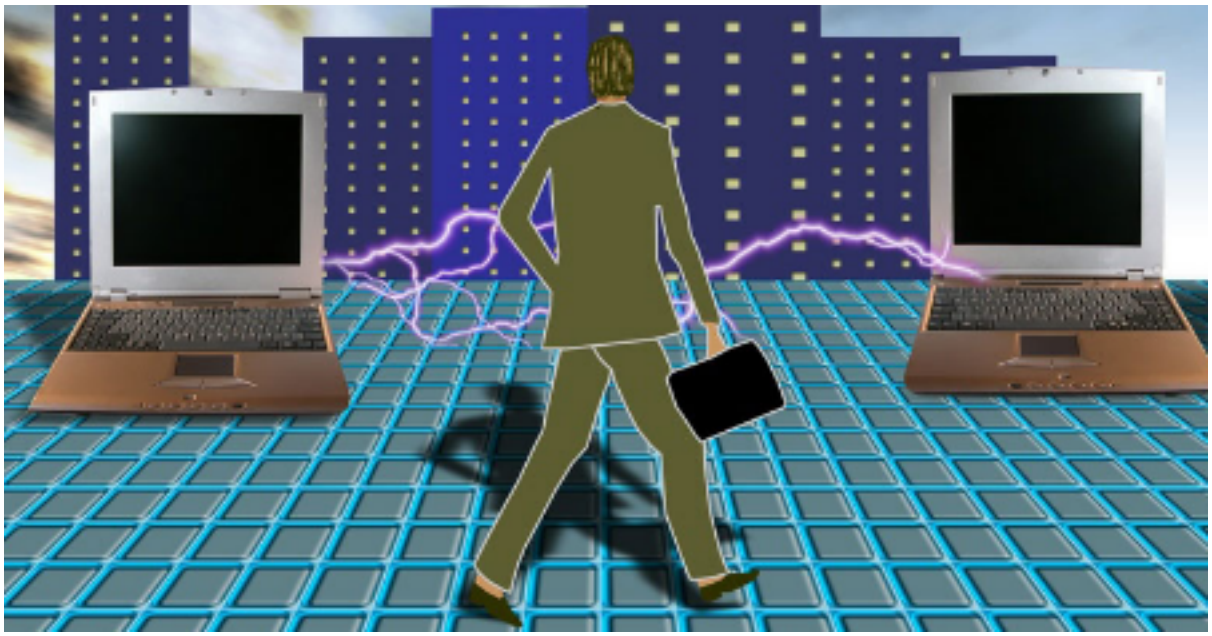
To access the second local variable, the subroutine would subtract 8 from the EBP register, and so on.

When the subroutine returns, it pops the EBP register from the stack. This restores the stack frame to the state it was in before the subroutine was called.

Disadvantages of Register Parameters

Register parameters can be used to pass arguments to subroutines in 32-bit programs using the fastcall calling convention. This can be more efficient than passing arguments on the stack, but it has a number of disadvantages:

Registers are used for other purposes. The registers used for parameters are also used for other purposes, such as holding loop counters and operands in calculations. Therefore, any registers used as parameters must be pushed on the stack and restored after the subroutine returns.



Code clutter. The extra pushes and pops can create code clutter and make it difficult to maintain. Potential for errors. Programmers must be careful to match every PUSH with a POP, even when there are multiple execution pathways through the code. Otherwise, registers may be left on the stack, which can lead to unexpected behavior.

The following code shows an example of how register parameters can be used to call the DumpMem subroutine from the Irvine32 library:

```
01 push ebx
02 ; save register values
03 push ecx
04 push esi
05 mov esi, OFFSET array
06 ; starting OFFSET
07 mov ecx, LENGTHOF array
08 ; size, in units
09 mov ebx, TYPE array
10 ; doubleword format
11 call DumpMem
12 ; display memory
13 pop esi
14 ; restore register values
15 pop ecx
16 pop ebx
```

Note the order of popping, LIFO - Last In First Out.

This code saves the values of the EAX, EBX, and ECX registers before calling DumpMem.

The DumpMem subroutine then uses these registers to access the memory to be displayed.

After the DumpMem subroutine returns, the values of the EAX, EBX, and ECX registers are restored.

However, this code is also susceptible to errors.

For example, if the eax register equals 1 on line 8, the procedure will not return to its caller on line 17 because three register values were left on the runtime stack.

Stack parameters

Stack parameters offer a more flexible and reliable approach to passing arguments to subroutines. To pass an argument to a subroutine using stack parameters, the argument is simply pushed onto the stack

before calling the subroutine.

For example, the following code shows how to call the DumpMem subroutine using stack parameters:

```
54 push
55 TYPE array
56 push
57 LENGTHOF array
58 push
59 OFFSET array
60 call
61 DumpMem
```

This code pushes the type, length, and offset of the array to be displayed onto the stack before calling the DumpMem subroutine. The DumpMem subroutine then uses these values to access the memory to be displayed.

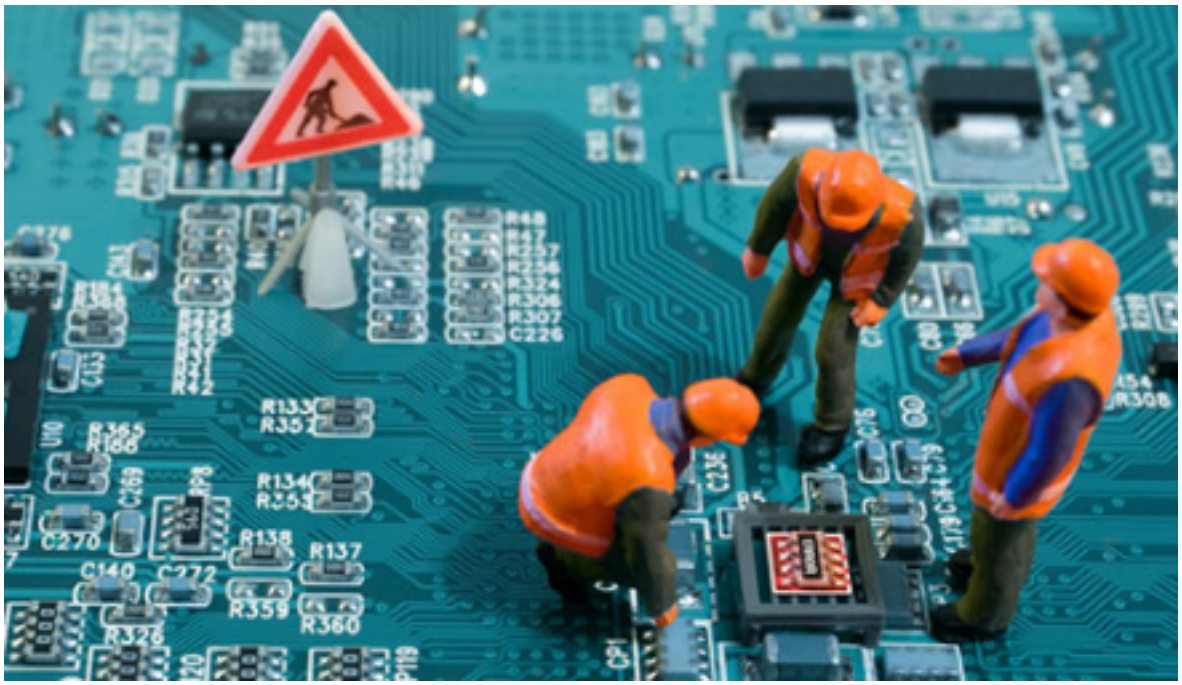
Advantages of Stack Parameters

Stack parameters have a number of advantages over register parameters:

More flexible. Stack parameters can be used to pass any number of arguments to a subroutine, regardless of the number of registers available.

More reliable. Stack parameters are less susceptible to errors than register parameters. For example, there is no need to worry about matching every PUSH with a POP.

Easier to maintain. Code that uses stack parameters is typically easier to read and maintain than code that uses register parameters.



Stack parameters are the preferred way to pass arguments to subroutines in most cases. They offer a more flexible, reliable, and maintainable approach than register parameters.

==========

Pass by Value

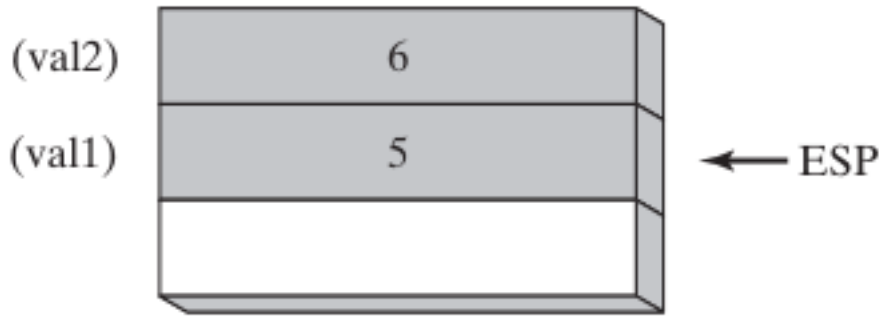
==========

When an argument is passed by value in MASM, a copy of the value is pushed onto the stack. The calling convention used by MASM pushes the arguments in reverse order, meaning that the last argument is pushed onto the stack first.

The following MASM code shows how to call a subroutine named AddTwo, passing it two 32-bit integers by value:

```
.data
    val1 DWORD 5
    val2 DWORD 6
.code
    push val2
    push val1
    call AddTwo
```

After the push instructions have been executed, the stack will look like this:



The diagram above shows the stack just prior to the CALL instruction for the AddTwo subroutine, which is passed two 32-bit integers by value. The arguments are pushed on the stack in reverse order, with val2 on top and val1 below.

The equivalent function call in C++ would be:

```
int sum = AddTwo(val1, val2);
```

In the image, the ESP register is pointing to val1 because it was the last value to be pushed onto the stack.

The stack grows downwards in MASM, meaning that when you push a value onto the stack, the ESP register is decremented by the size of the value.

So, if you push val2 onto the stack, the ESP register will be decremented by 4 bytes. Then, if you push val1 onto the stack, the ESP register will be decremented by another 4 bytes.

As a result, the ESP register will now be pointing to val1, which is the most recently pushed value.

As you can see, the ESP register is pointing to val1, which is the most recently pushed value.
When the subroutine AddTwo is called, it will read the arguments from the stack at offsets 0 and 4 bytes, respectively.

This means that it will read val2 from offset 0 and val1 from offset 4 bytes. After the subroutine AddTwo has finished executing, it will pop the arguments off the stack.

This will increment the ESP register by 8 bytes, so that it points to the next value on the stack.

==========

Pass by Reference

==========

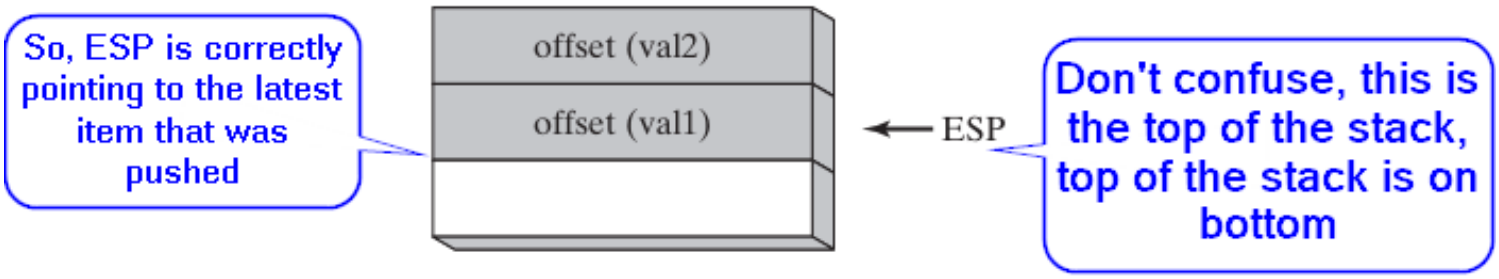
When an argument is passed by reference in MASM, the **address of the argument is pushed onto the stack.**

This allows the subroutine to modify the value of the original variable. The following MASM code shows how to call a subroutine named Swap, passing it two arguments by reference:

named Swap, passing it two arguments by reference:

```
084 .data
085     val1 DWORD 5
086     val2 DWORD 6
087 .code
088     push OFFSET val2 ; Push the address of the second argument onto the stack.
089     push OFFSET val1 ; Push the address of the first argument onto the stack.
090     call Swap ; Call the Swap subroutine.
```

After the push instructions have been executed, the stack will look like this:



Stack grows downwards!!! I won't stop repeating that.

The ESP register points to the top of the stack, so the subroutine Swap can access the arguments by reading from the stack at offsets 0 and 4 bytes, respectively.

The subroutine Swap can then use these addresses to modify the values of the original variables, val1 and val2.

The following C code is equivalent to the MASM code above:

```
;this is wrong code
int *val1;
int *val2;
Swap(&val1, &val2);

------------------------------

;this is the correct code
int val1;
int val2;
int *ptr1 = &val1;
int *ptr2 = &val2;
Swap(&val1, &val2);
```

The first code is wrong because the pointer variables val1 and val2 are not initialized. This means that they contain garbage values.

• When the Swap function is called, it will expect to receive pointers to integers as arguments. However, the pointer variables **val1 and val2 will not contain pointers to integers,** so the Swap function will not be able to work correctly.

• Additionally, the Swap function will try to modify the values of the variables pointed to by val1 and val2. However, the pointer variables val1 and val2 will contain garbage values, so the Swap function will try to modify the values of garbage memory. This can lead to undefined behavior.

The second code is correct because the pointer variables ptr1 and ptr2 are initialized to the addresses of the variables val1 and val2, respectively.

• When the Swap function is called, it will receive pointers to integers as arguments, because the pointer variables ptr1 and ptr2 contain pointers to the variables val1 and val2.

• The Swap function will then be able to modify the values of the variables val1 and val2, because it has pointers to those variables.

• Therefore, the second code is correct, and the first code is wrong.

• It is important to initialize pointer variables before using them. This will ensure that they contain valid addresses, and that they will be able to point to the correct variables.

Conclusion

Passing by value and passing by reference are two different ways to pass arguments to a subroutine in MASM.

Passing by value is the default, and it is the safest way to pass arguments, because it prevents the subroutine from modifying the original variables.

==========

Passing Arrays

==========

High-level languages always pass arrays to subroutines by reference because it is more efficient and safer than passing them by value.

When **passing an array by reference**, the address of the array is pushed onto the stack. The subroutine can then get the address from the stack and use it to access the array.

This is much more efficient than passing the array by value, because it does not require each array element to be pushed onto the stack separately.

Additionally, **passing an array by reference is safer** than passing it by value. If the array is large, passing it by value can quickly **overflow the stack.**

When passing an array by reference, the **stack only needs to store the address of the array**, which is much smaller than the size of the array itself.

The following MASM code shows how to pass an array to a subroutine by reference:

```
.data
    array DWORD 50 DUP(?)
.code
    push OFFSET array
    call ArrayFill
```

The OFFSET array operator returns the address of the array variable. So, the push OFFSET array instruction pushes the address of the array variable onto the stack.

The ArrayFill subroutine can then access the array by reading from the stack at offset 0 bytes. The following C code is equivalent to the MASM code above:

```
void ArrayFill(int *array) {
  for (int i = 0; i < 50; i++) {
    array[i] = i;
  }
}
```

The array parameter is a pointer to the array. The subroutine can then access the array elements by dereferencing the array pointer.

Passing arrays by reference is the standard way to pass arrays to subroutines in both MASM and C. It is more efficient and safer than passing arrays by value.