# Contents

# 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 **subroutine**, 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.

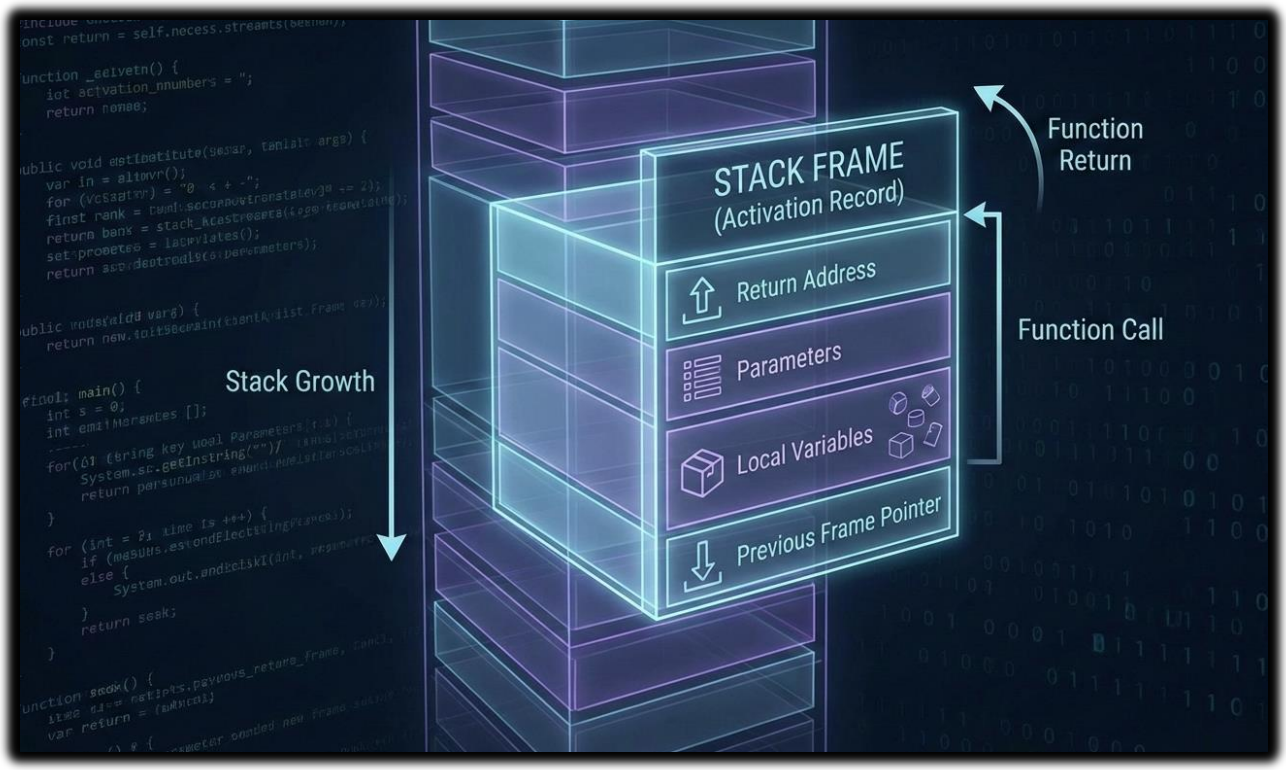
### I. Stack Parameters

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.




## II. The Anatomy of a Stack Frame

A **stack frame** (also called an **activation record**) is a specific chunk of memory created on the stack every time a function (subroutine) is called.
Think of it as the function's **personal workspace** while it's running.

This stack frame holds:

- Arguments passed to the function
- The return address (where execution should go after the function finishes)
- Local variables
- Saved registers

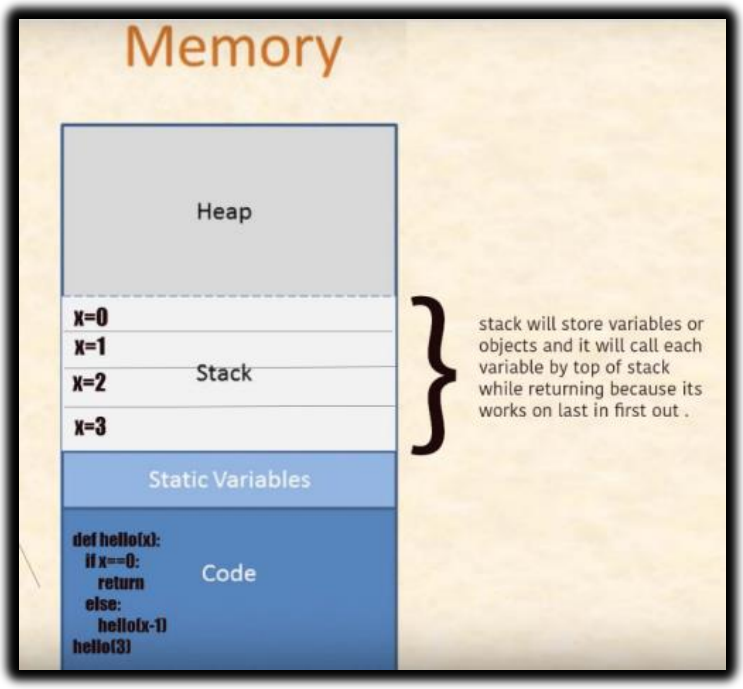
## III. How a Stack Frame Is Built (Step by Step)

1. **Passed arguments** (if any) are pushed onto the stack first.
2. When the subroutine starts executing, the current value of the **Extended Base Pointer (EBP)** is pushed onto the stack.
   This preserves the caller's stack frame.
3. The **EBP is then set equal to the current Stack Pointer (ESP)**.
   From this point on, EBP becomes a **stable reference point** for accessing:
   - Function parameters
   - Local variables
4. If the subroutine uses **local variables**, the **Stack Pointer (ESP) is decremented** to reserve space for them on the stack.

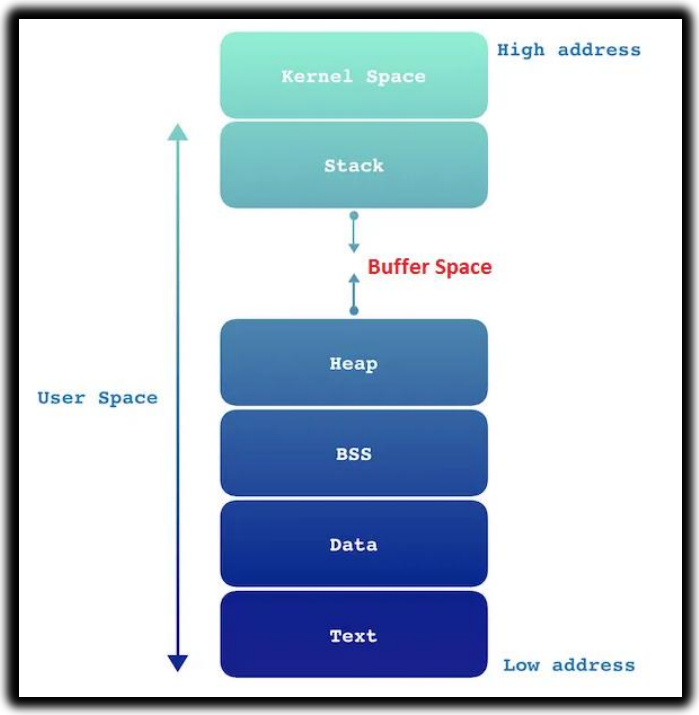
## IV. Important Stack Behavior to Remember

- The stack starts at **higher memory addresses**.
- As values are pushed, **ESP decreases** → the stack **grows downward**.
- When values are popped, **ESP increases**.

This downward growth explains why space for parameters and local variables is allocated by decrementing the stack pointer.



If certain **registers need to be preserved**, they are pushed onto the stack before the subroutine modifies them. This ensures their original values can be restored before returning control to the caller.




The **layout and organization of a stack frame** depend heavily on:

- The program's **memory model**
- The **calling convention** used to pass arguments

**Why Argument Passing on the Stack Matters**

Understanding how arguments are passed on the stack is **crucial**, because most high-level programming languages rely on this mechanism.

For example:

- In **32-bit Windows API programming**, function arguments are typically passed **on the stack**.
- In **64-bit programming**, a different calling convention is used (many arguments are passed through registers instead).
  We'll explore this newer convention in detail in later chapters.

---

**Calls and the Stack (MASM Example)**

Consider a scenario where **“Reese”** (the calling, external procedure) calls **“Rennex”** (the called, internal procedure) in MASM.

Here’s what actually happens:

- When **Reese calls Rennex**, it is **Reese** that pushes **Rennex’s return address** onto the stack.
- This return address points to the exact instruction in **Reese** where execution should continue after **Rennex** finishes.
- In other words, **the caller (Reese)** is responsible for saving the return address.

**During Execution**

- **Rennex** runs its code normally.
- When Rennex reaches the RET instruction, it **uses the saved return address** to transfer control back to Reese.

**After the Return**

- Control returns to **Reese**.
- At this point, it is typically **Reese’s responsibility to clean up the stack**.
- Reese issues a **POP instruction** to remove the return address from the stack.
- This POP:
    - Retrieves the value at the top of the stack
    - Adjusts the **Stack Pointer (ESP)** accordingly

By doing this, Reese ensures:

- The stack remains balanced
- Program flow continues correctly

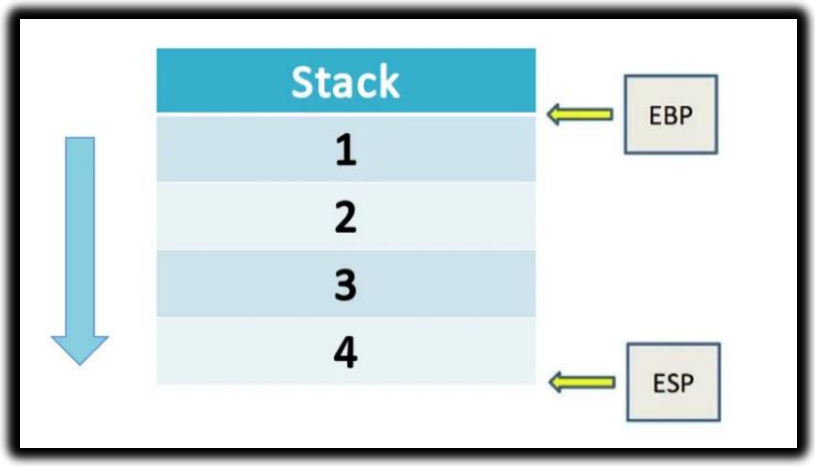
---

**About Stack Diagrams**

Some stack diagrams can look confusing at first, especially when multiple values are involved.
However, the rule **never changes**:

The stack pointer **always points to the top of the stack**.

Even when the diagram looks complex, the underlying behavior of the stack remains the same.



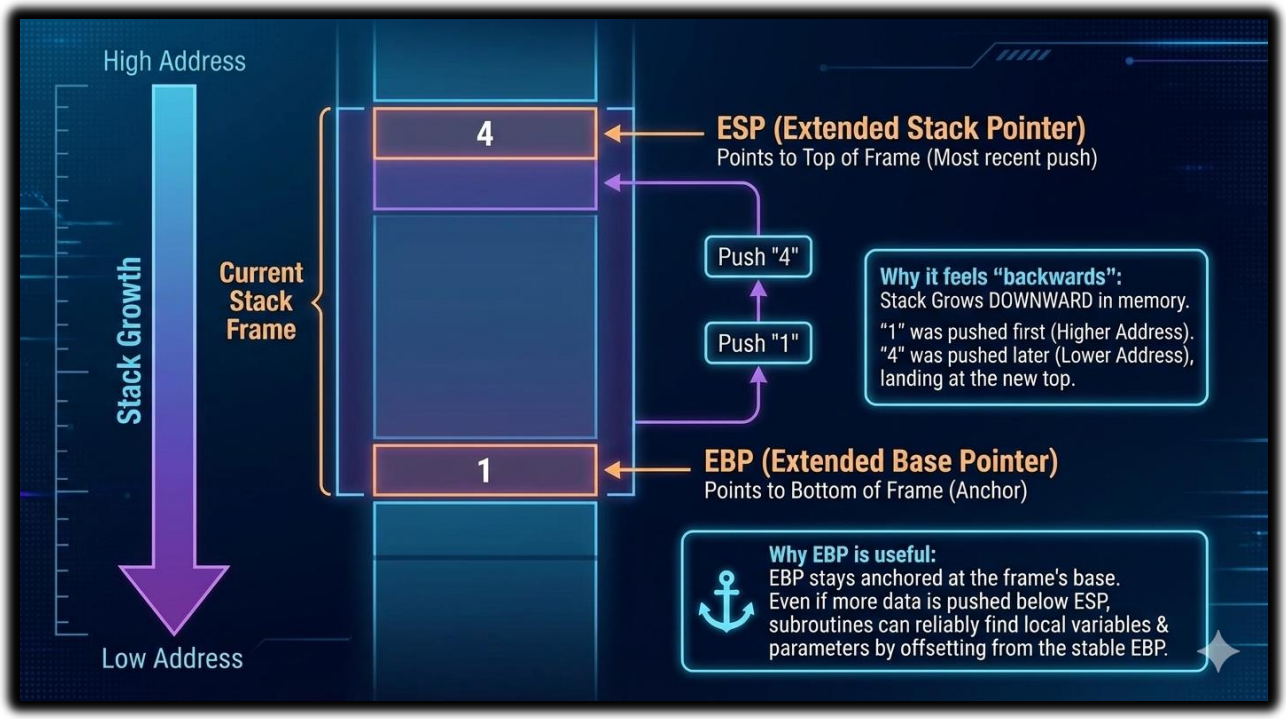

Right now, the stack pointer (ESP) is pointing at the very top of the current stack frame, and the value sitting there is 4.

The extended base pointer (EBP) is pointing lower down, at the bottom of this stack frame, where the value 1 lives.

At first glance this can feel a little backwards — because we usually think of memory addresses getting bigger as we go "up." But here's the key thing to remember:

The stack grows downward in memory.

That means every time you push something new onto the stack, it gets placed at a lower memory address than whatever was there before.



So, in the picture you're looking at:

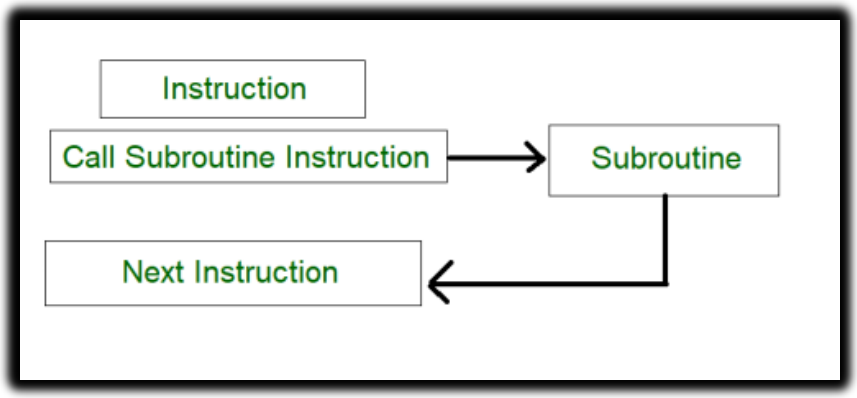
- First, the value 1 was pushed → it landed at a higher memory address
- Then the value 4 was pushed → it landed at a lower memory address

That's why 1 ends up at the bottom of the stack frame, and 4 ends up at the top (even though 4 is the most recently added item).

And that's exactly why we have EBP (the base pointer).

It stays parked at the bottom of the current stack frame — kind of like an anchor.

That way, even if the function you're currently in pushes more stuff onto the stack later, your subroutine can still reliably find its own local variables and parameters by counting offsets from EBP. Super handy!




## Stack Frames and Subroutine Calls

When a subroutine is called, the **EBP register** plays a central role in creating a stack frame:

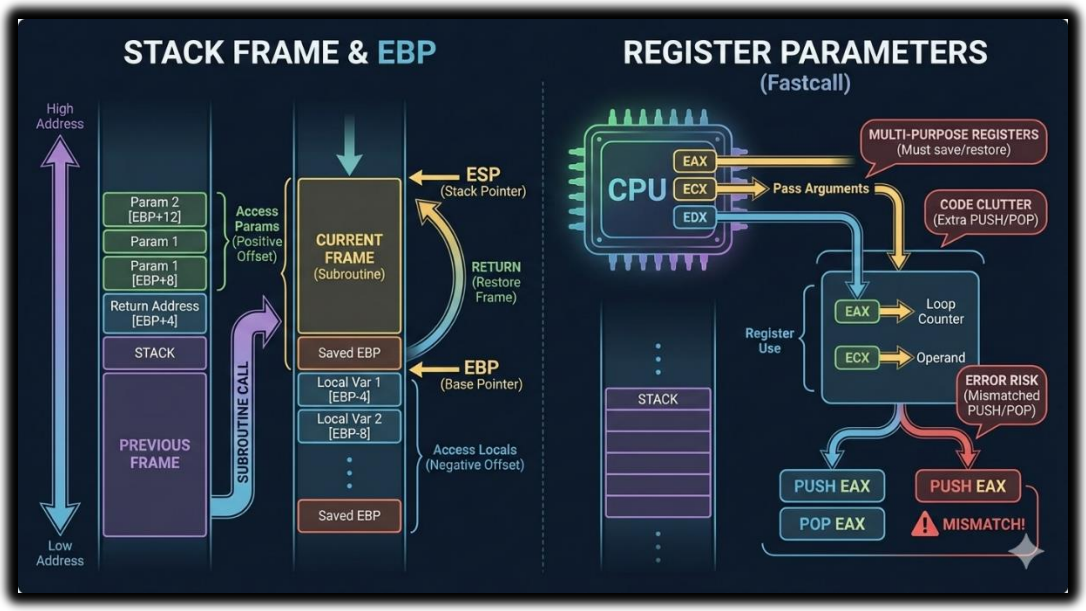
1. **Setup:**
   - The current value of EBP is pushed onto the stack.
   - EBP is then set equal to ESP, marking the start of the new frame.
2. **Accessing variables:**
   - Local variables are accessed using negative offsets from EBP.
     - First local variable → [EBP-4]
     - Second local variable → [EBP-8]
     - And so on.
   - Parameters are accessed using positive offsets from EBP.

3. **Return:**
    - When the subroutine ends, the saved EBP is popped back from the stack.
    - This restores the previous stack frame, returning control cleanly to the caller.

## Disadvantages of Register Parameters (Fastcall Convention)

Passing arguments in registers can be faster than using the stack, but it comes with trade-offs:

- **Registers are multi-purpose:**
    - The same registers used for parameters are also needed for loop counters, operands, and other tasks.
    - This means they often must be saved (pushed) before use and restored (popped) afterward.
- **Code clutter:**
    - Extra pushes and pops make the code harder to read and maintain.
- **Error risk:**
    - Every PUSH must be matched with a POP.
    - If execution paths diverge (e.g., multiple branches), it's easy to mismatch pushes/pops, leaving registers stuck on the stack and causing unpredictable 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
```

## Saving and Restoring Registers with the Stack

When working with subroutines, registers often need to be preserved. This is done by **pushing** them onto the stack before the call and **popping** them back afterward. Because the stack operates in **LIFO (Last In, First Out)** order, the registers must be popped in the reverse order they were pushed.

## Example Flow

1. **Before the call:**
   - EAX, EBX, and ECX are pushed onto the stack.
   - This saves their values so the subroutine can safely use them.
2. **Inside the subroutine (**DumpMem**):**
   - The registers are used to access and display memory.
3. **After the call:**
   - The saved values are popped back into EAX, EBX, and ECX.
   - This restores the original state of the registers.

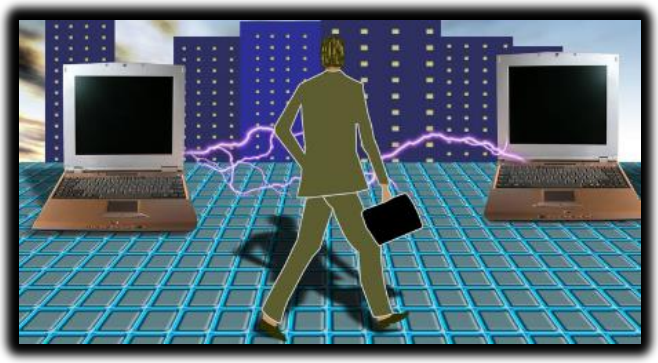
## Potential Pitfall

If the push/pop sequence is mismatched, the stack becomes unbalanced. For example:

- If EAX = 1 at line 8 and the procedure exits incorrectly, three register values remain on the stack.
- This prevents the procedure from returning properly at line 17, since the return address is buried under leftover data.

## Key Takeaway

Always respect the **push/pop balance**. Every register pushed must be popped in reverse order, or the stack frame will break — leading to errors like failed returns or corrupted data.



# Stack Parameters: The Nicer Way to Pass Arguments

When you want to send info (arguments) to a subroutine, using the stack is actually one of the cleanest and most flexible methods out there. The idea is super straightforward:

1. Before you call the subroutine, just push each argument you want to send onto the stack.
2. Then call the subroutine like normal.
3. Inside the subroutine, it can reach back and grab those values right off the stack.

That's it! No weird registers to juggle, no fixed limits on how many arguments you can pass — the stack just grows to hold whatever you need. Here's a quick real-world feel for it: Imagine you want to call a subroutine called DumpMem that needs three pieces of info:

- A starting memory address
- A length (how many bytes to dump)
- Maybe a flag or format option

Instead of cramming everything into registers (which can get messy fast), you'd do something like this in assembly:

```
; Prepare to call DumpMem(startAddr, length, formatFlag)

push    dword [formatFlag]     ; push last argument first (common convention)
push    dword [length]
push    dword [startAddr]      ; push first argument last

call    DumpMem

add     esp, 12                ; clean up the stack (3 × 4 bytes = 12)
```

(The exact push order and cleanup style can vary depending on the calling convention, but the core idea stays the same.)

Pushing arguments onto the stack before the call means:

- The subroutine can easily find them relative to the EBP (base pointer) once it sets up its own stack frame
- You can pass as many arguments as you want (limited only by available stack space)
- It's very reliable — especially useful in nested calls or when you have lots of parameters

It's one of those things that feels a little "extra" at first, but once you get used to it, it becomes super natural and way less error-prone than trying to squeeze everything into registers.

## I. 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

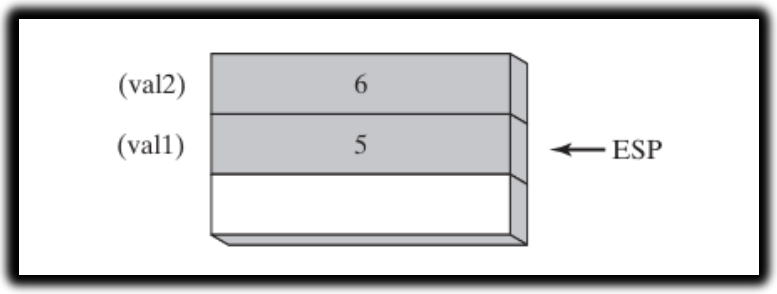
Let's look at a super common way to pass arguments to a subroutine in MASM: passing by value using the stack.

When you pass something by value, you're not sending the original variable — you're pushing a copy of its value onto the stack.

Here's the little MASM snippet we're working with:

```
73 .data
74     val1 DWORD 5
75     val2 DWORD 6
76 .code
77     push val2
78     push val1
79     call AddTwo
```

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




Notice something important:

- We pushed val2 (6) first
- Then we pushed val1 (5) second

Because the stack grows downward in memory (lower addresses), every new push makes ESP move to a lower address.

So, after both pushes:

- ESP is pointing at the most recently pushed value → val1 (5)
- val2 (6) is sitting just below it

This reverse-order pushing is the standard MASM / stdcall / Win32 calling convention trick.

It means the last argument you want the function to see gets pushed first.In C++ land, this code is basically doing the same thing as:

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

So, inside the AddTwo subroutine, once it sets up its stack frame (usually with push ebp; mov ebp, esp), the arguments are super easy to reach:

- At [ebp + 8] → first argument (val1 = 5)
- At [ebp + 12] → second argument (val2 = 6)

(Why +8 and +12? Because [ebp] holds the old EBP, [ebp+4] holds the return address, so arguments start at +8.)

When AddTwo is done and wants to clean up, it pops those two arguments off the stack (or more commonly, the caller cleans up by doing add esp, 8 after the call).

Either way, ESP gets bumped back up by 8 bytes (4 bytes × 2 arguments), so it points to whatever was on the stack before we started pushing.Quick recap of why this feels "backwards" at first:

- Stack grows down → new pushes → lower addresses → ESP moves down
- We push arguments right-to-left (last arg first) → so the first arg ends up closest to ESP
- ESP always points to the top (most recent) item

# Pass by Reference

## I. Passing Arguments by Reference in MASM - ("Let the Function Change My Stuff")

Sometimes you don't just want to hand a subroutine a copy of your value — you want it to be able to actually change the original variable back in your main code.

That's when you pass by reference.

Instead of pushing the value itself onto the stack, you push the memory address (the location) of the variable.

Then inside the subroutine, it can go to that address and read or write whatever it wants — so changes stick around even after the subroutine finishes.

Here's what that looks like in MASM (just the calling part for now):

```
.data
    val1 DWORD 5
    val2 DWORD 6
.code
    push OFFSET val2 ; Push the address of the second argument onto the stack.
    push OFFSET val1 ; Push the address of the first argument onto the stack.
    call Swap ; Call the Swap subroutine.
```

OR

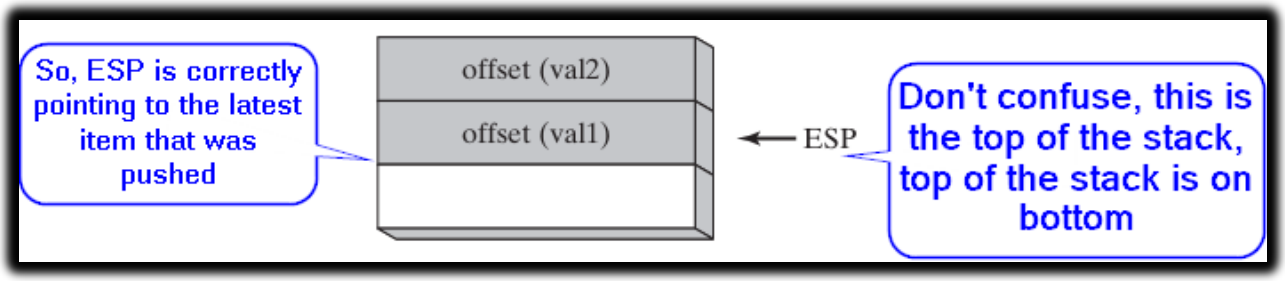
```
; Assume we have two variables we want to swap
.data
    num1    DWORD   42
    num2    DWORD   99

.code
    ; Push the *addresses* (not the values!) of the two variables
    lea     eax, [num2]       ; load effective address of num2 into eax
    push    eax               ; push address of num2

    lea     eax, [num1]       ; load effective address of num1
    push    eax               ; push address of num1

    call    Swap              ; now Swap can modify num1 and num2 directly!
```

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




After those pushes, the stack has the addresses sitting on it (not 42 and 99, but something like 00403000h and 00403004h — whatever memory locations hold num1 and num2).

Inside the Swap subroutine, it would do something like this (super simplified):

```
Swap PROC
    push ebp
    mov  ebp, esp

    mov  eax, [ebp+8]     ; eax = address of first arg (num1)
    mov  ebx, [ebp+12]    ; ebx = address of second arg (num2)

    mov  ecx, [eax]       ; ecx = actual value at num1 (42)
    mov  edx, [ebx]       ; edx = actual value at num2 (99)

    mov  [eax], edx       ; num1 now gets 99
    mov  [ebx], ecx       ; num2 now gets 42

    pop  ebp
    ret
Swap ENDP
```

## Key differences from pass-by-value:

1. We push addresses (using lea or offset) instead of values
2. The subroutine uses those addresses to dereference (go get or set the real data). It's basically like passing a pointer in C/C++:
3. Any changes the subroutine makes affect the original variables — magic!

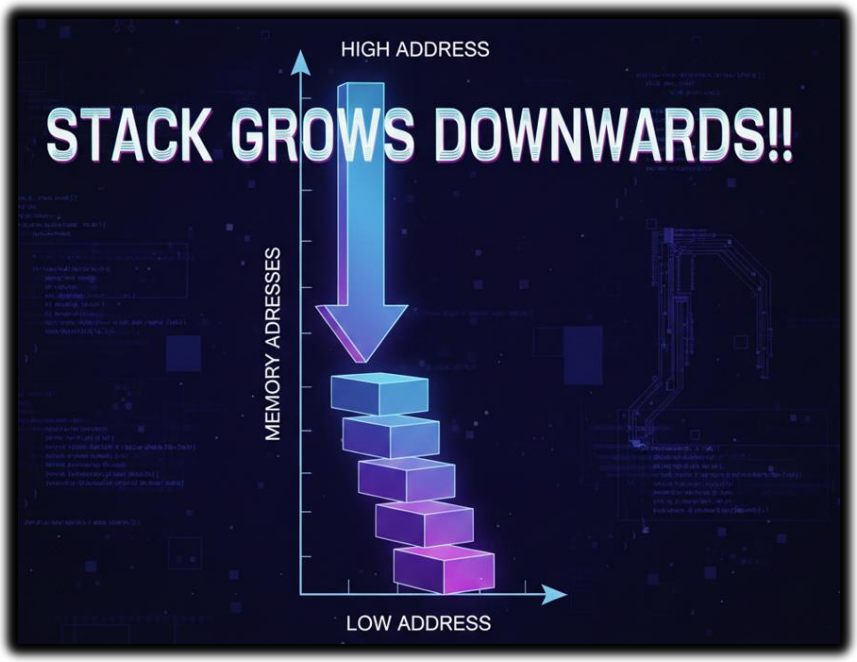
```
void Swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
 }
```

Super useful when you want the subroutine to return multiple results, update big structures, or just modify variables without copying them.

Stack Grows Downwards!!! (Yes, We're Repeating It Because It's Important)

Okay, let's pick up right where we left off with passing by reference in that Swap example.

After pushing the addresses of num1 and num2 onto the stack (remember: we pushed the address of the second one first, then the first one — classic reverse order), the stack ends up looking something like this:




```
int *vall;      // ← uninitialized pointer — contains garbage!
int *valz;      // ← also garbage!

Swap(vall, valz);   // ← disaster! You're passing random memory
    addresses
```

Why this is dangerous:

- vall and valz are pointers, but you never told them where to point.
- They're holding junk values (whatever random bits were in memory).
- When Swap tries to use those junk addresses (*a = ...), it's writing to random places in memory → crashes, corrupted data, undefined behavior, etc.
- Never pass uninitialized pointers!

Correct way (this actually works):

```
int va11 = 42;      // real integer variable
int va12 = 99;      // another one

// Option 1: Pass addresses directly (most common & cleanest)
Swap(&va11, &va12);

// Option 2: If you really want to use pointer variables first
int *ptr1 = &va11;   // ptr1 now holds the address of va11
int *ptr2 = &va12;   // ptr2 holds address of va12

Swap(ptr1, ptr2);    // same effect — Swap gets the addresses
```

Or even shorter (and what most people do):

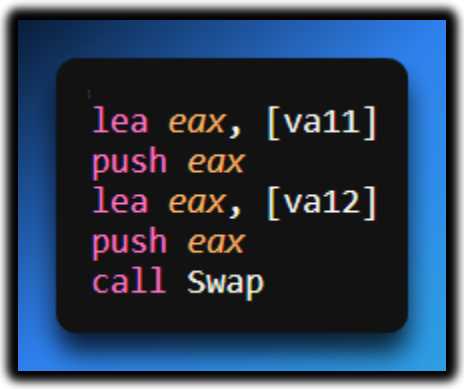
```
int va11 = 42;
int va12 = 99;

Swap(&va11, &va12);   // boom — passes the addresses directly
```

Inside Swap, it receives two int* (pointers), dereferences them with *, and swaps the actual values at those addresses.

After the call, va11 is 99 and va12 is 42 — the originals got changed, exactly like in the assembly version.

Quick Side-by-Side: Assembly vs C++Assembly (MASM, passing by reference):




C++ Equivalent:


Same idea:

- Assembly → push addresses manually
- C++ → compiler does it for you when you write &

## Wrapping It Up: By Value vs By Reference

- **Pass by value (what we did earlier with AddTwo):**
  Pushes copies of the values.
  Safe — subroutine can't touch your originals.
  Good when you just want to send data in and not change it.
- **Pass by reference (what we're doing with Swap):**
  Pushes addresses (pointers to the originals).
  Lets the subroutine read and change the real variables.
  Perfect for swap, sorting arrays, returning multiple results, etc.

And remember (you already know this by heart now)

Stack grows downwards!!! → ESP points to the newest thing → arguments are at positive offsets from EBP after the frame is set up.

No more confusion — we've untangled it.

## Passing Arrays to Subroutines: Why We Almost Always Do It by Reference

In high-level languages like C, C++, Java, etc., arrays are always passed by reference (well, technically by pointer/address) when you send them to a function.

And guess what? MASM does the exact same thing for basically the same reasons: it's way more efficient and much safer.

Here's why passing by value would be a nightmare with arrays:

- If you tried to pass an array by value, you'd have to push every single element onto the stack one by one.
- For a small array? Annoying but doable.
- For a big array (say 1,000 elements or 100,000)? You'd blow up the stack in seconds— stack overflow city.

Instead, we do the smart thing: push just the address of the array (a single 4-byte or 8-byte value, depending on 32-bit or 64-bit).

The subroutine grabs that address and can then read from or write to any part of the array using normal memory access.

- **Super efficient:** one push instead of hundreds/thousands.
- **Super safe:** no risk of overflowing the stack with a giant copy.

### Real MASM Example: Passing an Array by Reference

Here's how it looks in code:

```
118 .data
119     array DWORD 50 DUP(?)
120 .code
121     push OFFSET array
122     call ArrayFill
```

- OFFSET array is the MASM way to say "give me the memory address where array starts."
- That single push puts the address on the stack.
- ArrayFill can pull it off the stack (usually at [ebp + 8] after setting up the frame) and then treat it like a pointer to start walking through the array.

Inside ArrayFill, it might look something like this (simplified):

```
ArrayFill PROC
    push ebp
    mov  ebp, esp

    mov  esi, [ebp + 8]       ; esi = starting address of the array

    xor  ecx, ecx             ; ecx = 0 (our counter)
fill_loop:
    cmp  ecx, 50
    jge  done

    mov  [esi + ecx*4], ecx   ; array[i] = i   (each element is 4 bytes)

    inc  ecx
    jmp  fill_loop

done:
    pop  ebp
    ret
ArrayFill ENDP
```

```
void ArrayFill(int* array) {          // array is really just a
    pointer
    for (int i = 0; i < 50; i++) {
        array[i] = i;                 // same as *(array + i) = i;
    }
}

// Calling it:
int array[50];
ArrayFill(array);                     // array "decays" to &array[0]
    automatically
// or more explicitly: ArrayFill(&array[0]);
```

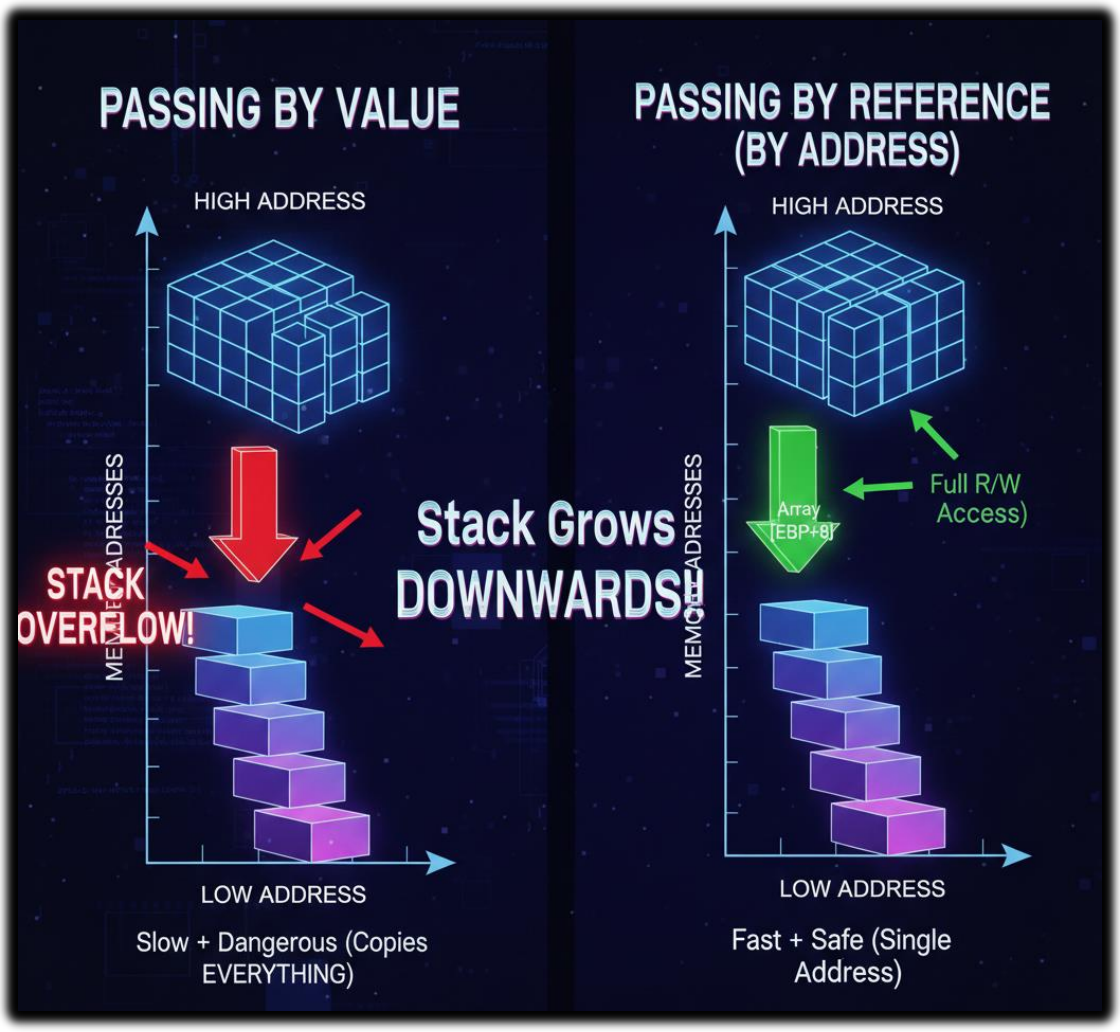
**See how natural that is?**

In C, when you pass an array name to a function, it automatically turns into a pointer to the first element — exactly like pushing OFFSET array in assembly.

## Quick Summary (Why This Is the Standard Way)

- Passing arrays by value → copies the whole thing → slow + dangerous (stack overflow risk)
- Passing by reference (aka by address/pointer) → just one address → fast + safe
- This is why both MASM and C/C++ default to passing arrays this way
- You get full read/write access inside the subroutine without duplicating anything

And of course… stack grows downwards!!! → that one address sits nicely on top (most recently pushed), ready for the subroutine to grab at [ebp + 8] or similar.



# ACCESSING STACK PARAMETERS

Starting with a quick true/false question you asked.

**The register ESP is used to point to the next item on the stack and is referred to as the 'stack pointer'.**

**→ False!**

## The ESP (Extended Stack Pointer)

The ESP (Extended Stack Pointer) register always points to the top of the stack — meaning the most recently pushed item (or the exact spot where the next push will happen). It does not point to some "next item" that's already there waiting.

- Before any push → ESP points to the empty spot where the first item will go.
- After you push something → ESP now points directly at that just-pushed item (the current top).
- When you push again → ESP moves down to point at the new top.

Stack grows downwards!!! → Pushing decrements ESP (by 4 bytes on 32-bit, usually), so newer items live at lower memory addresses.

Popping increments ESP, revealing the previous item as the new top.

### Quick example:

```
push 1      ; ESP now points at 1 (top)
push 2      ; ESP now points at 2 (top)
push 3      ; ESP now points at 3 (top)
```

Right now:

- ESP → 3 (most recent)
- Below it: 2
- Below that: 1

If you pop eax:

- eax gets 3
- ESP moves up → now points at 2 (the new top)

So yes — ESP always tracks the current top of the stack, not some future "next" spot in a vague way.

## EBP: The Stable “Home Base” for Your Stack Frame

EBP, often called the frame pointer or stack frame pointer.

Unlike ESP (which jumps around every push/pop), EBP stays fixed once a function sets it up.

It acts like an **anchor** — a stable reference point so the function can reliably find:

- Its local variables
- Its parameters (passed on the stack)
- The return address
- Even the caller’s saved registers

Typical setup inside a function:

```
push ebp          ; save old EBP
mov  ebp, esp     ; EBP now points to the bottom of THIS function's stack frame
sub  esp, 20      ; make room for locals (example: 20 bytes)
```

Now the stack frame layout is predictable (from EBP):

- [ebp] → old EBP (saved)
- [ebp + 4] → return address
- [ebp + 8] → first parameter
- [ebp + 12] → second parameter
- [ebp - 4] → first local variable
- [ebp - 8] → second local, etc.

No matter how many more pushes happen inside the function (temporary ones), EBP stays put — so your offsets to parameters and locals never break.

## What's a Stack Frame, Anyway?

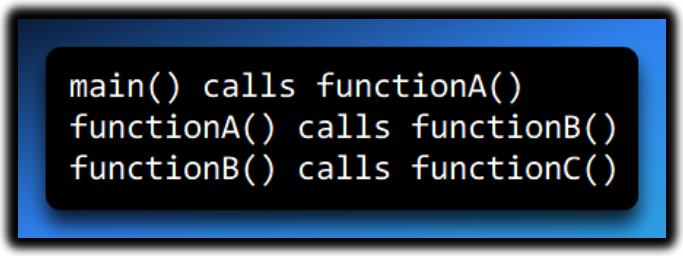
A stack frame is just the chunk of stack memory that belongs to one function call:

- Saved registers (like old EBP)
- Return address
- Function parameters
- Local variables

Each time a function calls another function, a new stack frame gets built on top (because stack grows downwards!!!).
The frames are nested:

- Deepest/most recent call = top frame
- Oldest call (main or whatever started it) = bottom frame

Example call chain:




Active stack frames (top to bottom):

1. functionC's frame ← current top (ESP here)
2. functionB's frame
3. functionA's frame
4. main's frame

When functionC returns:

- Its frame is "destroyed" (ESP moves back up)
- Now functionB is the top/active frame again

### How Many Stack Frames Can You Have?

As many as the stack memory allows — but there's a limit!

- Stack size is finite (default is often 1 MB on Windows, 8 MB on Linux, etc.)
- Each frame takes some space (parameters + locals + saved stuff)
- Too deep? → stack overflow → crash (classic recursion gone wrong)

Programs with heavy recursion or tons of nested calls can eat stack fast.

Most of the time you want shallow call stacks — easier to debug and way less likely to blow up.

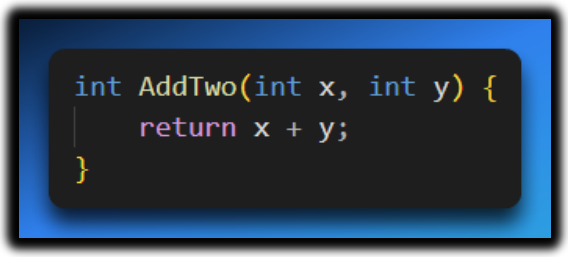
## Accessing Stack Parameters 2

High-level languages provide different ways to initialize and access parameters during function calls. In **C and C++**, this is typically handled using a **stack frame**.

A **stack frame** is a block of memory created on the stack when a function is called. It stores:

- Function parameters
- Local variables
- Saved registers

### I. Example: A Simple C Function

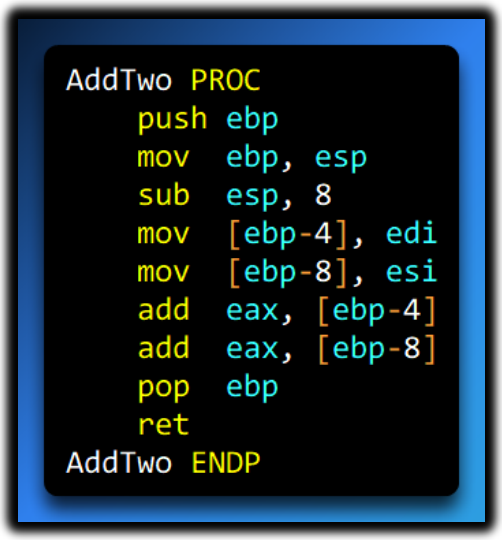



When this function is compiled, the compiler automatically generates two key sections:

- A **prolog**, which sets up the stack frame
- An **epilog**, which tears it down before returning to the caller

The **prolog** saves the current EBP and updates it to point to the top of the stack.
The **epilog** restores EBP and transfers control back to the calling function.

## II. Assembly Implementation of AddTwo



```
AddTwo PROC
    push ebp
    mov  ebp, esp
    sub  esp, 8
    mov  [ebp-4], edi
    mov  [ebp-8], esi
    add  eax, [ebp-4]
    add  eax, [ebp-8]
    pop  ebp
    ret
AddTwo ENDP
```

- push ebp
  Saves the caller's base pointer on the stack.
- mov ebp, esp
  Sets EBP to the current stack top, establishing a new stack frame for AddTwo.
- sub esp, 8
  Allocates **8 bytes** of space on the stack for local storage.
- mov [ebp-4], edi
  mov [ebp-8], esi
  Stores the function's parameters inside the stack frame at fixed offsets from EBP.
- add eax, [ebp-4]
  add eax, [ebp-8]
  Adds the two values and places the result in EAX, which is commonly used for return values.
- pop ebp
  Restores the caller's base pointer.
- ret
  Uses the saved return address to transfer control back to the caller.

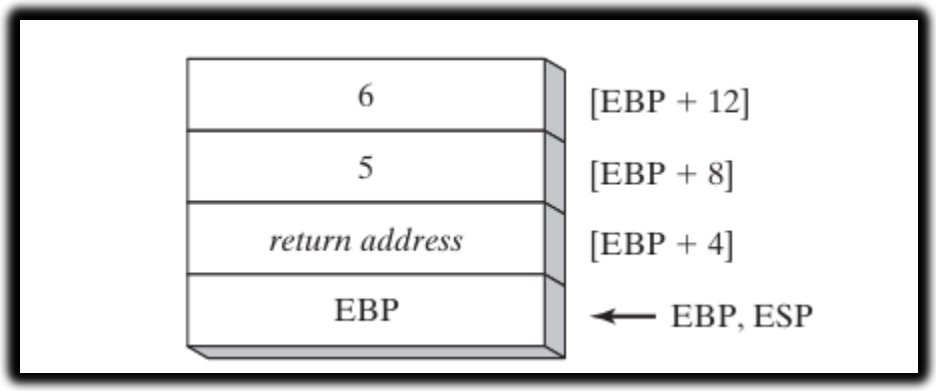
## III. How Parameters Are Pushed onto the Stack

When AddTwo is called, the compiler pushes the parameters onto the stack **in reverse order**:

- The **second argument** is pushed first
- The **first argument** is pushed last

This happens because the stack grows **downward**, and pushing arguments this way ensures consistent access using fixed offsets from EBP.

The following figure shows the contents of the stack frame after the function call AddTwo(5, 6):

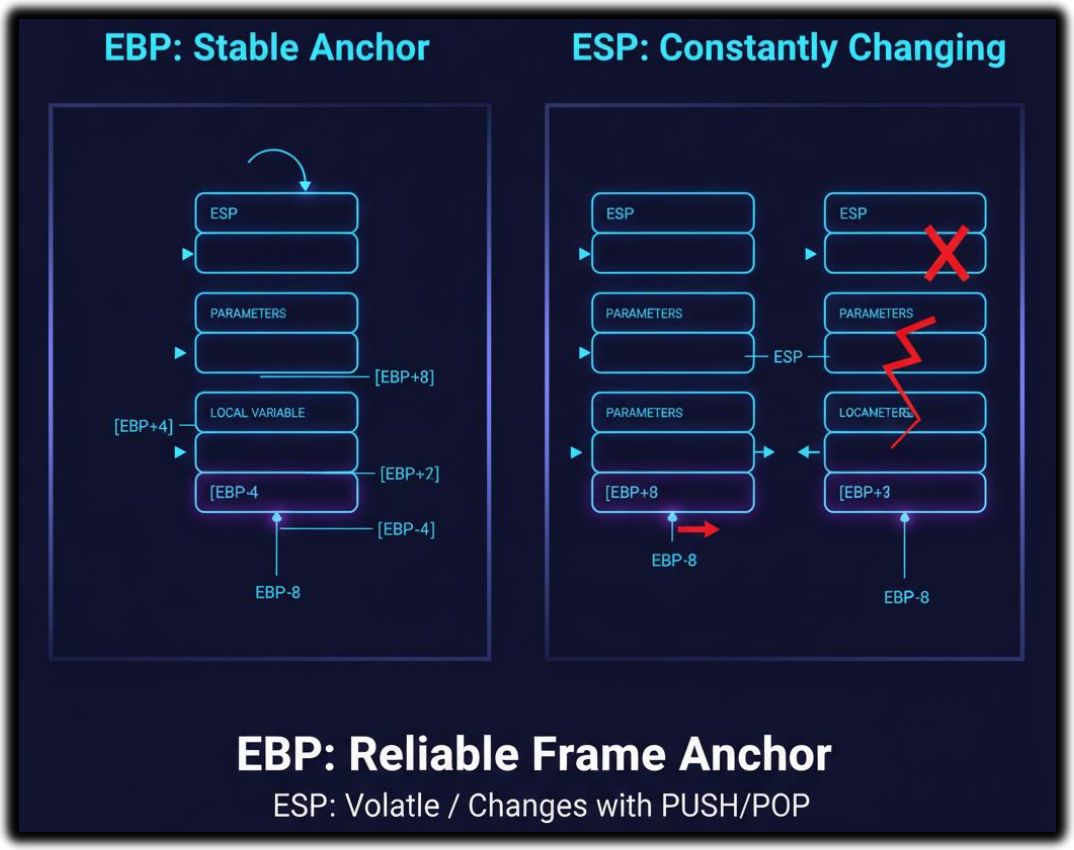



## IV. Why EBP Matters (Even When ESP Changes)

The AddTwo function can push additional registers onto the stack **without changing the offsets of its parameters** from EBP.
This is because:

- **ESP changes** as values are pushed and popped.
- **EBP stays fixed** for the lifetime of the function.

That's why parameters and local variables are accessed relative to EBP and *not* ESP. EBP acts as a **stable anchor** inside a constantly moving stack.




## V. Understanding the Stack Frame in the Image

The image represents a **stack frame** for a function that takes **two parameters**.
A stack frame is created when a function is called and contains:

- Function parameters
- Local variables
- Saved registers

Since the stack **grows downward**, parameters are pushed in **reverse order**.

## VI. Example: AddTwo(5, 6)

When AddTwo(5, 6) is called:

- 6 is the **second parameter**, so it is pushed **first**.
- 5 is the **first parameter**, so it is pushed **last**.

This reverse order ensures predictable offsets from EBP.

## VII. What Each Stack Entry Represents

- **6**
  The second parameter passed to AddTwo.
- **5**
  The first parameter passed to AddTwo.
- **[EBP + 12]**
  The memory address of the **second parameter** (6).
- **[EBP + 8]**
  The memory address of the **first parameter** (5).
- **[EBP + 4]**
  Contains the **saved EBP of the calling function**.
  This links the current stack frame to the caller's stack frame.

## VIII. How the Stack Frame Is Created

When the function is called, the compiler:

1. Pushes the caller's EBP onto the stack.
2. Sets EBP to the current value of ESP.

This marks the **start of a new stack frame**.

## IX. Returning from the Function

When the function finishes executing:

- The **return address** is used to transfer control back to the caller.
- The saved EBP is restored.
- The caller's stack frame becomes active again.

This process ensures:

- Stack integrity
- Correct control flow
- Safe access to parameters and locals

## X. Key Registers (Quick Recap)

- **EBP (Base Pointer)**
  Fixed reference point for the current stack frame.
- **ESP (Stack Pointer)**
  Points to the top of the stack and changes as data is pushed or popped.

Here is an example of how EBP is used to access the parameters and local variables for a function:

```
; Function prologue
push ebp
mov ebp, esp
sub esp, 8 ; Reserve space for two parameters
mov [ebp-4], edi ; Store the first parameter
mov [ebp-8], esi ; Store the second parameter

; Function body
; ...
; Function epilogue
mov esp, ebp
pop ebp
ret

; Access the first parameter
mov eax, [ebp-4]

; Access the second parameter
mov eax, [ebp-8]
```

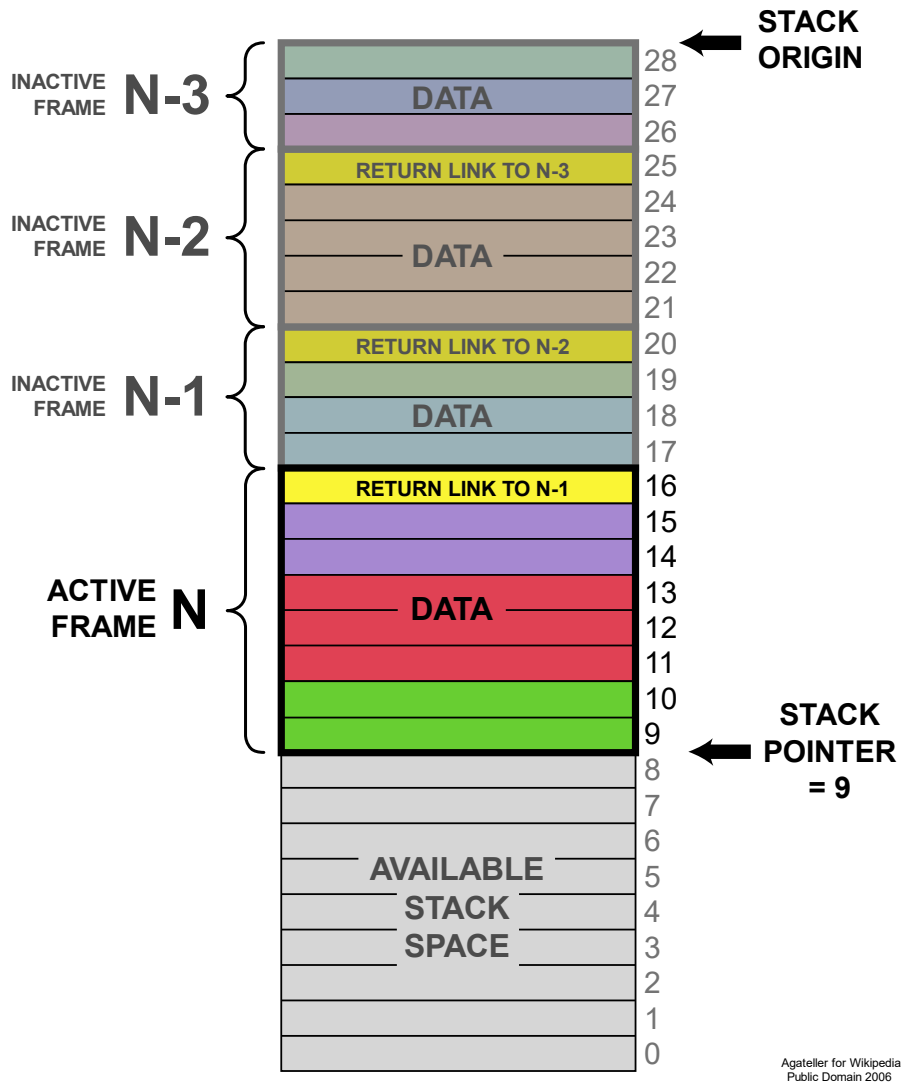
# BASE OFFSET ADDRESSING

The following code is a rewritten and explained implementation of AddTwo using base-offset addressing to access stack parameters:

```
189 ; AddTwo - Add two parameters and return their sum in EAX
190 AddTwo PROC
191     ; Push the base register (EBP) onto the stack
192     push ebp
193     ; Move the stack pointer (ESP) to the base register (EBP)
194     mov ebp, esp
195     ; Calculate the offset of the second parameter (12 bytes from the base of the stack frame)
196     mov eax, 12
197     ; Add the offset to the base register to get the address of the second parameter
198     add eax, ebp
199     ; Load the second parameter into the accumulator (EAX)
200     mov eax, [eax]
201     ; Calculate the offset of the first parameter (8 bytes from the base of the stack frame)
202     mov eax, 8
203     ; Add the offset to the base register to get the address of the first parameter
204     add eax, ebp
205     ; Load the first parameter into the accumulator (EAX)
206     mov eax, [eax]
207     ; Add the first and second parameters
208     add eax, [eax]
209     ; Restore the base register (EBP) from the stack
210     pop ebp
211     ; Return the result in EAX
212     ret
213 AddTwo ENDP
```

The first instruction, **push ebp**, saves the base register (EBP) onto the stack. This is important because EBP will be used as the base register for accessing stack parameters.

The next instruction, **mov ebp, esp**, moves the stack pointer (ESP) to the base register (EBP). This effectively sets the base of the stack frame.




The next two instructions, mov eax, 12 followed by add eax, ebp, work together to find the location of the second parameter on the stack. That parameter sits 12 bytes away from the base of the stack frame. Once the address is calculated, mov eax, [eax] loads the actual value of the second parameter into the EAX register.

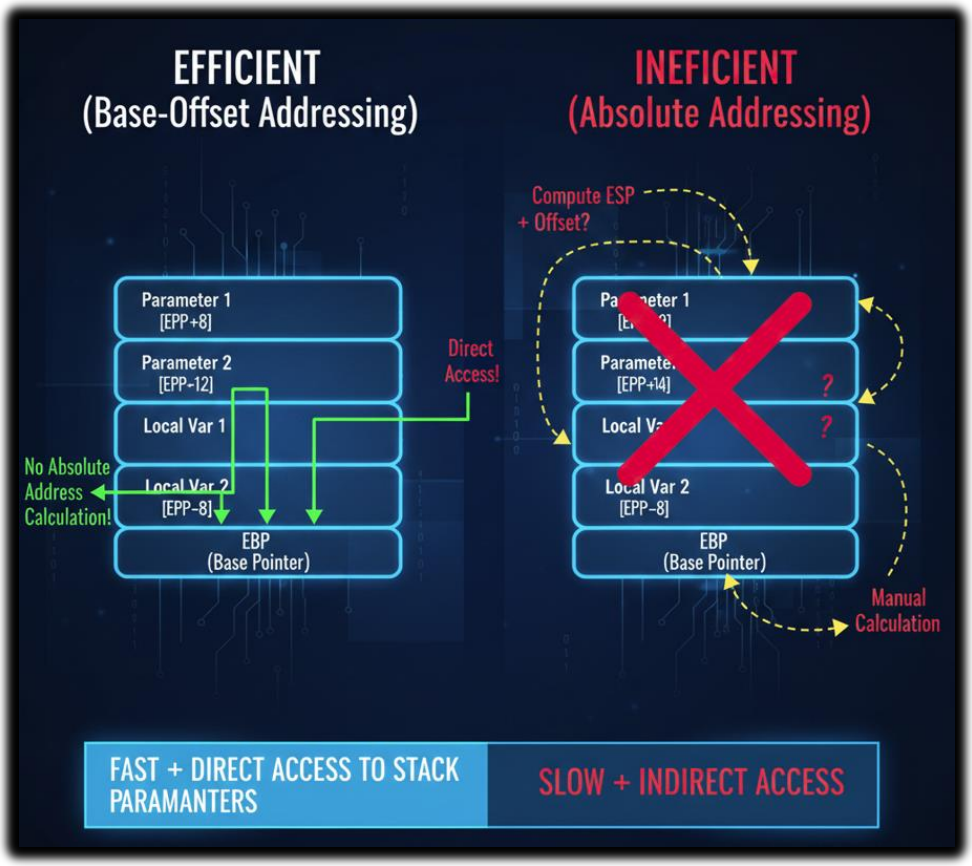
After that, the same process is repeated for the first parameter. The instructions mov eax, 8 and add eax, ebp compute its offset, since the first parameter is located 8 bytes from the base of the stack frame. The instruction mov eax, [eax] then loads the first parameter into EAX.

Next, add eax, [eax] adds the two parameters together. When the computation is done, pop ebp restores the base pointer to its original value.


This step is important because the base register must be restored before the function returns. Finally, the ret instruction exits the function.

Using base-offset addressing has a couple of big advantages. It's efficient because it lets us access stack parameters directly, without needing to compute their absolute memory addresses.

It's also flexible—since everything is referenced relative to the base of the stack frame, the stack can move around without requiring changes to the code that accesses those parameters.



# EXPLICIT STACK PARAMETERS

**Explicit stack parameters** are stack parameters that are referenced by their offset from the base pointer register (EBP).

This is in contrast to implicit stack parameters, which are referenced by their position in the stack frame.

## A Simple Procedure That Adds Two Numbers (Passed on the Stack)

This little procedure takes two arguments (let's call them x and y), adds them together, and returns the sum in EAX.

Both arguments come in on the stack (classic pass-by-value style), and the procedure uses the standard prologue/epilogue to set up and tear down its stack frame.

### I. Here's what the code typically looks like:

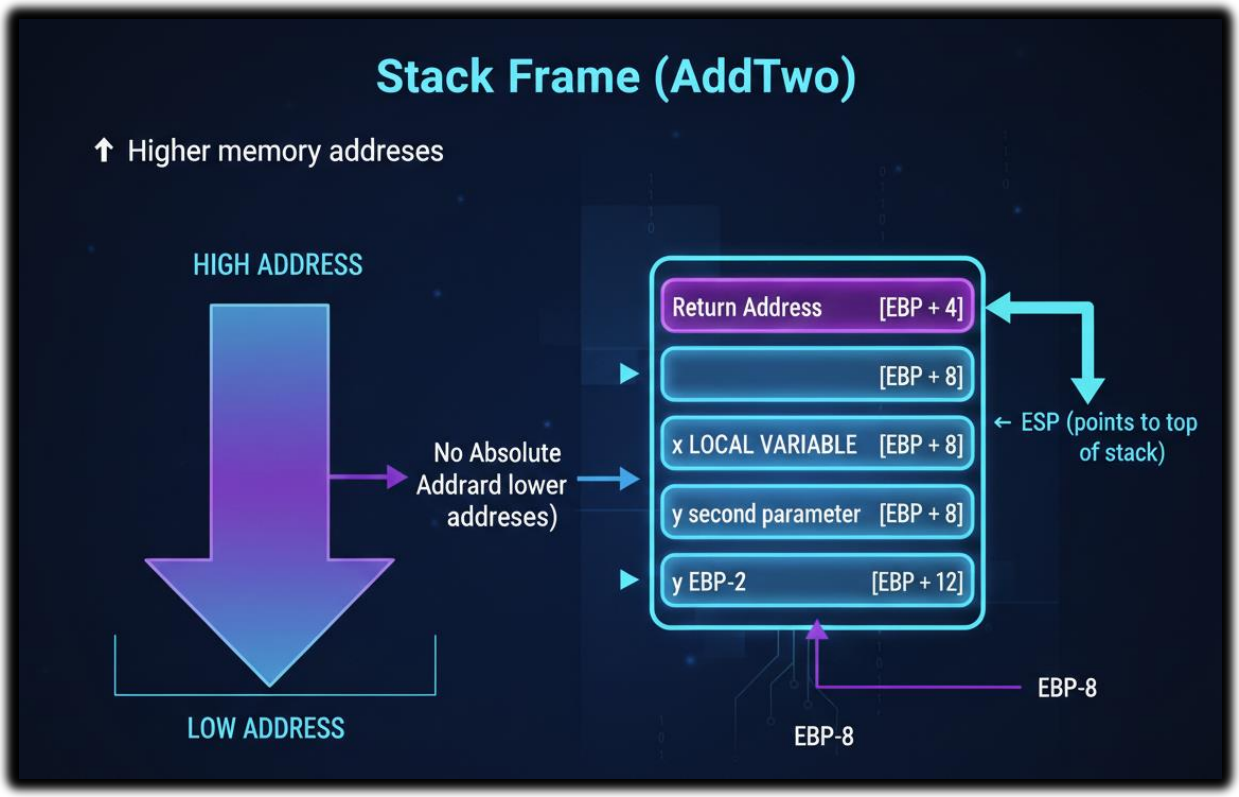
```
227 AddTwo PROC
228   push ebp
229   mov ebp, esp
230   mov eax, [ebp + 12] ; y_param
231   add eax, [ebp + 8] ; x_param
232   pop ebp
233   ret
234 AddTwo ENDP
```

Let's pretend the caller did this:

```
push y      ; push second argument first
push x      ; push first argument last
call AddTwo
```

(That reverse push order is standard in many conventions like stdcall/cdecl.)Right after the call instruction:

- The return address gets pushed automatically
- The stack looks like this (top to bottom, remember stack grows downwards!!!):




## II. Now inside the procedure of the above code:

1. **push ebp**
   Saves the caller's EBP value on the stack (so we don't mess up whoever called us).
   ESP moves down by 4 bytes.

2. **mov ebp, esp**
   Copies the current ESP into EBP.
   Now EBP is locked in place at the bottom of this function's stack frame.
   This is the magic part — EBP becomes our stable reference point. No matter what else gets pushed later, our parameters stay at fixed offsets from EBP.

3. **mov eax, [ebp + 12]**
   Grabs y (the first argument the function sees logically, even though it was pushed first).
   Why +12?

   - [ebp] = saved old EBP
   - [ebp + 4] = return address
   - [ebp + 8] = first pushed arg after return addr → x
   - [ebp + 12] = second pushed arg → y

4. **add eax, [ebp + 8]**
   Adds x to whatever is already in EAX (which is y).
   Result: eax = x + y

5. **pop ebp**
   Restores the caller's original EBP value.
   (In some conventions the caller cleans the stack with add esp, 8 after the call; here we assume the callee doesn't clean params.)

6. **ret**
   Pops the return address into EIP and jumps back to the caller.
   The sum is left in EAX (standard return value register for integers).

## III. Why This Setup Rocks

**Using Symbolic Constants for Explicit Stack Parameters**

The following code uses **symbolic constants** instead of raw offsets for stack parameters.

This approach:

- Improves readability
- Makes the code easier to maintain
- Reduces mistakes when offsets change

In short: **same logic, cleaner code, happier future-you** 😌

```
239 y_param EQU [ebp + 12]
240 x_param EQU [ebp + 8]
241
242 AddTwo PROC
243   push ebp
244   mov ebp, esp
245   mov eax, y_param
246   add eax, x_param
247   pop ebp
248   ret
249 AddTwo ENDP
```

# CLEANING UP THE STACK

## Stack Cleanup in Plain Terms

When a subroutine finishes, its parameters must be removed from the stack. Two approaches exist:


**Explicit cleanup** → The subroutine itself pops parameters off the stack (in reverse order).


**Implicit cleanup** → The caller handles it. The return instruction (RET n) automatically removes *n* bytes from the stack.

The following example shows how to perform **explicit stack cleanup** in the AddTwo subroutine:

```
273 AddTwo PROC
274   push ebp
275   mov ebp, esp
276   ; ...
277   ; Calculate the sum of the two parameters.
278   ; ...
279   pop ebp
280   ret
281 AddTwo ENDP
```

The pop ebp instruction at the end of the subroutine removes the base pointer register (EBP) from the stack.

This is done to restore the original value of EBP, which was pushed onto the stack at the beginning of the subroutine.

The following example shows how to use **implicit stack cleanup** in the AddTwo subroutine:

```
289 AddTwo PROC
290   push ebp
291   mov ebp, esp
292   ; ...
293   ; Calculate the sum of the two parameters.
294   ; ...
295   ret 8
296 AddTwo ENDP
```

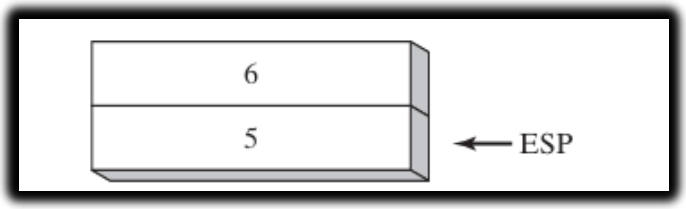
The ret 8 instruction at the end of the subroutine tells the caller to remove 8 bytes from the stack when the subroutine returns.

This is the same as the size of the two parameters that were pushed onto the stack at the beginning of the subroutine.

## Stack OverFlow

Assuming that AddTwo leaves the two parameters on the stack, the following illustration shows the stack after returning from the call:

This image shows the stack after the call AddTwo instruction in main has been executed:




## I. Stack Growth Problem

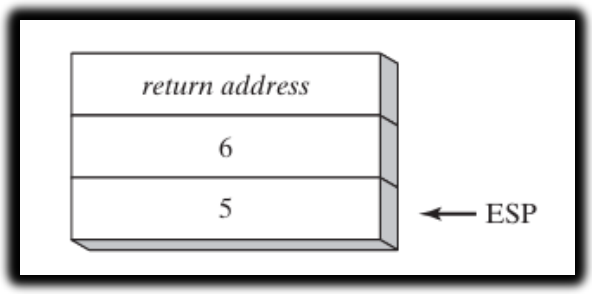
- Every call to AddTwo consumes **12 bytes** on the stack:
  - 4 bytes per parameter (2 parameters = 8 bytes)
  - 4 bytes for the return address
- If AddTwo is called repeatedly in a loop without proper cleanup, the stack keeps growing → **stack overflow risk**.
- The issue gets worse if another subroutine (like Example1) calls AddTwo inside it, since each nested call adds more stack usage.

A more serious problem could result if we called Example1 from main, which in turn calls AddTwo:

```
319 main PROC
320   call Example1
321   exit
322 main ENDP
323
324 Example1 PROC
325   push 6
326   push 5
327   call AddTwo
328   ret                    ;stack is corrupted
329 Example1 ENDP
```

In the image below, the return address for the **call AddTwo** instruction is still on the stack. This is because the AddTwo subroutine did not perform any stack cleanup.

When the RET instruction in Example1 is about to execute, ESP points to the integer 5 rather than the return address that would take it back to main:




## II. Stack Cleanup Failure and Its Consequences

In the image below, the **return address for the AddTwo call** is still on the stack.
This happens because the AddTwo subroutine **does not perform stack cleanup** before returning.

When the RET instruction in **Example1** is about to execute, ESP is pointing to the integer **5** instead of the correct return address that should transfer control back to main.

As a result:

- The RET instruction loads the value **5** into the instruction pointer (EIP)
- The processor attempts to jump to memory address **0x00000005**

Because this address lies **outside the program's code segment**, the processor raises a **runtime exception**, and the operating system terminates the program.

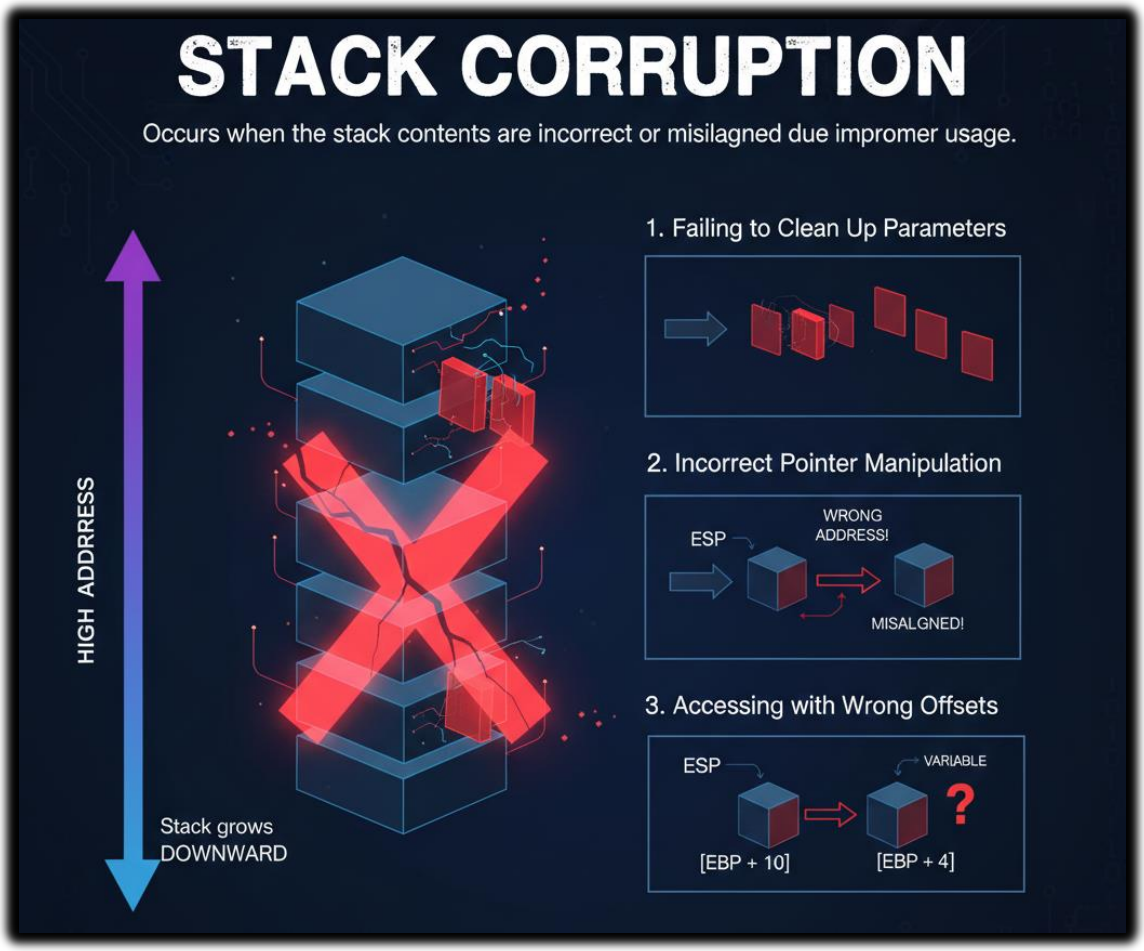
This behavior is a classic example of **stack corruption**.

# STACK CORRUPTION VS STACK OVERFLOW

## I. Stack corruption

Occurs when the stack contents are incorrect or misaligned due to improper usage. Common causes include:

- Failing to clean up stack parameters
- Incorrect stack pointer manipulation
- Accessing the stack using wrong offsets

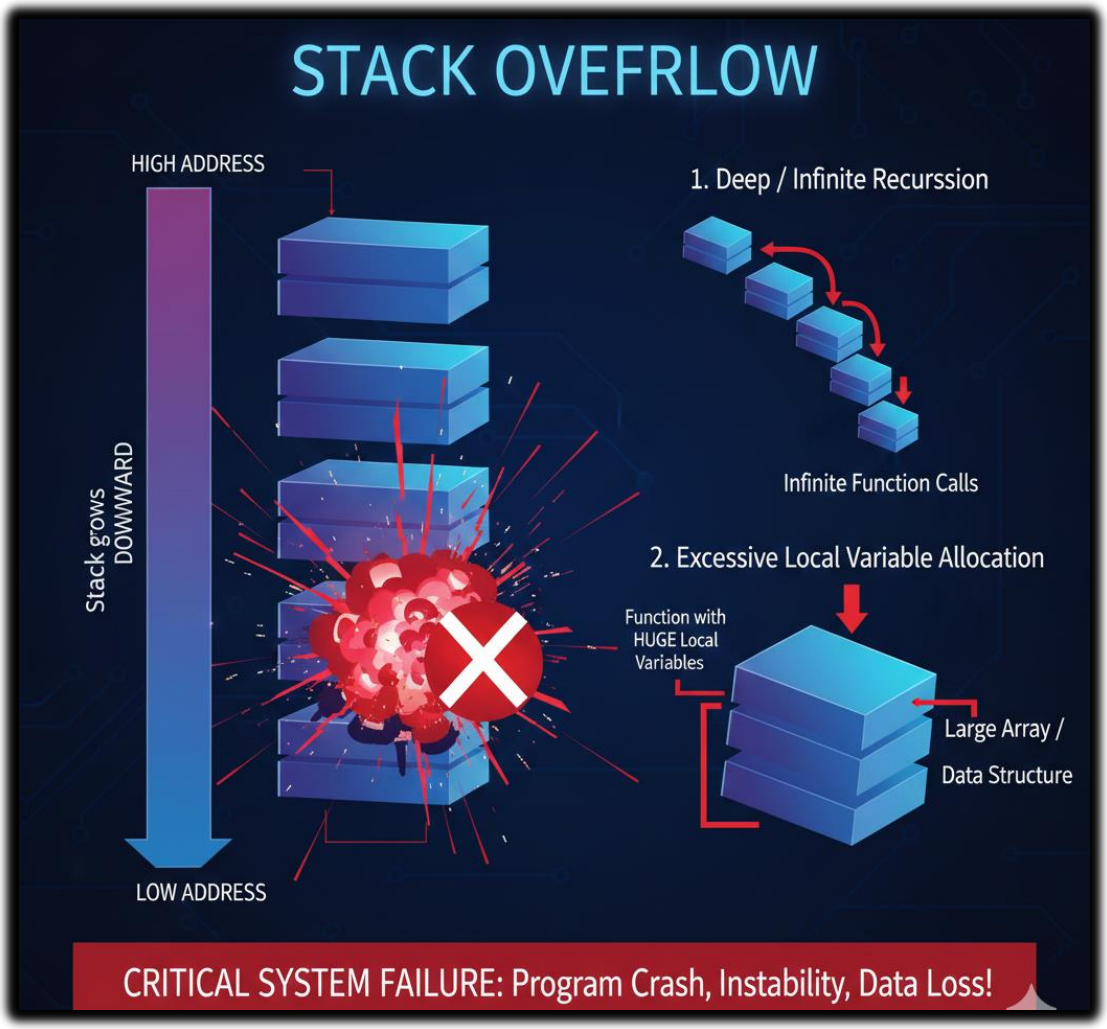


## II. Stack overflow

Occurs when the stack exceeds its allocated memory space.
This typically happens due to:

- Deep or infinite recursion
- Excessive local variable allocation

Note: Leaving parameters on the stack does **not directly cause stack overflow**, but it *does* corrupt the stack and may eventually lead to undefined behavior or crashes.



## III. Preventing Stack Corruption

To prevent stack corruption:

- Subroutines **must clean up the stack** before returning
- Stack cleanup can be:
  - **Implicit** (handled by the callee using ret n)
  - **Explicit** (handled by the caller using add esp, n)

In **Example1**, the subroutine does not clean up its parameters.

As a result, when RET executes, it uses an incorrect value as the return address, leading to a runtime exception and program termination.


# CALLING CONVENTIONS

This line is the problem 👇

*"With the C calling convention, the caller has to tell the called function how many parameters it is passing."*

That's **not how it works**, and that phrasing will absolutely confuse you later.

The callee **never needs to be told** how many parameters there are.
The real difference is **who adjusts the stack pointer after the call**.

## C (cdecl) Calling Convention

- **Parameter passing:**
  Arguments are pushed onto the stack **in reverse order** (right to left).
- **Stack cleanup:**
  The **caller** cleans up the stack after the function returns
  (usually with add esp, N).
- **Why this exists:**
  Because the caller cleans the stack, cdecl supports:
  Functions with a **variable number of arguments** (e.g., printf)

## STDCALL Calling Convention

- **Parameter passing:**
  Arguments are pushed onto the stack **in reverse order** (right to left).
- **Stack cleanup:**
  The **callee** cleans up the stack before returning
  (often using ret N).
- **Why this exists:**
  Since the callee knows exactly how many parameters it has, it can safely clean up the stack.

## The Correct Way to Think About It

❌ *Wrong mental model:*

The caller tells the function how many arguments it passed

✅ *Right mental model:*

- Both sides already know the function's signature at **compile time**
- The only difference is **who adjusts ESP after the call**

## One-Line Memory Trick

- **cdecl** → *caller* cleans
- **stdcall** → *standard callee cleans*

## Why This Matters (and Why Your Earlier Crash Happened)

If:

- The caller thinks the **callee** will clean the stack
- But the callee **doesn't**

Then:

- ESP points to the wrong value
- RET jumps to garbage
- 💥 runtime exception

Which is *exactly* what you just analyzed earlier. So yeah — you're connecting the dots correctly.

Here is an example of a function call using the C calling convention:

```
337 int AddTwo(int a, int b) {
338   return a + b;
339 }
340
341 int main() {
342   int a = 5;
343   int b = 6;
344
345   ;Push the parameters onto the stack in reverse order.
346   push(b);
347   push(a);
348
349   ;Call the AddTwo function.
350   call AddTwo;
351
352   ;Add the size of the parameters to the stack pointer to clean up the stack.
353   add esp, 8;
354
355   ;Store the return value in a variable.
356   int result = eax;
357
358   ; ...
359 }
```

Here is an example of a function call using the STDCALL calling convention:

```
361 int AddTwo(int a, int b) {
362   return a + b;
363 }
364
365 int main() {
366   int a = 5;
367   int b = 6;
368
369   ;Push the parameters onto the stack in reverse order.
370   push(b);
371   push(a);
372
373   ;Call the AddTwo function.
374   call AddTwo;
375
376   ;...
377 }
```

## Key Difference Between cdecl and stdcall

The only practical difference between the **C (cdecl)** and **STDCALL** calling conventions is **who cleans up the stack** after a function call:

- **cdecl** → the **caller** cleans the stack
- **stdcall** → the **callee** (called function) cleans the stack

The **STDCALL calling convention** is used extensively by the **Windows API**, so understanding it is essential when writing programs that interact with Windows system functions.

## Optional: Other Calling Conventions (Reference Only)

The following calling conventions exist but are **out of scope** for this discussion and are not required to understand 32-bit stack frames:

- Pascal
- FORTRAN
- x64 System V / Microsoft x64
- Specialized OS or ABI-specific conventions

Stop there. No more explanations for these.

# SAVING AND RESTORING REGISTERS

## SAVING AND RESTORING REGISTERS, YEAH MY BUOY/GUUURRRL

Subroutines often **save the contents of registers on the stack** before modifying them.

This is good practice because it allows the original register values to be **restored before returning**, ensuring that the caller's state is not accidentally corrupted.

In other words:

A subroutine should leave the CPU exactly how it found it (except for return values).

### When Should Registers Be Saved?

The **ideal time** to save registers is:

- **After** setting up the stack frame
- **Before** allocating space for local variables

Why?

- Once EBP is set to ESP, it becomes a **fixed reference point**
- The stack grows **below EBP**
- Pushing registers below EBP does **not affect parameter offsets**

This guarantees that function parameters remain accessible at the same offsets throughout execution.

## Example: Saving and Restoring Registers

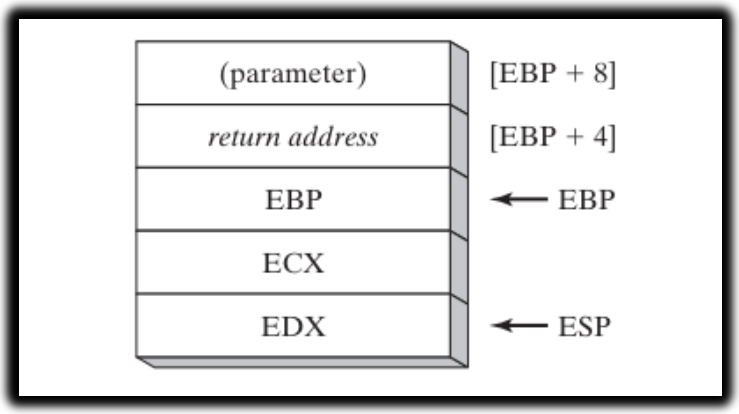
Here is a diagram of a stack frame for the MySub procedure:

```
MySub PROC
    push ebp
    mov  ebp, esp

    push ecx
    push edx

    ; ---- subroutine body goes here ----

    pop  edx
    pop  ecx
    pop  ebp
    ret
MySub ENDP
```




## What This Code Is Doing (Step by Step)

1. **push ebp**
   Saves the caller's base pointer on the stack.
2. **mov ebp, esp**
   Establishes a new stack frame for MySub.
   From this point on, EBP is the stable base of the frame.
3. **push ecx / push edx**
   Saves the values of registers that the subroutine intends to use.

4. **Subroutine work happens here**
   The function is free to modify ECX and EDX.
5. **pop edx / pop ecx**
   Restores the original register values.
6. **pop ebp**
   Restores the caller's stack frame.
7. **ret**
   Pops the return address into EIP and transfers control back to the caller.

## Stack Frames (Clean Explanation)

A **stack frame** is a portion of the stack dedicated to a single subroutine call.
It exists **only while the function is executing**.

A stack frame typically contains:

- Function parameters
- The return address
- Saved EBP
- Saved registers
- Local variables

The **EBP register** points to the **base of the stack frame**, allowing parameters and locals to be accessed using fixed offsets.

## Stack Layout for MySub (Conceptual)

After the prologue and register saves, the stack looks like this:

- [EBP + 8] → first parameter
- [EBP + 4] → return address
- [EBP] → saved EBP
- [EBP - 4] → saved ECX
- [EBP - 8] → saved EDX
- ESP → top of the stack

**Key idea:** Parameters are **above EBP**, saved registers and locals are **below EBP**.

## Important Clarification (Fixing a Major Bug)

❌ **Wrong:**

ECX and EDX are callee-saved registers

✅ **Correct:**

- In **32-bit x86**, ECX and EDX are **caller-saved**
- They are saved here **by choice**, not by convention

This is allowed and often done for safety, but it is **not required by the ABI**.

## Accessing Parameters and Locals

Once EBP is established:

- Parameters are accessed using **positive offsets**
- Locals and saved registers use **negative offsets**

Example:

```
mov eax, [ebp + 8]   ; first parameter
mov ebx, [ebp - 4]   ; saved ECX
```

## Returning from the Subroutine

When MySub finishes:

- Saved registers are restored
- The old EBP is restored
- RET jumps back to the caller using the saved return address

The caller then resumes execution normally.

## Conclusion

Saving and restoring registers:

- Protects the caller's state
- Prevents stack corruption
- Makes nested calls safe and predictable

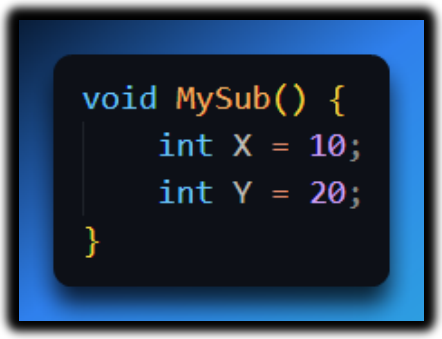
Stack frames:

- Provide structure
- Enable stable parameter access
- Are the backbone of function calls in 32-bit systems

This version is **tight, correct, and aligned** with everything you've already built.

# LOCAL VARIABLES IN ASSEMBLY

When a C++ function declares **local variables**, they are allocated on the **stack** when the function is called. For example, consider this C++ function:



When compiled to assembly, the generated code typically looks like this:

```
MySub PROC
    push ebp
    mov  ebp, esp
    sub  esp, 8             ; allocate space for X and Y
    mov  DWORD PTR [ebp-4], 10   ; X = 10
    mov  DWORD PTR [ebp-8], 20   ; Y = 20
    mov  esp, ebp           ; deallocate locals
    pop  ebp
    ret
MySub ENDP
```

## Step-by-Step Explanation

1. **push ebp**
   Saves the caller's base pointer onto the stack so the function can restore it before returning.

2. **mov ebp, esp**
   Establishes a new **stack frame** for MySub. From now on, EBP is the stable reference point for accessing local variables and parameters.

3. **sub esp, 8**
   Allocates 8 bytes of space for the two local variables X and Y (4 bytes each for 32-bit integers).

4. **mov DWORD PTR [ebp-4], 10**
   Initializes local variable X with 10.
   Offset -4 means 4 bytes **below EBP**.

5. **mov DWORD PTR [ebp-8], 20**
   Initializes local variable Y with 20.
   Offset -8 means 8 bytes **below EBP**.

6. **mov esp, ebp**
   Deallocates the local variables from the stack.

7. **pop ebp**
   Restores the caller's base pointer.

8. **ret**
   Returns to the caller using the saved return address on the stack.

## Stack Frame Layout for MySub

Here's a conceptual view of the stack frame after the prologue and local allocation:

↑ HIGH MEMORY ADDRESSES

| Stack Content | Size (bytes) | Offset from EBP |
|---|---|---|
| First Parameter (if any) | 4 | [EBP + 8] |
| Return Address | 4 | [EBP + 4] |
| ➤ Saved EBP (Frame Pointer) | 4 | [EBP] |
| Local Variable X | 4 | [EBP - 4] |
| Local Variable Y | 4 | [EBP - 8] |

↓ LOW MEMORY ADDRESSES (STACK GROWS HERE)

Notes:

- **Local variables** live below EBP (negative offsets).
- **Parameters** live above EBP (positive offsets).
- The stack grows **downward** in memory.

## Accessing Local Variables

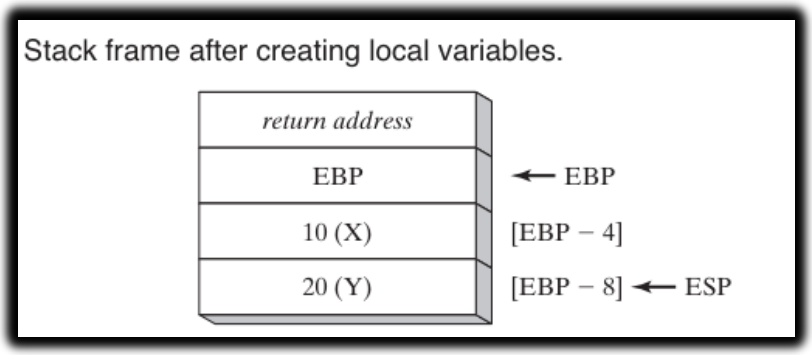
You can access a local variable using its offset from EBP. For example:

```
mov eax, [ebp-4]   ; Load X into EAX
mov ebx, [ebp-8]   ; Load Y into EBX
```

Because EBP is fixed, the offsets to locals remain constant regardless of any pushes/pops that happen elsewhere in the function.

## Key Takeaways

- **EBP acts as the anchor** for all stack frame references.
- **Local variables** are allocated below EBP.
- **Parameters** are accessed above EBP.
- The stack frame is **created at function entry** and **destroyed at function exit**.
- Proper stack frame management prevents **stack corruption** and makes nested function calls safe.




# LOCAL VARIABLE SYMBOLS IN ASSEMBLY

When working with local variables in assembly, it can be tedious and error-prone to repeatedly reference them by **stack offsets** like [ebp-4] or [ebp-8].

A better practice is to define a **symbol** for each local variable using the EQU directive. This gives a meaningful name to the stack offset.

## Defining a Symbol

For example, to define a symbol for a local variable X at offset -4 from EBP:

```
X_local EQU DWORD PTR [ebp-4]
```

- X_local is now a symbolic name representing the memory location of X.
- [ebp-4] indicates that the variable is 4 bytes **below the base pointer**.
- DWORD PTR specifies that the variable is 32 bits (4 bytes).

## Using the Symbol

Instead of writing:

```
mov eax, [ebp-4]  ; load X into EAX
mov [ebp-4], 16   ; store 16 into X
```

You can use the symbol:

```
mov eax, X_local   ; load X into EAX
mov X_local, 16    ; store 16 into X
```

This improves **readability** and **maintainability**, especially for functions with multiple locals.

## Full Example: MySub with Symbolic Local

```
X_local EQU DWORD PTR [ebp-4]
Y_local EQU DWORD PTR [ebp-8]

MySub PROC
    push ebp
    mov  ebp, esp
    sub  esp, 8           ; allocate locals
    mov  X_local, 16      ; initialize X
    mov  Y_local, 26      ; initialize Y
    mov  esp, ebp         ; deallocate locals
    pop  ebp
    ret
MySub ENDP
```

**Benefits of using symbols:**

1. Easier to read — [ebp-4] becomes X_local.
2. Easier to maintain — if the offset changes, you only update the EQU directive.
3. Reduces errors — you avoid miscalculating offsets manually.

# REFERENCE PARAMETERS

## REFERENCE PARAMETERS AND THE ArrayFill PROCEDURE

In assembly, **reference parameters** are passed to a procedure by **address**. Instead of receiving a copy of a variable, the procedure gets a pointer to the variable in the caller's scope. This allows the procedure to **directly modify** the caller's data.

Reference parameters are accessed using **base-pointer offsets** relative to the EBP register. Since parameters are pushed onto the stack in **reverse order**, the first reference parameter is typically located **12 bytes above EBP** ([ebp+12]).

### I. Example: ArrayFill Procedure

The ArrayFill procedure fills an array with a pseudorandom sequence of 16-bit integers. It takes:

1. **Pointer to the array** (passed by reference)
2. **Array length** (passed by value)

## II. Procedure Structure

```
ArrayFill PROC
    ; --- Prologue: Save stack frame ---
    push ebp
    mov  ebp, esp

    ; --- Save general-purpose registers ---
    pushad

    ; --- Get parameters ---
    mov  esi, [ebp+12]  ; pointer to array
    mov  ecx, [ebp+8]   ; array length

    ; --- Loop to fill array ---
L1:
    mov  eax, 16666h     ; seed / constant for RandomRange
    call RandomRange     ; generate random value
    mov  [esi], ax       ; store value in array
    add  esi, TYPE WORD  ; move to next element
    loop L1              ; repeat until ECX = 0

    ; --- Epilogue: Restore registers ---
    popad
    pop ebp
    ret
ArrayFill ENDP
```

## III. Step-by-Step Breakdown

**Procedure Declaration:** ArrayFill PROC begins the procedure.

**Prologue – Setting Up Stack Frame:**

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

Saves the caller's base pointer and establishes a new stack frame.

**Save Registers:**

```
pushad
```

Saves all general-purpose registers (EAX, ECX, EDX, EBX, ESI, EDI, EBP) so the procedure doesn't disturb the caller's state.

**Load Parameters:**

```
mov esi, [ebp+12]  ; array pointer
mov ecx, [ebp+8]   ; array length
```

[ebp+12] = address of the array (reference parameter)

[ebp+8] = length of the array (value parameter)

**Array Filling Loop:**

- Loads a constant into EAX
- Calls RandomRange to generate a pseudorandom number
- Stores the number in the array [ESI]
- Moves ESI to the next element (add esi, TYPE WORD)
- Decrements ECX and loops (loop L1) until all elements are filled

**Restore Registers:**

```
popad
pop ebp
ret
```

Restores general-purpose registers and previous stack frame, then returns control to the caller.

## Key Concepts

- **Reference Parameters:** The procedure can modify the caller's variable directly.
- **Base-Offset Addressing:** Access parameters via [ebp+offset].
- **Stack Discipline:** Prologue (push ebp; mov ebp, esp) and epilogue (pop ebp; ret) ensure stack integrity.
- **Register Preservation:** pushad / popad protect caller state during execution.
- **Array Processing:** Use ESI as a pointer to walk through the array.

## Why Reference Parameters Matter

- Enables **efficient data manipulation** without copying large structures.
- Makes **procedures reusable** and flexible for different caller data.
- Combined with proper stack handling, ensures **safe, predictable execution** in assembly.

# LEA INSTRUCTION AND STACK PARAMETERS

In assembly language, the OFFSET directive lets you obtain the address of a variable or label **at compile time**. However, this **does not work with stack parameters** because their addresses are **unknown until runtime**.

For example, the following will **not assemble**:

```
mov esi, OFFSET [ebp-30]
```

This fails because EBP is a **runtime register** pointing to the top of the current stack frame. While the local variable myString has a fixed offset -30 from EBP, the actual memory address of EBP is unknown at compile time.

## Example in C++

```cpp
void makeArray() {
    char myString[36];
    for (int i = 0; i < 30; i++)
        myString[i] = '*';
}
```

This code initializes the first 30 elements of myString with the character '*'. After execution, myString contains 30 asterisks.

## Using LEA in Assembly

The LEA (Load Effective Address) instruction allows you to **calculate the address of a stack variable at runtime**. Once loaded, this address can be used to access the variable safely.

Equivalent assembly code for the makeArray function:

```asm
makeArray PROC
    push ebp
    mov  ebp, esp
    sub  esp, 36          ; allocate space for myString (36 bytes)

    lea  esi, [ebp-36]    ; load address of myString into ESI
    mov  ecx, 36          ; loop counter

L1:
    mov  BYTE PTR [esi], '*' ; fill one byte
    inc  esi                 ; move to next byte
    loop L1                  ; repeat until ECX = 0

    add  esp, 36           ; deallocate myString
    pop  ebp
    ret
makeArray ENDP
```

## Step-by-Step Explanation

Prologue: Setup Stack Frame.

- Saves the caller's EBP
- Establishes a new stack frame
- Reserves space for myString on the stack

Load Address of Stack Variable

- Computes the **effective address** of myString
- Stores it in ESI
- Unlike OFFSET, this works at **runtime**

Loop to Fill Array

- The instruction mov BYTE PTR [esi], '*' stores the character '*' into the memory location pointed to by the ESI register.
- The instruction inc esi increments ESI to point to the next byte in memory.
- The loop L1 instruction decreases the ECX register by 1 and, if ECX is not zero, jumps back to the label L1 to repeat the process.
- The loop continues until ECX reaches zero, meaning all bytes in the sequence have been set to '*'.
- After the loop, the epilogue restores the stack to its state before the function call.

Epilogue: Restore Stack

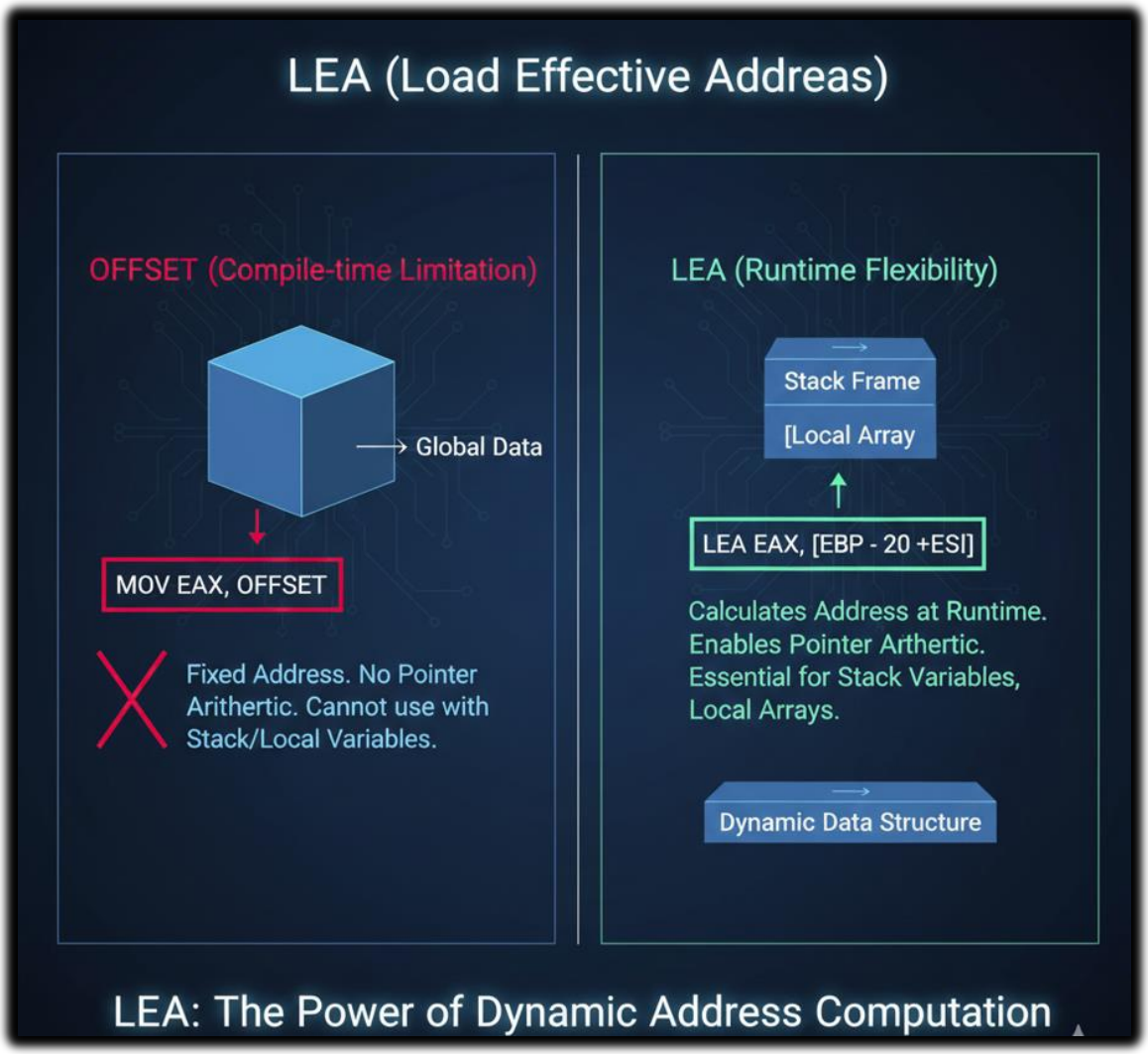
```
add  esp, 36
pop  ebp
ret
```

- Deallocates myString
- Restores previous stack frame
- Returns to caller

## Key Concepts

- **OFFSET**: Compile-time addresses (cannot be used for stack locals).
- **LEA**: Calculates **effective addresses at runtime**.
- **Stack Parameters**: Accessed via [EBP ± offset], but actual address is runtime-dependent.
- **Efficient Array Access**: LEA + pointer register (like ESI) allows dynamic manipulation of stack-based arrays.

✅ **Takeaway:** LEA is essential for working with **stack variables, local arrays, and dynamic data structures** in assembly. It avoids the compile-time limitations of OFFSET and makes pointer arithmetic possible.



# ENTER AND LEAVE INSTRUCTIONS

**ENTER and LEAVE Instructions**

The ENTER and LEAVE instructions are used to **manage stack frames** in assembly language. ENTER **creates** a stack frame for a procedure, and LEAVE **destroys** it when the procedure finishes.

## ENTER Instruction

```
ENTER bytes, nestingLevel
```

- bytes: Number of bytes to reserve for local variables.
- nestingLevel: Lexical nesting level (usually 0 for most functions).

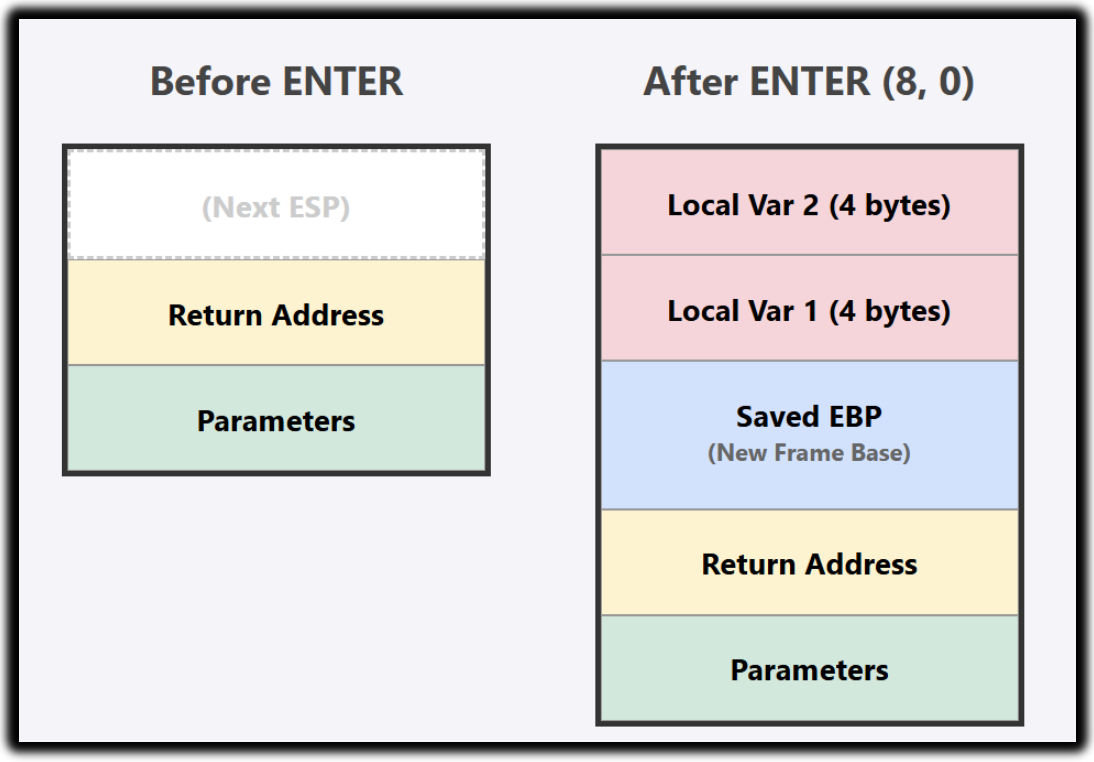
## What it does:

1. Pushes the current EBP onto the stack (saving the caller's base pointer).
2. Copies ESP into EBP to set the base of the new stack frame.
3. Allocates bytes of stack space for local variables.

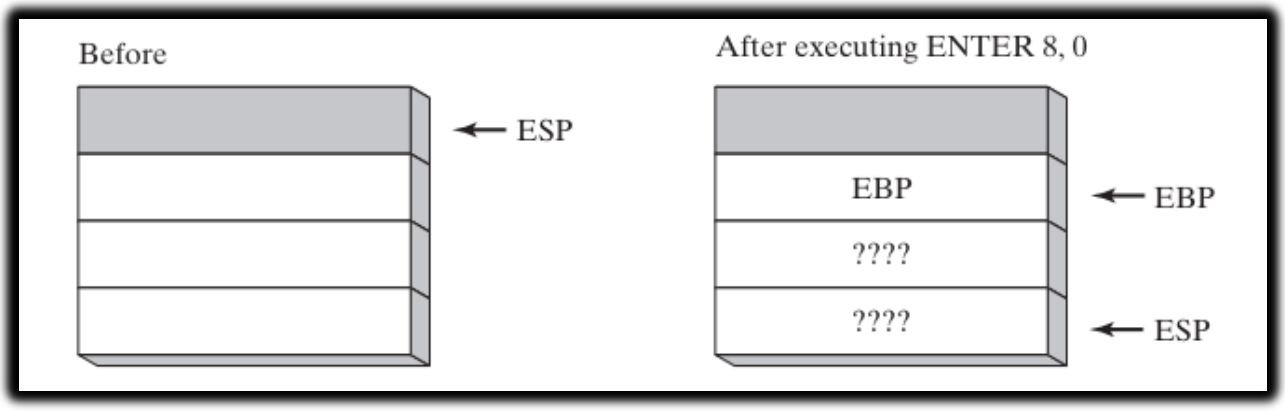
## Example:

```
416 MySub PROC
417     enter 8, 0 ; Reserve 8 bytes of stack space for local variables.
418     ; ...
419     leave ; Destroy the stack frame.
420     ret
421 MySub ENDP
```

## Stack effect:




Or




## LEAVE Instruction

- Restores the stack frame created by ENTER.
- Performs two actions:
  1. Sets ESP back to the current EBP (removes local variables).
  2. Pops the saved EBP value from the stack (restores the caller's base pointer).

## Key Point:

ENTER and LEAVE **must be used together**. Using ENTER without LEAVE will leave local stack space allocated, which can cause **stack growth** and potentially **crash the program**.

## Why Use ENTER/LEAVE?

- They **simplify stack frame management** compared to manually pushing/popping EBP and adjusting ESP.
- They are particularly useful for procedures with **nested calls** or **multiple local variables**.
- They make code **readable and maintainable**, while ensuring proper cleanup of the stack.

## Visual Summary

- **ENTER 8, 0**:
  - Push EBP
  - Set EBP = ESP
  - Reserve 8 bytes for locals
- **LEAVE**:
  - ESP = EBP (remove locals)
  - Pop EBP (restore caller frame)

The accompanying image shows the stack before and after executing ENTER, illustrating saved EBP and reserved local space.

## ✅ Takeaway:

Always pair ENTER with LEAVE to ensure proper stack frame creation and destruction. Mismanaging this can lead to stack corruption or runtime crashes.

# LOCAL DIRECTIVE

## LOCAL Directive in Assembly Language

The LOCAL directive is used to **declare local variables** within a procedure. These variables exist only in the procedure where they are declared and **reserve space on the stack**.

```
LOCAL varlist
```

- varlist is a comma-separated list of variable definitions.
- Each variable definition has the form:

```
label:type
```

**Example:** Declare a BYTE variable:

```
MySub PROC
LOCAL var1:BYTE
```

**Example:** Declare multiple variables of different types:

```
BubbleSort PROC
LOCAL temp:DWORD, SwapFlag:BYTE
```

**Example:** Declare a pointer to a 16-bit integer:

```
Merge PROC
LOCAL pArray:PTR WORD
```

**Example:** Declare an array of 10 doublewords:

```
LOCAL TempArray[10]:DWORD
```

## Key Points

1. **Placement:** Must be immediately after the PROC directive.
2. **Stack Space Allocation:**
   - The amount of stack space reserved depends on the type of each variable:
     - BYTE → 1 byte
     - WORD → 2 bytes
     - DWORD → 4 bytes
3. **Scope:** Local variables are only accessible within the procedure. They **cannot be accessed by other procedures**.
4. **Not a Stack Frame:**
   - The LOCAL directive **does not create a stack frame**.
   - Use ENTER to create a stack frame and LEAVE to destroy it.
5. **Best Practice:** Always declare all local variables using LOCAL to improve **readability and maintainability**.

## Example in context:

```
MySub PROC
    LOCAL x:DWORD, y:DWORD, flag:BYTE
    ; procedure code using x, y, flag
    RET
MySub ENDP
```

- This reserves 9 bytes of stack space (4 + 4 + 1) for local variables.
- The variables x, y, and flag exist only during the execution of MySub.

## ✅ Takeaway:

The LOCAL directive is a convenient way to declare local variables of any type, including arrays and pointers.

It ensures stack space is reserved and keeps code organized, but **it must be combined with ENTER/LEAVE** if you want a full stack frame.

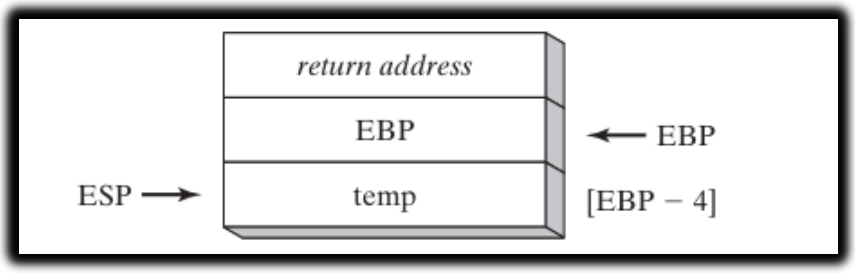
The following is a more in-depth explanation of the MASM code generation for the LOCAL directive:

```
485 Example1 PROC
486     LOCAL temp:DWORD
487     mov
488     eax,temp
489     ret
490 Example1 ENDP
```

**MASM generated code:**

```
499 push
500 ebp
501 mov
502 ebp,esp
503 add
504 esp,0FFFFFFFCh
505 ; add -4 to ESP
506 mov
507 eax,[ebp-4]
508 leave
509 ret
```

**Stack frame diagram:**



# MASM Code Generation and Stack Frames

## How MASM Sets Up a Stack Frame

When a procedure is called in MASM, the code generator performs the following steps:

**Save the base pointer (EBP):**

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

- Saves the caller's stack frame pointer.
- Sets EBP as the base of the new stack frame.

**Allocate space for local variables:**

```
sub esp, 4  ; reserve 4 bytes for local variable temp
```

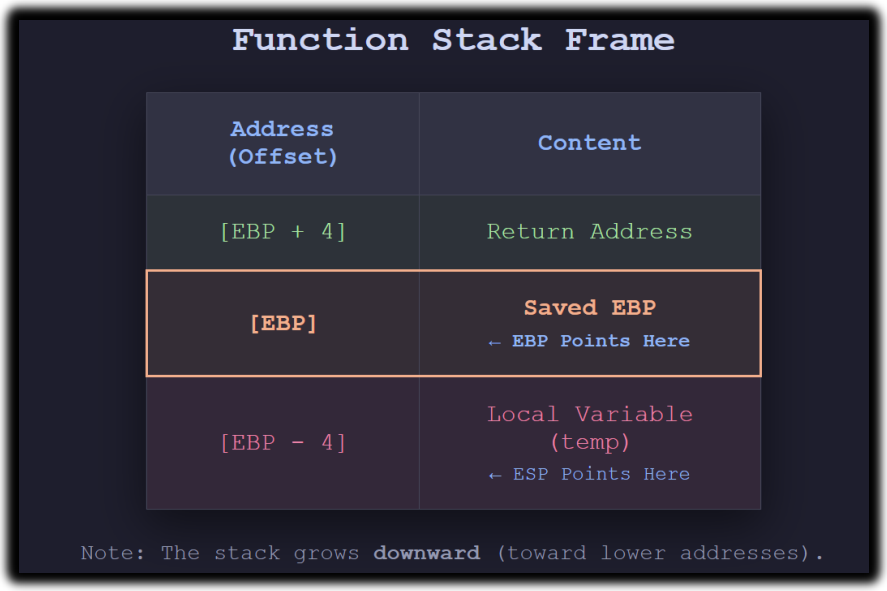
- The stack grows downward.
- Local variables are accessed relative to EBP (e.g., temp is at [ebp-4]).

**Use local variables:**

```
mov eax, [ebp-4]  ; load temp into EAX
```

**Tear down the stack frame:**

```
leave  ; restores ESP and EBP
ret    ; return to caller
```




## Microsoft x64 Calling Convention

The Microsoft x64 calling convention defines how functions pass parameters and return values in **64-bit Windows**:

1. **Parameter passing:**
   - First 4 parameters → RCX, RDX, R8, R9
   - Remaining parameters → pushed onto the stack (left-to-right)
2. **Return values:**
   - Integers ≤ 64 bits → RAX
   - Larger or complex types → placed on the stack; RCX points to the return location
3. **Stack requirements:**
   - At least 32 bytes of **shadow space** must be allocated by the caller
   - RSP must be **16-byte aligned** before a call
4. **Register preservation:**
   - Caller-saved (may be overwritten): RAX, RCX, RDX, R8, R9, R10, R11
   - Callee-saved (must be preserved by function): RBX, RBP, RDI, RSI, R12-R15
5. **CALL instruction:** subtracts 8 from RSP (64-bit address)

Understanding this is critical for writing **64-bit Windows programs** or calling Windows API functions.

## Stack Frames and Parameters

**Subroutine Stack Frame**

A subroutine's stack frame contains:

- Caller's **return address**
- Subroutine's **local variables**
- Optionally, saved registers

## True/False Quiz Highlights:

1. **Stack frame always contains return address + locals → True**
2. **Arrays passed by reference to avoid copying → True**
3. **Prologue pushes EBP → True**
4. **Local variables created by adding positive value to ESP → True** (actually usually subtracting since stack grows down)
5. **Last argument at [EBP+8] in 32-bit → False** (correct: last argument is [EBP+4])
6. **Passing by reference stores argument's address on the stack → True**

## Stack Parameter Types

1. **Value parameters**
   - Copied onto the stack
   - Modifications inside function **do not affect caller**
2. **Reference parameters**
   - Address of variable stored on stack
   - Modifications inside function **affect caller**

## C++ Example:

```cpp
void swap_values(int a, int b) {  // value parameters
    int temp = a;
    a = b;
    b = temp;
}

void swap_references(int* a, int* b) { // reference parameters
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;

    swap_values(x, y);      // x=10, y=20
    swap_references(&x,&y); // x=20, y=10
}
```

Value parameters are copied; reference parameters allow functions to modify caller's variables directly.

## ✅ Takeaways:

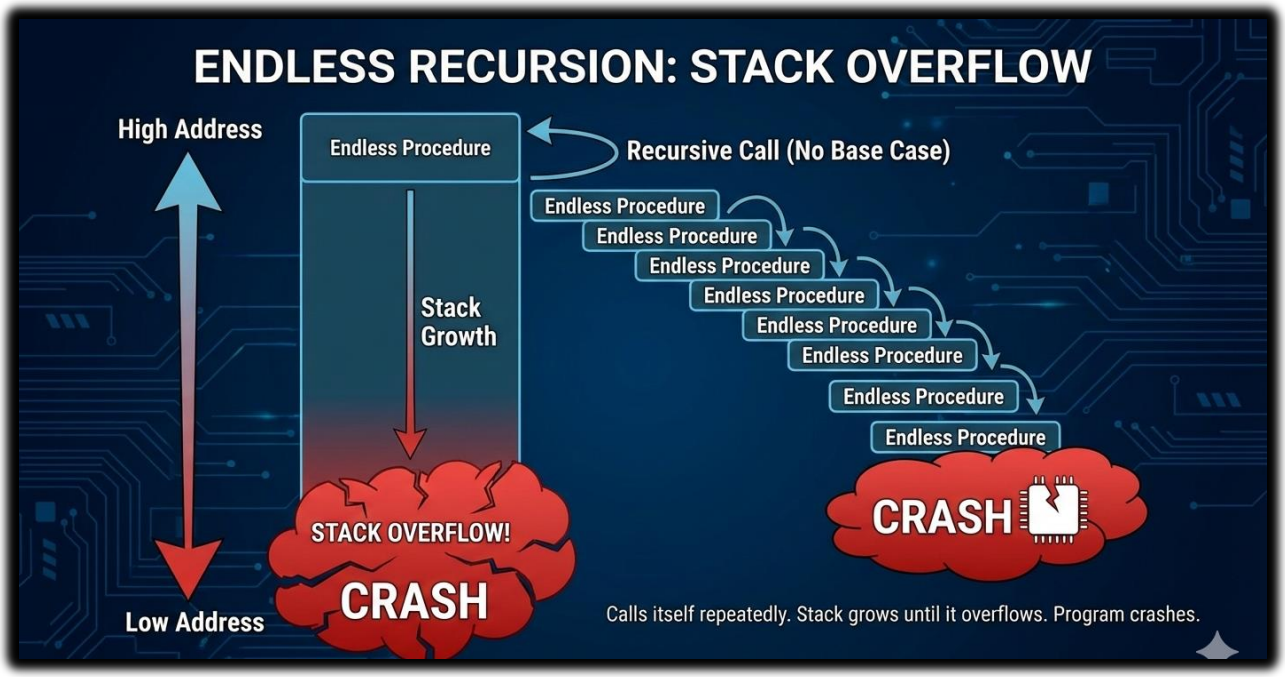
- MASM uses push ebp / mov ebp, esp / sub esp, N for stack frames.
- leave and ret restore the stack and return.
- x64 calling conventions differ significantly from 32-bit, especially with registers.
- Know the difference between value vs reference parameters—they determine whether changes propagate back to the caller.

# RECURSION IN ASSEMBLY LANGUAGE

- Recursion is a programming technique where a function calls itself directly or indirectly.
- It can be a powerful tool for solving complex problems, but it is important to understand how it works and how to avoid writing recursive functions that can cause stack overflows.

## Endless Recursion:

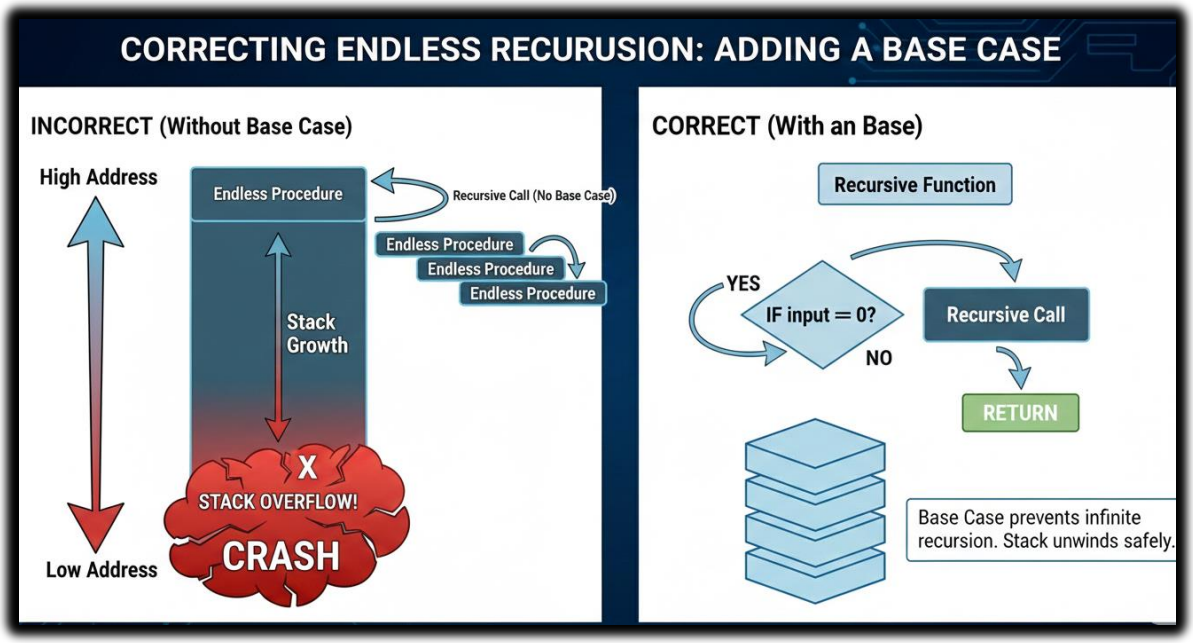
- The example of endless recursion illustrates what can go wrong when recursion is not used correctly.
- The Endless procedure calls itself repeatedly without ever checking for a base case.
- As a result, the stack continues to grow until it overflows, causing the program to crash.




## Correcting Endless Recursion:

- To rewrite the Endless procedure correctly, a base case must be added.
- A base case is a condition that will cause the procedure to terminate instead of calling itself again.
- For the Endless procedure, a suitable base case could be: *"if the input is 0, then return"*.

- Including a base case prevents infinite recursion and ensures the program runs safely.




```asm
; Endless Recursion (Endless.asm)
INCLUDE Irvine32.inc
.data
    endlessStr BYTE "This recursion never stops",0
.code
    main PROC
        call
        Endless
        exit
    main ENDP
    Endless PROC
        mov ecx, 1 ; input parameter
        ; base case
        cmp ecx, 0
        je endless_exit

        ; recursive call
        call Endless

        ; decrement input parameter
        dec ecx
        ; and call again
        jmp Endless

        endless_exit:
        ret
    Endless ENDP
END main
```

Here's your text rewritten as bullet points:

- The rewritten version of the Endless procedure will terminate correctly when the input is 0.
- Before terminating, it will print the message: *"This recursion never stops"* to the console.

## When to Use Recursion:

- Recursion is not suitable for all problems; it can be inefficient and difficult to debug.
- Recursion is a powerful tool for solving problems with repeating patterns.
- It is commonly used in algorithms for traversing linked lists and trees.
- Before using recursion, ensure that the problem is a good fit for a recursive solution.
- Carefully design recursive functions to avoid stack overflows and infinite recursion.

| Pushed on Stack | Value in ECX | Value in EAX |
|---|---|---|
| L1 | 5 | 0 |
| L2 | 4 | 5 |
| L2 | 3 | 9 |
| L2 | 2 | 12 |
| L2 | 1 | 14 |
| L2 | 0 | 15 |

## Recursively Calculating a Sum:

- A recursive procedure is one that calls itself.
- Recursion is useful for solving problems that can be broken down into smaller subproblems of the same type.
- To calculate the sum of integers from 1 to n, a recursive procedure can be used.

```
554 CalcSum(n):
555   if n == 0:
556     return 0
557   else:
558     return n + CalcSum(n - 1)
```

The procedure works by:

- Recursively calling itself to calculate the sum of integers from 1 to n - 1.
- Adding n to the result of that recursive call.

The **base case** occurs when n == 0, in which case the sum is 0.

The following table illustrates a **stack trace** for the recursive call CalcSum(5).

| Stack Frame | ECX (counter) | EAX (sum) |
|---|---|---|
| main() | 5 | 0 |
| CalcSum(5) | 4 | 0 |
| CalcSum(4) | 3 | 4 |
| CalcSum(3) | 2 | 7 |
| CalcSum(2) | 1 | 10 |
| CalcSum(1) | 0 | 11 |

**Stack Frame Behavior:**

- Each recursive call pushes a stack frame onto the stack.
- The stack frame contains the **return address**, which is the address of the next instruction to execute after the call returns.

**Registers:**

- **ECX** contains the counter value for the current recursive call.
- **EAX** contains the sum of integers calculated so far.

**Recursive Call Flow:**

- First recursive call: CalcSum(5) calls CalcSum(4)
  - ECX = 4, EAX = 0
  - The program calculates the sum of integers from 1 to 4 recursively.
- Second recursive call: CalcSum(4) calls CalcSum(3)
  - ECX = 3, EAX = 0
  - The program calculates the sum of integers from 1 to 3 recursively.
- This continues until the **base case** (n == 0) is reached, returning 0.

**Returning from Recursion:**

- The program returns from CalcSum(0) to CalcSum(1), then to CalcSum(2), and so on.
- By the time it returns from CalcSum(5), the sum of integers from 1 to 5 is stored in **EAX**.
- The program can then return this sum from the main() function.

```
571 ;Sum of Integers (RecursiveSum.asm)
572 INCLUDE Irvine32.inc
573 .code
574 main PROC
575     mov ecx, 5   ; Set ECX to 5, the number of integers to sum.
576     mov eax, 0   ; Initialize EAX to 0; it will hold the sum.
577     call CalcSum ; Call the CalcSum function to calculate the sum.
578 L1:
579     call WriteDec ; Display the result in EAX.
580     call Crlf     ; Print a new line.
581     exit
582 main ENDP
583 ;-----------------------------------------------------
584 CalcSum PROC
585     ; Calculates the sum of a list of integers
586     ; Receives: ECX = count
587     ; Returns: EAX = sum
588     ;-----------------------------------------------------
589     cmp ecx, 0    ; Compare ECX (counter) with 0.
590     jz L2        ; If it's zero, jump to L2 and quit.
591     add eax, ecx  ; Add ECX to EAX, updating the sum.
592     dec ecx       ; Decrement the counter.
593     call CalcSum  ; Recursively call CalcSum to process the next integer.
594
595 L2:
596     ret
597 CalcSum ENDP
598 end main
```

## Main Procedure and CalcSum Explanation

- The main procedure initializes:
  - **ECX** to 5, representing the number of integers to sum.
  - **EAX** to 0, which will store the running total (sum).
- It then calls the CalcSum procedure to compute the sum.
- After the calculation:
  - The result is printed using WriteDec.
  - A new line is added using Crlf.
- **CalcSum Procedure:**
  - CalcSum is a recursive procedure that calculates the sum of integers.
  - It checks whether **ECX** (the counter) is zero.
  - If ECX is not zero:
    - The current value of ECX is added to **EAX**.
    - ECX is decremented.
    - CalcSum calls itself recursively.
  - This process continues until ECX reaches 0.
  - When ECX is 0, the procedure returns using ret.

## Factorial of an Integer

- The Factorial procedure uses recursion to calculate the factorial of a number.
- It receives one stack parameter, **N**, which is the value whose factorial is to be calculated.
- The return address of the calling program is automatically pushed onto the stack by the CALL instruction.
- **Stack Frame Setup:**
  - Factorial first pushes **EBP** onto the stack to save the caller's base pointer.
  - It then sets **EBP** to point to the beginning of the current stack frame.
  - This setup allows access to parameters and local variables using base-offset addressing.

- **Base Case:**
  - The base case occurs when **N = 0**.
  - In this case, Factorial returns **1**, since the factorial of 0 is 1.
- **Recursive Case:**
  - If **N ≠ 0**, Factorial calls itself recursively with **N − 1**.
  - This continues until the base case is reached.
- **Returning from Recursion:**
  - After returning from the recursive call, the procedure multiplies the returned value by **N**.
  - This follows the definition: *factorial(N) = N × factorial(N − 1)*

## Example Stack Trace

The following table shows a stack trace for a call to Factorial(3):

| Stack Frame | EBP | ESP | N |
|---|---|---|---|
| main() | 0x00000000 | 0x00000004 | 3 |
| Factorial(3) | 0x00000004 | 0x00000000 | 3 |
| Factorial(2) | 0x00000000 | 0x00000004 | 2 |
| Factorial(1) | 0x00000004 | 0x00000000 | 1 |
| Factorial(0) | 0x00000000 | 0x00000004 | 0 |

## Explanation of the Stack and Recursive Calls (Factorial)

- A stack frame for each recursive call is pushed onto the stack when the CALL instruction is executed.
- Each stack frame contains:
  - The return address (the next instruction to execute after the call returns)
  - The saved base pointer (EBP)
  - The parameter **N**
- **Registers:**
  - **EBP** contains the base pointer for the current stack frame.
  - **ESP** contains the stack pointer, which points to the top of the stack.
  - **N** is a parameter stored on the stack and accessed using base-offset addressing (e.g., [EBP + 8]).
  - **EAX** is used to store the return value of the function.

## Recursive Execution Flow

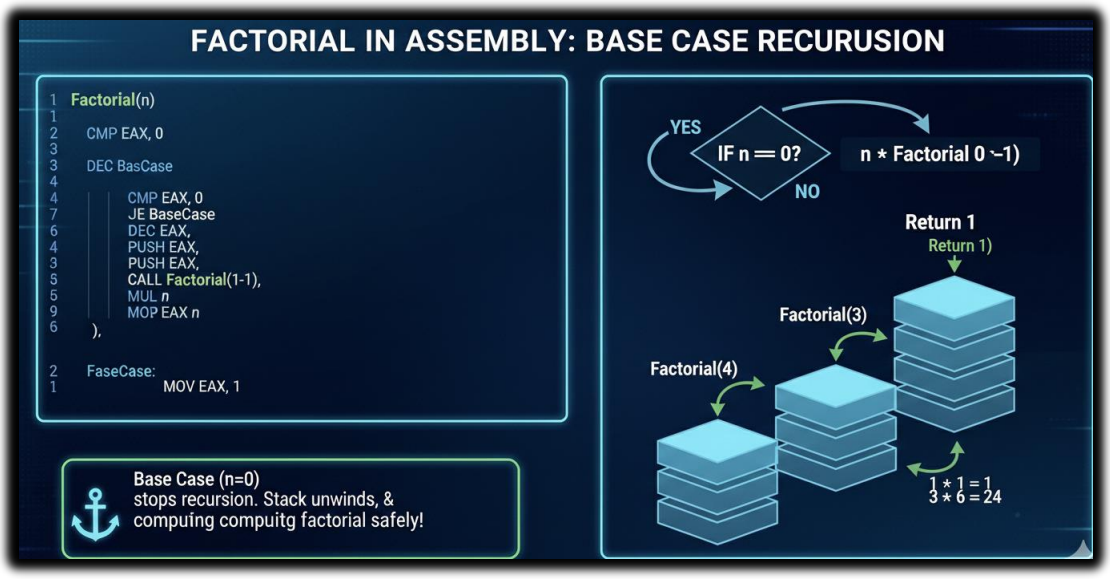
- **First call: Factorial(3)**
  - A new stack frame is created.
  - EBP is set to the start of the current stack frame.
  - The parameter N is set to 3.
  - The base case (N == 0) is checked and fails.
  - Factorial calls itself with N - 1 → Factorial(2).
- **Second call: Factorial(2)**
  - A new stack frame is created.
  - EBP is updated to the new stack frame.
  - The parameter N is set to 2.
  - The base case is checked and fails.
  - Factorial calls itself with N - 1 → Factorial(1).
- This process continues until **Factorial(0)** is reached.

## Base Case and Returning

- **Base case:**
  - When N == 0, Factorial returns **1**.
  - The value 1 is placed in the **EAX** register.
- **Returning from recursion:**
  - Returning to Factorial(1):
    - EAX = 1
    - Multiply EAX by N (1) → result remains 1
  - Returning to Factorial(2):
    - EAX = 1
    - Multiply EAX by N (2) → result is 2
  - Returning to Factorial(3):
    - EAX = 2
    - Multiply EAX by N (3) → result is 6

## Final Result

- The program returns to main().
- The **EAX register contains 6**, which is the factorial of 3.



## Step-by-Step Explanation of the Factorial Assembly Code

- The main procedure:
  - Pushes the initial value (5) onto the stack.
  - Calls the Factorial procedure to compute the factorial.
  - Receives the result in **EAX** and displays it.
- The Factorial procedure:
  - Is a recursive function that computes the factorial of an integer.
  - Saves the caller's base pointer by pushing **EBP** onto the stack.
  - Sets up a new stack frame by moving **ESP** into **EBP**.
  - Retrieves the parameter n from the stack into **EAX**.
  - Compares n to 0 using the cmp instruction.
    - If n > 0, execution jumps to label **L1**.
    - If n == 0, execution jumps to label **L2** (base case).
- **Recursive Case (L1):**
  - Decrements n.
  - Pushes n − 1 onto the stack.
  - Calls Factorial recursively.
  - After returning, multiplies the returned value in **EAX** by n.
- **Base Case (L2):**
  - Occurs when n == 0.
  - Returns 1 in **EAX**.
  - Restores **EBP** and returns to the caller.
- This recursive process continues until n reaches 0.
- The final result is accumulated in **EAX** and returned to main.
- For input 5, the program correctly computes **5! = 120**.

### Important Tip

When writing recursive procedures, it is crucial to:

- Track which registers are modified.
- Save and restore registers when their values must persist across recursive calls.

Failure to do so can corrupt data and cause incorrect results.

## Questions and Answers

### 1. True / False

- **Statement:** Given the same task, a recursive subroutine usually uses more memory than a nonrecursive one.
- **Correct Answer:** ✅ **True**
- **Explanation:**
  - Recursive subroutines require additional stack space for:
    - Return addresses
    - Saved base pointers
    - Parameters
  - Each recursive call adds a new stack frame.

### 2. What condition terminates the recursion in the Factorial function?

- The recursion terminates when the input parameter **n equals 0**.

### 3. Which instructions execute after each recursive call finishes?

The instructions that:

- Multiply the value returned in **EAX** by n.

This follows the definition:

- **factorial(n) = n × factorial(n − 1)**

## 4. What happens if you try to calculate 13!?

- The program will **overflow**.
- **Reason:**
  - 13! = 6,227,020,800
  - This value exceeds the maximum value of a 32-bit signed integer.
- The result stored in **EAX** will be incorrect due to integer overflow.

## Challenge Question: Stack Space Usage (Corrected)

*How many bytes of stack space would be used by the Factorial procedure when calculating 5!?*

❌ **Original claim: *20 bytes per call, total 100 bytes***

This is **incorrect**.

✅ **Correct Explanation:**

- Each recursive call uses:
  - **4 bytes** → return address
  - **4 bytes** → saved EBP
  - **4 bytes** → parameter n
- **Total per call: 12 bytes**
- For Factorial(5):
  - Calls made: Factorial(5), (4), (3), (2), (1), (0) → **6 calls**
  - **Total stack usage:**
    - 12 bytes × 6 calls = 72 bytes

✅ **Correct answer: 72 bytes of stack space**

# INVOKE, ADDR, PROC AND PROTO

## Advanced Procedure Directives (INVOKE, PROC, PROTO)

- The **INVOKE**, **PROC**, and **PROTO** directives provide powerful tools for defining and calling procedures in 32-bit mode.
- These directives are more convenient to use than traditional CALL and PROC instructions.
- However, they **mask the underlying structure of the runtime stack**, which can make it harder to understand what is happening at a low level.
- The **PROTO** directive:
  - Allows the assembler to validate procedure calls.
  - Checks argument lists against procedure declarations.
  - Helps prevent errors and makes programs more robust.
- Advanced procedure directives:
  - Improve program readability and maintainability.
  - Are especially useful when procedures are called across module boundaries.
  - Abstract away low-level stack management details.

### Important Considerations

- Because advanced directives hide stack mechanics:
  - Programmers should develop a solid understanding of how subroutine calls work internally.
  - This includes knowledge of stack frames, parameter passing, and return addresses.

### Recommendation

If you are new to assembly language: Start by learning traditional CALL and PROC directives.

After gaining a strong understanding of low-level subroutine mechanics: Use advanced procedure directives to improve code clarity and maintainability.

# INVOKE Directive (32-bit Assembly)

The INVOKE directive is a high-level MASM feature used to call procedures in 32-bit mode.

It allows you to pass multiple arguments to a procedure using a single, readable statement, instead of manually pushing each argument.

## General Syntax

```
INVOKE procedureName [, argumentList]
```

- **procedureName**: Name of the procedure being called
- **argumentList**: Optional, comma-separated list of arguments

## Valid Argument Types

Arguments passed to INVOKE can be **any valid expression**, including:

Immediate values

```
10, 3000h, OFFSET myList
```

Integer expressions

```
(10 + 20), COUNT
```

Variables

```
myList, array, myWord, myDword
```

Address expressions

```
[myList + 2], [ebx + esi]
```

Registers

```
eax, bl, edi
```

## Argument Order (Very Important)

Arguments in an INVOKE statement are **pushed onto the stack in reverse order**, just like standard 32-bit calling conventions (cdecl, stdcall).

## Example: Calling DumpArray

```
INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

This pushes the arguments in this order:

1. TYPE array (element size)
2. LENGTHOF array (number of elements)
3. OFFSET array (address of array)

Then the procedure is called.

### Equivalent Low-Level Code

```
push TYPE array
push LENGTHOF array
push OFFSET array
call DumpArray
```

✅ INVOKE just saves you from writing this manually.

### Multiline INVOKE (Readable Style)

You can split arguments across multiple lines to improve clarity:

```
INVOKE DumpArray,       ; displays an array
    OFFSET array,       ; pointer to the array
    LENGTHOF array,     ; number of elements
    TYPE array          ; size of each element
```

This is **purely a style choice**—both forms compile to the same machine code.

## Important Considerations

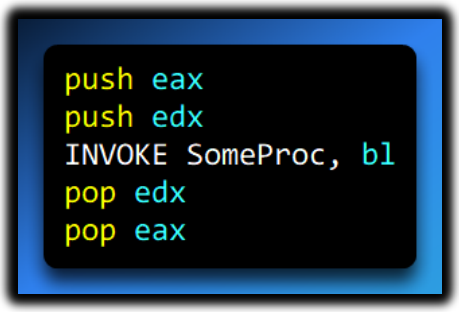
### 1. Register Side Effects (EAX & EDX)

When passing arguments **smaller than 32 bits** (BYTE or WORD), MASM may:

- **Widen the value to 32 bits**
- Use **EAX and EDX internally**
- Overwrite their contents

How to avoid problems:

- Pass **32-bit arguments**, OR
- Save and restore registers:




## 2. 32-bit Only

⚠ The INVOKE directive:

- Works **only in 32-bit mode**
- Is **not used in x64**, where parameters are passed in registers

## 3. Short note:

- INVOKE is a **convenient, readable way** to call procedures
- Arguments are pushed **right-to-left**
- Supports many argument types
- May overwrite **EAX/EDX** when widening small arguments
- Available **only in 32-bit MASM**

Use INVOKE for clean, readable procedure calls in 32-bit assembly—but always be aware of argument size and register side effects.

# ADDR Operator (32-bit MASM)

The ADDR operator is used with the INVOKE directive to **pass addresses (pointers)** to procedures in **32-bit mode**. It is essentially a **clean, readable shortcut** for passing OFFSET values when calling procedures.

## Key Characteristics

- ✅ **32-bit only**
- ✅ **Used only with INVOKE**
- ✅ Returns the **address** of its operand
- ❌ Cannot be used with MOV, CALL, or other instructions
- ❌ Operand must be known at **assembly time**

## Syntax

```
ADDR operand
```

- operand must be an **assembly-time constant**
- Examples of valid operands:
  - Labels
  - Static arrays
  - Fixed memory addresses

## Basic Example

```
INVOKE FillArray, ADDR myArray
```

This passes the **address of myArray** to FillArray.

**Equivalent Code (Manual Version)**

```
mov esi, OFFSET myArray
INVOKE FillArray, esi
```

✅ The ADDR version is shorter, clearer, and preferred.

## Important Restrictions

### 1. Only Works with INVOKE

❌ Invalid:

```
mov esi, ADDR myArray
call SomeProc
```

✅ Valid:

```
INVOKE SomeProc, ADDR myArray
```

### 2. Assembly-Time Constants Only

You **cannot** use ADDR with values computed at runtime.

❌ Invalid:

```
ADDR [eax+4]
```

✅ Valid:

```
ADDR Array
ADDR Array+4
```

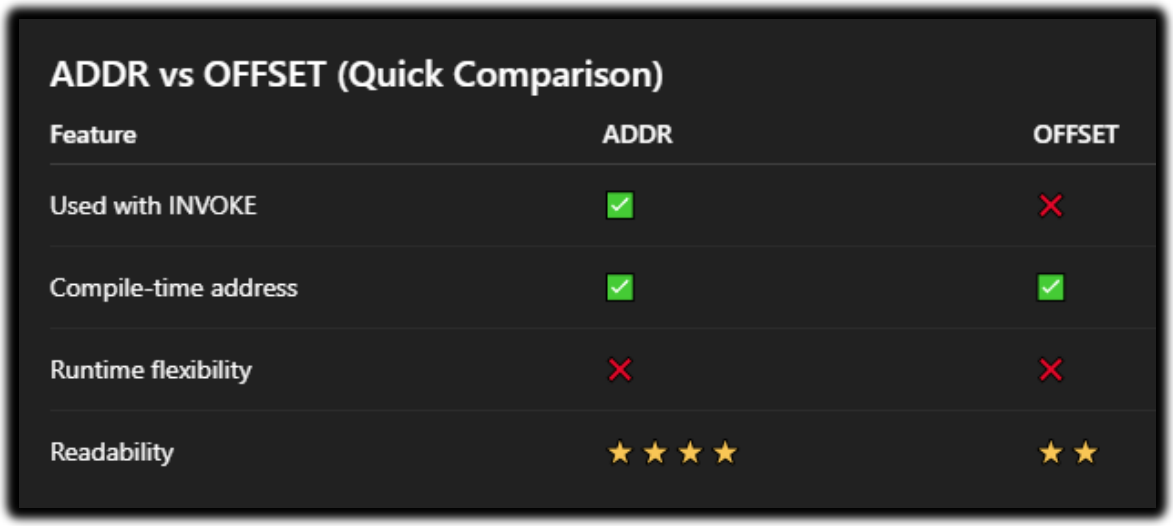
## Passing Multiple Addresses (Example)

```
.data
Array DWORD 20 DUP(?)

.code
INVOKE Swap, ADDR Array, ADDR Array+4
```

So ADDR is just a **clean wrapper around OFFSET pushes**.

## ADDR vs OFFSET (Quick Comparison)



## Passing Function Addresses (Callbacks)

ADDR can also be used to pass the **address of a procedure** to another procedure (useful for callbacks).

### Example

```
PrintArray PROC Near
    ; prints array elements
    ret
PrintArray ENDP


DoSomething PROC Near
    INVOKE PrintArray, ADDR PrintArray
    ret
DoSomething ENDP
```

Here, ADDR PrintArray passes the **address of the function itself**.

### Summary

- ADDR is a **high-level helper** for passing pointers using INVOKE
- It expands to push OFFSET ... behind the scenes
- Only works in **32-bit MASM**
- Improves readability and reduces boilerplate
- Perfect for arrays, structures, and callback-style designs

If you're using INVOKE and need to pass a pointer to something known at assembly time, **ADDR is the cleanest and safest choice**.

# PROC Directive (32-bit MASM)

The PROC directive is used to **define a procedure** in 32-bit assembly language. It tells the assembler where a procedure begins and allows you to specify **parameters, calling conventions, register usage, and visibility**.

## General Syntax

```
label PROC [attributes] [, parameter_list]
```

- **label**
  User-defined name of the procedure (must follow identifier rules).
- **attributes** *(optional)*
  Control how the procedure behaves.
- **parameter_list** *(optional)*
  Declares named parameters passed to the procedure.

## PROC Attributes

### 1. Distance

Specifies how the procedure is called:

- NEAR – same code segment (most common)
- FAR – different segment (rare in modern code)

### 2. Language Type (Calling Convention)

Specifies the **parameter passing convention**, such as:

- C (cdecl)
- STDCALL
- PASCAL

This affects:

- Argument order
- Stack cleanup responsibility
- Name decoration

### 3. Visibility

Controls whether the procedure is visible outside the module:

- PUBLIC – accessible from other modules
- PRIVATE – only accessible within the same module

### 4. Prologue/Epilogue Arguments

Used to control or customize stack frame generation (advanced usage).

## Declaring Parameters

Parameters are declared directly in the PROC line using:

```
paramName:type
```

```
AddTwo PROC,
    val1:DWORD,
    val2:DWORD
```

- Parameters are automatically placed on the stack
- They are accessed using symbolic names instead of [ebp+8], [ebp+12]

## USES Clause

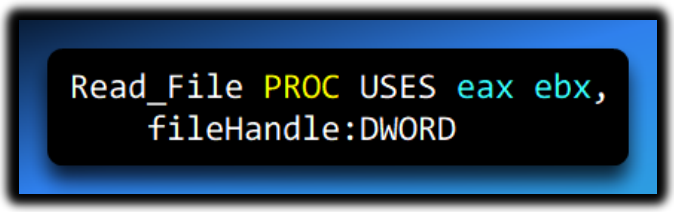
The USES clause tells MASM **which registers the procedure will modify**.

MASM automatically:

- Pushes these registers at the start of the procedure
- Pops them before returning

Syntax:

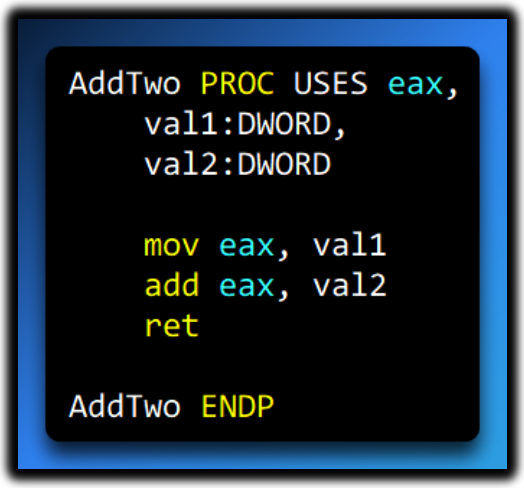

Example




This means:

- EAX and EBX are preserved
- Caller state remains intact
- No manual push / pop needed

## Full Example



```
AddTwo PROC USES eax,
    val1:DWORD,
    val2:DWORD

    mov eax, val1
    add eax, val2
    ret

AddTwo ENDP
```

## What MASM Generates Automatically

- Stack frame setup
- Parameter access logic
- Register preservation (eax)
- Clean, consistent epilogue

## Why PROC Is Powerful

- Named parameters → **readable code**
- USES → **automatic register saving**
- Calling convention support
- Works seamlessly with INVOKE

## Summary

- PROC defines a procedure in 32-bit MASM
- Supports parameters, calling conventions, and visibility
- USES automatically preserves registers
- Eliminates manual stack and register bookkeeping
- Designed to work cleanly with INVOKE

If you're writing structured 32-bit MASM, PROC + parameters + USES is the cleanest, safest, and most maintainable way to define procedures.

The following example shows a simple procedure named AddTwo():

```
779 AddTwo PROC
780     val1: DWORD   ; Define a DWORD parameter named val1.
781     val2: DWORD   ; Define another DWORD parameter named val2.
782     mov eax, val1 ; Move the value of val1 into the EAX register.
783     add eax, val2 ; Add the value of val2 to EAX.
784     ret           ; Return from the procedure, effectively returning the result in EAX.
785 AddTwo ENDP
```

This procedure takes two doublewords as parameters and returns their sum.

The following shows the assembly code generated by MASM when assembling the AddTwo() procedure:

```
794 AddTwo PROC
795     push ebp        ; Save the current base pointer (BP).
796     mov ebp, esp    ; Set up a new base pointer, making ESP the stack frame pointer.
797
798     mov eax, dword ptr [ebp+8]   ; Load the first parameter from the stack into EAX.
799     add eax, dword ptr [ebp+0Ch] ; Add the second parameter from the stack to EAX.
800
801     leave  ; Release the current stack frame.
802     ret    ; Return from the procedure, effectively returning the result in EAX.
803
804     8      ; Indication of the number of bytes used by parameters. (Not part of the actual code.)
805
806 AddTwo ENDP
```

1. The first two lines save the old **EBP** value and set **EBP** to the current stack pointer.
   - This creates a stack frame for the procedure.
2. The next lines copy the function parameters from the stack into registers.
   - The first parameter goes into **EAX**.
   - The second parameter is read from the stack later.
3. The code adds the second parameter to the value in **EAX**.
   - The result stays in **EAX**.
4. The **leave** instruction restores the old stack frame.
   - It puts the old **EBP** back and resets **ESP**.
5. The **ret** instruction exits the procedure.
   - The value in **EAX** is returned.

6. The number **8** at the end means the parameters use 8 bytes total.
    - Each parameter is 4 bytes (32-bit).
7. The **PROC** directive helps define procedures in 32-bit mode.
    - It supports named parameters and manages the stack frame.

## Instruction details:

1. push ebp
    - Saves the previous base pointer on the stack.
2. mov ebp, esp
    - Sets up a new base pointer for this procedure.
3. mov eax, [ebp+8]
    - Loads the first parameter from the stack into **EAX**.
4. add eax, [ebp+0Ch]
    - Adds the second parameter to **EAX**.
5. leave
    - Cleans up the stack frame and restores **EBP**.
6. ret
    - Returns from the procedure using the value in **EAX**.

## Specifying the Parameter Passing Protocol

A **parameter passing protocol** (calling convention) defines **how arguments are passed**, **who cleans up the stack**, and **how procedures return values**. Common protocols include **C (cdecl)**, **STDCALL**, and **Pascal**.

In MASM, you specify the calling convention using the **attributes field** of the PROC directive. Syntax:

```
procName PROC callingConvention,
    parameterList
```

The calling convention controls:

- Argument order
- Stack cleanup responsibility
- Name decoration
- Compatibility with high-level languages

### C Calling Convention (cdecl)

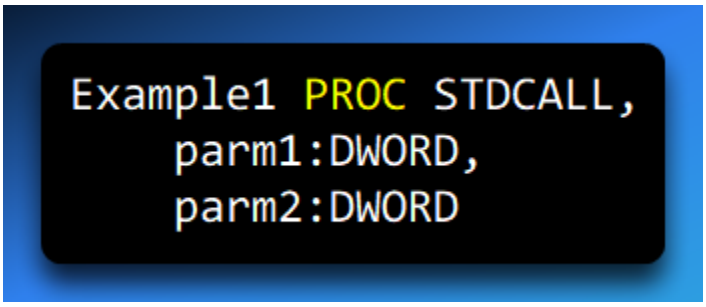
```
Example1 PROC C,
    parm1:DWORD,
    parm2:DWORD
```

**Key characteristics:**

- Arguments pushed **right-to-left**
- **Caller** cleans up the stack
- Commonly used by **C/C++**
- Allows variable argument functions (e.g., printf)

When called using INVOKE, MASM generates code consistent with the **C calling convention**.

### STDCALL Calling Convention



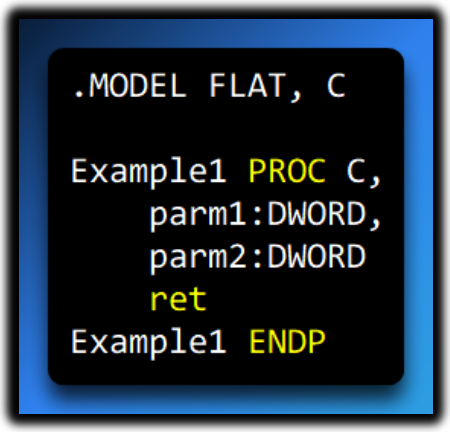

**Key characteristics:**

- Arguments pushed **right-to-left**
- **Callee** cleans up the stack
- Used extensively by the **Windows API**
- Fixed number of parameters

When called using INVOKE, MASM generates code consistent with **STDCALL**.

## Full Examples

### Using C Calling Convention



Using STDCALL Calling Convention

```
.MODEL FLAT, STDCALL

Example2 PROC STDCALL,
    parm1:DWORD,
    parm2:DWORD
    ret
Example2 ENDP
```

## Why This Matters

- Ensures **binary compatibility** with C/C++ and Windows API functions
- Prevents **stack corruption**
- Allows assembly procedures to be safely called from high-level languages
- Makes INVOKE generate correct push/cleanup code automatically

## Short note:

| Convention | Stack Cleanup | Usage |
|---|---|---|
| C (cdecl) | Caller | C / C++ |
| STDCALL | Callee | Windows API |

✅ **Final takeaway:**
Specifying the parameter passing protocol using the PROC directive is essential for writing **safe, interoperable assembly code** that works correctly with other languages and libraries.

## PROTO Directive in 32-bit Mode

The **PROTO directive** is used to **declare a procedure prototype** in MASM. A prototype specifies a procedure's name, parameter list, and optionally the calling convention, allowing you to:

- Call a procedure **before its definition**
- **Verify** that the number and types of arguments match the procedure
- Enable safer and more readable code when using INVOKE

Syntax

```
label PROTO [attributes] [, parameter_list]
```

- **label** – Name of the procedure
- **attributes** – Optional; specifies the calling convention (C, STDCALL, etc.)
- **parameter_list** – Optional; defines parameters with types

### Example: ArraySum Prototype

```
ArraySum PROTO,
    ptrArray:PTR DWORD,   ; Pointer to the array
    szArray:DWORD         ; Array size
```

This declares that ArraySum takes:

1. A **pointer to an array** of doublewords
2. The **size of the array**

### Calling a Procedure with INVOKE

Once a prototype is declared, you can safely call the procedure:

```
INVOKE ArraySum, ptrArray, szArray
```

- INVOKE automatically pushes the arguments in the correct order
- Checks that the **number and types** of arguments match the prototype

This eliminates many common errors, such as stack misalignment or incorrect argument types.

### Important Notes

1. Every procedure called with INVOKE **must have a prototype**.
2. The prototype must appear **before the procedure is called**.
3. The **parameter types and number** in the prototype must match the actual procedure definition.
4. PROTO improves **code readability, maintainability, and safety**.

PROTO allows assembly programs to **declare procedure interfaces ahead of time**, enabling safer, cleaner, and more modular code, especially when combined with INVOKE.

## ASSEMBLY TIME ARGUMENT CHECKING

The **PROTO directive** allows MASM to perform **assembly-time argument checking**. This helps verify:

- The **number** of arguments passed to a procedure
- The **types** of arguments (to a limited extent)

⚠ MASM is **less strict than C/C++**. It checks basic types and number of arguments but won't catch every type mismatch.

## Example Prototype

```
879 Sub1 PROTO,
880 p1:BYTE,
881 p2:WORD,
882 p3:PTR BYTE
```

Sub1 expects **three arguments**:

- A **byte**
- A **word** (2 bytes)
- A **pointer to a byte**

## Valid INVOKE Call

```
INVOKE Sub1, byte_1, word_1, ADDR byte_1
```

The assembler will generate the following code for this INVOKE statement:

```
890 push 404000h ; Push the pointer to byte_1 onto the stack.
891 sub esp, 2  ; Reserve 2 bytes on the stack for padding.
892 push word ptr ds:[00404001h] ; Push the value of word_1 onto the stack.
893 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.
894 push eax ; Push the value from EAX onto the stack.
895 call 00401071 ; Call the function at address 00401071.
```

The assembler pads the stack with two bytes because the second argument (word_1) is a word, which is two bytes long.

1. push 00404000h → Pushes the pointer (p3)
2. sub esp, 2 → Padding for alignment
3. push word ptr ds:[00404001h] → Push 2-byte word (p2)
4. mov al, byte ptr ds:[00404000h] → Load 1-byte value (p1)
5. push eax → Push as DWORD
6. call 00401071 → Call the procedure

## Errors Detected by MASM

1. Argument exceeds declared size

```
INVOKE Sub1, word_1, word_2, ADDR byte_1
; ❌ Error: p1 expects BYTE, word_1 is WORD
```

2. Too few or too many arguments

```
INVOKE Sub1, word_2, ADDR byte_1
; ❌ Error: too few arguments

INVOKE Sub1, byte_1, word_2, ADDR byte_1, byte_2
; ❌ Error: too many arguments
```

## Errors NOT Detected by MASM

MASM **does not detect smaller arguments**. It will **expand smaller types** to the expected parameter size.

```
INVOKE Sub1, byte_1, byte_1, ADDR byte_1
```

Instead, MASM will expand the smaller argument (byte_1) to the size of the declared parameter (WORD).

MASM expands byte_1 to match p2:WORD

Generates code:

```
mov al, byte ptr ds:[00404000h]
movzx eax, al    ; expand byte → DWORD
push eax
```

✅ MASM ensures stack consistency but may silently widen arguments.

In the following code generated by MASM, the second argument (byte_1) is expanded into EAX before pushing it on the stack:

```
925 push 404000h ; Push the address of byte_1 onto the stack.
926 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.
927 movzx eax, al ; Expand the value in AL into EAX.
928 push eax ; Push the value from EAX onto the stack.
929 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.
930 push eax ; Push the value from EAX onto the stack.
931 call 00401071 ; Call the function at address 00401071 (Assuming it's a function).
```

## Key Points

- PROTO + INVOKE helps **catch argument mismatches** at assembly time.
- MASM checks: **number of arguments** & **basic type size**
- MASM **does not catch all type errors**, e.g., smaller types are expanded.
- Stack padding ensures **alignment** when pushing arguments of different sizes.

## ArraySum

```
952 ; ArraySum Procedure
953 ; Parameters:
954 ;   esi: Points to the array
955 ;   ecx: Size of the array
956 ; Returns:
957 ;   eax: The sum of the array
958 ArraySum PROC USES esi ecx,
959     ptrArray: PTR DWORD, ; Pointer to the array
960     szArray: DWORD      ; Array size
961
962     mov esi, ptrArray  ; Load the address of the array into esi.
963     mov ecx, szArray  ; Load the size of the array into ecx.
964     mov eax, 0        ; Initialize the sum to zero.
965
966     cmp ecx, 0         ; Check if the array size is zero.
967     je L2             ; If yes, quit.
968
969 L1:
970     add eax, [esi]    ; Add the value at esi to the sum in eax.
971     add esi, 4        ; Move to the next integer in the array (4 bytes forward).
972     loop L1           ; Repeat for the remaining array size.
973
974 L2:
975     ret               ; Return with the sum in EAX.
976
977 ArraySum ENDP
```

**ArraySum()** takes two parameters: A pointer to the array (address of the first element). The size of the array (number of elements).

The procedure uses registers:

- **ESI** to hold the array pointer.
- **ECX** to hold the array size.
- **EAX** will hold the running sum of the elements.

**Steps of the procedure:**

1. Set **EAX** to 0 (start the sum at zero).
2. Check if the array size (**ECX**) is zero. If zero, return immediately.
3. Otherwise, enter a loop:
    - Add the value pointed to by **ESI** to **EAX**.
    - Move **ESI** to the next element in the array.
    - Repeat until all elements are added.
4. After the loop, **EAX** contains the total sum.
5. Return from the procedure.

**Result:**

- **EAX** holds the sum of all elements in the array.

**Example of calling ArraySum():**

- You pass the array pointer and its size as arguments.
- After the procedure, **EAX** will have the sum.

```
0983 .data
0984     array DWORD 10000h, 20000h, 30000h, 40000h, 50000h
0985     theSum DWORD ?
0986
0987 .code
0988 main PROC
0989     INVOKE ArraySum, ADDR array, LENGTHOF array
0990     ; Call the ArraySum procedure, passing the address of the array and the number of elements.
0991
0992     mov theSum, eax
0993     ; Store the sum returned by ArraySum in theSum.
0994
0995     ; Your program logic can continue here, using the calculated sum.
0996
0997 main ENDP
```

The **INVOKE** statement calls the **ArraySum()** procedure.

It passes two arguments:

- The address of the array.
- The number of elements in the array.

The **LENGTHOF** operator calculates the number of elements in the array automatically.

After **ArraySum()** finishes, the total sum of the array elements is stored in the **theSum** variable.

This example shows how to use:

- The **PROC** directive to define procedures with stack parameters.
- The **INVOKE** directive to call procedures and pass arguments via the stack.

**Key point:** Using **PROC** and **INVOKE** makes procedure calls cleaner and safer, especially when working with parameters.

```
.data
    array DWORD 10000h, 20000h, 30000h, 40000h, 50000h
    theSum DWORD ?

.code
    ; ArraySum Procedure
    ArraySum PROC USES esi ecx,
        ptrArray: PTR DWORD, ; Pointer to the array
        szArray: DWORD      ; Array size
        mov esi, ptrArray  ; Load the address of the array into esi.
        mov ecx, szArray  ; Load the size of the array into ecx.
        mov eax, 0        ; Initialize the sum to zero.
        cmp ecx, 0         ; Check if the array size is zero.
        je L2             ; If yes, quit.
    L1:
        add eax, [esi]    ; Add the value at esi to the sum in eax.
        add esi, 4        ; Move to the next integer in the array (4 bytes forward).
        loop L1           ; Repeat for the remaining array size.
    L2:
        ret               ; Return with the sum in EAX.
    ArraySum ENDP
    main PROC
        INVOKE ArraySum, ADDR array, LENGTHOF array
        ; Call the ArraySum procedure, passing the address of the array and the number of elements.
        mov theSum, eax
        ; Store the sum returned by ArraySum in theSum.
        ; Your program logic can continue here, using the calculated sum.
    main ENDP
```

## .data Section

- An array named **array** is defined with **5 DWORD (32-bit) elements** and initial values.
- A DWORD variable named **theSum** is declared with a **question mark**, meaning it is **uninitialized**.

## .code Section – ArraySum Procedure

- **ArraySum** calculates the sum of a DWORD array.
- It expects **two parameters**:
    1. **ptrArray** – a pointer to the array.
    2. **szArray** – the number of elements in the array.

## Inside ArraySum:

- **ESI** holds the array address.
- **ECX** stores the array size.
- **EAX** is set to **0** and will accumulate the sum.
- The code checks if the array size is **0**.
- If zero, it jumps to **L2**, effectively quitting the procedure.

**Loop (L1):**

- Add the value at the address in **ESI** to **EAX**.
- Increment **ESI** by 4 to move to the next DWORD.
- Repeat for all elements using the **loop** instruction.

**End of procedure (L2):** Return with the sum in **EAX**.

## Main Procedure

- Calls **ArraySum** using the **INVOKE** directive.
- Passes **address of array** and **number of elements (LENGTHOF array)** as arguments.
- Stores the result returned in **EAX** into **theSum**.
- The program can continue using the sum stored in **theSum**.

## Parameter Classifications

- **Input Parameters:**
  - Passed **from the calling program to the procedure**.
  - Procedure can use the data but **cannot change it**.
  - Example: reading an array to calculate a sum.
- **Output Parameters:**
  - Passed **from the procedure back to the calling program**.
  - Procedure can **change the value**, and the calling program will see the updated data.
  - Example: returning the sum of the array elements.

## Key Points:

- **ArraySum** efficiently calculates the sum of an array and returns it in **EAX**.
- **INVOKE** and **PROC** make procedure calls and parameter passing easier in assembly.
- Using **ESI, ECX, and EAX** keeps track of the array, its size, and the running sum.

Here is an example of an input parameter:

```
1033 .data
1034     buffer BYTE 80 DUP(?)
1035     inputHandle DWORD ?
1036 .code
1037     INVOKE ReadConsole, inputHandle, ADDR buffer
1038     ; ReadConsole is expected to store user input in the 'buffer' variable.
```

and

```
1042 procedure add_two_numbers(x: DWORD, y: DWORD): DWORD
1043   ; ...
1044   add eax, x
1045   add eax, y
1046   ret
1047 endp
1048
1049 ; Calling the procedure
1050 mov eax, 10
1051 mov ebx, 20
1052 call add_two_numbers
1053 mov ecx, eax ; ecx will now contain the value 30
```

- In the **add_two_numbers()** example:
  - **x** and **y** are **input parameters**.
  - The procedure **reads** their values to calculate a sum.
  - It **does not change** x or y.
- **Output parameters** are different:
  - They are used to **return data** from the procedure to the calling program.
  - The procedure **can change** the value of an output parameter.
  - The calling program will see the updated value after the procedure returns.

## Example of an output parameter:

A procedure that calculates the sum of two numbers and stores it in a variable provided by the caller.

- Caller passes a **pointer or variable** to hold the result.
- Procedure updates that variable directly.

Here is an example of an output parameter:

```
1057 procedure get_system_time(time: PTR DWORD)
1058   ; ...
1059   mov [time], eax
1060   ret
1061 endp
1062
1063 ; Calling the procedure
1064 mov eax, OFFSET time_variable
1065 call get_system_time
1066
1067 ; The time variable will now contain the system time
```

In this example:

- **time** is an **output parameter**.
- The procedure **get_system_time()** uses the pointer in **time** to store the system time in the memory location the pointer points to.

**Input and output parameters can be used together** in a procedure.

- Example: A procedure could take an **input parameter** for the size of an array and use an **output parameter** to return the sum of the array elements.

**Input/output parameters**:

- Example: a **buffer** (a block of memory for temporary storage).
- A procedure can take a buffer as an input/output parameter to:
  - Read data into the buffer from a file (input).
  - Return the data to the calling program (output).
  - Optionally, modify the data and return the updated version (input/output).

## Key idea:

**Input parameters** provide data to the procedure.

- **Output parameters** return data from the procedure.
- **Input/output parameters** allow the procedure to both read and update data in the same memory location.

Here is an example of how to use an input/output parameter in assembly language:

```
1071 procedure read_file(buffer: PTR BYTE, size: DWORD): DWORD
1072   ; ...
1073   ; Read data from the file into the buffer
1074   ; ...
1075   ret
1076 endp
1077
1078 ; Calling the procedure
1079 mov eax, OFFSET buffer
1080 mov ebx, size
1081 call read_file
1082
1083 ; The buffer variable will now contain the data that was read from the file
```

## In this example:

- **buffer** is an **input/output parameter**.
- The procedure **read_file()** reads data from a file into the buffer.
- The procedure can also **modify the buffer**, so the calling program receives the updated data.

## Additional output:

- **read_file()** returns the **number of bytes read** from the file.
- The calling program can use this to know how much valid data is in the buffer.

## Key idea:

- Input/output parameters let the procedure **both read and update** data in the same memory location.
- Procedures can also return additional information (like the number of bytes read) through the **return value** or another output parameter.

## Next example: Exchanging Two Integers

- This is another example of using **input/output parameters**.
- A procedure can **swap two numbers** by taking pointers to the integers as parameters and modifying their values directly.

```
include Irvine32.inc

Swap PROTO, pValX:PTR DWORD, pValY:PTR DWORD
; Exchange the values of two 32-bit integers
; Returns: nothing
Swap PROC USES eax esi edi,
pValX:PTR DWORD,
; pointer to first integer
pValY:PTR DWORD
; pointer to second integer
; get pointers
mov esi, pValX
mov edi, pValY
; get first integer
mov eax, [esi]
; exchange with second
xchg eax, [edi]
; replace first integer
mov [esi], eax
; PROC generates RET 8 here
ret
Swap ENDP
```

## Swap procedure:

- Takes **two input/output parameters**: pValX and pValY.
- These parameters are **pointers** to the two integers that need to be swapped.

## Steps inside Swap:

1. Get the pointers to the two integers.
2. Load the value of the first integer into **EAX**.
3. Use **XCHG** to exchange the value in **EAX** with the second integer.
4. Store the value in **EAX** back into the first integer.

## Return behavior:

- Swap does **not return a value**.
- Ends with a **RET** instruction.
- Because it uses the **STDCALL calling convention**, the **PROC directive generates RET 8**, which cleans up the two parameters from the stack.

## Calling Swap:

- It can be called from the main procedure by passing the **addresses of the two integers** as arguments.

```
1117 ; Display the array before the exchange:
1118 mov esi, OFFSET Array
1119 mov ecx, 2
1120 ; count = 2
1121 mov ebx, TYPE Array
1122 call
1123 DumpMem
1124 ; dump the array values
1125 INVOKE Swap, ADDR Array, ADDR [Array+4]
1126 ; Display the array after the exchange:
1127 call
1128 DumpMem
```

## Calling Swap:

- The **INVOKE** statement calls the Swap procedure.
- It passes the **addresses of the first two elements** of the array.
- After the procedure returns, the **first two elements are swapped**.

### Limitations / missing checks:

- Swap does **not check for invalid addresses**.
  - If a pointer is incorrect, the program may **crash**.
- Swap is **not optimized for speed**.
  - Using a temporary variable could make swapping slightly faster.

### Key takeaway:

- Swap is a **simple example of input/output parameters** in assembly.
- It demonstrates **how a procedure can directly modify data** in memory through pointers.

## Debugging Tips for Procedure Calls

### 1. Argument Size Mismatch

Make sure the arguments you pass to a procedure are the **correct size**.

- Example: If a procedure expects a **doubleword pointer**, pass a **doubleword pointer**.
- Passing a smaller pointer, like a **word pointer**, will prevent the procedure from accessing the data correctly.

Example:

- Swap() expects **two doubleword pointers**.
- If you pass **two word pointers**, the procedure cannot access the integers correctly.
- This may cause incorrect results or program crashes.

Here is an example of an argument size mismatch:

```
1134 ; Swap procedure from Section 8.4.6
1135 Swap PROC, pValX:PTR DWORD, pValY:PTR DWORD
1136 ...
1137
1138 ; Incorrect call to Swap
1139 INVOKE Swap, ADDR [DoubleArray + 0], ADDR [DoubleArray + 1]
```

## 2. Passing the Wrong Type of Pointer

Make sure the arguments are the **correct type** of pointer.

- Example: If a procedure expects a **doubleword pointer**, do **not** pass a **byte pointer**.
- Using the wrong type of pointer prevents the procedure from reading/writing memory correctly.

**Example:**

- Swap() expects **DWORD pointers**.
- Passing a **BYTE pointer** will cause memory access errors.

Here is an example of passing the wrong type of pointer:

```
1145 ; Swap procedure from Section 8.4.6
1146 Swap PROC, pValX:PTR DWORD, pValY:PTR DWORD
1147 ...
1148
1149 ; Incorrect call to Swap
1150 INVOKE Swap, ADDR [ByteArray + 0], ADDR [ByteArray + 1]
```

## 3. Wrong Pointer Size

- Swap() expects **two doubleword pointers**.
- Passing **two byte pointers** instead will prevent the procedure from accessing the integers correctly.
- This can cause **incorrect results or crashes**.

## 4. Passing Immediate Values to Reference Parameters

- **Reference parameters** expect a **pointer to a variable**, not the value itself.
- If you pass an **immediate value** (like 5) instead of a pointer, the procedure cannot access memory correctly.
- This will lead to **errors or crashes**.

Small note:

- Always pass the **correct pointer type and size** for reference parameters.
- Never pass an immediate value to a parameter that expects a pointer.

Here is an example of passing an immediate value to a reference parameter:

```
1156 ; Sub2 procedure
1157 Sub2 PROC, dataPtr:PTR WORD
1158 mov
1159 esi,dataPtr
1160 ; get the address
1161 mov
1162 WORD PTR [esi],0
1163 ; dereference, assign zero
1164 ret
1165 Sub2 ENDP
1166
1167 ; Incorrect call to Sub2
1168 INVOKE
1169 Sub2, 1000h
```

## 5. Passing Immediate Values to Reference Parameters

- Sub2() expects **a pointer to a word** as its only parameter.
- Passing an **immediate value** instead will prevent the procedure from accessing memory correctly.
- This can cause **crashes or incorrect results**.

## 6. General Advice

- Always be careful when **passing arguments** to procedures in assembly.
- Check the **procedure documentation** to confirm:
  - The **number of parameters**
  - The **type of each parameter**
  - The **size of each parameter**
- Mistakes can lead to **program crashes or wrong results**.

Short note:

- Always pass **pointers** for reference parameters.
- Never pass immediate values where a pointer is expected.
- Confirm argument type, size, and count before calling a procedure.

# WIRESTACKFRAME PROCEDURE

## WriteStackFrame and WriteStackFrameName (Irvine32 Library)

### WriteStackFrame:

- A procedure from the **Irvine32 library**.
- Displays the **contents of the current procedure's stack frame**.
- Shows details like:
  - **Stack parameters** (arguments passed to the procedure)
  - **Return address**
  - **Local variables**
  - **Saved registers**

### WriteStackFrameName:

- Similar to **WriteStackFrame**, but also displays the **name of the procedure** whose stack frame is being shown.

### Usefulness:

- Helps **debug and visualize the stack**.
- Makes it easier to **understand how parameters, local variables, and registers are stored**.
- Very helpful when learning or troubleshooting **stack behavior in assembly programs**.

```
1198 WriteStackFrame PROTO,
1199     numParam:DWORD,
1200     ; number of passed parameters
1201     numLocalVal: DWORD,
1202     ; number of DWordLocal variables
1203     numSavedReg: DWORD
1204     ; number of saved registers
```

# WriteStackFrame Procedure (Irvine32 Library)

Displays the **contents of the current procedure's stack frame**.

- Shows:
  - Stack parameters
  - Return address
  - Saved registers
  - Local variables

Parameters (3):

**numParam** – Number of parameters passed to the procedure.

Determines how many DWORDs to display at the **top of the stack**.

**numLocalVal** – Number of DWORD local variables for the procedure.

**numSavedReg** – Number of registers saved on the stack. Typically 2 (for **EAX** and **EBX**).

## How it displays the stack:

Starts at **EBP** and moves downward toward **ESP**.

For each DWORD, shows:

- Offset from **EBP**
- Hex value stored there

Order of display (from higher to lower offsets):

- Parameters (highest offsets)
- Return address
- Saved EBP
- Local variables
- Saved registers

Stops when reaching **ESP** (last used stack location).

## WriteStackFrameName Procedure

Works like **WriteStackFrame**, but with an extra parameter:

- **procName** – Pointer to a null-terminated string with the procedure's name.
- Displays the **procedure name** at the top of the output.
- Useful when **multiple procedures call WriteStackFrame/WriteStackFrameName**.

## Summary / Usefulness

Let's you **see the contents of the stack** at any point in a procedure.

Helps debug issues with:

- Stack parameters
- Local variables
- Saved registers
- Return addresses

Great for understanding **stack behavior in assembly programs**.

Here is an explanation of the MASM code example that was shown in the original text:

```
1173 ; In main procedure
1174 main PROC
1175     mov eax, 0EAEAEAEAh    ; Save test value in EAX
1176     mov ebx, 0EBEBEBEBh    ; Save test value in EBX
1177     INVOKE myProc, 1111h, 2222h ; Call myProc, passing 2 parameters
1178     exit main              ; Exit program
1179
1180 main ENDP
1181 ; In myProc procedure
1182 myProc PROC
1183     ; Procedure uses EAX and EBX, so they will be saved
1184     USES eax ebx
1185     x: DWORD, y:DWORD      ; Declare parameter variables
1186     LOCAL a:DWORD, b:DWORD  ; Declare local variables
1187     PARAMS = 2             ; 2 parameters
1188     LOCALS = 2             ; 2 local DWORD variables
1189     SAVED_REGS = 2         ; 2 saved registers (EAX and EBX)
1190     mov a,0AAAAh           ; Load value into local variable a
1191     mov b,0BBBBh           ; Load value into local variable b
1192     ; Display stack frame contents
1193     INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS
1194 myProc ENDP
```

The following sample output was produced by the call:

```
Stack Frame
00002222 ebp+12 (parameter)
00001111 ebp+8 (parameter)
00401083 ebp+4 (return address)
0012FFF0 ebp+0 (saved ebp) <--- ebp
0000AAAA ebp-4 (local variable)
0000BBBB ebp-8 (local variable)
EAEAEAEA ebp-12 (saved register)
EBEBEBEB ebp-16 (saved register) <--- esp
```

A second procedure, named WriteStackFrameName, has an additional parameter that holds the name of the procedure owning the stack frame:

```
WriteStackFrameName PROTO,
    numParam:DWORD,
    ; number of passed parameters
    numLocalVal:DWORD,
    ; number of DWORD local variables
    numSavedReg:DWORD,
    ; number of saved registers
    procName:PTR BYTE
    ; null-terminated string
```

Like WriteStackFrame, but takes an extra parameter:

- **procName** – the name of the procedure owning the stack frame.

Displays the procedure name at the top of the stack output.

## Main Procedure Example

- Loads sample values into **EAX** and **EBX** to be saved on the stack later.
- Calls **myProc** with **two DWORD arguments**: 1111h and 2222h.
- Exits the program after the call.

## myProc Procedure

Uses **EAX** and **EBX** registers (so they will be saved on the stack).

Declares **x and y parameters** and **a and b local variables**.

Loads sample values into the local variables.

Calls **WriteStackFrame**, passing:

- 2 – number of parameters
- 2 – number of local DWORD variables
- 2 – number of saved registers (EAX and EBX)

## Displayed Stack Contents

The procedure's stack frame shows:

- Parameters: 1111h and 2222h
- Return address back to **main**
- Saved **EBP** from **main**
- Local variables: a and b
- Saved registers: **EAX** and **EBX**

## Purpose:

- Demonstrates how WriteStackFrame helps visualize a procedure's stack usage.
- Useful for understanding stack layout, parameter passing, local variables, and saved registers.

## Additional Info

- Irvine32 library source code can be found in: \Examples\Lib32\Irvine32.asm (usually C:\Irvine)

# MULTIMODULE PROGRAMS

## Multimodule Programs in Assembly

Large programs can be divided into **multiple modules** (separately assembled units).

- Easier to manage and assemble.
- Change in one module only requires reassembling that module.
- The linker combines all assembled modules (.OBJ files) into a single executable.

## Two Approaches to Multimodule Programs

1. **Traditional Approach**
   - Use **EXTERN** to declare procedures defined in other modules.
   - Use **PUBLIC** to export procedures for other modules.
2. **Modern MASM Approach**
   - Use **INVOKE** and **PROTO** directives.
   - Simplifies procedure calls.
   - Hides some low-level details of stack handling and calling conventions.

## Hiding and Exporting Procedure Names

By default, MASM makes all procedures public.

Any procedure can be called from any module in the program.

**You can restrict visibility:**

PRIVATE qualifier – makes a single procedure private.

OPTION PROC:PRIVATE – makes all procedures in a module private by default.

- You can still export specific procedures using the PUBLIC directive.
- **Important:**
  - *If you use OPTION PROC:PRIVATE in the startup module, make sure the startup procedure (usually main) is declared PUBLIC.*
  - *Otherwise, the OS loader will not find it, and the program will fail to start.*

## Key takeaway:

- Use **PRIVATE** and **PUBLIC** to control which procedures are visible outside a module.
- Use **EXTERN** and **PUBLIC** (traditional) or **INVOKE/PROTO** (modern MASM) for modular procedure calls.

The following example shows how to create a multimodule program using the traditional approach:

```
1235 ; Module 1: mod1.asm
1236
1237 myProc PROC PUBLIC
1238 ; ...
1239
1240 myProc ENDP
1241
1242 ; Module 2: mod2.asm
1243
1244 EXTERN myProc
1245
1246 main PROC
1247 ; ...
1248
1249 INVOKE myProc
1250
1251 main ENDP
```

## Assembling a Multimodule Program

To assemble a program with multiple modules, use **MASM commands** for each module.

- Example: assemble myprog.asm → produces myprog.obj.
- Use the **linker** to combine all .OBJ files → produces the executable myprog.exe.

## Key Directives for Multimodule Programs

**EXTERN**

- Tells the assembler that a procedure (e.g., myProc()) is **defined in another module**.
- Example: EXTERN myProc:PROC

**PUBLIC**

- Tells the assembler that a procedure can be **called from other modules**.
- Example: PUBLIC myProc

**INVOKE and PROTO (Modern MASM)**

- **PROTO** declares the procedure's interface (parameters and types).
- **INVOKE** calls the procedure using its declared interface.
- Simplifies **stack handling** and **procedure calls**, hiding low-level details.

**Key takeaway:**

- **EXTERN/PUBLIC** → traditional way to connect modules.
- **PROTO/INVOKE** → modern MASM way, safer and easier to manage parameters.
- Always ensure **startup procedure (main)** is PUBLIC if using PRIVATE by default.

## Using the INVOKE and PROTO Directives

The following example shows how to create a multimodule program using Microsoft's advanced INVOKE and PROTO directives:

```
1257 ; Module 1: mod1.asm
1258
1259 PROTO myProc
1260
1261 myProc PROC PUBLIC
1262 ; ...
1263
1264 myProc ENDP
1265
1266 ; Module 2: mod2.asm
1267
1268 INVOKE myProc
1269
1270 main PROC
1271 ; ...
1272
1273 main ENDP
```

To assemble the program, you would use the following commands:

```
1282 ml /c /obj mod1.asm
1283 ml /c /obj mod2.asm
1284 link mod1.obj mod2.obj /out:myprog.exe
```

This would create an executable file called myprog.exe.

The **/c option** tells MASM to compile the source code but not link it. The **/obj option** tells MASM to generate object files. The **/out: option** tells MASM to generate an executable file with the specified name.

The PROTO directive tells the assembler about the prototype of the myProc() procedure, including its name and the number and types of its arguments.

The INVOKE directive tells the assembler to call the myProc() procedure.

The INVOKE and PROTO directives simplify procedure calls and hide some low-level details, such as the need to push and pop arguments on the stack.

# CALLING EXTERNAL PROCEDURES

## Calling External Procedures in MASM

When you want to call a procedure defined in another module, MASM provides two directives:

1. **EXTERN** – Tells the assembler that a procedure is defined elsewhere.
2. **PROTO** – Declares the **prototype** of an external procedure, allowing **argument checking** and better optimization.

## 1. Using EXTERN

The **EXTERN directive** declares an external procedure and optionally its **stack frame size**.

**Syntax:**

```
EXTERN procedureName@stackSize:PROC
```

- procedureName → Name of the external procedure
- @stackSize → Total size of the stack used for parameters (in bytes)
- :PROC → Tells the assembler this is a procedure

**Example 1: External procedure with no parameters**

```
EXTERN Sub1@0:PROC  ; Sub1 takes 0 parameters

.code
main PROC
    call Sub1@0       ; Call external procedure
    ret
main ENDP
```

- @0 → Indicates 0 bytes of parameters
- call uses the decorated name with stack size

**Example 2: External procedure with two DWORD parameters**

```
EXTERN AddTwo@8:PROC  ; AddTwo takes 2 DWORDs (2 x 4 bytes = 8)

.data
val1 DWORD 10
val2 DWORD 20

.code
main PROC
    push val2          ; Push second parameter
    push val1          ; Push first parameter
    call AddTwo@8      ; Call external procedure
    ret
main ENDP
```

- @8 → 2 parameters × 4 bytes each = 8
- Parameters are pushed **right to left** (last parameter first)

## 2. Using PROTO (Recommended)

The **PROTO directive** declares the **procedure prototype**, including:

- Name
- Number of parameters
- Type of each parameter

This allows **MASM to check arguments** and **optimize calls**. Syntax:

```
procedureName PROTO [attributes], param1:type, param2:type, ...
```

## Example: Declaring the prototype for AddTwo

```
AddTwo PROTO STDCALL, val1:DWORD, val2:DWORD
```

- STDCALL → Parameter passing convention
- MASM now knows **AddTwo expects two DWORDs**

## Calling the external procedure with INVOKE:

```
.data
val1 DWORD 10
val2 DWORD 20

.code
main PROC
    INVOKE AddTwo, val1, val2  ; MASM automatically pushes args in correct order
    ret
main ENDP
```

- **No need to manually push parameters**
- MASM checks **number and types** of arguments at **assembly time**

## Key Differences


**Recommendation:** Use **PROTO** if you are calling multiple external procedures or want MASM to check arguments and optimize code.

## Summary

1. **EXTERN** is simple but less safe:
   - Requires stack size suffix
   - Manual parameter pushing
2. **PROTO** is safer and more readable:
   - Supports INVOKE
   - Performs argument checking
   - Optimized for performance

# Using Variables and Symbols Across Module Boundaries

## Exporting and Importing Variables

In MASM, you can share variables and symbols between modules using **EXTERNDEF**.

## What is EXTERNDEF?

- EXTERNDEF combines the functionality of **PUBLIC** and **EXTERN**.
- It allows you to **export a variable or symbol** from one module and **import it into another**.

## How to Use EXTERNDEF

### Step 1: Create an Include File

Define the variables and symbols you want to share in a .inc file.

```
; vars.inc
EXTERNDEF count:DWORD, SYM1:ABS
```

- count → a DWORD variable
- SYM1 → an absolute symbol

### Step 2: Export Variables in a Module

In the module that defines the variables, **include the .inc file**:

```
; sub1.asm
.386
.model flat, STDCALL
INCLUDE vars.inc

.data
count DWORD 0       ; initialize exported variable
SYM1 10             ; define absolute symbol

END
```

EXTERNDEF ensures these symbols are available to other modules.

**Step 3: Import and Use Variables in Another Module**

In a different module, include the same .inc file and use the variables:

```
; main.asm
.386
.model flat, STDCALL
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD
INCLUDE vars.inc

.code
main PROC
    mov count, 5        ; use imported variable
    mov eax, SYM1       ; use imported symbol
    INVOKE ExitProcess, 0
main ENDP

END main
```

- count and SYM1 are now accessible in main.asm.
- No need to redeclare them manually.

## Benefits of EXTERNDEF

1. **Modular Code:** Easily share variables and symbols across modules.
2. **Reduces Duplicate Code:** No need to redefine shared variables in multiple modules.
3. **Reusability:** Makes your code more maintainable and organized.

## Summary

- **EXTERNDEF = PUBLIC + EXTERN**
- Use .inc files to define shared variables for multiple modules.
- Always include the .inc file in modules that export or import the variables.

**Tip:** For large projects, using EXTERNDEF keeps your global symbols organized and avoids naming conflicts.

## Making the ArraySum Program Modular

The **ArraySum program** is a great example of a **multimodule program**.

Modules and Responsibilities:

- **main.asm** – Startup module.
  - Calls other modules to perform tasks.
- **promptforintegers.asm** – Handles user input.
  - Prompts for array elements and reads them from the console.
- **arraysum.asm** – Performs computation.
  - Calculates the **sum of the integers** in the array.
- **writeinteger.asm** – Handles output.
  - Writes an integer to the console.

## Program Structure

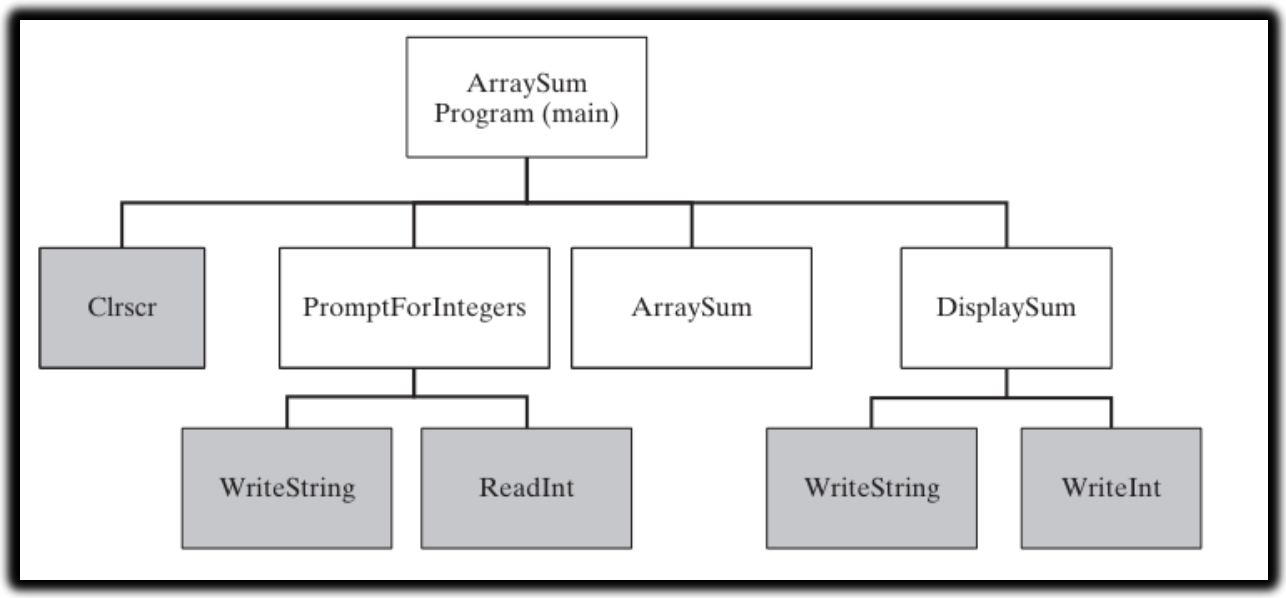
Each module is **assembled independently** into an .OBJ file.

The **linker** combines all .OBJ files into a **single executable**.

Modular design allows:

- Easy management of large programs.
- Only reassembling a module when its source changes.
- Clear separation of responsibilities.

## Structure Chart (Conceptual)




- **main.asm** coordinates the program flow.
- The other modules perform specialized tasks.

## Short note:

- Modular design demonstrates **EXTERN/PUBLIC or INVOKE/PROTO directives** in action.
- Each module focuses on **one task**, making the program easier to maintain and debug.
- Read the .asm files in the advanced procedures folder, then continue…

## ✅ Benefits of this style:

1. Each module explains **what it does and what its parameters are**.
2. You can see **how the stack and registers are used** at a glance.
3. No need for separate lengthy notes — the code is **self-documenting**.

# Creating Modules Using INVOKE and PROTO (Notes)

## Purpose of INVOKE, PROTO, and PROC

- These directives are used to create **multimodule programs** in 32-bit mode.
- They simplify calling procedures across modules and ensure **type and argument checking** at assembly time.

## PROC Directive

- Used to **define a procedure**.
- Can also **declare parameters** for the procedure.
- Example:

```
ArraySum PROC, ptrArray:PTR DWORD, arraySize:DWORD
...
ArraySum ENDP
```

## PROTO Directive

- Declares a **prototype** for a procedure.
- Tells the assembler the procedure's **name, parameters, and calling convention**.
- Allows calling a procedure **before it is defined**.
- Example:

```
ArraySum PROTO, ptrArray:PTR DWORD, arraySize:DWORD
```

## INVOKE Directive

- Used to **call a procedure** in another module.
- Automatically **passes parameters** according to the prototype.
- Performs **basic argument type checking** at assembly time.
- Example:

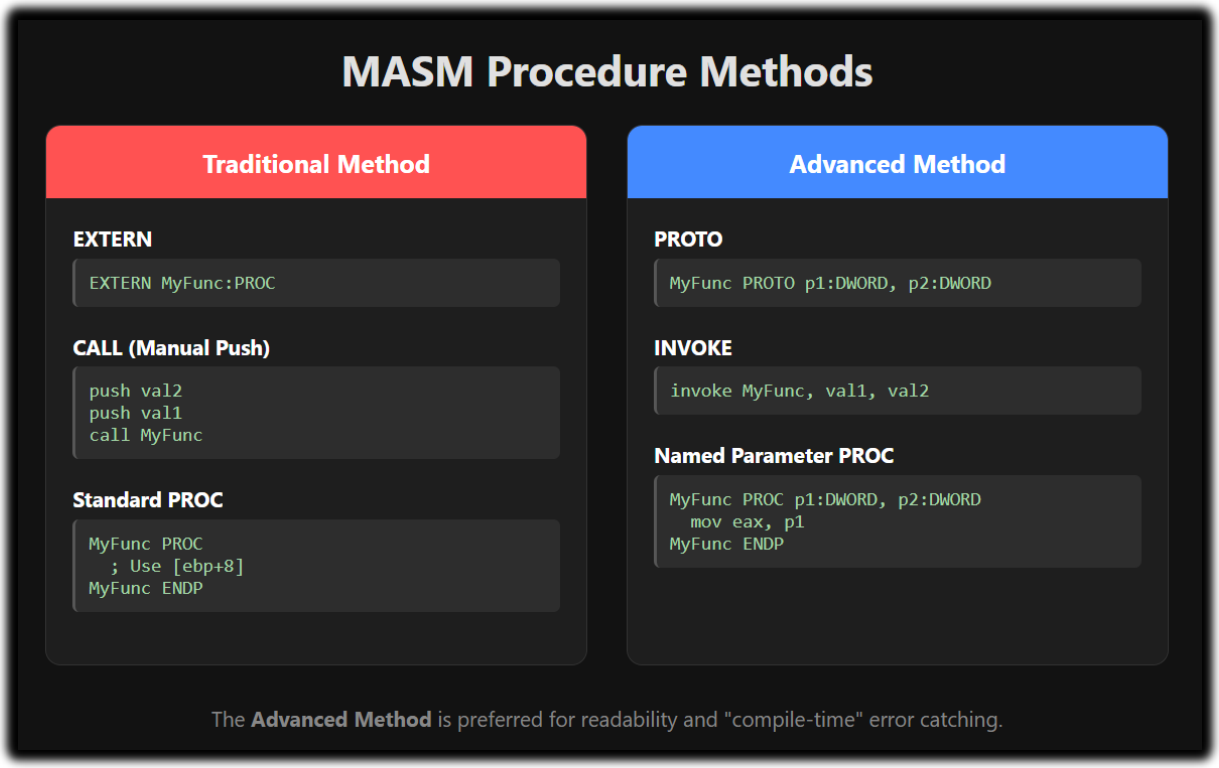
```
INVOKE ArraySum, ADDR array, Count
```

## Advantages of Using INVOKE, PROTO, and PROC

- **Argument matching:** Assembler ensures that arguments passed to procedures match the parameters.
- **Error prevention:** Reduces runtime crashes due to mismatched arguments.
- **Modular design:** Makes it easy to create reusable, maintainable code across multiple modules.
- **Readability:** Improves the clarity of procedure calls compared to using CALL and manual stack manipulation.

## Workflow in a Multimodule Program

- Define procedures in separate modules using **PROC**.
- Declare their prototypes in the main module or a header include file using **PROTO**.
- Call the procedures using **INVOKE**, passing the appropriate arguments.




Example Using INVOKE, PROTO, and PROC:

```
1430 ; Include the necessary include file
1431 INCLUDE sum.inc
1432
1433 ; Define data and code sections
1434 .data
1435 Count = 3
1436 prompt1 BYTE "Enter a signed integer: ",0
1437 prompt2 BYTE "The sum of the integers is: ",0
1438 array DWORD Count DUP(?)
1439 sum DWORD ?
1440
1441 .code
1442 main PROC
1443     call Clrscr
1444
1445     ; Call PromptForIntegers using INVOKE with argument lists
1446     INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count
1447
1448     ; Call ArraySum using INVOKE with argument lists
1449     INVOKE ArraySum, ADDR array, Count
1450     mov sum, eax
1451
1452     ; Call DisplaySum using INVOKE with argument lists
1453     INVOKE DisplaySum, ADDR prompt2, sum
1454
1455     call Crlf
1456     exit
1457 main ENDP
```

These are all the functions using advanced methods:

```
; sum.inc
INCLUDE Irvine32.inc

PromptForIntegers PROTO,
    ptrPrompt:PTR BYTE,
    ptrArray:PTR DWORD,
    arraySize:DWORD

ArraySum PROTO,
    ptrArray:PTR DWORD,
    arraySize:DWORD

DisplaySum PROTO,
    ptrPrompt:PTR BYTE,
    theSum:DWORD
```

```
; prompt.asm
INCLUDE sum.inc

.code
PromptForIntegers PROC,
    ptrPrompt:PTR BYTE,
    ptrArray:PTR DWORD,
    arraySize:DWORD

    pushad
    mov ecx, arraySize
    cmp ecx, 0
    jle L2
    mov edx, ptrPrompt
    mov esi, ptrArray
L1:
    call WriteString
    call ReadInt
    call Crlf
    mov [esi], eax
    add esi, 4
    loop L1
L2:
    popad
    ret
PromptForIntegers ENDP
END
```

```
1530 ; arraysum.asm
1531 INCLUDE sum.inc
1532
1533 .code
1534 ArraySum PROC,
1535     ptrArray:PTR DWORD,
1536     arraySize:DWORD
1537
1538     push ecx
1539     mov eax, 0
1540     mov esi, ptrArray
1541     mov ecx, arraySize
1542     cmp ecx, 0
1543     jle L2
1544 L1:
1545     add eax, [esi]
1546     add esi, 4
1547     loop L1
1548 L2:
1549     pop ecx
1550     ret
1551 ArraySum ENDP
1552 END
```


```
1530 ; arraysum.asm
1531 INCLUDE sum.inc
1532
1533 .code
1534 ArraySum PROC,
1535     ptrArray:PTR DWORD,
1536     arraySize:DWORD
1537
1538     push ecx
1539     mov eax, 0
1540     mov esi, ptrArray
1541     mov ecx, arraySize
1542     cmp ecx, 0
1543     jle L2
1544 L1:
1545     add eax, [esi]
1546     add esi, 4
1547     loop L1
1548 L2:
1549     pop ecx
1550     ret
1551 ArraySum ENDP
1552 END
```

```
1557 ; display.asm
1558 INCLUDE sum.inc
1559
1560 .code
1561 DisplaySum PROC,
1562     ptrPrompt:PTR BYTE,
1563     theSum:DWORD
1564
1565     push eax
1566     push edx
1567     mov edx, ptrPrompt
1568     call WriteString
1569     mov eax, theSum
1570     call WriteInt
1571     call Crlf
1572     pop edx
1573     pop eax
1574     ret
1575 DisplaySum ENDP
1576 END
```

```
1580 ; sum_main.asm
1581 INCLUDE sum.inc
1582
1583 Count = 3
1584
1585 .data
1586     prompt1 BYTE "Enter a signed integer: ", 0
1587     prompt2 BYTE "The sum of the integers is: ", 0
1588     array DWORD Count DUP(?)
1589     sum DWORD ?
1590 .code
1591     main PROC
1592         call Clrscr
1593         INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count
1594         INVOKE ArraySum, ADDR array, Count
1595         mov sum, eax
1596         INVOKE DisplaySum, ADDR prompt2, sum
1597         call Crlf
1598         exit
1599     main ENDP
1600     END
```

Here is a summary of the two ways to create multimodule programs:

# Methods for Creating Multimodule Programs

## I. Traditional Method

- Use the **EXTERN** directive to declare symbols defined in another module.
- Use the **CALL** instruction to call procedures in other modules.
- Requires manual handling of parameters and calling details.

## II. Advanced Method (32-bit Mode)

- Use the **PROTO** directive to declare procedure prototypes from other modules.
- Use the **INVOKE** directive to call procedures.
- Use the **PROC** directive to define procedures and declare parameters.
- Simplifies procedure calls and automatically handles stack details.
- Available **only in 32-bit mode**.

## III. Conclusion

- The **advanced method** is the **preferred approach** for multimodule programs in 32-bit mode.
- It is **simpler, safer, and more efficient** than the traditional method.
- The **traditional method** is still supported and may be required for certain programs or older code.

# ADVANCED OPTIONAL TOPIC 1 – USES OPERATOR

## USES Operator in Procedures (MASM / x86)

### Purpose

The **USES operator** is a MASM feature that automatically:

- **Saves specified registers** at the start of a procedure
- **Restores them** before returning

This helps ensure that a procedure does **not accidentally modify registers** that the caller expects to remain unchanged.

### How the USES Operator Works

When you write:

```
MySub1 PROC USES ecx edx
```

MASM automatically generates code equivalent to:

```
push ecx
push edx
; procedure body
pop edx
pop ecx
ret
```

### Key Behavior

Registers listed after USES are:

- Pushed onto the stack on entry
- Popped in reverse order on exit

This follows standard stack discipline (LIFO).

```
1604 MySub1 PROC USES ecx edx
1605 ret
1606 MySub1 ENDP
```

What Happens Internally:

1. ECX is pushed onto the stack
2. EDX is pushed onto the stack
3. Procedure executes
4. EDX is restored
5. ECX is restored
6. Control returns to the caller

The following code is generated by MASM when it assembles MySub1:

```
1610 push ecx
1611 push edx
1612 pop edx
1613 pop ecx
1614 ret
```

Suppose we combine USES with a stack parameter, as does the following MySub2 procedure. Its parameter is expected to be located on the stack at EBP+8:

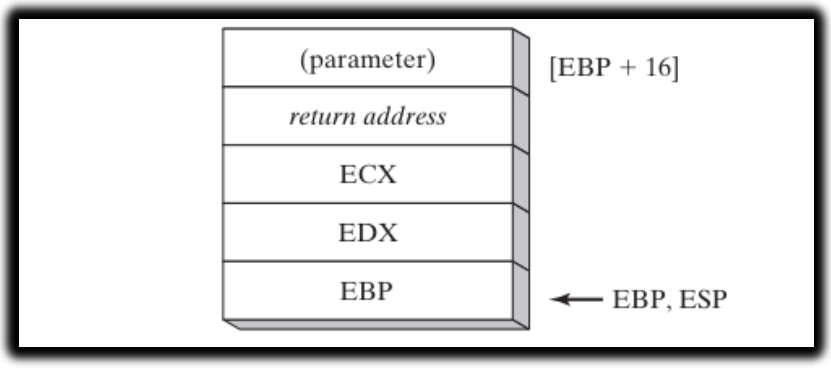
```
1620 MySub2 PROC USES ecx edx
1621 push ebp
1622 mov ebp,esp
1623 mov eax,[ebp+8]
1624 ; this is wrong!
1625 pop ebp
1626 ret 4
1627 MySub2 ENDP
```

Here is the corresponding code generated by MASM for MySub2:

```
1630 push ecx
1631 push edx
1632 push ebp
1633 mov ebp,esp
1634 mov eax,dword ptr [ebp+8]
1635 pop ebp
1636 pop edx
1637 pop ecx
1638 ret 4
```

An error results because the assembler inserted the PUSH instructions for ECX and EDX at the beginning of the procedure, altering the offset of the stack parameter.

Figure 8-6 below shows how the stack parameter must now be referenced as [EBP+16]. USES modifies the stack before saving EBP, which corrupts the standard prologue code commonly used for subroutines.




## Why the USES Operator Can Break Stack-Based Parameters

Core Rule (High-Level) - Do not use the USES operator in procedures that access parameters using constant stack offsets such as:

```
mov eax, [ebp + 8]
```

If you need to preserve registers in such procedures, **save and restore them manually** using PUSH and POP.

## Standard Procedure Call & Stack Setup (x86, 32-bit)

Typical Call Sequence

- **Caller** pushes arguments onto the stack (right-to-left).
- CALL instruction: Pushes the return address.
- **Callee** sets up a stack frame (standard prologue):

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

Resulting Stack Layout

```
[ebp + 8]   → first argument
[ebp + 4]   → return address
[ebp]       → saved EBP
```

📌 This layout is what makes constant offsets like [ebp + 8] valid.

## What the USES Operator Actually Does

When you declare:

```
MySub PROC USES ecx edx
```

at the **start** of the procedure, and corresponding POPs at the end.

## Important Detail

- These PUSH instructions are **hidden**
- They **modify the stack layout**
- You don't see them unless you inspect the generated code

## Why This Causes Problems with Constant Offsets

### The Core Issue

Each PUSH instruction:

- Decrements ESP
- Adds extra data to the stack

When USES is applied **before or during stack-frame setup**, the stack pointer no longer matches the layout assumed by constant offsets.

### Result

[ebp + 8] no longer points to the intended argument

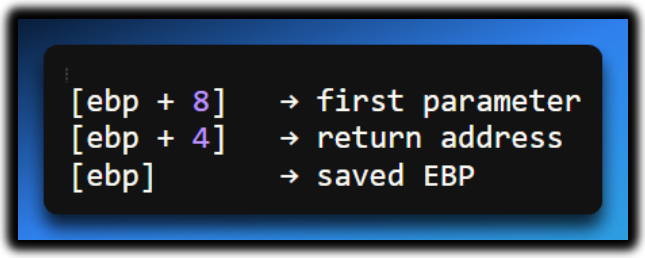
Parameters are **shifted**

Leads to:

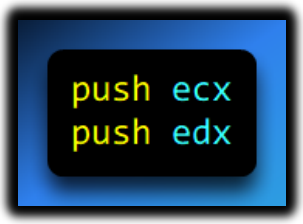
- Wrong values
- Subtle bugs
- Hard-to-debug crashes

## Conceptual Stack Illustration

**Without USES**




**With USES ecx edx -** Hidden code:



Now the stack contains **extra entries before parameters**, breaking the assumed offsets.

📌 **Constant offsets + USES = broken assumptions**

## Important Correction (Fixing the 2022 Confusion)

❌ **Incorrect statement (common in older notes):**

"The procedure saves its caller-saved registers (EBP, ESI, EDI, EBX)."

**Correct Version**

- **EBP, ESI, EDI, EBX are callee-saved registers**
- They are saved **only if the procedure explicitly saves them**
- USES saves **only the registers you list**, nothing more

This distinction matters when reasoning about stack layout.

## When It Is Safe to Use USES

✅ **Safe When**

- The procedure **does not use fixed [ebp + constant] offsets**
- Parameters are accessed via:
    - Registers
    - MASM-generated stack frames (PROTO, INVOKE)
- You let MASM fully manage the calling convention

❌ **Unsafe When**

- You manually manage the stack frame
- You rely on exact offsets like [ebp + 8]
- You mix USES with custom prologue/epilogue logic

## Recommended Alternative: Manual Register Saving

When precise stack control is required:

```
push ecx
push edx

; procedure body (safe to use [ebp + 8])

pop edx
pop ecx
```

## Why This Works

- You control **exactly when** the stack changes
- Stack layout remains predictable
- No hidden side effects

## Common Pitfalls

- Assuming USES is “free”
- Forgetting that PUSH changes stack offsets
- Mixing automatic (USES) and manual stack management
- Debugging wrong parameters instead of the stack layout

## Key Takeaways (Exam / Interview Ready)

- USES automatically saves registers using hidden PUSH/POP
- These instructions **modify the stack**
- Fixed stack offsets depend on an **unchanged stack frame**
- Never mix USES with [ebp + constant] addressing
- For low-level or manual stack control:
  - Save registers explicitly
  - Or avoid USES

# PASSING 8-BIT AND 16-BIT ARGUMENTS ON THE STACK

## Passing 8-bit Arguments (32-bit Mode)

When passing stack arguments to procedures in 32-bit mode, it is best to push **32-bit operands**.

The stack pointer (**ESP**) should remain aligned on a **doubleword (32-bit) boundary**.

Pushing **16-bit operands** can misalign ESP.

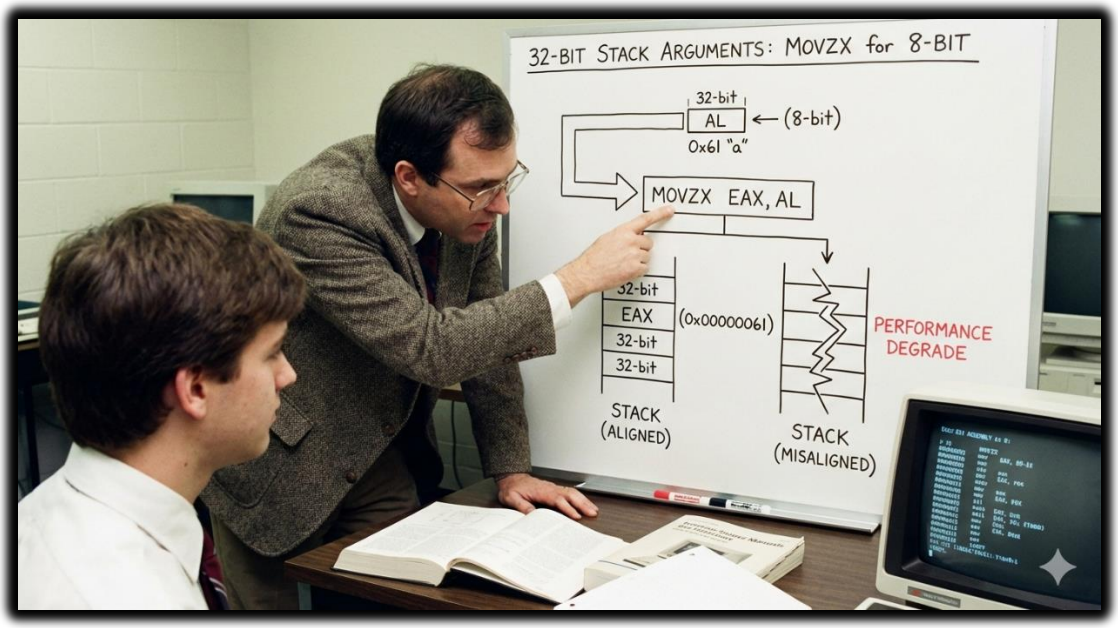
- Misalignment can **degrade runtime performance**.
- It can also cause problems with **calling conventions** that assume 4-byte alignment.

If a **16-bit operand** must be passed to a procedure in 32-bit mode:

- Use the **MOVZX** instruction to extend the operand to **32 bits** before pushing it onto the stack.

For example:

- The Uppercase procedure receives a **character argument**.
- It returns the character's **uppercase equivalent** in **AL**.



```
1645 Uppercase PROC
1646 push ebp
1647 mov ebp,esp
1648 mov al,[esp+8]
1649 ; AL = character
1650 cmp al,'a'
1651 ; less than 'a'?
1652 jb L1
1653 ; yes: do nothing
1654 cmp al,'z'
1655 ; greater than 'z'?
1656 ja L1
1657 ; yes: do nothing
1658 sub al,32
1659 ; no: convert it
1660 L1:
1661 pop ebp
1662 ret 4
1663 ; clean up the stack
1664 Uppercase ENDP
```

If we pass a character literal to Uppercase, the PUSH instruction automatically expands the value to 32 bits in 32-bit mode.

```
1667 push 'x'
1668 call Uppercase
```

However, if we pass a character variable to Uppercase, the PUSH instruction does not allow an 8-bit operand to be pushed directly onto the stack.

To work around this, we can use the MOVZX instruction to expand the character into a 32-bit register (such as EAX) before pushing it onto the stack.

```
1672 .data
1673     charVal BYTE 'x'
1674 .code
1675     movzx eax,charVal
1676     ; move with extension
1677     push eax
1678     call Uppercase
```

This preserves **doubleword alignment of ESP** and avoids potential performance or calling-convention issues when calling Uppercase.

## Passing 16-bit Arguments.

The AddTwo procedure expects two **32-bit integer arguments**. However, the word1 and word2 variables are **16-bit integers**.

If word1 and word2 are pushed directly onto the stack, they will not be passed in the expected 32-bit format, and AddTwo will not correctly interpret the arguments.

To fix this, each argument must be **extended to 32 bits** before being pushed onto the stack.

- If the values are **unsigned**, zero-extension can be used, which sets the high-order 16 bits to zero.
- If the values are **signed**, sign-extension must be used to preserve the sign.

This converts each 16-bit argument into a proper 32-bit argument, allowing AddTwo to correctly add the two integers.

The following code correctly calls AddTwo by extending each argument to 32 bits before pushing it onto the stack:

```
1686 .data
1687     word1 WORD 1234h
1688     word2 WORD 4111h
1689 .code
1690     movzx eax,word1
1691     push eax
1692     movzx eax,word2
1693     push eax
1694     call AddTwo
1695     ; sum is in EAX
```

The MOVZX instruction is used to zero-extend the 16-bit word1 and word2 variables into the 32-bit EAX register.

After being extended to 32 bits, the arguments are pushed onto the stack in **reverse order** (right-to-left), with word2 pushed first and word1 pushed last.

When AddTwo executes, it accesses the arguments from the stack and adds them together. The sum of the two integers is returned in the **EAX** register.

It is important that the **caller** ensures the arguments passed to a procedure match the parameters expected by the procedure.

For stack parameters, both the **order** and **size** of the arguments are critical. Passing an incorrect number of arguments, using the wrong order, or passing arguments of the wrong size can lead to incorrect results or program failure.

In the past notes we had said and What we corrected ⚠

## 1. Stack arguments are not popped by the CALL

This sentence is inaccurate:

"When AddTwo is called, it will pop the two arguments off the stack…"

The **CALL instruction does not pop arguments**.

Depending on the calling convention:

- Either the **callee** removes them (e.g., RET 8)
- Or the **caller** removes them (e.g., ADD ESP, 8)

## 2. Clarify zero-extension assumption

- MOVZX is correct **only if the values are unsigned**.
- If signed, MOVSX should be used.

# Passing 64-bit Arguments

To pass a 64-bit integer argument to a procedure in 32-bit mode, the argument must be split into two 32-bit doublewords.

The **high-order doubleword** is pushed first, followed by the **low-order doubleword**. This ensures that the **lower-order doubleword is at the lower memory address** on the stack, consistent with the stack growing downward.

Proper ordering allows procedures, like WriteHex64, to access the 64-bit argument correctly.

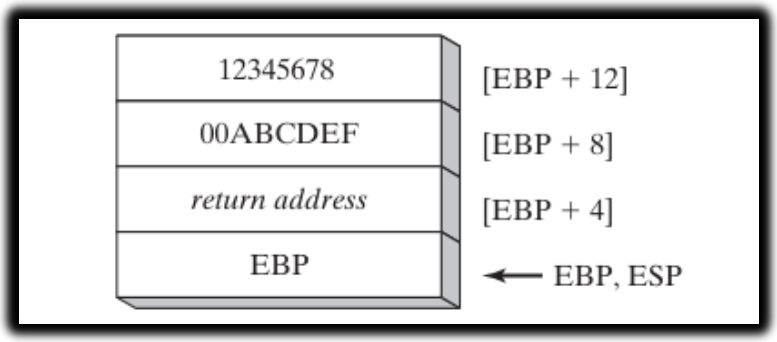
As with all stack arguments, the **caller** must ensure that the size and order of the arguments match what the procedure expects.

```
1700 WriteHex64 PROC
1701     push ebp
1702     mov ebp,esp
1703     mov eax,[ebp+12]
1704     ; high doubleword
1705     call WriteHex
1706     mov eax,[ebp+8]
1707     ; low doubleword
1708     call WriteHex
1709     pop ebp
1710     ret 8
1711 WriteHex64 ENDP
```

The following sample call to WriteHex64 pushes the upper half of longVal, followed by the lower half:

```
1715 .data
1716     longVal QWORD 1234567800ABCDEFh
1717 .code
1718     push DWORD PTR longVal + 4
1719     ; high doubleword
1720     push DWORD PTR longVal
1721     ; low doubleword
1722     call WriteHex64
```

Figure below shows the stack frame inside WriteHex64 just after EBP was pushed on the stack and ESP was copied to EBP:




Or

```
       High Memory
+---------------------+
|    ... (higher)     |  [EBP + 20] (Return Address)
+---------------------+
|         EBP         |   [EBP + 16] (Previous EBP)
+---------------------+
| High Doubleword     |  [EBP + 12] (high doubleword of longVal)
+---------------------+
| Low Doubleword      |  [EBP + 8] (low doubleword of longVal)
+---------------------+
       Low Memory
```

The WriteHex64 procedure can retrieve the **high-order and low-order doublewords** of a 64-bit integer argument from the stack and display them in hexadecimal.

As with all procedures, the **caller must ensure** that the arguments it passes match the parameters expected by the procedure.

For stack-based arguments, both the **order** and **size** of the arguments are important. Passing the wrong number, size, or order of arguments can lead to incorrect results or program crashes.

In 32-bit mode, the **stack grows downward**: pushing a value **decrements ESP**, and popping a value **increments ESP**.

To pass a 64-bit integer to a procedure, we must push the **high-order doubleword first**, followed by the **low-order doubleword**.

This ensures that the integer is stored on the stack in **little-endian order**, so that the **low-order doubleword is at the lower memory address**. This matches x86 expectations for 64-bit values in memory.

If we pushed the **low-order doubleword first**, followed by the high-order doubleword, the 64-bit integer would be stored in **reverse order** relative to what the procedure expects, causing incorrect behavior when the procedure accesses the stack.

The following diagram shows how a 64-bit integer is stored on the stack in little-endian order:

```
[EBP + 12] high doubleword of the integer
[EBP + 8] low doubleword of the integer
```

The WriteHex64 procedure can retrieve the high-order and low-order doublewords of a 64-bit integer from the stack and display them in hexadecimal.

It is important to ensure that the arguments passed to a procedure **match the expected parameters**. Procedures expect arguments to be passed in a **specific order, number, and size**.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or of the wrong size, the procedure may produce **incorrect results or crash**.

For example:

- If WriteHex64 expects **one 64-bit integer**, but the caller passes **two 64-bit integers**, the procedure will not display the values correctly.
- If the caller passes a **32-bit integer** instead of a 64-bit integer, the procedure will also fail to display the correct value.

In assembly, the **compiler does not check** argument consistency; it is the programmer's responsibility to ensure correctness.

In 32-bit mode, the **stack grows downward**. Pushing a value onto the stack **decrements ESP**, and popping a value **increments ESP**.

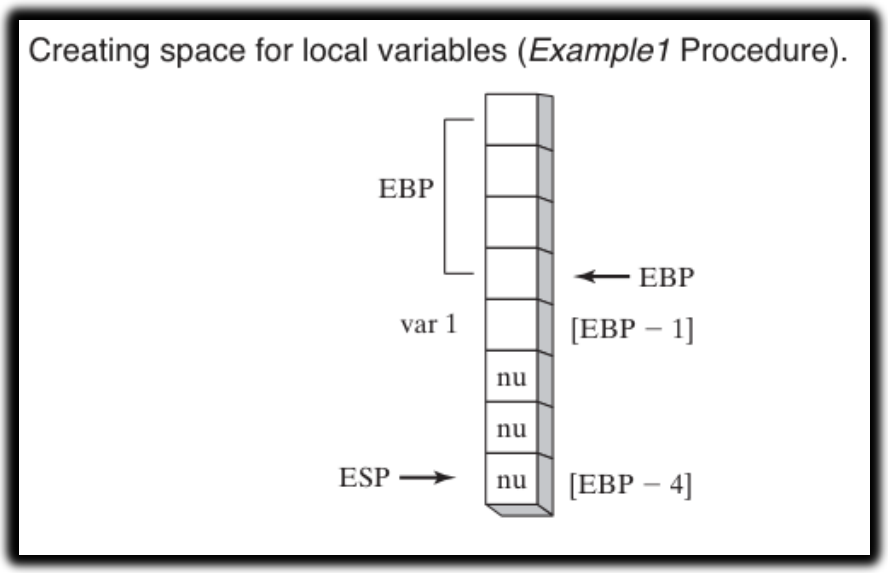
When a procedure declares a **local variable**, MASM allocates space for it on the stack. The allocated size depends on the variable's type:

- A **byte variable** is allocated 1 byte.
- A **word variable** is allocated 2 bytes.
- A **doubleword variable** is allocated 4 bytes.

If the variable size is **not a multiple of 4 bytes**, MASM **rounds up** the allocation to the next multiple of 4 bytes. This maintains **doubleword alignment**, which is required for proper stack operation.

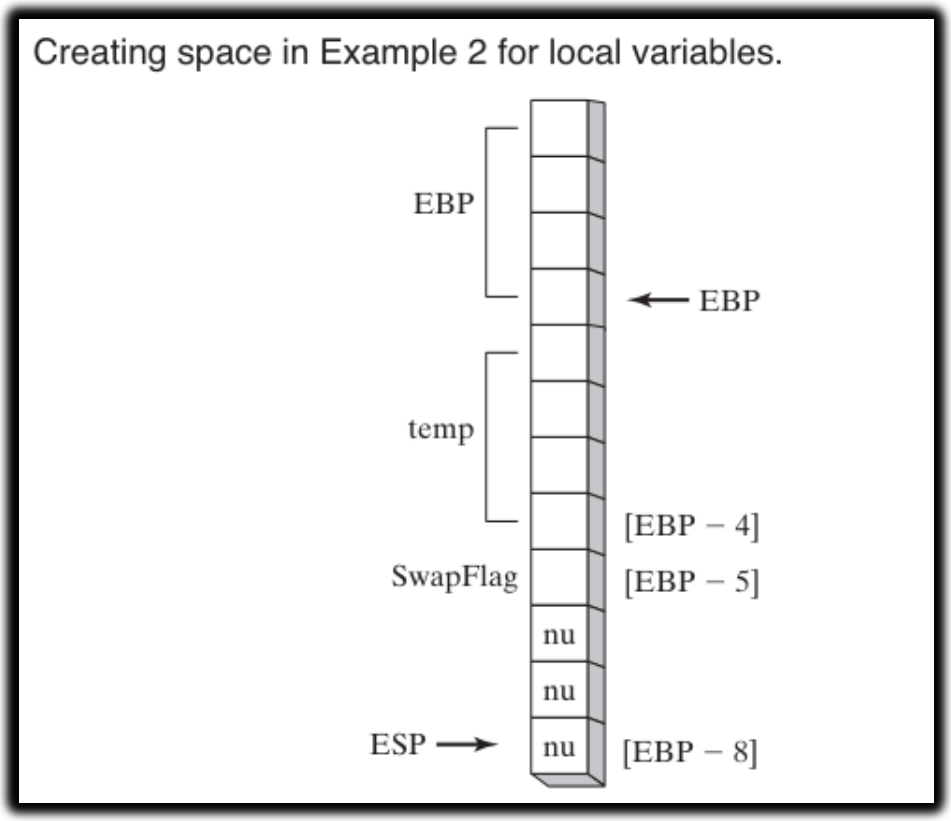
For example, if a byte variable var1 is declared in the Example1 procedure, MASM will allocate **4 bytes** on the stack. The extra 3 bytes are unused but ensure that the stack remains aligned on a **4-byte boundary**.

The following diagram shows how the stack looks after the Example1 procedure has been compiled and assembled:




The nu blocks represent unused bytes.

The following diagram shows how the stack looks after the Example2 procedure has been compiled and assembled:




The temp variable is a **doubleword variable**, so it is naturally aligned on a **doubleword (4-byte) boundary**.

The SwapFlag variable is a **byte variable**, but MASM still allocates **4 bytes** for it on the stack.

This ensures that the **stack remains aligned on a doubleword boundary**, which is required for proper stack operation and efficient access.

## Stack Size for Nested Procedure Calls

The stack size required for **nested procedure calls** is the **sum of the stack sizes required for each individual procedure call**.

This is because the stack stores the **local variables and return addresses** for all active procedures. Each time a procedure is called, space is allocated on the stack, and this space remains in use until the procedure returns.

For example, consider the following code:
*(example code would go here)*

```
1750 Sub1 PROC
1751 local array1[50]:dword
1752 ; 200 bytes
1753 callSub2
1754 .
1755 .
1756 ret
1757 Sub1 ENDP
1758 Sub2 PROC
1759 local array2[80]:word
1760 ; 160 bytes
1761 callSub3
1762 .
1763 .
1764 ret
1765 Sub2 ENDP
1766 Sub3 PROC
1767 local array3[300]:dword
1768 ; 1200 bytes
1769 .
1770 .
1771 ret
1772 Sub3 ENDP
```

## 1. Stack Size for Nested Procedure Calls

The stack size required for nested procedure calls is the **sum of the stack sizes required for each individual procedure**.

For example:

- Sub1 requires 200 bytes
- Sub2 requires 160 bytes
- Sub3 requires 1200 bytes

Therefore, the **total stack size** required for this code is **1560 bytes**.

This represents the **minimum stack space** required for correct execution. If the program attempts to use more stack space than is available, it may **crash**.

## 2. Recursive Procedure Calls

When a procedure is called **recursively**, the stack space it uses is approximately:

stack space ≈ size of local variables + parameters × estimated recursion depth

For example, if a procedure has 100 bytes of local variables and parameters, and it is called recursively to a depth of 10, it will require approximately **1000 bytes of stack space**.

## 3. Stack Overflow

A **stack overflow** occurs when a program requires more stack space than is available.

To avoid stack overflows:

- Be aware of the **stack space requirements** of your program.
- Use the **STACK directive** to reserve additional stack space if necessary.
- Understand the impact of **nested and recursive procedure calls**, since each active call consumes stack space.

## 4. Key Points from the Chapter

- There are two types of procedure parameters:
  - **Register parameters**: faster, used by Irvine libraries
  - **Stack parameters**: more flexible
- A **stack frame** contains a procedure's parameters, local variables, saved registers, and return address.
- Parameters can be passed **by value** (copied) or **by reference** (address passed). Arrays should usually be **passed by reference**.
- **Stack parameters** are accessed using **EBP offset addressing**, e.g., [EBP-8]. The **LEA instruction** is useful for obtaining addresses of stack parameters.
- **ENTER/LEAVE instructions** manage stack frame setup and teardown.
- **Recursive procedures** call themselves directly or indirectly; recursion is effective for repeating data structures.
- **Local variables**:
  - Have restricted scope
  - Lifetime tied to the procedure
  - Avoid naming conflicts
  - Enable recursion
- **INVOKE directive** calls procedures with multiple arguments; **ADDR** passes pointers.
- **PROC declares procedures**, **PROTO declares prototypes** of existing procedures.

- Large programs should be split into **multiple source code modules** for manageability.
- **Java bytecode** is the compiled form of Java programs; the **JVM executes bytecodes** using a **stack-oriented model**.