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.

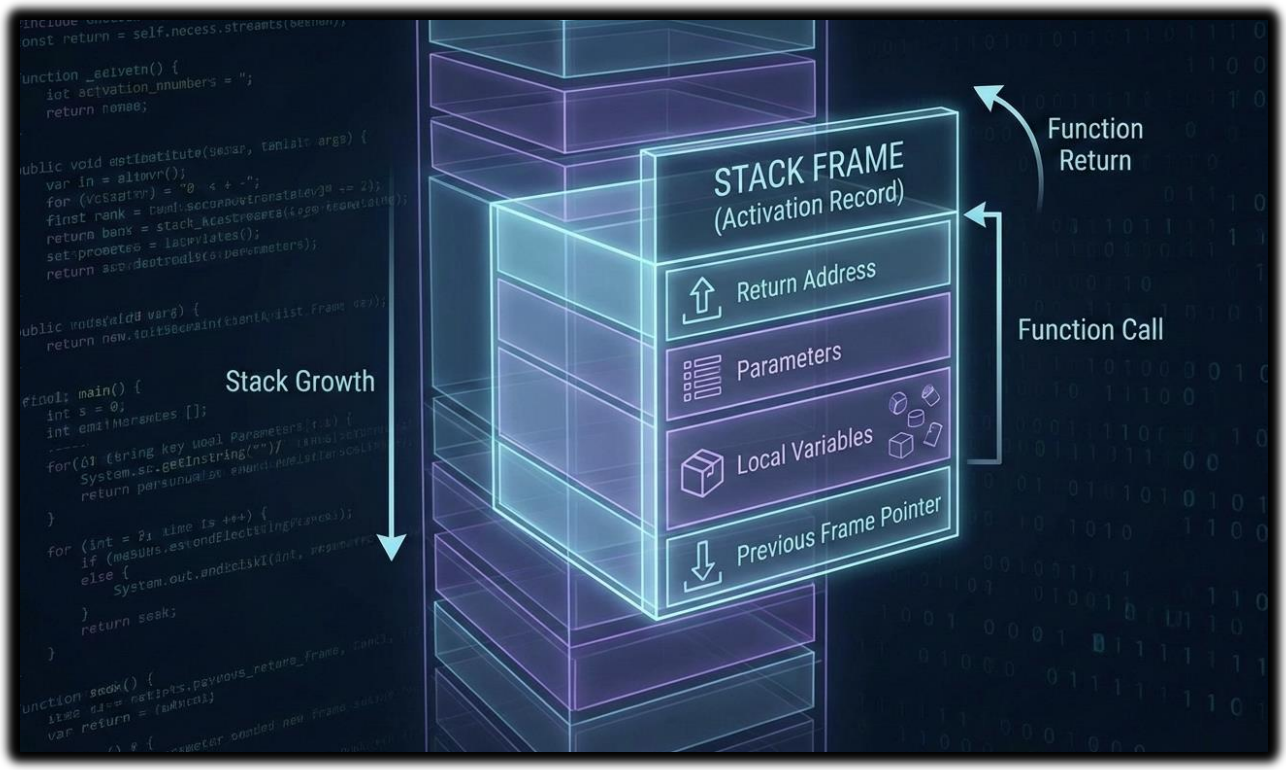
I. Stack Parameters

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

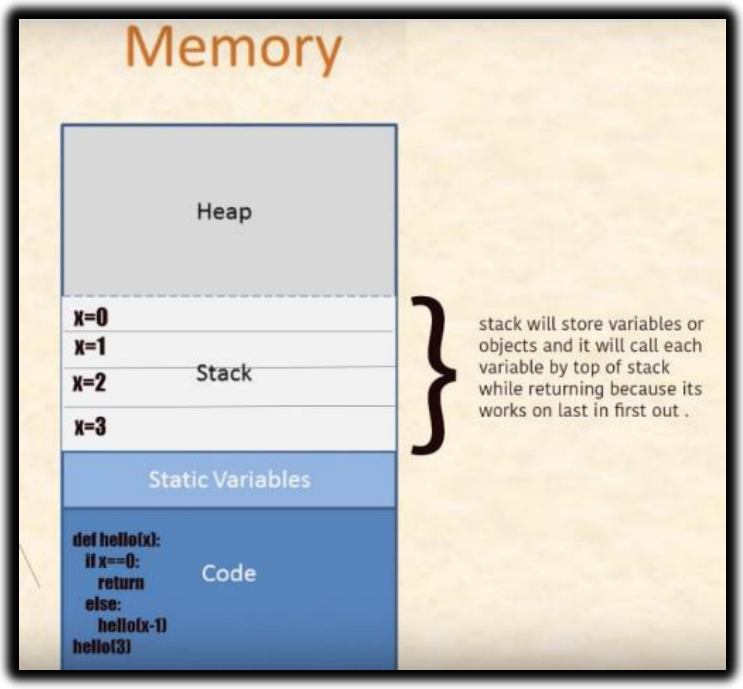
III. How a Stack Frame Is Built (Step by Step)

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.

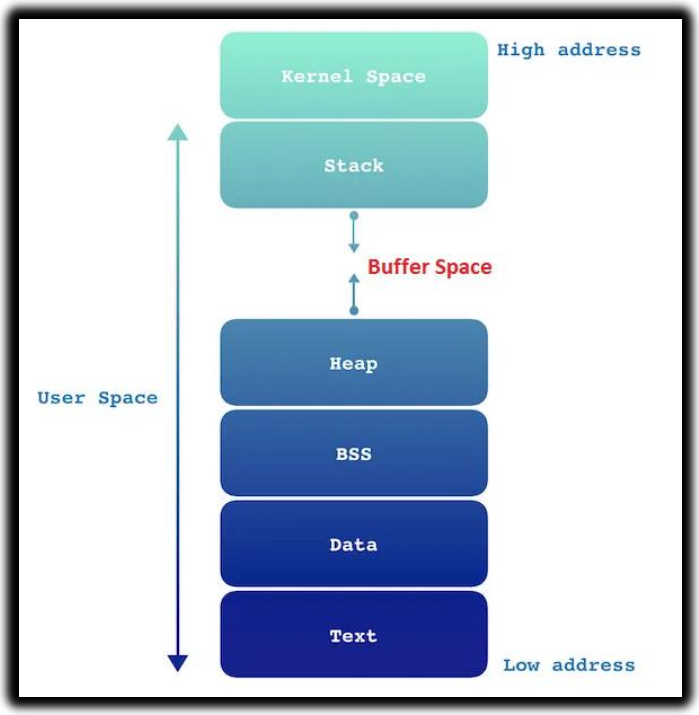
IV. Important Stack Behavior to Remember

- 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:

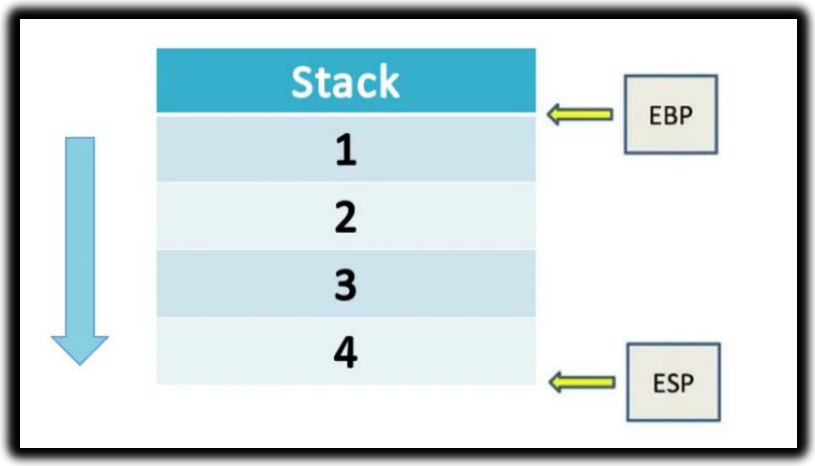
- The stack remains balanced
- Program flow continues correctly

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.



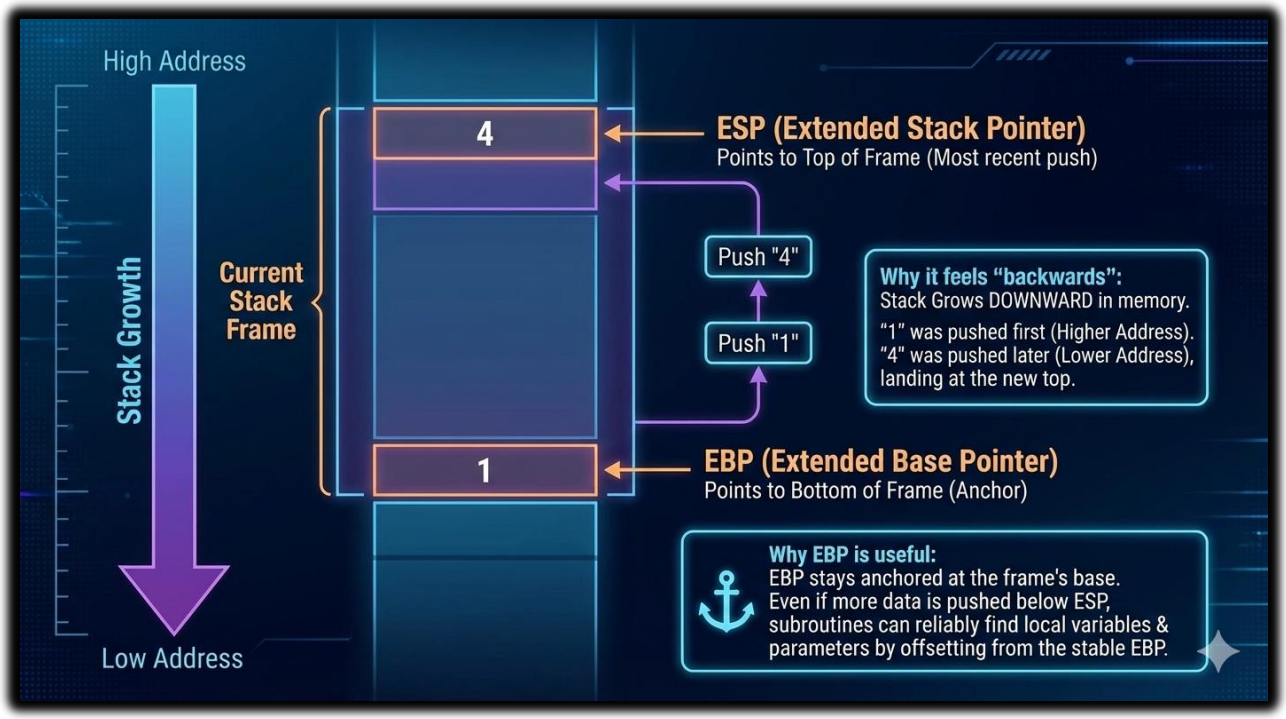
Right now, the stack pointer (ESP) is pointing at the very top of the current stack frame, and the value sitting there is 4.

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:

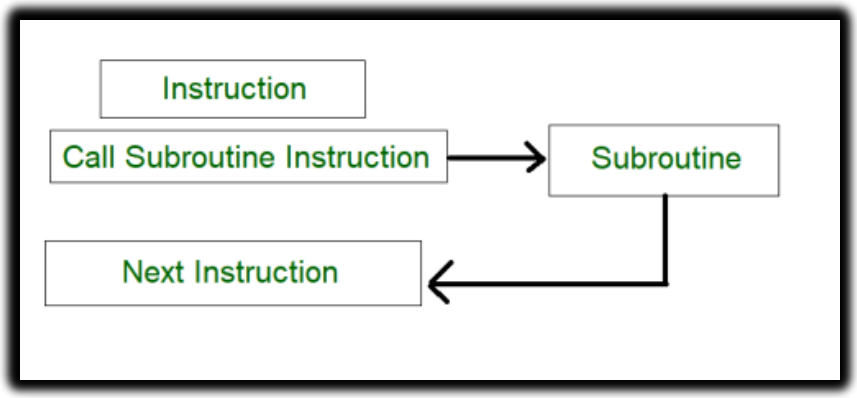
- First, the value 1 was pushed → it landed at a higher memory address
- Then the value 4 was pushed → it landed at a lower memory address

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:

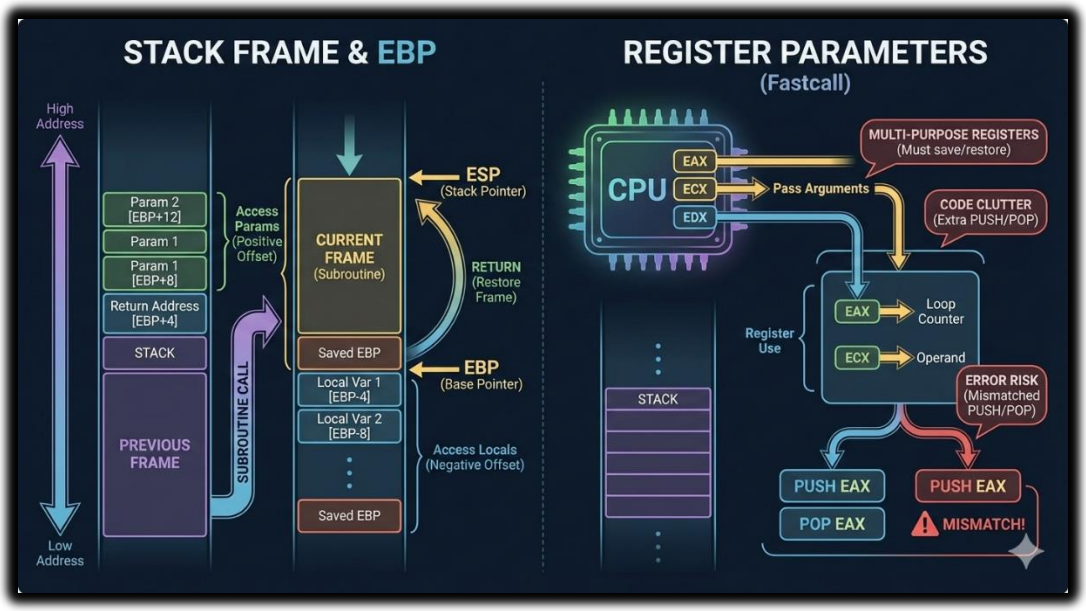
1. **Setup:**
   - The current value of EBP is pushed onto the stack.
   - EBP is then set equal to ESP, marking the start of the new frame.
2. **Accessing variables:**
   - Local variables are accessed using negative offsets from EBP.
     - First local variable → [EBP-4]
     - Second local variable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

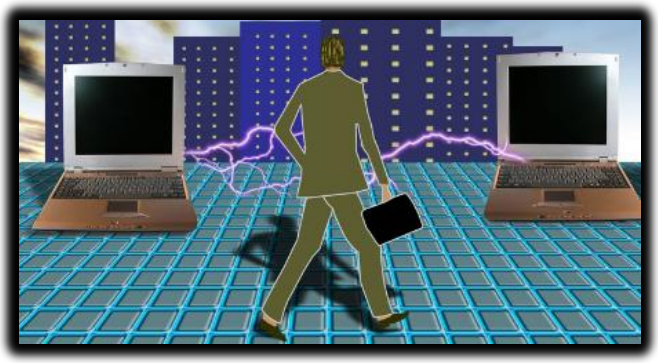
Potential Pitfall

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

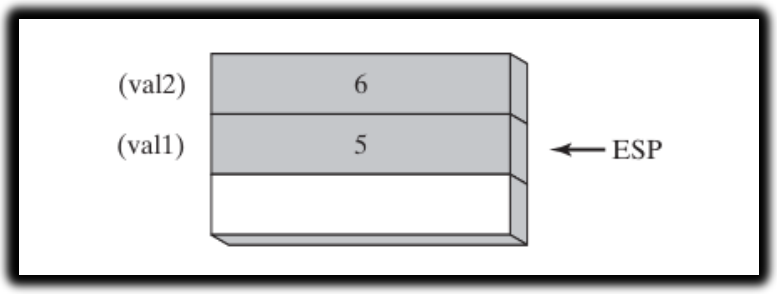
Let's look at a super common way to pass arguments to a subroutine in MASM: passing by value using the stack.

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):

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

OR

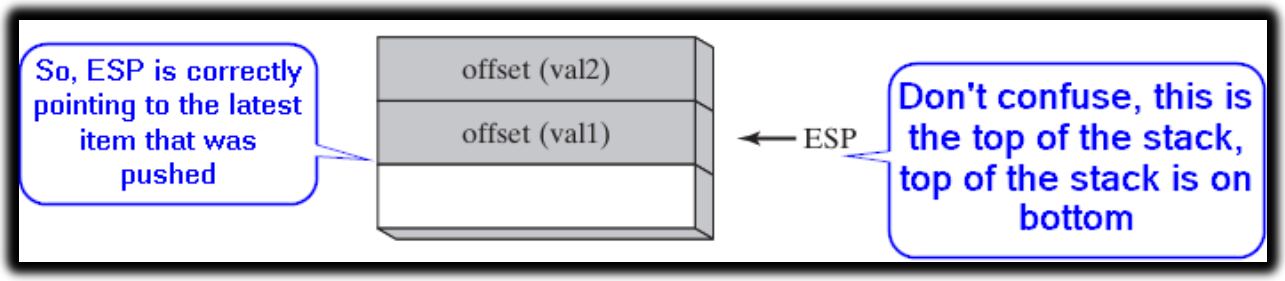
```
; Assume we have two variables we want to swap
.data
    num1    DWORD   42
    num2    DWORD   99

.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!

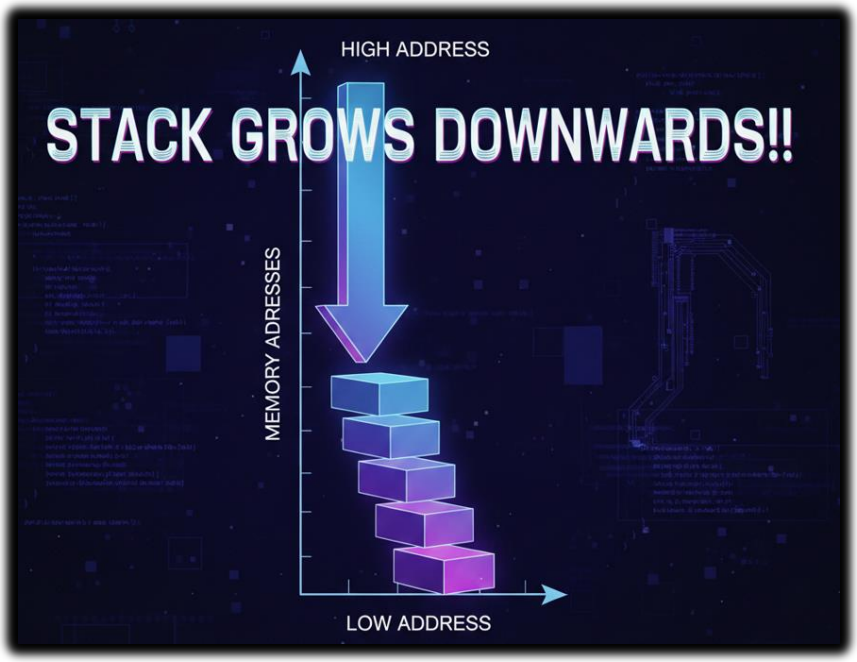
```
void Swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
 }
```

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):

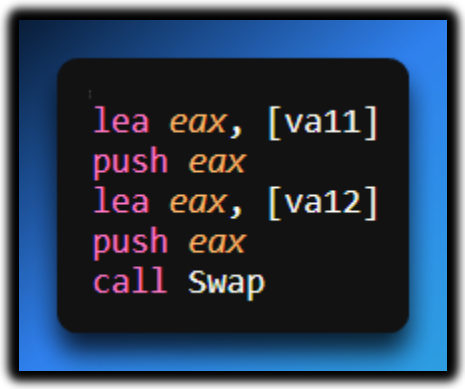
```
int va11 = 42;
int va12 = 99;

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]);
```

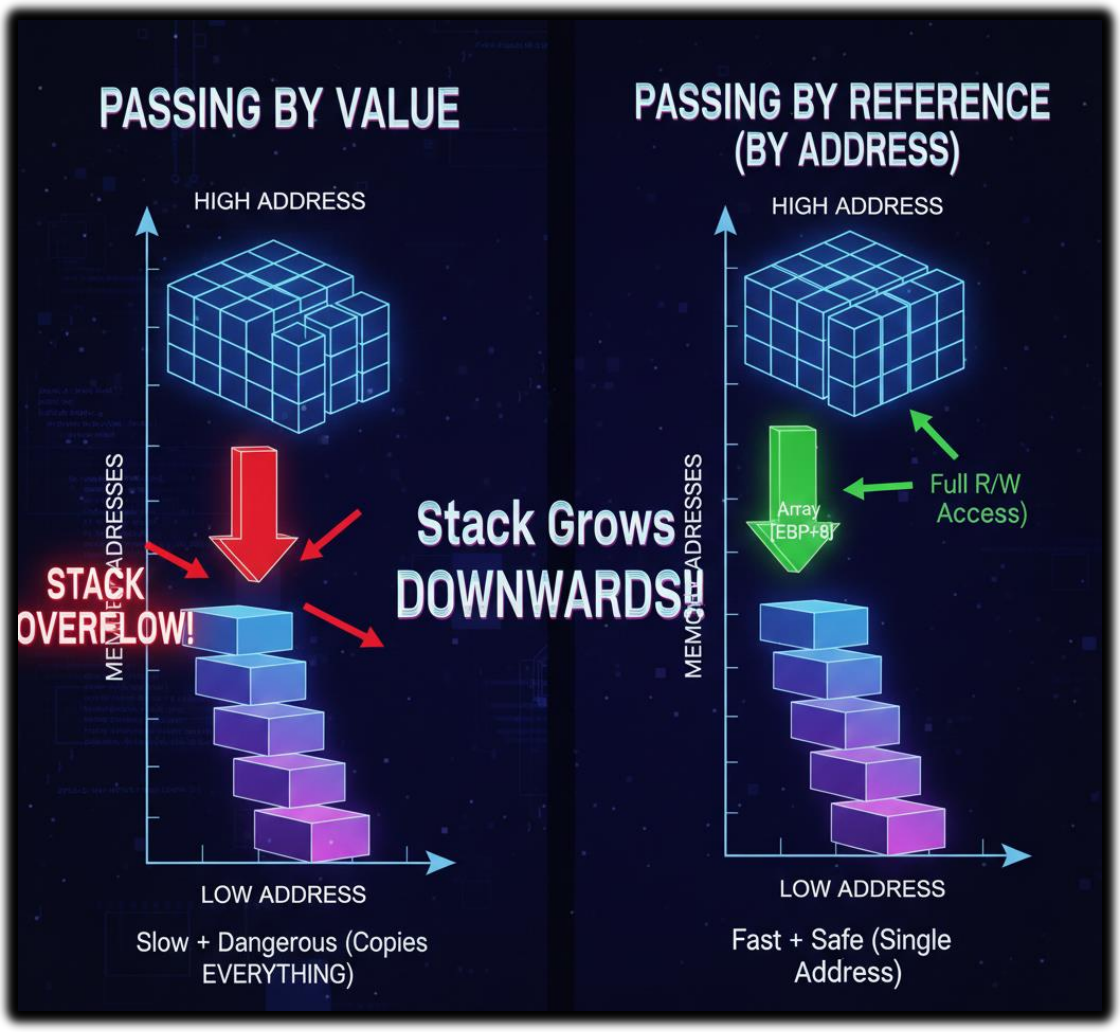
See how natural that is?

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?

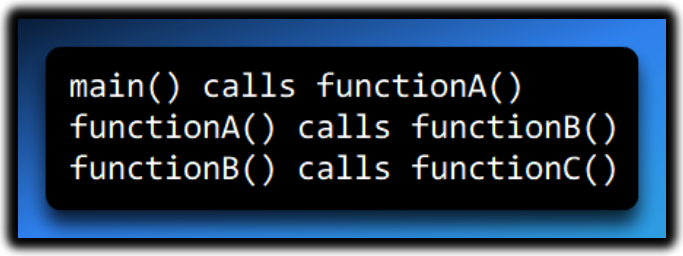
A stack frame is just the chunk of stack memory that belongs to one function call:

- 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.

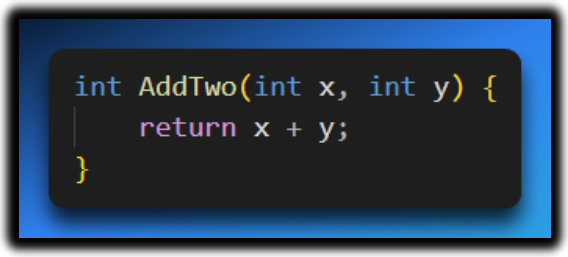
Accessing Stack Parameters 2

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

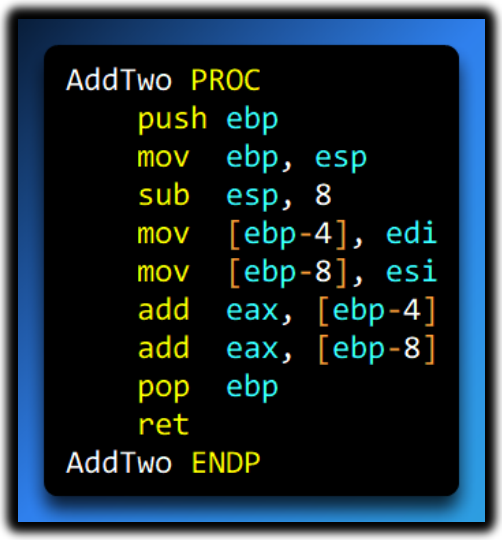


When this function is compiled, the compiler automatically generates two key sections:

- 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.

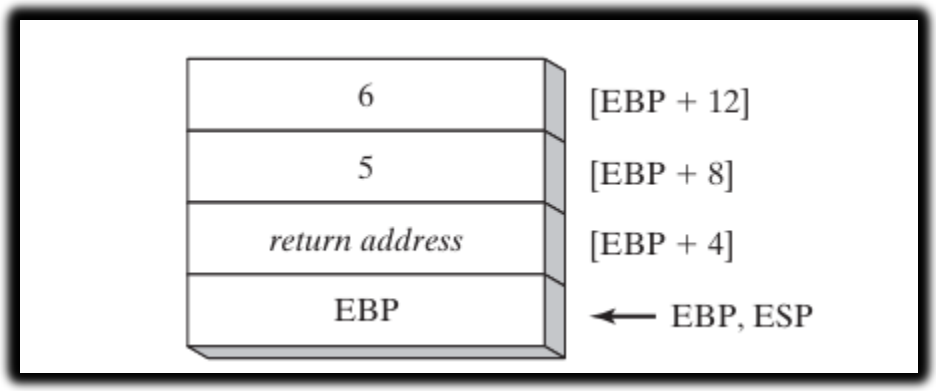
III. How Parameters Are Pushed onto the Stack

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):

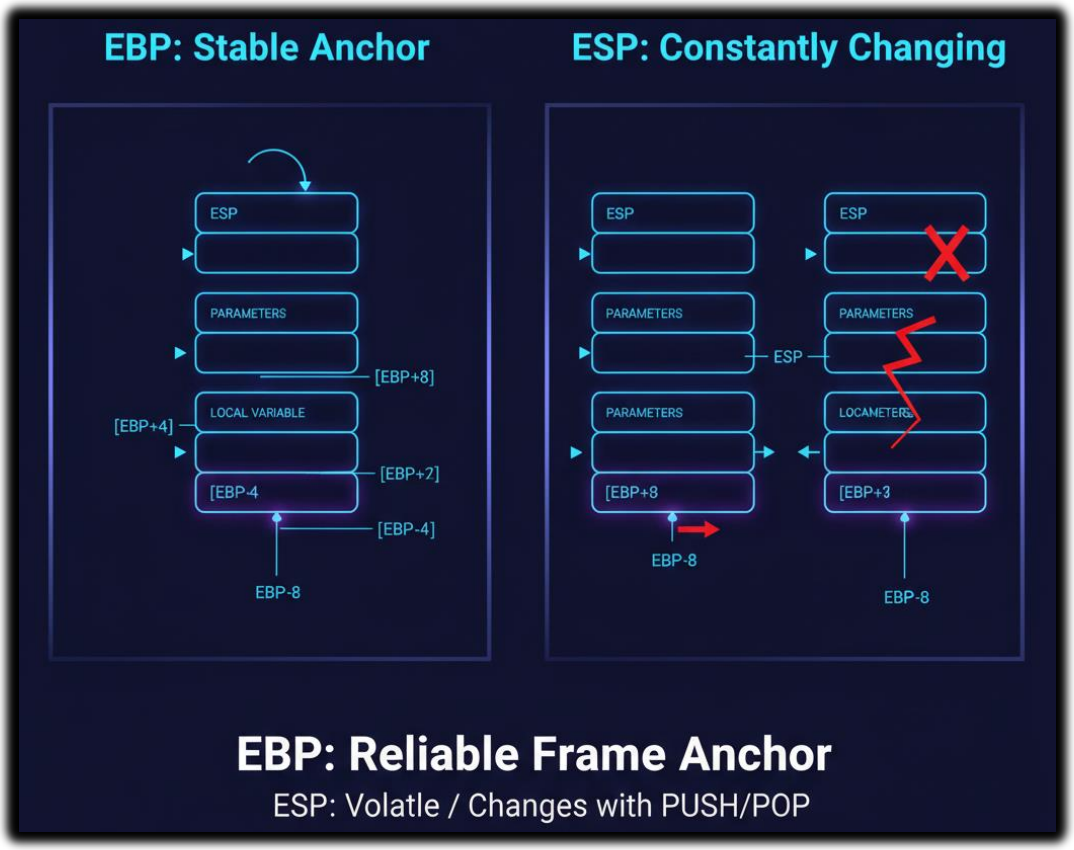


IV. Why EBP Matters (Even When ESP Changes)

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]
```

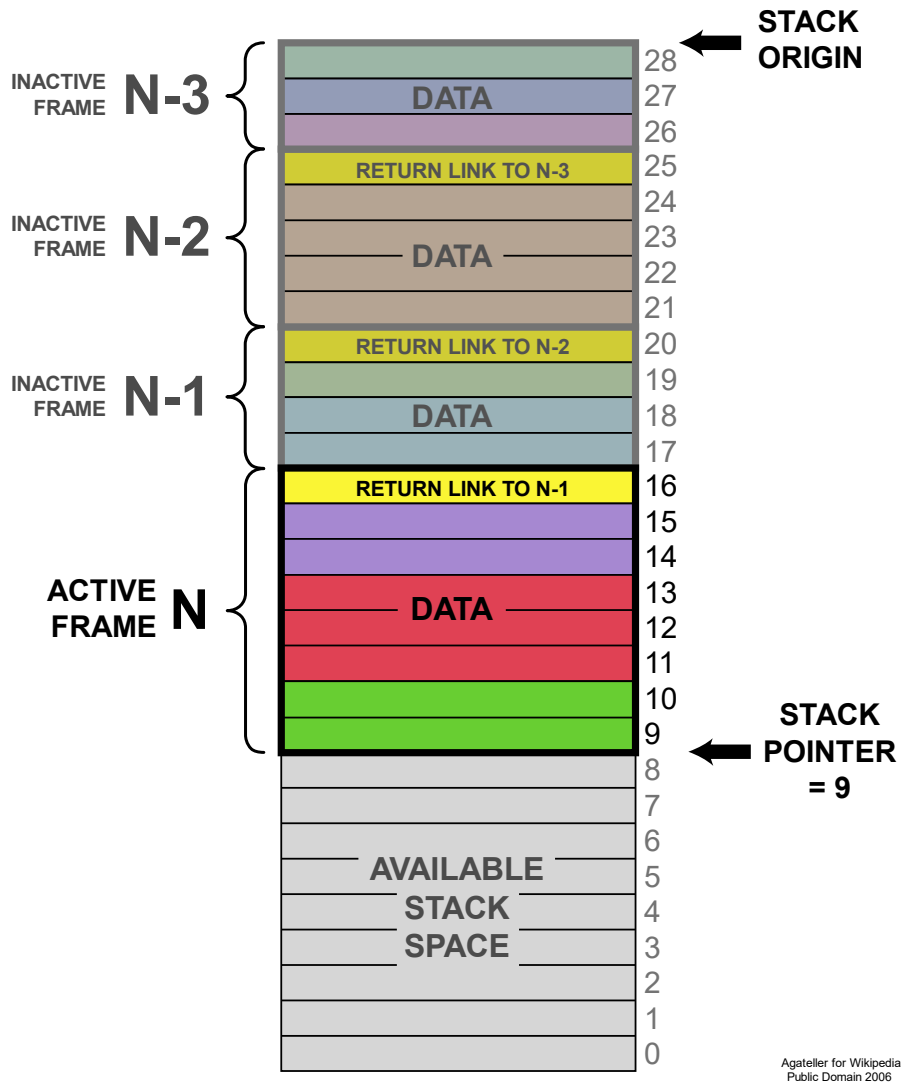
BASE OFFSET ADDRESSING

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.

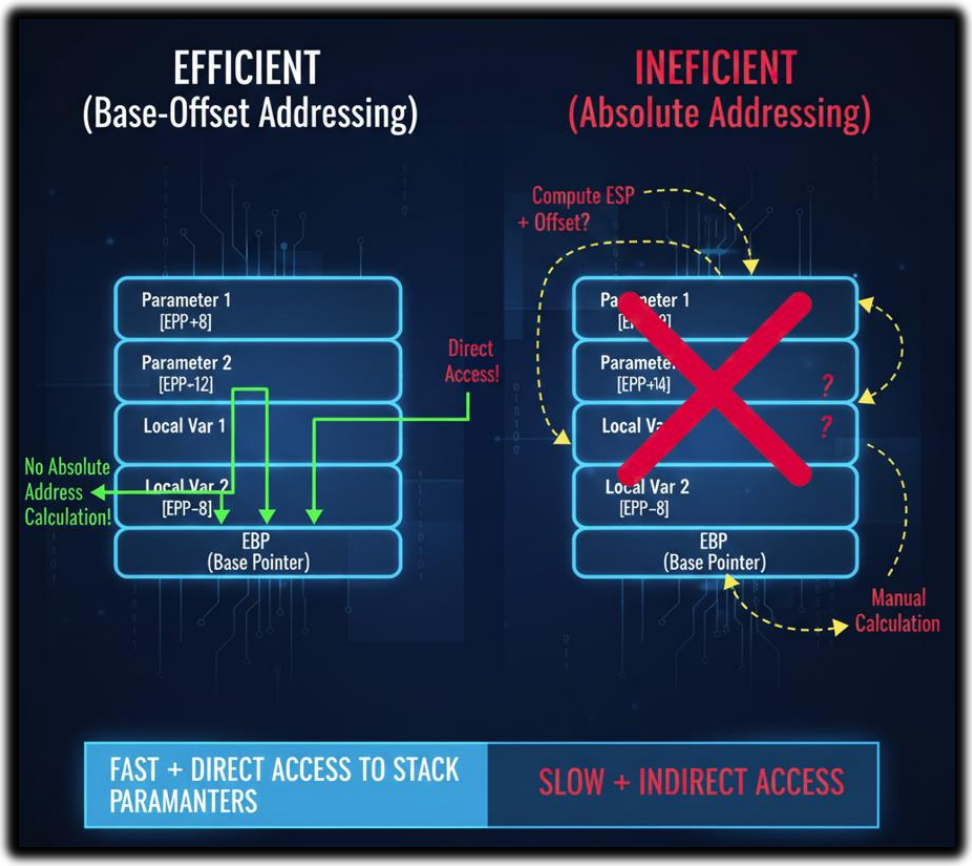
After that, the same process is repeated for the first parameter. The instructions mov eax, 8 and add eax, ebp compute its offset, since the first parameter is located 8 bytes from the base of the stack frame. The instruction mov eax, [eax] then loads the first parameter into EAX.

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:

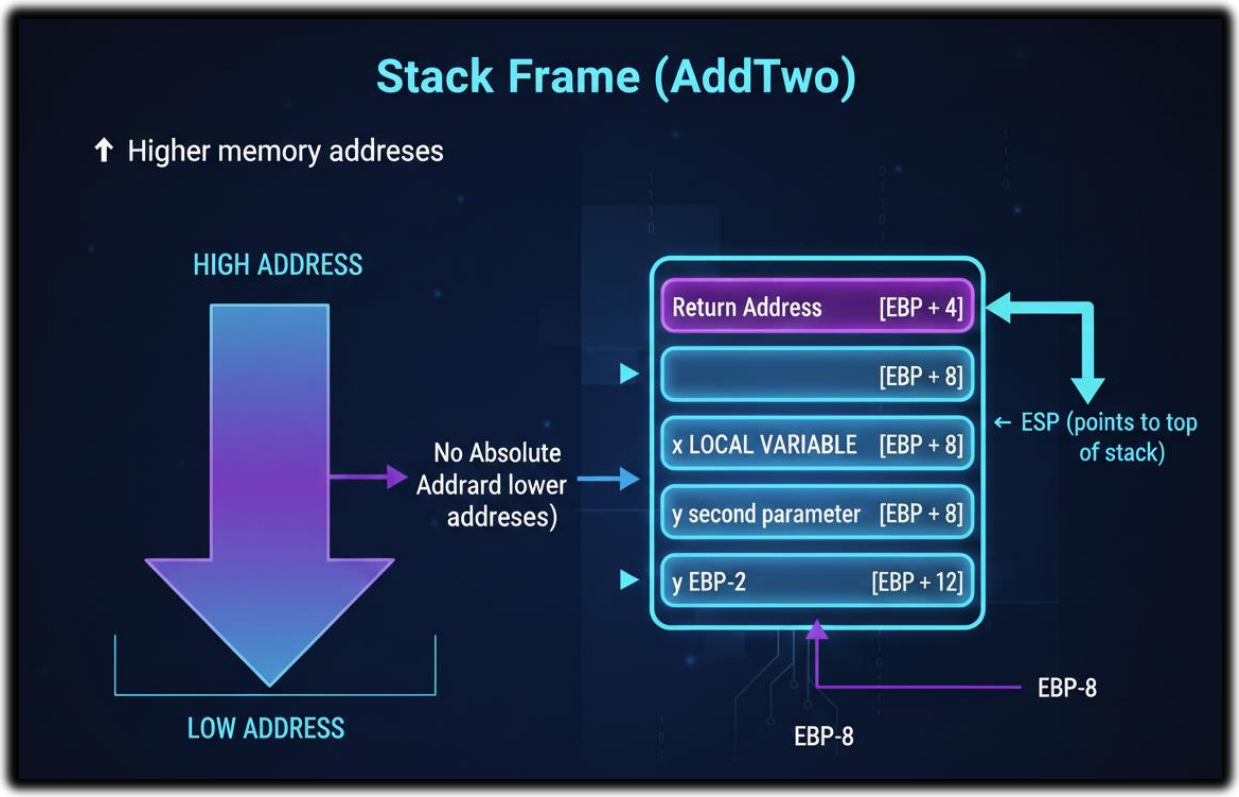
```
227 AddTwo PROC
228   push ebp
229   mov ebp, esp
230   mov eax, [ebp + 12] ; y_param
231   add eax, [ebp + 8] ; x_param
232   pop ebp
233   ret
234 AddTwo ENDP
```

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
```

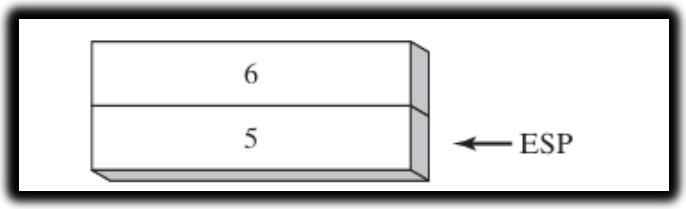
The ret 8 instruction at the end of the subroutine tells the caller to remove 8 bytes from the stack when the subroutine returns.

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

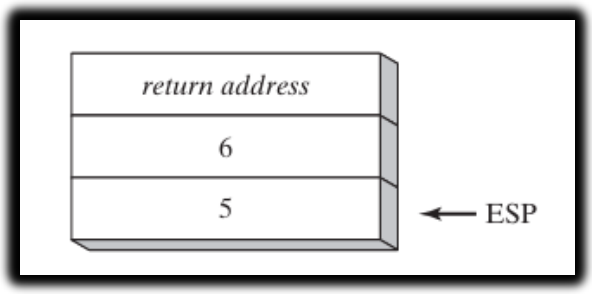
- Every call to AddTwo consumes **12 bytes** on the stack:
  - 4 bytes per parameter (2 parameters = 8 bytes)
  - 4 bytes for the return address
- If AddTwo is called repeatedly in a loop without proper cleanup, the stack keeps growing → **stack overflow risk**.
- The issue gets worse if another subroutine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

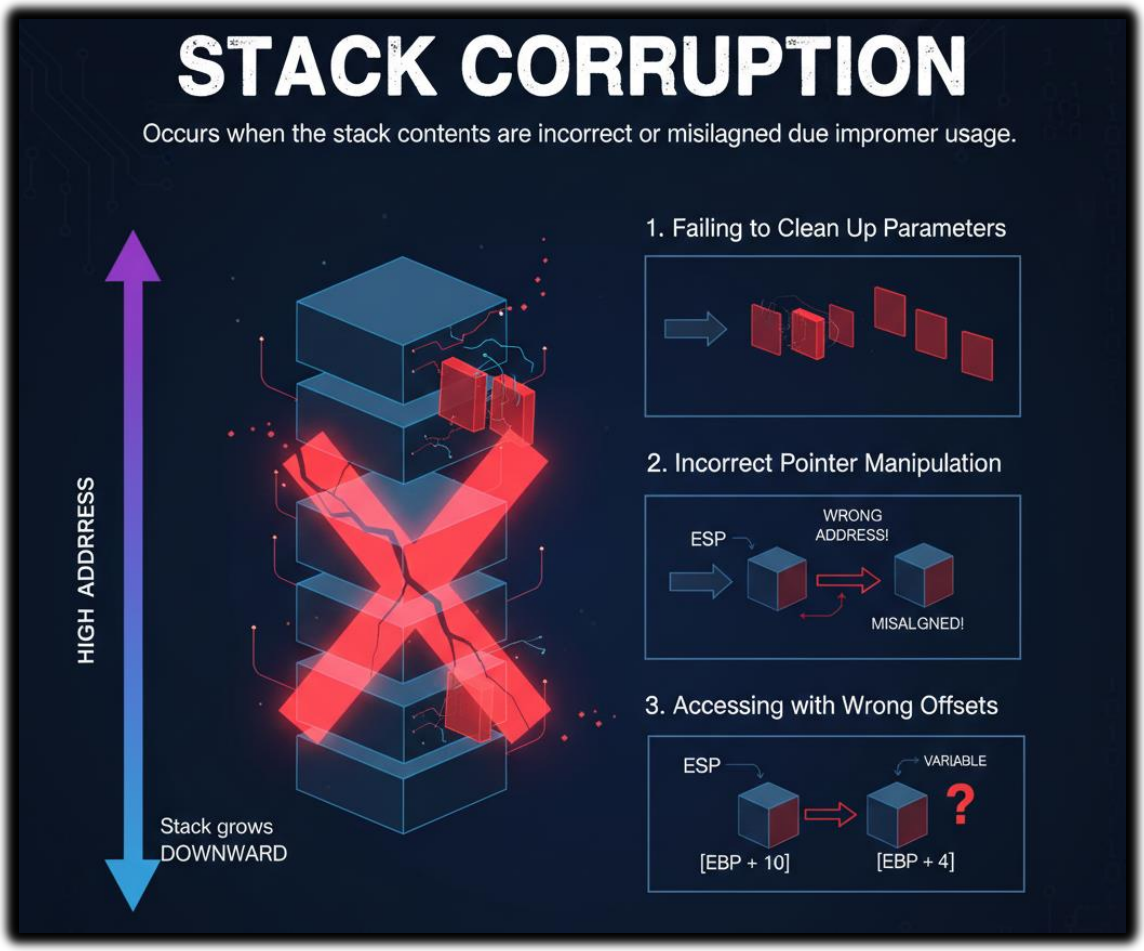
This behavior is a classic example of **stack corruption**.

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

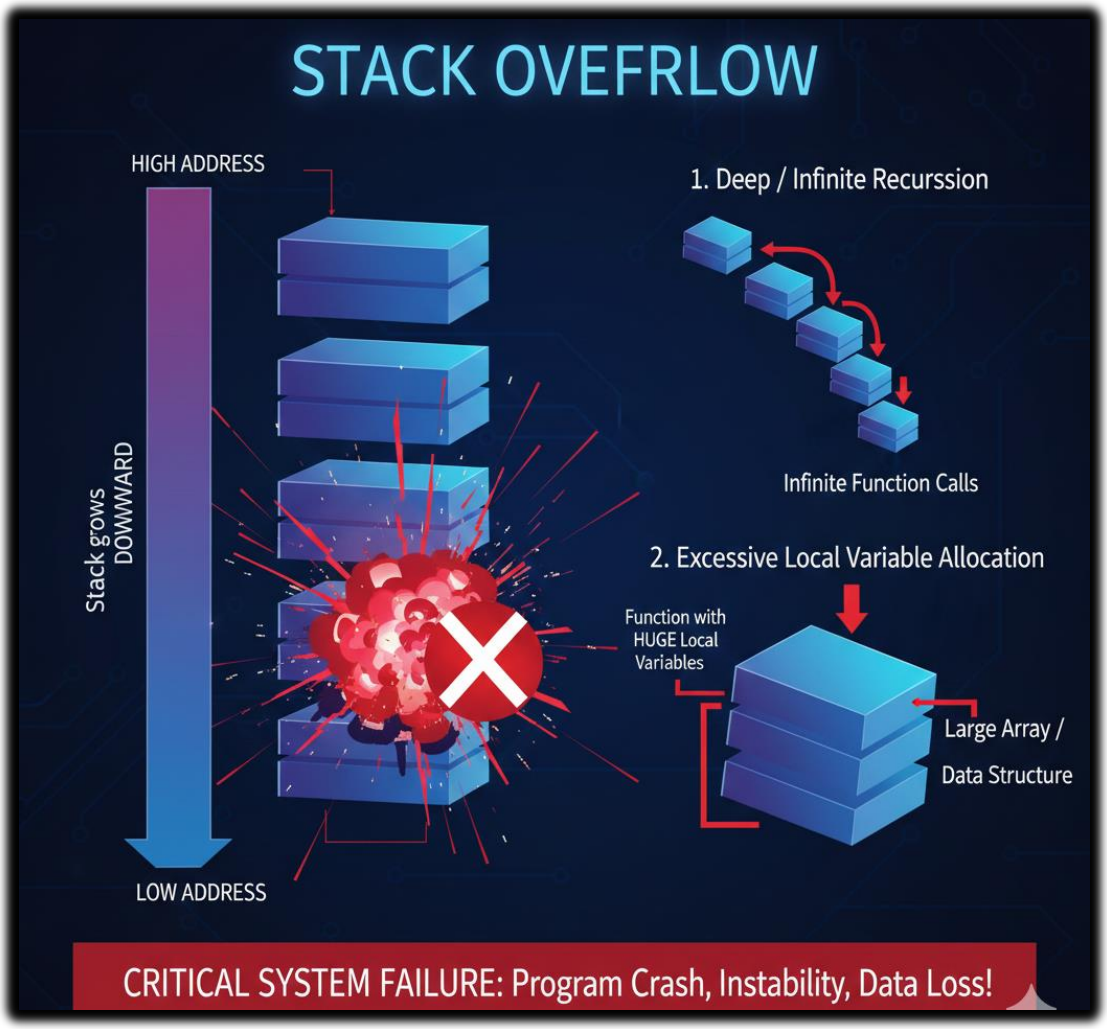


II. Stack overflow

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

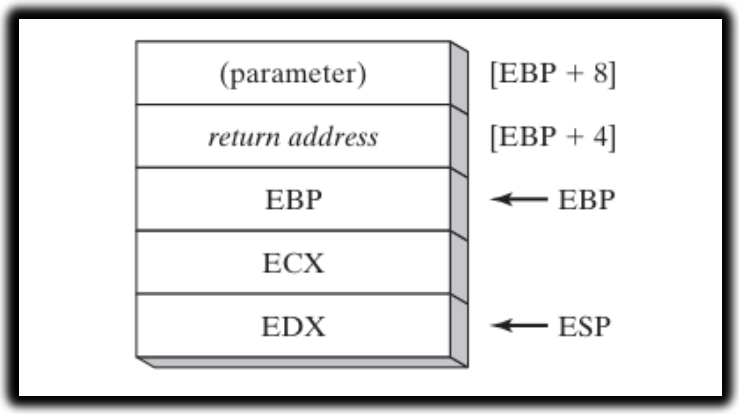
Here is a diagram of a stack frame for the MySub procedure:

```
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

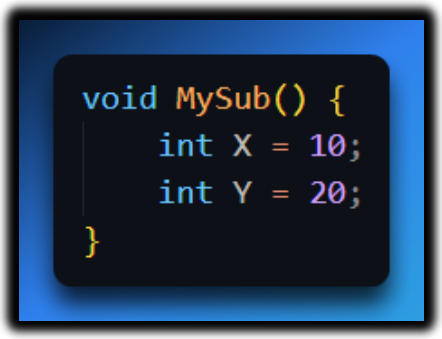
Stack frames:

- 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

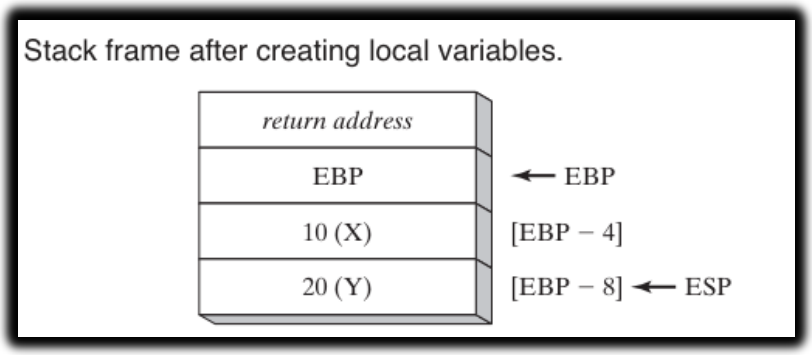
You can access a local variable using its offset from EBP. For example:

```
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

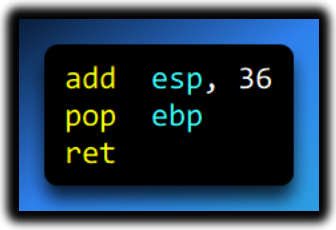
Load Address of Stack Variable

- 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

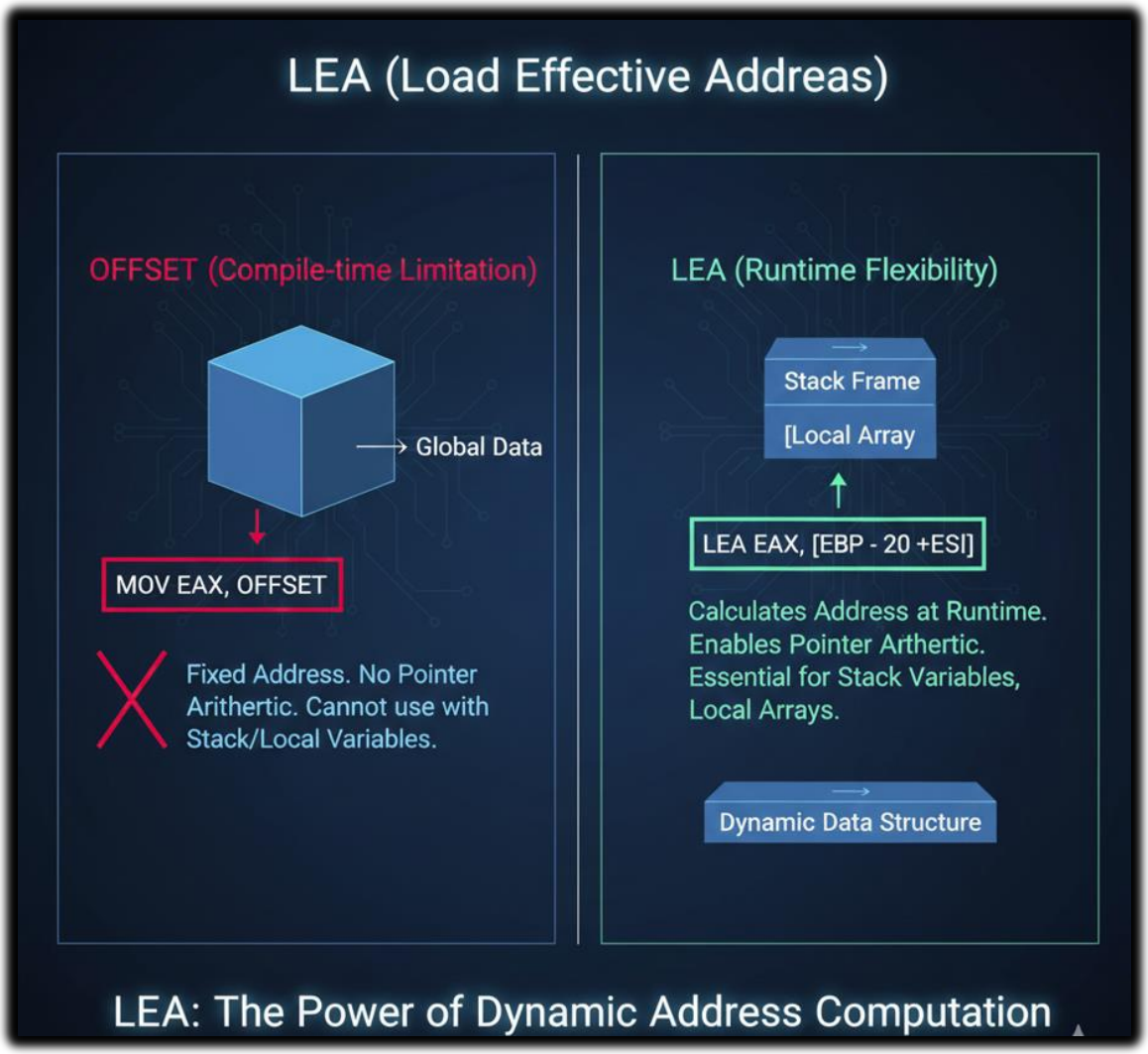


- Deallocates myString
- Restores previous stack frame
- Returns to caller

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

The ENTER and LEAVE instructions are used to manage stack frames in assembly language.

The **ENTER instruction** creates a stack frame for a called procedure, while the **LEAVE instruction** destroys the stack frame for the current procedure.

The ENTER instruction takes two operands: the number of bytes of stack space to reserve for local variables and the lexical nesting level of the procedure.

The lexical nesting level is the number of nested function calls that have occurred to reach the current function. In most cases, the lexical nesting level is zero.

The ENTER instruction performs the following actions:

Pushes the value of the EBP register onto the stack.

Sets the EBP register to the address of the current stack frame.

Reserves the specified number of bytes of stack space for local variables.

The LEAVE instruction performs the following actions:

Pops the value of the EBP register from the stack.

Restores the ESP register to its value before the ENTER instruction was executed.

The following example shows how to use the ENTER and LEAVE instructions to create and destroy a stack frame for a procedure:

```
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
```

It is important to note that the ENTER and LEAVE instructions should be used together. If you use the ENTER instruction to create a stack frame, you must also use the LEAVE instruction to destroy the stack frame. Otherwise, the stack space that you reserved for local variables will not be released.

The image that you provided shows the stack before and after the ENTER instruction has executed. The ENTER instruction has pushed the value of the EBP register onto the stack and set the EBP register to the address of the current stack frame. The ENTER instruction has also reserved 8 bytes of stack space for local variables.

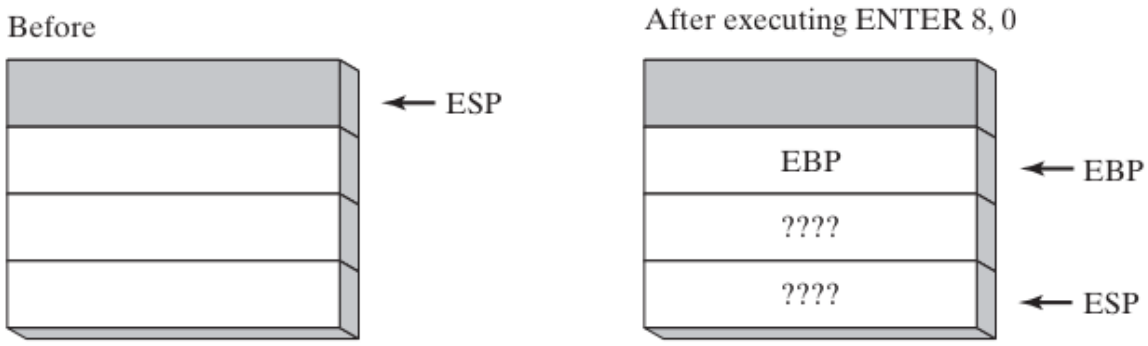
Why is it important to use the ENTER and LEAVE instructions together?

The ENTER and LEAVE instructions should be used together because they work together to manage the stack.

The ENTER instruction creates a stack frame for a called procedure, while the LEAVE instruction destroys the stack frame for the current procedure.

If you use the ENTER instruction to create a stack frame but do not use the LEAVE instruction to destroy the stack frame, **the stack space that you reserved for local variables will not be released.**

This will cause the stack to grow larger and larger, and it could eventually cause the program to crash.



The image above shows the stack before and after the ENTER instruction has executed. The ENTER instruction has pushed the value of the EBP register onto the stack and set the EBP register to the address of the current stack frame. The ENTER instruction has also reserved 8 bytes of stack space for local variables.

LOCAL DIRECTIVE

The LOCAL directive in assembly language is used to declare local variables.

It can be used to declare named local variables of any type, including standard types such as BYTE, DWORD, and PTR WORD, as well as user-defined types such as structures.

The LOCAL directive must be placed on the line immediately following the PROC directive. Its syntax is as follows:

```
LOCAL varlist
```

Where varlist is a list of variable definitions, separated by commas. Each variable definition takes the following form:

label:type where label is the name of the local variable and type is the type of the local variable.

For example, the following assembly language code declares a local variable named var1 of type BYTE:

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

The following assembly language code declares a doubleword local variable named temp of type DWORD and a variable named SwapFlag of type BYTE:

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

The following assembly language code declares a PTR WORD local variable named pArray, which is a pointer to a 16-bit integer:

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

The following assembly language code declares a local variable named TempArray which is an array of 10 doublewords:

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

The LOCAL directive reserves stack space for the local variables that it declares. The amount of stack space reserved depends on the type and size of each local variable.

For example, a BYTE variable requires 1 byte of stack space, while a DWORD variable requires 4 bytes of stack space.

Local variables are accessible within the procedure in which they are declared. They are not accessible to other procedures.

It is important to note that the LOCAL directive is not equivalent to the ENTER instruction. The ENTER instruction creates a stack frame for a called procedure, while the LOCAL directive simply declares local variables.

The ENTER instruction must be used in conjunction with the LEAVE instruction to destroy the stack frame.

The LOCAL directive is a convenient and easy-to-use way to declare local variables in assembly language.

It is a good idea to use the LOCAL directive for all local variables, even if they are only used in a single procedure. This will make your code more readable and maintainable.

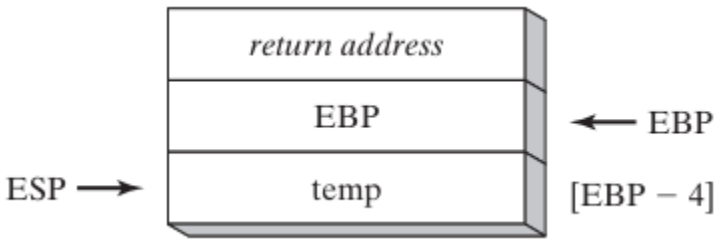
The following is a more in-depth explanation of the MASM code generation for the LOCAL directive:

```
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:



The MASM code generator works as follows:

It pushes the value of the EBP register onto the stack.

This saves the current value of the base pointer register. It sets the EBP register to the address of the current stack frame.

This makes the base pointer register point to the top of the new stack frame.

It subtracts 4 bytes from the ESP register.

This reserves 4 bytes of stack space for the local variable temp.

It loads the value of the local variable temp into the EAX register.

It calls the leave instruction to destroy the stack frame and restore the ESP register to its value before the enter instruction was executed.

It returns from the procedure.

Image explanation:

The image shows the stack before and after the Example1 procedure has executed.

The Example1 procedure has reserved 4 bytes of stack space for the local variable temp. The local variable temp is now located at the address ebp-4.

The esp register points to the top of the stack.

The ebp register points to the base of the stack frame. The base of the stack frame is the address of the first local variable.

The example1 procedure has no parameters.

Therefore, the ebp register points to the same address before and after the procedure has executed.

Microsoft x64 calling convention

The **Microsoft x64 calling convention** is a set of rules that govern how parameters are passed to and from functions in 64-bit Windows programs. It is used by C and C++ compilers, as well as by the Windows API library.

Here is a summary of the key points of the Microsoft x64 calling convention:

The first four parameters to a function are passed in registers: RCX, RDX, R8, and R9. Additional parameters are pushed onto the stack in left-to-right order.

Parameters less than 64 bits long are not zero extended, so the upper bits have indeterminate values. The return value from a function is returned in the RAX register, if it is an integer whose size is less than or equal to 64 bits.

Otherwise, the return value is placed on the stack and RCX points to its location. The caller is responsible for allocating at least 32 bytes of shadow space on the runtime stack, so called functions can optionally save the register parameters in this area.

The stack pointer (RSP) must be aligned on a 16-byte boundary when calling a function.

The caller is responsible for removing all parameters and shadow space from the runtime stack after the called function has finished.

Here are some additional details about the Microsoft x64 calling convention:

The CALL instruction subtracts 8 from the RSP register, since addresses are 64 bits long.

The RAX, RCX, RDX, R8, R9, R10, and R11 registers are often altered by functions, so if the calling program wants them preserved, it must push them onto the stack before the function call and pop them off the stack afterwards.

The values of the RBX, RBP, RDI, RSI, R12, R14, R14, and R15 registers must be preserved by functions.

The Microsoft x64 calling convention is a complex topic, but it is important to understand it if you are writing 64-bit Windows programs.

Questions

1. **(True/False): A subroutine's stack frame always contains the caller's return address and the subroutine's local variables.**

Answer: True

Explanation: A subroutine's stack frame is a region of memory on the stack that is used to store information about the subroutine, such as its local variables and the caller's return address. The caller's return address is the address of the instruction in the calling function that will be executed after the subroutine returns.

1. **(True/False): Arrays are passed by reference to avoid copying them onto the stack.**

Answer: True

Explanation: Arrays are typically passed by reference to functions to avoid copying them onto the stack. This is because arrays can be very large, and copying them onto the stack would be inefficient.

1. **(True/False): A subroutine's prologue code always pushes EBP on the stack.**

Answer: True

Explanation: The prologue code for a subroutine is the code that is executed at the beginning of the subroutine. The prologue code typically saves the value of the EBP register on the stack. The EBP register is used to point to the base of the current stack frame.

1. **(True/False): Local variables are created by adding a positive value to the stack pointer.**

Answer: True

Explanation: Local variables are created by adding a positive value to the stack pointer. This value is the size of the local variable in bytes.

1. **(True/False): In 32-bit mode, the last argument to be pushed on the stack in a subroutine call is stored at location EBP + 8.**

Answer: False

Explanation: In 32-bit mode, the last argument to be pushed on the stack in a subroutine call is stored at location EBP + 4.

1. **(True/False): Passing by reference means that an argument's address is stored on the runtime stack.**

Answer: True

Explanation: Passing by reference means that an argument's address is stored on the runtime stack. This means that the function can directly access the argument in the calling function.

1. **What are the two common types of stack parameters?**

Answer: The two common types of stack parameters are:

Value parameters: Value parameters are copied onto the stack when the function is called. When the function returns, the changes made to the parameter on the stack are not reflected in the calling function.

Reference parameters: Reference parameters are passed by address. When a function is called with a reference parameter, the stack contains the address of the parameter in the calling function. Changes made to the parameter in the called function are also reflected in the calling function. Example:

```
void swap_values(int a, int b) {
  // a and b are value parameters

  int temp = a;
  a = b;
  b = temp;
}

void swap_references(int* a, int* b) {
  // a and b are reference parameters

  int temp = *a;
  *a = *b;
  *b = temp;
}

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

```
  // Call swap_values() with the value parameters x and y
  swap_values(x, y);

  // x and y will still be 10 and 20, respectively

  // Call swap_references() with the reference parameters &x and &y
  swap_references(&x, &y);

  // x and y will now be 20 and 10, respectively

  return 0;
}
```

Explanation:

In the above example, the swap_values() function takes two value parameters, a and b. When the function is called, the values of a and b are copied onto the stack. The function then swaps the values of the two parameters on the stack. When the function returns, the values of a and b in the calling function are not affected.

The swap_references() function takes two reference parameters, a and b. When the function is called, the stack contains the addresses of the a and b parameters in the calling function. The function then swaps the values of the two parameters in the calling function. When the function returns, the values of a and b in the calling function are affected.

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 you provided is a good illustration of 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 will continue to grow until it overflows, causing the program to crash.

To rewrite the Endless procedure correctly, we need to add a base case. This is a condition that will cause the procedure to terminate instead of calling itself again.

In the case of the Endless procedure, the base case could be something like "if the input is 0, then return".

Here is a rewritten version of the Endless procedure that includes a base case:

```
519 ; Endless Recursion (Endless.asm)
520 INCLUDE Irvine32.inc
521 .data
522     endlessStr BYTE "This recursion never stops",0
523 .code
524     main PROC
525         call
526         Endless
527         exit
528     main ENDP
529     Endless PROC
530         mov ecx, 1 ; input parameter
531         ; base case
532         cmp ecx, 0
533         je endless_exit
534 
535         ; recursive call
536         call Endless
537 
538         ; decrement input parameter
539         dec ecx
540         ; and call again
541         jmp Endless
542 
543         endless_exit:
544         ret
545     Endless ENDP
546 END main
```

This rewritten version of the Endless procedure will now terminate correctly when the input is 0. It will also print the message "This recursion never stops" to the console before it terminates.

When to Use Recursion

Recursion is not a good choice for all problems. It can be inefficient and difficult to debug. However, it can be a powerful tool for solving problems that have repeating patterns. For example, recursion is often used to implement algorithms for traversing linked lists and trees.

If you are considering using recursion in your program, it is important to make sure that the problem you are trying to solve is a good fit for recursion. You should also carefully design your recursive function to avoid stack overflows.

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. This can be useful for solving problems that can be broken down into smaller subproblems of the same type.

To calculate the sum of the integers from 1 to n, we can use the following recursive procedure:

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

This procedure works by recursively calling itself to calculate the sum of the integers from 1 to n - 1, and then adding n to the result. The base case is when n == 0, in which case the sum is simply 0.

The following table shows a stack trace for the recursive call of 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

Explanation of the Table:

The stack frame for each recursive call is pushed onto the stack when the CALL instruction is executed. The stack frame contains the return address, which is the address of the next instruction to be executed after the recursive call returns.

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

At the first recursive call to CalcSum(5), the counter value is 4 and the sum is 0. The program calculates the sum of the integers from 1 to 4 by recursively calling CalcSum(4).

At the second recursive call to CalcSum(4), the counter value is 3 and the sum is 0. The program calculates the sum of the integers from 1 to 3 by recursively calling CalcSum(3).

This process continues until the base case is reached, when n == 0. At this point, the program returns 0 from the recursive call. The program then returns from the recursive call to CalcSum(3), and so on.

By the time the program returns from the recursive call to CalcSum(5), the sum of the integers from 1 to 5 has been calculated and stored in the EAX register. The program can then return the sum from the main() function.

```
;Sum of Integers (RecursiveSum.asm)
INCLUDE Irvine32.inc
.code
main PROC
    mov ecx, 5   ; Set ECX to 5, the number of integers to sum.
    mov eax, 0   ; Initialize EAX to 0; it will hold the sum.
    call CalcSum ; Call the CalcSum function to calculate the sum.
L1:
    call WriteDec ; Display the result in EAX.
    call Crlf     ; Print a new line.
    exit
main ENDP
;-----------------------------------------------------
CalcSum PROC
    ; Calculates the sum of a list of integers
    ; Receives: ECX = count
    ; Returns: EAX = sum
    ;-----------------------------------------------------
    cmp ecx, 0    ; Compare ECX (counter) with 0.
    jz L2         ; If it's zero, jump to L2 and quit.
    add eax, ecx  ; Add ECX to EAX, updating the sum.
    dec ecx       ; Decrement the counter.
    call CalcSum  ; Recursively call CalcSum to process the next integer.

L2:
    ret
CalcSum ENDP
end main
```

This code first sets up the main procedure, where it initializes ecx to 5 (the number of integers to sum) and eax to 0 (to store the sum). It then calls the CalcSum procedure to calculate the sum. Afterward, it prints the result using WriteDec and adds a new line with Crlf.

The CalcSum procedure is a recursive function that calculates the sum of integers. It checks if ecx (the counter) is zero; if not, it adds the current value of ecx to the sum in eax, decrements ecx, and then makes a recursive call to CalcSum. This process continues until ecx reaches 0, at which point the function returns (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 number to calculate. The calling program's return address is automatically pushed on the stack by the CALL instruction.

The first thing Factorial does is to push EBP on the stack, to save the base pointer to the calling program's stack. It then sets EBP to the beginning of the current stack frame. This

allows the procedure to access its parameters and local variables using base-offset addressing.

Next, Factorial checks the base case, which is when N equals zero. In this case, Factorial returns 1, which is the factorial of 0.

If N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter. This process continues until the base case is reached.

When Factorial returns from a recursive call, it multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of 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

The stack frame for each recursive call is pushed onto the stack when the CALL instruction is executed. The stack frame contains the return address, which is the address of the next instruction to be executed after the recursive call returns.

The EBP register contains the base pointer to the current stack frame. The ESP register contains the stack pointer, which points to the top of the stack.

The N register contains the value of the parameter passed to Factorial.

At the first recursive call to Factorial(3), the EBP register is set to the beginning of the current stack frame. The N register is loaded with the value 3, which is the parameter passed to Factorial.

Factorial checks the base case, which is when N equals zero. Since N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter.

At the second recursive call to Factorial(2), the EBP register is set to the beginning of the new stack frame. The N register is loaded with the value 2, which is the parameter passed to Factorial.

Factorial checks the base case, which is when N equals zero. Since N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter.

This process continues until the base case is reached, when N equals zero. At this point, Factorial returns 1, which is the factorial of 0.

The program then returns from the recursive call to Factorial(2). The N register is loaded with the value 2, which is the result of the recursive call.

Factorial multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

The program then returns from the recursive call to Factorial(3). The N register is loaded with the value 6, which is the result of the recursive call.

Factorial multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

The program then returns to the main() function. The EAX register contains the value 6, which is the factorial of 3.

Let's break down the provided assembly code for calculating the factorial of an integer, explained above, step by step, and I'll explain the key parts in detail.

```
605 INCLUDE Irvine32.inc           ;Calculating a Factorial (Fact.asm)
606 .code
607 main PROC
608     push 5                     ;Push the initial value (e.g., 5) on the stack.
609     call Factorial             ;Call the Factorial procedure to calculate the factorial.
610     call WriteDec              ;Display the result (EAX) on the console.
611     call Crlf                  ;Print a new line.
612     exit
613 main ENDP
614 ;-----------------------------------------------
615 Factorial PROC
616     ; Calculates a factorial.
617     ; Receives: [ebp+8] = n, the number to calculate
618     ; Returns: eax = the factorial of n
619     ;-----------------------------------------------
620     push ebp            ; Save the current base pointer.
621     mov ebp, esp        ; Set up a new base pointer for the current stack frame.
622     mov eax, [ebp+8]    ; Get the value of n from the stack.
623     cmp eax, 0          ; Check if n is zero.
624     ja L1               ; If n is greater than zero, continue; otherwise, go to L2.
625     mov eax, 1          ; If n is zero, return 1 as the value of 0!
626     jmp L2              ; Jump to the point where we clean up the stack and return.
627 L1:
628     dec eax             ; Decrement n.
629     push eax            ; Push the decremented value onto the stack.
630     call Factorial      ; Recursively call the Factorial procedure with n-1.
631 L2:
632     pop ebp             ; Clean up the stack by restoring the previous base pointer.
633     ret                 ; Return with the result (EAX).
634 Factorial ENDP
635 END main
```

Here's an in-depth explanation:

The main procedure begins by pushing the initial value (5 in this case) onto the stack and then calls the Factorial procedure to calculate the factorial.

The Factorial procedure is a recursive function for calculating the factorial of an integer. It first saves the current base pointer on the stack and sets up a new base pointer for the current stack frame.

It retrieves the value of n from the stack (passed as a parameter) into the eax register.

It compares n to 0 using the cmp instruction. If n is greater than 0 (ja - jump above), it proceeds to L1; otherwise, it jumps to L2.

In L1, it decrements n and pushes the new value onto the stack. Then, it makes a recursive call to the Factorial procedure with n-1.

In L2, it pops the base pointer from the stack to clean up the stack frame and returns with the result in EAX.

This recursive approach continues to reduce n until it reaches 0, accumulating the product of each multiplication in EAX.

The result is then returned and displayed in the main procedure.

The program calculates factorials using recursion, and the result for the provided input of 5 would be 120.

Tip:

It is important to keep track of which registers are modified when making recursive calls to a procedure, so that you can save and restore their values if necessary. This is especially important if the register values are needed across recursive procedure calls.

---

1. 1. **1. (True/False): Given the same task to accomplish, a recursive subroutine usually uses more memory than a nonrecursive one.** False: Recursive subroutines typically use more memory than nonrecursive subroutines, because they require additional stack space to store the return addresses of the recursive calls.

1. 2. **2. In the Factorial function, what condition terminates the recursion?** The recursion terminates when the input parameter, n, is equal to 0.

1. **Which instructions in the assembly language Factorial procedure execute after each recursive call has finished?** The following instructions in the assembly language Factorial procedure execute after each recursive call has finished:

```
669 mov ebx, [ebp+8]
670 mul ebx
```

These instructions multiply the result of the recursive call by n. This is necessary because the factorial of n is equal to n multiplied by the factorial of n - 1.

What would happen to the Factorial program's output if you tried to calculate 13!? The Factorial program would fail to calculate 13! because the factorial of 13 is too large to be represented in a 32-bit integer.

Challenge: How many bytes of stack space would be used by the Factorial procedure when calculating 5!? The Factorial procedure would use 20 bytes of stack space when calculating 5!. This is because the stack frame for each recursive call requires 4 bytes for the return address and 16 bytes for the local variables.

Here is a breakdown of the stack space requirements:

```
676 Return address: 4 bytes
677 Local variables: 16 bytes
678 Total: 20 bytes
```

The Factorial procedure makes 5 recursive calls when calculating 5!, so the total stack space requirement is 20 bytes per recursive call * 5 recursive calls = 100 bytes.

INVOKE, ADDR, PROC AND PROTO

The INVOKE, PROC, and PROTO directives provide powerful tools for defining and calling procedures in 32-bit mode.

They are more convenient to use than the traditional CALL and PROC directives, but they mask the underlying structure of the runtime stack.

In such cases, the **PROTO directive** helps the assembler to validate procedure calls by checking argument lists against procedure declarations. This can help to prevent errors and make programs more robust.

Advanced procedure directives are more convenient to use than traditional CALL and PROC directives, but they mask the underlying structure of the runtime stack.

It is important to develop a detailed understanding of the low-level mechanics involved in subroutine calls before using advanced procedure directives.

Advanced procedure directives can be used to improve program readability and maintainability, especially when programs execute procedure calls across module boundaries.

The PROTO directive helps the assembler to validate procedure calls by checking argument lists against procedure declarations. Recommendation:

If you are new to assembly language, it is recommended that you start by learning the traditional CALL and PROC directives.

Once you have a good understanding of how subroutine calls work, you can then consider using advanced procedure directives to improve your code.

==========

INVOKE Directive

==========

The INVOKE directive is a powerful tool for calling procedures in 32-bit mode. It allows you to pass multiple arguments to a procedure using a single line of code.

The general syntax of the INVOKE directive is as follows:

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

procedureName is the name of the procedure to be called.

argumentList is an optional comma-delimited list of arguments passed to the procedure.

Arguments to INVOKE

Arguments to INVOKE can be any valid expression, including:

- **Immediate values (e.g., 10, 3000h, OFFSET myList).**
- **Integer expressions (e.g., (1020), COUNT).**
- **Variables (e.g., myList, array, myWord, myDword).**
- **Address expressions (e.g., [myList + 2], [ebx+ esi]).**
- **Registers (e.g., eax, bl, edi).**

Arguments to INVOKE are pushed onto the stack in the reverse order that they are specified in the INVOKE statement.

The following example shows how to use the INVOKE directive to call a procedure named DumpArray():

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

This statement will push the following values onto the stack:

The address of the array

The length of the array

The size of the array elements

The DumpArray() procedure will then be called with these arguments.

This statement is equivalent to the following code using the CALL instruction:

```
692 push TYPE array
693 push LENGTHOF array
694 push OFFSET array
695 call DumpArray
```

The INVOKE directive can handle almost any number of arguments, and individual arguments can appear on separate source code lines. This can be useful for documenting complex INVOKE statements or for breaking up long argument lists.

The following example shows an INVOKE statement with arguments on separate source code lines:

```
700 INVOKE DumpArray,
701 ; displays an array
702 ; points to the array
703 OFFSET array,
704 ; the array length
705 LENGTHOF array,
706 ; array component size
707 TYPE array
```

Which form you choose is a matter of personal preference. Some programmers prefer to document their code extensively, while others prefer to keep their code as concise as possible.

Important Considerations

When passing arguments to INVOKE that are smaller than 32 bits, the assembler may overwrite the EAX and EDX registers when it widens the arguments before pushing them onto the stack.

To avoid this behavior, you can either pass 32-bit arguments to INVOKE or save and restore the EAX and EDX registers before and after the procedure call.

The INVOKE directive is only available in 32-bit mode.

The INVOKE directive is a powerful tool for calling procedures in 32-bit mode.

It allows you to pass multiple arguments to a procedure using a single line of code.

However, it is important to be aware of the potential for overwriting the EAX and EDX registers when passing small arguments to INVOKE.

==========

ADDR Operator

==========

The ADDR operator is a powerful tool for passing pointer arguments to procedures using INVOKE. It is only available in 32-bit mode.

The ADDR operator takes a single operand, which must be an assembly time constant. This means that the operand must be known at compile time, and cannot be a variable or expression that is evaluated at runtime.

The ADDR operator returns the address of the operand. This address can then be passed to a procedure using INVOKE.

The following example shows how to use the ADDR operator to pass the address of an array to a procedure named FillArray():

```
INVOKE FillArray, ADDR myArray
```

This statement is equivalent to the following code:

```
715 mov esi, myArray
716 INVOKE FillArray, esi
```

However, the first form is more concise and readable.

The ADDR operator can only be used with the INVOKE directive.

It is not valid to use the ADDR operator with other instructions, such as MOV or CALL.

The ADDR operator can only be used to pass the address of an assembly time constant.

It is not valid to pass the address of a variable or expression that is evaluated at runtime.

The following code shows how to use the ADDR operator to call a procedure named Swap() and pass it the addresses of the first two elements in an array of doublewords:

```
722 .data
723     Array DWORD 20 DUP(?)
724 .code
725     ...
726     INVOKE Swap,
727     ADDR Array,
728     ADDR [Array+4]
```

The assembler will generate the following code:

```
734 push
735 OFFSET Array+4
736 push
737 OFFSET Array
738 call
739 Swap
```

The ADDR operator is a powerful tool for passing pointer arguments to procedures using INVOKE. It allows you to write more concise and readable code.

The ADDR operator can also be used to pass the address of a function to another function. This can be useful for implementing callback functions.

For example, the following code shows how to define a function named PrintArray() that prints the elements of an array to the console:

```
.code
PrintArray PROC Near
...
ENDP
```

The following code shows how to pass the address of the PrintArray() function to a function named DoSomething():

```
.code
    DoSomething PROC Near
    INVOKE PrintArray, ADDR PrintArray
    ENDP
```

When the DoSomething() function is called, it will call the PrintArray() function to print the elements of an array to the console.

==========

PROC Operator

==========

The PROC directive is used to define a procedure in 32-bit mode. It has the following syntax:

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

- • • **label** is a user-defined label that follows the rules for identifiers.
- • • **attributes** is a list of optional attributes that can be used to control the behavior of the procedure. These attributes are:
- • • **distance:** Specifies whether the procedure is near or far.
- • • **langtype:** Specifies the calling convention (parameter passing convention) to use for the procedure.
- • • **visibility:** Specifies the visibility of the procedure to other modules.
- • • **prologuearg:** Specifies arguments affecting generation of prologue and epilogue code.
- • • **parameter_list** is a list of optional parameters that can be passed to the procedure.

Parameters

Parameters can be of any type, including bytes, words, doublewords, floating-point numbers, and pointers. To declare a parameter, you use the following syntax:

```
paramName:type
```

- **paramName** is the name of the parameter.
- **type** is the type of the parameter.

For example, the following procedure declares two parameters, val1 and val2, both of which are doublewords:

```
767 AddTwo PROC,
768 val1:DWORD,
769 val2:DWORD
```

USES

The USES clause is an optional clause that can be used to specify which registers the procedure will need to use. This can be useful for optimizing the procedure's code.

For example, the following procedure declares that it will need to use the EAX and EBX registers:

```
Read_File PROC USES eax ebx,
```

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
```

The first two lines of the generated code push the EBP register onto the stack and move the stack pointer to EBP. This is done to create a stack frame for the procedure.

The next two lines move the parameters from the stack to the EAX and EDX registers.

The next line adds the two parameters together in the EAX register.

The next two lines restore the EBP register and return from the procedure.

The constant at the end of the procedure is the size of the procedure's stack frame. This value is used by the RET instruction to pop the correct number of bytes off the stack when returning from the procedure.

The PROC directive is a powerful tool for defining procedures in 32-bit mode. It allows you to create procedures with named parameters and to control the behavior of the procedure's stack frame.

Here's a detailed explanation of the code:

push ebp: This instruction saves the current base pointer (BP) by pushing it onto the stack. This is a common practice to establish a proper stack frame for the procedure.

mov ebp, esp: The ebp register is set to the current value of esp, establishing a new stack frame for this procedure. This step aligns the base pointer with the current top of the stack (ESP) and makes it easier to access function parameters and local variables.

mov eax, dword ptr [ebp+8]: This line loads the first parameter (at offset +8 from the base pointer) from the stack into the EAX register. The [ebp+8] notation indicates that the first parameter is located 8 bytes above the base pointer.

add eax, dword ptr [ebp+0Ch]: Here, the code loads the second parameter (at offset +12 from the base pointer) from the stack and adds it to the value in EAX.

leave: This instruction is often used to clean up the stack frame. It's the opposite of the enter instruction. It effectively performs the following operations:

Restores the previous value of the base pointer (EBP) from the stack. Adjusts the stack pointer (ESP) to remove the local variables and parameters of the current function.

Essentially, it unwinds the stack frame to the previous state. ret: This instruction returns from the procedure, and the value in EAX becomes the return value of the function.

The **8 at the end i**s likely a comment indicating that the parameters take up 8 bytes in total (4 bytes each), which is common for two 32-bit integers.

In summary, the AddTwo procedure adds two 32-bit integers passed as parameters, and the result is returned in the EAX register. The use of the base pointer (EBP) simplifies parameter access within the stack frame.

==========

Specifying the Parameter Passing Protocol:

==========

The parameter passing protocol specifies how parameters are passed to and from procedures. There are different parameter passing protocols, such as C, Pascal, and STDCALL.

To specify the parameter passing protocol for a procedure in assembly language, you can use the attributes field of the PROC directive.

For example, the following procedure declares that it uses the C calling convention:

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

If you execute Example1() using the INVOKE directive, the assembler will generate code that is consistent with the C calling convention.

Similarly, the following procedure declares that it uses the STDCALL calling convention:

```
835 Example1 PROC STDCALL,
836 parm1:DWORD, parm2:DWORD
```

If you execute Example1() using the INVOKE directive, the assembler will generate code that is consistent with the STDCALL calling convention.

The following example shows how to use the PROC directive to declare a procedure with a specific parameter passing protocol:

```
835 Example1 PROC STDCALL,
836 parm1:DWORD, parm2:DWORD
837
838
839
840
841 .MODEL FLAT,STDCALL
842
843 ; Declare a procedure with the C calling convention.
844 Example1 PROC C,
845 parm1:DWORD, parm2:DWORD
846
847 ; ...
848
849 Example1 ENDP
850
851 ; Declare a procedure with the STDCALL calling convention.
852 Example2 PROC STDCALL,
853 parm1:DWORD, parm2:DWORD
854
855 ; ...
856
857 Example2 ENDP
```

The ability to specify the parameter passing protocol for a procedure is a powerful feature that allows you to write assembly language code that can be called from other programming languages.

==========

PROTO Directive in 32-bit Mode

==========

In 32-bit mode, the **PROTO directive** is used to create a prototype for an existing procedure.

A prototype declares a procedure's name and parameter list.

It allows you to call a procedure before defining it and to verify that the number and types of arguments match the procedure definition.

The syntax of the PROTO directive is as follows:

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

- • • **label** is the name of the procedure.
- • • **attributes** is an optional field that can be used to specify the parameter passing protocol for the procedure.
- • • **parameter_list** is an optional list of parameters that the procedure takes. Example

The following example shows how to create a prototype for a procedure named ArraySum():

```
866 ArraySum PROTO,
867 ptrArray:PTR DWORD,
868 ; points to the array
869 szArray:DWORD
870 ; array size
```

This prototype declares that the ArraySum() procedure takes two parameters: a pointer to an array of doublewords and the size of the array.

Once you have created a prototype for a procedure, you can call it using the INVOKE directive.

The INVOKE directive will verify that the number and types of arguments match the prototype before calling the procedure.

For example, the following code calls the ArraySum() procedure:

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

This code will call the ArraySum() procedure with the pointer to the array ptrArray and the size of the array szArray as arguments.

Important Considerations:

Every procedure called by the INVOKE directive must have a prototype.

The prototype for a procedure must appear before the procedure is called.

The number and types of arguments in the prototype must match the number and types of arguments in the procedure definition.

The PROTO directive is a powerful tool for writing reusable and reliable assembly language code.

It allows you to call procedures before defining them and to verify that the number and types of arguments match the procedure definition.

ASSEMBLY TIME ARGUMENT CHECKING

The PROTO directive helps the assembler check the number and types of arguments passed to a procedure when it is called. This is called assembly time argument checking.

However, assembly time argument checking is not as precise as you would find in languages like C and C++. MASM only checks for the correct number of parameters and to a limited extent, matches argument types to parameter types.

Suppose the following prototype is declared for a procedure named Sub1():

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

This prototype declares that the Sub1() procedure takes three parameters: a byte, a word, and a pointer to a byte.

The following is a valid call to Sub1():

```
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.

Errors Detected by MASM

MASM will generate an error if an argument exceeds the size of a declared parameter. For example, the following INVOKE statement will generate an error:

```
908 INVOKE Sub1, word_1, word_2, ADDR byte_1
909 ;arg 1 error
```

MASM will also generate errors if an INVOKE statement has too few or too many arguments. For example, the following INVOKE statements will generate errors:

```
913 INVOKE Sub1, byte_1, word_2
914 ; error: too few arguments
915 INVOKE Sub1, byte_1,
916 ; error: too many arguments
917 word_2, ADDR byte_1, word_2
```

Errors Not Detected by MASM

MASM will not detect an error if an argument's type is smaller than a declared parameter. For example, the following INVOKE statement will not generate an error:

```
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).

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).
```

Here's a more detailed explanation:

push 404000h: This instruction pushes the pointer to byte_1 onto the stack. It's pushing an address to the stack, which may be used as a parameter for the function you're calling (at address 00401071).

sub esp, 2: This instruction subtracts 2 from the stack pointer (esp). It's used to reserve 2 bytes on the stack for padding. This padding might be needed to align the stack correctly, especially when dealing with functions or system calls that expect specific stack alignment.

push word ptr ds:[00404001h]: Here, the code pushes the value of word_1 onto the stack. It's assumed that word_1 is a 16-bit (2-byte) value. The word ptr specifies that you are dealing with a word-sized value, and it's loaded from memory address 00404001h.

mov al, byte ptr ds:[00404000h]: This instruction loads the value of byte_1 into the AL register. It's assumed that byte_1 is an 8-bit (1-byte) value, and it's loaded from memory address 00404000h.

push eax: The value from the EAX register is pushed onto the stack. This is likely done to make it available as a parameter for the function being called at address 00401071.

call 00401071: This instruction calls a function located at address 00401071. The behavior of this function depends on its implementation and the purpose it serves within your program.

Overall, this code appears to be setting up some parameters on the stack and then calling a function at address 00401071, passing these parameters. The specifics of how these parameters are used and the purpose of the function being called would require more context to fully understand.

==========

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
```

The ArraySum() procedure takes two parameters: a pointer to an array of doublewords and the size of the array. The procedure uses the ESI and ECX registers to store the address of the array and the size of the array, respectively.

The procedure begins by setting the EAX register to zero. This will be the sum of the array elements. Then, the procedure checks the size of the array. If the size is zero, the procedure simply returns. Otherwise, the procedure enters a loop.

In the loop, the procedure adds the value at the current address in the array to the EAX register. Then, the procedure increments the ESI register to point to the next element in the array. The loop repeats until all of the elements in the array have been added.

After the loop has finished, the sum of the array elements is stored in the EAX register. The procedure then returns.

Here is an example of how to call the ArraySum() procedure:

```
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 with the address of the array variable and the number of elements in the array variable as arguments.

The LENGTHOF operator is used to calculate the number of elements in the array variable.

After the ArraySum() procedure has returned, the sum of the array elements is stored in the theSum variable.

The ArraySum() example is a good example of how to use the PROC directive to declare stack parameters and how to use the INVOKE directive to call procedures with stack 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
```

In the .data section:

- An array named array is defined with five DWORD (32-bit) elements and initial values.
- A DWORD variable named theSum is declared with a question mark to indicate that it's uninitialized.

In the .code section:

- The ArraySum procedure is defined to calculate the sum of an array of DWORDs. It expects two parameters:
- • **ptrArray:** A pointer to the array.
- • **szArray:** The size (number of elements) of the array. Inside ArraySum:
- • **esi** is used to hold the address of the array.
- • **ecx** stores the size of the array.
- • **eax** is initialized to zero and used to accumulate the sum.
- The code checks if the array size is zero. If it is, it immediately jumps to L2, effectively quitting the procedure.
- In L1, it adds the value at the address pointed by esi to the sum in eax, increments esi by 4 to move to the next DWORD in the array, and repeats this process for the entire array size using the loop instruction.

- Finally, in L2, it returns with the sum stored in eax.

The main procedure:

- Calls the ArraySum procedure using the INVOKE directive and passes the address of the array and the number of elements (LENGTHOF array) as parameters.
- It stores the result (the sum) returned by ArraySum in the theSum variable.
- After this code, your program logic can continue, making use of the calculated sum stored in theSum.
- This code efficiently calculates the sum of the elements in the array and stores it in theSum.

Parameter Classifications:

In the context of procedure parameters, these parameters can be classified based on the direction of data transfer between the calling program and the called procedure:

Here is a simpler explanation of input and output parameters in assembly language:

Input parameters are passed to a procedure from the calling program. The procedure can use the data, but it cannot change it. This means that when the procedure returns, the data in the calling program will be the same as it was before the procedure was called. Input parameters are typically used when the procedure needs data to operate on, but does not need to return any data.

Output parameters are used to return data from a procedure to the calling program. The procedure can change the data in the output parameter, and the calling program will see the change after the procedure returns. Output parameters are typically used when the procedure needs to return data to the calling program, such as the result of a calculation.

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 this example, the x and y parameters are input parameters. The procedure add_two_numbers() uses the data in these parameters to calculate the sum of the two numbers. However, the procedure does not change the values of x and y.

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

Input and output parameters can be used together in a procedure. For example, a procedure could take an input parameter that specifies the size of an array, and it could use an output parameter to return the sum of the elements in the array.

One example of an input/output parameter is a buffer. A buffer is a block of memory that is used to store data temporarily. A procedure might take an input/output parameter of type buffer to read data from a file and then return the data to the calling program.

The procedure could also use the buffer to modify the data and then return the modified data to the calling program.

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

```
procedure read_file(buffer: PTR BYTE, size: DWORD): DWORD
  ; ...
  ; Read data from the file into the buffer
  ; ...
  ret
endp

; Calling the procedure
mov eax, OFFSET buffer
mov ebx, size
call read_file

; The buffer variable will now contain the data that was read from the file
```

In this example, the buffer parameter is an input/output parameter. The read_file() procedure reads data from the file into the buffer.

The read_file() procedure also returns the number of bytes that were read from the file. The calling program can use this information to determine how much data is in the buffer.

Example: Exchanging Two Integers

```
1090 include Irvine32.inc
1091
1092 Swap PROTO, pValX:PTR DWORD, pValY:PTR DWORD
1093 ; Exchange the values of two 32-bit integers
1094 ; Returns: nothing
1095 Swap PROC USES eax esi edi,
1096 pValX:PTR DWORD,
1097 ; pointer to first integer
1098 pValY:PTR DWORD
1099 ; pointer to second integer
1100 ; get pointers
1101 mov esi, pValX
1102 mov edi, pValY
1103 ; get first integer
1104 mov eax, [esi]
1105 ; exchange with second
1106 xchg eax, [edi]
1107 ; replace first integer
1108 mov [esi], eax
1109 ; PROC generates RET 8 here
1110 ret
1111 Swap ENDP
```

The Swap procedure takes two input/output parameters: pValX and pValY. These parameters contain the addresses of the two integers that need to be swapped.

The procedure begins by getting the pointers to the two integers.

Then, the procedure gets the value of the first integer and stores it in the EAX register.

Next, the procedure uses the **XCHG instruction** to exchange the values of the EAX register and the second integer.

Finally, the procedure stores the value of the EAX register in the first integer.

The Swap procedure does not return any value, so it simply ends with a RET instruction.

However, the PROC directive generates a RET 8 instruction at the end of the procedure, assuming that the STDCALL calling convention is being used.

The Swap procedure can be called from the main procedure as follows:

```
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
```

The INVOKE statement calls the Swap procedure with the addresses of the first two elements of the Array variable as arguments. After the Swap procedure returns, the first two elements of the Array variable will be swapped.

Missing information:

The Swap procedure does not check for errors. For example, if the addresses of the two integers are not valid, the procedure will crash.

The Swap procedure is not optimized for speed. For example, the procedure could use a temporary variable to store the value of the first integer while the second integer is being swapped.

Overall, the Swap procedure is a simple example of how to use input/output parameters in assembly language.

==========

Debugging Tips

==========

Argument Size Mismatch

When passing arguments to a procedure, it is important to make sure that the arguments are the correct size.

For example, if a procedure expects a doubleword pointer, you should pass a doubleword pointer.

If you pass a smaller pointer, such as a word pointer, the procedure will not be able to access the data correctly.

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]
```

The Swap() procedure expects two doubleword pointers. However, the incorrect call to Swap() passes two word pointers.

This will cause the procedure to not be able to access the data correctly.

Passing the Wrong Type of Pointer

When passing arguments to a procedure, it is also important to make sure that the arguments are the correct type.

For example, if a procedure expects a doubleword pointer, you should pass a doubleword pointer. If you pass a different type of pointer, such as a byte pointer, the procedure will not be able to access the data correctly.

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]
```

The Swap() procedure expects two doubleword pointers. However, the incorrect call to Swap() passes two byte pointers.

This will cause the procedure to not be able to access the data correctly.

Passing Immediate Values

You should not pass immediate values to reference parameters.

A **reference parameter** is a parameter that expects a pointer to data.

If you pass an immediate value to a reference parameter, the procedure will not be able to access the data correctly.

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

```
; Sub2 procedure
Sub2 PROC, dataPtr:PTR WORD
mov
esi,dataPtr
; get the address
mov
WORD PTR [esi],0
; dereference, assign zero
ret
Sub2 ENDP

; Incorrect call to Sub2
INVOKE
Sub2, 1000h
```

The Sub2() procedure expects a pointer to a word as its only parameter. However, the incorrect call to Sub2() passes an immediate value. This will cause the procedure to not be able to access the data correctly.

It is important to be careful when passing arguments to procedures in assembly language. If you make a mistake, it can cause the program to crash or produce incorrect results. Be sure to check the documentation for the procedure that you are calling to make sure that you are passing the correct type and number of arguments.

WIRESTACKFRAME PROCEDURE

Here is a more in-depth explanation of the WriteStackFrame and WriteStackFrameName procedures:

The Irvine32 library contains a useful procedure named WriteStackFrame that displays the contents of the current procedure's stack frame. It shows the procedure's stack parameters, return address, local variables, and saved registers.

```
WriteStackFrame PROTO,
    numParam:DWORD,
    ; number of passed parameters
    numLocalVal: DWORD,
    ; number of DWordLocal variables
    numSavedReg: DWORD
    ; number of saved registers
```

The **WriteStackFrame procedure** displays the contents of the current stack frame, which contains the stack parameters, local variables, saved registers, and return address for the current procedure.

It takes 3 parameters:

- • • **numParam -** The number of parameters passed to the current procedure. This determines how many DWORDs to show for the parameters at the top of the stack.
- • • **numLocalVal -** The number of DWORD local variables allocated on the stack for the current procedure.
- • • **numSavedReg -** The number of registers saved on the stack for the current procedure. Typically this is 2 for EAX and EBX.

It displays the stack contents by starting at EBP and moving downward to ESP. For each DWORD it displays the offset from EBP and the hex value stored there.

The parameters passed to the procedure are displayed first at the highest offsets from EBP. Then the return address, saved EBP, local variables, and saved registers are displayed in descending offset order.

ESP points to the last used stack location, so the display stops when it reaches ESP.

WriteStackFrameName does the same thing, but takes an additional parameter:

- • • **procName** - A pointer to a null-terminated string containing the name of the current procedure. This is displayed at the top of the output.

So WriteStackFrameName allows you to identify which procedure's stack frame is being displayed. This is useful when multiple procedures call WriteStackFrame/Name.

In summary, these procedures give visibility into the stack contents at any point within a procedure. This helps debug issues with stack parameters, local variables, register saving, etc.

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

```
; In main procedure
main PROC
    mov eax, 0EAEAEAEAh    ; Save test value in EAX
    mov ebx, 0EBEBEBEBh    ; Save test value in EBX
    INVOKE myProc, 1111h, 2222h ; Call myProc, passing 2 parameters
    exit main              ; Exit program

main ENDP
; In myProc procedure
myProc PROC
    ; Procedure uses EAX and EBX, so they will be saved
    USES eax ebx
    x: DWORD, y:DWORD      ; Declare parameter variables
    LOCAL a:DWORD, b:DWORD  ; Declare local variables
    PARAMS = 2             ; 2 parameters
    LOCALS = 2             ; 2 local DWORD variables
    SAVED_REGS = 2         ; 2 saved registers (EAX and EBX)
    mov a,0AAAAh           ; Load value into local variable a
    mov b,0BBBBh           ; Load value into local variable b
    ; Display stack frame contents
    INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS
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:

```
1221 WriteStackFrameName PROTO,
1222     numParam:DWORD,
1223     ; number of passed parameters
1224     numLocalVal:DWORD,
1225     ; number of DWORD local variables
1226     numSavedReg:DWORD,
1227     ; number of saved registers
1228     procName:PTR BYTE
1229     ; null-terminated string
```

The main procedure:

- Loads some sample values into EAX and EBX to be saved on the stack later.
- Calls the myProc procedure, passing two DWORD arguments (1111h and 2222h).
- Exits the program.

The myProc procedure:

- Uses EAX, 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 for the number of parameters

- 2 for the number of local DWORD variables

- 2 for the number of saved registers (EAX and EBX)

This displays the contents of myProc's stack frame, including:

- **The 1111h and 2222h parameters**
- **The return address back to main**
- **The saved EBP from main**
- **The local variables a and b**
- **The saved EAX and EBX registers from main**

So, this demonstrates how WriteStackFrame can display a procedure's stack contents to help understand and debug the stack usage.

You can find the source code for the Irvine32 library in the \Examples\Lib32 directory of our book's install directory (usually C:\Irvine). Look for the file named Irvine32.asm.

MULTIMODULE PROGRAMS

A large program can be divided into multiple modules (assembled units) to make it easier to manage and assemble. Each module is assembled independently, so a change to one module's source code only requires reassembling the single module. The linker combines all assembled modules (OBJ files) into a single executable file.

There are two general approaches to creating multimodule programs:

Traditional approach:

Using the EXTERN directive to declare external procedures and the PUBLIC directive to export procedures to other modules. Microsoft's advanced INVOKE and PROTO directives: simplify procedure calls and hide some low-level details.

Hiding and Exporting Procedure Names

By default, MASM makes all procedures public, permitting them to be called from any other module in the same program.

You can override this behavior using the PRIVATE qualifier or the OPTION PROC:PRIVATE directive.

PRIVATE qualifier: makes a single procedure private.

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

You can use the PUBLIC directive to export any procedures you want.

If you use OPTION PROC:PRIVATE in your program's startup module, be sure to designate your startup procedure (usually main) as PUBLIC, or the operating system's loader will not be able to find it.

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
```

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

This would create an executable file called myprog.exe.

The EXTERN directive tells the assembler that the myProc() procedure is defined in another module.

The PUBLIC directive tells the assembler that the myProc() procedure can be called from other modules.

Using the INVOKE and PROTO Directives

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

```
; Module 1: mod1.asm

PROTO myProc

myProc PROC PUBLIC
; ...

myProc ENDP

; Module 2: mod2.asm

INVOKE myProc

main PROC
; ...

main ENDP
```

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

```
ml /c /obj mod1.asm
ml /c /obj mod2.asm
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

To call an external procedure in MASM, you use the EXTERN directive.

The EXTERN directive tells the assembler that the procedure is defined in another module and gives the procedure's name and stack frame size.

The following example shows how to call an external procedure named sub1():

```
1297 INCLUDE Irvine32.inc
1298 EXTERN sub1@0:PROC
1299
1300 .code
1301     main PROC
1302       call sub1@0
1303       exit
1304     main ENDP
1305     END main
```

The **@0 suffix** at the end of the procedure name indicates that the procedure **does not have any parameters.**

If the procedure has parameters, you must include the **stack frame size** in the EXTERN directive.

The stack frame size is the total amount of stack space that the procedure uses to store its parameters and local variables.

The following example shows how to call an external procedure named AddTwo(), which has two doubleword parameters:

```
1310 INCLUDE Irvine32.inc
1311 EXTERN AddTwo@8:PROC
1312
1313 .code
1314     main PROC
1315       call AddTwo@8
1316       exit
1317     main ENDP
1318     END main
```

The @8 suffix at the end of the procedure name indicates that the procedure uses 8 bytes of stack space for its parameters.

You can also use the PROTO directive in place of the EXTERN directive. The PROTO directive tells the assembler about the prototype of the procedure, including its name and the number and types of its arguments.

The following example shows how to use the PROTO directive to declare the prototype of the AddTwo() procedure:

```
1323 INCLUDE Irvine32.inc
1324 PROTO AddTwo,
1325 val1:DWORD,
1326 val2:DWORD
1327
1328 .code
1329     main PROC
1330       call AddTwo
1331       exit
1332     main ENDP
1333     END main
```

The PROTO directive tells the assembler that the AddTwo() procedure has two doubleword parameters.

When the assembler sees a call to the AddTwo() procedure, it can check to make sure that the correct number of arguments are passed to the procedure.

The EXTERN and PROTO directives are both used to call external procedures.

The EXTERN directive is simpler to use, but the PROTO directive provides more information to the assembler, which can help to prevent errors.

Which directive should you use?

If you are only calling a few external procedures and you are not concerned about the performance of your program, then you can use the EXTERN directive.

If you are calling a large number of external procedures or if you are concerned about the performance of your program, then you should use the PROTO directive.

The PROTO directive provides more information to the assembler, which can help to optimize the code.

==========

Using Variables and Symbols across Module Boundaries:

Exporting Variables and Symbols

==========

What is EXTERNDEF?

EXTERNDEF is a directive that combines the functionality of the PUBLIC and EXTERN directives. It can be used to export variables and symbols from one module and import them into another module.

How to use EXTERNDEF?

To use EXTERNDEF, you first need to create an include file that contains the EXTERNDEF declarations for the variables and symbols that you want to share. For example, the following include file defines two variables, count and SYM1:

```
1336 ; vars.inc
1337 EXTERNDEF count:DWORD, SYM1:ABS
```

Then, you can include the include file in any module that needs to access the variables or symbols. For example, the following module exports the count and SYM1 variables:

```
1343 ; sub1.asm
1344 .386
1345 .model flat,STDCALL
1346 INCLUDE vars.inc
1347 SYM1 = 10
1348 .data
1349     count DWORD 0
1350     END
```

Finally, you can import the variables or symbols into any module that needs to use them. For example, the following module imports the count and SYM1 variables and uses them to calculate a value:

```
; main.asm
.386
.model flat,stdcall
.stack 4096
ExitProcess proto, dwExitCode:dword
INCLUDE vars.inc
.code
    main PROC
    mov
    count,2000h
    mov
    eax,SYM1
    INVOKE ExitProcess,0
    main ENDP
    END main
```

Benefits of using EXTERNDEF

There are several benefits to using EXTERNDEF to share variables and symbols across module boundaries:

It makes it easy to share variables and symbols between modules. It helps to reduce the amount of duplicate code.

It makes the code more modular and reusable. Conclusion

EXTERNDEF is a powerful directive that can be used to share variables and symbols across module boundaries.

It is a good practice to use EXTERNDEF to share variables and symbols between modules, as it makes the code more modular and reusable.

Let's make our program modular and see these concepts in action:

==========

ArraySum program

==========

The ArraySum program, first presented in Chapters before, is a good example of a multimodule program. The program can be divided into the following modules:

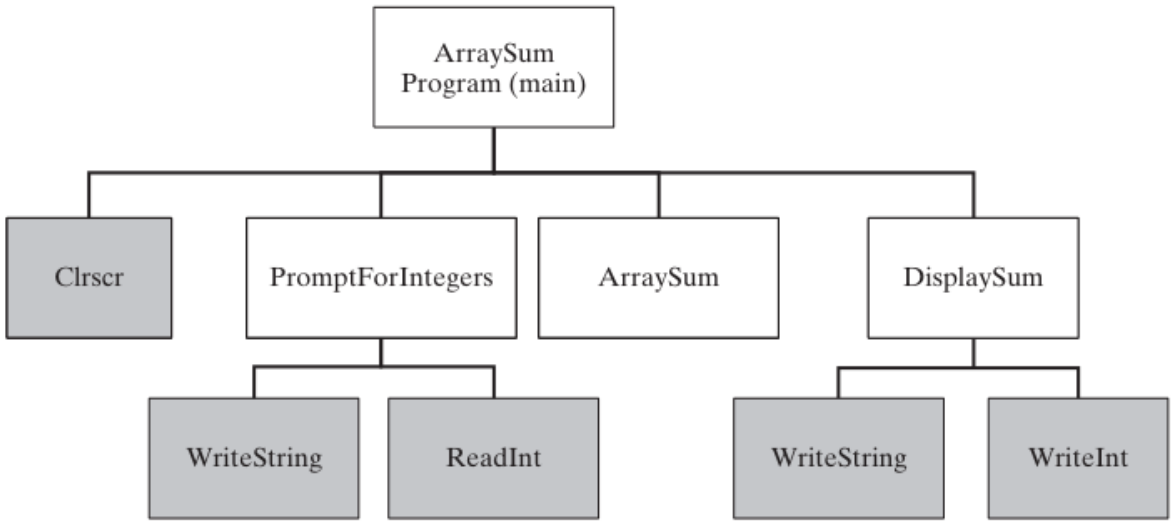
main.asm: The startup module, which calls the other modules to perform the program's tasks.

promptforintegers.asm: Prompts the user to enter an array of integers and reads the integers from the console.

arraysum.asm: Calculates the sum of the integers in the array.

writeinteger.asm: Writes an integer to the console.

The following diagram shows the structure chart of the ArraySum program:



==========

Prompt for Integers

==========

```
; Prompt For Integers (_prompt.asm)
INCLUDE Irvine32.inc
.code
        ;-----------------------------------------------------
        PromptForIntegers PROC
        ; Prompts the user for an array of integers and fills
        ; the array with the user's input.
        ; Receives:
        ; ptrPrompt: PTR BYTE - prompt string
        ; ptrArray: PTR DWORD - pointer to array
        ; arraySize: DWORD - size of the array
        ; Returns: nothing
        ;-----------------------------------------------------
        arraySize EQU [ebp+16]
        ptrArray EQU [ebp+12]
        ptrPrompt EQU [ebp+8]
```

```
enter 0,0
pushad
; save all registers
mov ecx,arraySize
cmp ecx,0
; array size != 0?
jle L2
; yes: quit
mov edx,ptrPrompt
; address of the prompt
mov esi,ptrArray
L1:
call WriteString
; display string
call ReadInt
; read integer into EAX
call Crlf
; go to next output line
mov [esi],eax
; store in array
add esi,4
; next integer
loop L1
L2:
popad
; restore all registers
leave
ret
12
; restore the stack
PromptForIntegers ENDP
end
```

The prompt.asm file contains the source code for the PromptForIntegers() procedure. This procedure prompts the user for an array of integers and fills the array with the user's input.

- The **PromptForIntegers()** procedure takes three parameters:
- • **ptrPrompt:** A pointer to the prompt string.
- • **ptrArray:** A pointer to the array.
- • **arraySize:** The size of the array.

The PromptForIntegers() procedure works as follows:

It saves all of the registers.

It compares the array size to 0. If the array size is 0, the procedure exits. It displays the prompt string using the WriteString() procedure. It reads an integer from the console using the ReadInt() procedure. It stores the integer in the array.

It increments the array pointer. It repeats steps 3-6 until all of the integers have been read. It restores all of the registers. It leaves the procedure. Here is a more detailed explanation of each step:

Step 1: Save all of the registers

The PromptForIntegers() procedure saves all of the registers because it needs to use them and it does not want to overwrite any of the values that are in the registers when the procedure is called.

Step 2: Compare the array size to 0

The PromptForIntegers() procedure checks the array size to make sure that it is not 0. If the array size is 0, the procedure exits. This is because it does not make sense to prompt the user for an array of integers if the array is empty.

Step 3: Display the prompt string

The PromptForIntegers() procedure displays the prompt string using the WriteString() procedure. The WriteString() procedure is a library procedure that writes a string to the console.

Step 4: Read an integer from the console

The PromptForIntegers() procedure reads an integer from the console using the ReadInt() procedure. The ReadInt() procedure is a library procedure that reads an integer from the console and stores it in the EAX register.

Step 5: Store the integer in the array

The PromptForIntegers() procedure stores the integer in the array by moving the EAX register to the array element that is pointed to by the array pointer.

Step 6: Increment the array pointer

The PromptForIntegers() procedure increments the array pointer so that it points to the next element in the array.

Step 7: Repeat steps 3-6 until all of the integers have been read

The PromptForIntegers() procedure repeats steps 3-6 until all of the integers have been read. This is done by using the loop instruction. The loop instruction repeats a block of instructions until a specified condition is met. In this case, the condition is that the array pointer is not equal to the end of the array.

Step 8: Restore all of the registers

The PromptForIntegers() procedure restores all of the registers that it saved in step 1.

Step 9: Leave the procedure

The PromptForIntegers() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The **PromptForIntegers() procedure** is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses library procedures to perform common tasks, such as writing a string to the console and reading an integer from the console.

==========

ArraySum program

==========

```
; ArraySum Procedure (_arraysum.asm)
INCLUDE Irvine32.inc
.code
	;-----------------------------------------------------
	ArraySum PROC
	;
	; Calculates the sum of an array of 32-bit integers.
	; Receives:
	; ptrArray - pointer to array
	; arraySize - size of array (DWORD)
	; Returns: EAX = sum
	;-----------------------------------------------------
	ptrArray EQU [ebp+8]
	arraySize EQU [ebp+12]
	enter 0,0
	push ecx
	; don't push EAX
	push esi
	mov eax,0
	; set the sum to zero
	mov esi,ptrArray
	mov ecx,arraySize
	cmp ecx,0
	; array size != 0?
	jle L2
	; yes: quit
```

```
L1:
add eax,[esi]
; add each integer to sum
add esi,4
; point to next integer
loop L1
; repeat for array size
L2:
pop esi
pop ecx
; return sum in EAX
leave
ret
8
; restore the stack
ArraySum ENDP
END
```

The code you provided is the implementation of the ArraySum() procedure in assembly language. The ArraySum() procedure calculates the sum of an array of 32-bit integers.

The ArraySum() procedure takes two parameters:

- ptrArray: A pointer to the array.
- arraySize: The size of the array.

The ArraySum() procedure works as follows:

It saves the ECX register, because it needs to use it and it does not want to overwrite the value that is in the register when the procedure is called. It sets the EAX register to 0.

This is because the EAX register will be used to store the sum of the integers in the array. It moves the pointer to the first element of the array into the ESI register. It compares the array size to 0.

If the array size is 0, the procedure exits. This is because it does not make sense to calculate the sum of an empty array.

It adds the integer at the current position in the array to the EAX register. It increments the ESI register so that it points to the next element in the array.

It repeats steps 5 and 6 until all of the integers in the array have been added to the EAX register. It restores the ECX register.

It leaves the procedure.

Here is a more detailed explanation of each step:

Step 1: Save the ECX register

The ArraySum() procedure saves the ECX register because it needs to use it and it does not want to overwrite the value that is in the register when the procedure is called.

Step 2: Set the EAX register to 0

The ArraySum() procedure sets the EAX register to 0 because it will be used to store the sum of the integers in the array.

Step 3: Move the pointer to the first element of the array into the ESI register

The ArraySum() procedure moves the pointer to the first element of the array into the ESI register. This is because the ESI register will be used to iterate through the array.

Step 4: Compare the array size to 0

The ArraySum() procedure checks the array size to make sure that it is not 0. If the array size is 0, the procedure exits. This is because it does not make sense to calculate the sum of an empty array.

Step 5: Add the integer at the current position in the array to the EAX register

The ArraySum() procedure adds the integer at the current position in the array to the EAX register. This is done using the add instruction. The add instruction adds two operands and stores the result in the first operand.

Step 6: Increment the ESI register so that it points to the next element in the array

The ArraySum() procedure increments the ESI register so that it points to the next element in the array. This is done using the inc instruction. The inc instruction increments the value of the operand by 1.

Step 7: Repeat steps 5 and 6 until all of the integers in the array have been added to the EAX register

The ArraySum() procedure repeats steps 5 and 6 until all of the integers in the array have been added to the EAX register.

This is done using the loop instruction. The loop instruction repeats a block of instructions until a specified condition is met.

In this case, the condition is that the ESI register is not equal to the value of the ptrArray parameter.

Step 8: Restore the ECX register

The ArraySum() procedure restores the ECX register.

Step 9: Leave the procedure

The ArraySum() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The ArraySum() procedure is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses a loop to iterate through the array, which is a common technique in assembly language.

==========

Display Sum Proc

==========

```
; DisplaySum Procedure (_display.asm)
INCLUDE Irvine32.inc
.code
	;-----------------------------------------------------
	DisplaySum PROC
	; Displays the sum on the console.
	; Receives:
	; ptrPrompt - offset of the prompt string
	; theSum - the array sum (DWORD)
	; Returns: nothing
	;-----------------------------------------------------
	theSum EQU [ebp+12]
	ptrPrompt EQU [ebp+8]
	enter 0,0
	push eax
	push edx
	mov edx,ptrPrompt
	; pointer to prompt
	call WriteString
	mov eax,theSum
	call WriteInt
	; display EAX
	call Crlf
	pop edx
	pop eax
	leave
	ret
	8
	; restore the stack
	DisplaySum ENDP
	END
```

The code you provided is the implementation of the DisplaySum() procedure in assembly language. The DisplaySum() procedure displays the sum of an array of 32-bit integers on the console.

The DisplaySum() procedure takes two parameters:

- • • **ptrPrompt:** A pointer to the prompt string.
- • • **theSum:** The sum of the integers in the array. The DisplaySum() procedure works as follows:

It saves the EAX and EDX registers, because it needs to use them and it does not want to overwrite the values that are in the registers when the procedure is called. It moves the pointer to the prompt string into the EDX register.

It calls the WriteString() procedure to display the prompt string on the console. It moves the sum of the integers in the array into the EAX register.

It calls the WriteInt() procedure to display the sum of the integers in the array on the console.

It calls the Crlf() procedure to move the cursor to the next line on the console. It restores the EAX and EDX registers.

It leaves the procedure.

Here is a more detailed explanation of each step:

Step 1: Save the EAX and EDX registers

The DisplaySum() procedure saves the EAX and EDX registers because it needs to use them and it does not want to overwrite the values that are in the registers when the procedure is called.

Step 2: Move the pointer to the prompt string into the EDX register

The DisplaySum() procedure moves the pointer to the prompt string into the EDX register. This is because the EDX register will be used to pass the pointer to the prompt string to the WriteString() procedure.

Step 3: Call the WriteString() procedure to display the prompt string on the console

The DisplaySum() procedure calls the WriteString() procedure to display the prompt string on the console. The WriteString() procedure is a library procedure that writes a string to the console.

Step 4: Move the sum of the integers in the array into the EAX register

The DisplaySum() procedure moves the sum of the integers in the array into the EAX register. This is because the EAX register will be used to pass the sum of the integers in the array to the WriteInt() procedure.

Step 5: Call the WriteInt() procedure to display the sum of the integers in the array on the console

The DisplaySum() procedure calls the WriteInt() procedure to display the sum of the integers in the array on the console. The WriteInt() procedure is a library procedure that writes an integer to the console.

Step 6: Call the Crlf() procedure to move the cursor to the next line on the console

The DisplaySum() procedure calls the Crlf() procedure to move the cursor to the next line on the console. The Crlf() procedure is a library procedure that moves the cursor to the next line on the console.

Step 7: Restore the EAX and EDX registers

The DisplaySum() procedure restores the EAX and EDX registers.

Step 8: Leave the procedure

The DisplaySum() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The DisplaySum() procedure is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses library procedures to perform common tasks, such as writing a string to the console and writing an integer to the console.

==========

Sum_main.asm

==========

```
; Integer Summation Program (Sum_main.asm)
; Multimodule example:
; This program inputs multiple integers from the user,
; stores them in an array, calculates the sum of the
; array, and displays the sum.
INCLUDE Irvine32.inc
INCLUDE macros.asm ; Include for INVOKE and PROTO

EXTERN PromptForIntegers:PROC
EXTERN ArraySum:PROC
EXTERN DisplaySum:PROC
```

```
; Modify Count to change the size of the array:
Count = 3

.data
prompt1 BYTE "Enter a signed integer: ",0
prompt2 BYTE "The sum of the integers is: ",0
array DWORD Count DUP(?)
sum DWORD ?

.code
main PROC
   call Clrscr
   ; PromptForIntegers(ADDR prompt1, ADDR array, Count)
   INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count

   ; sum = ArraySum(ADDR array, Count)
   INVOKE ArraySum, ADDR array, Count
   mov sum, eax

   ; DisplaySum(ADDR prompt2, sum)
   INVOKE DisplaySum, ADDR prompt2, sum
   call Crlf
   exit
main ENDP

END main
```

This code retains the same functionality as the original version but utilizes Microsoft's INVOKE and PROTO directives for calling procedures, making the code more structured and easier to read.

The code you provided is a multimodule example of an integer summation program. The program inputs multiple integers from the user, stores them in an array, calculates the sum of the array, and displays the sum.

The program is divided into three modules:

Sum_main.asm: This is the main module, which contains the main() procedure. The main() procedure is responsible for calling the other modules to perform the program's tasks.

promptforintegers.asm: This module contains the PromptForIntegers() procedure, which prompts the user for multiple integers and stores them in an array.

arraysum.asm: This module contains the ArraySum() procedure, which calculates the sum of the integers in an array.

display.asm: This module contains the DisplaySum() procedure, which displays the sum of the integers in an array on the console.

The Sum_main.asm module is the main module of the program. The main() procedure in this module performs the following steps:

It calls the Clrscr() procedure to clear the console screen. It calls the PromptForIntegers() procedure to prompt the user for multiple integers and store them in an array. It calls the ArraySum() procedure to calculate the sum of the integers in the array.

It calls the DisplaySum() procedure to display the sum of the integers in the array on the console. It calls the Crlf() procedure to move the cursor to the next line on the console. It calls the exit() procedure to exit the program.

The promptforintegers.asm module contains the PromptForIntegers() procedure. This procedure prompts the user for multiple integers and stores them in an array. The procedure takes the following parameters:

- • • **ptrPrompt:** A pointer to the prompt string.
- • • **ptrArray:** A pointer to the array.
- **Count:** The number of integers to prompt the user for.

The PromptForIntegers() procedure works as follows:

It iterates over the array and prompts the user for each integer. It stores the integer that the user enters in the array.

It repeats steps 1 and 2 until all of the integers have been entered. The arraysum.asm module contains the ArraySum() procedure. This procedure calculates the sum of the integers in an array. The procedure takes the following parameters:

- ptrArray: A pointer to the array.
- Count: The number of integers in the array.

The ArraySum() procedure works as follows:

It initializes the sum to 0.

It iterates over the array and adds each integer to the sum. It returns the sum. The display.asm module contains the DisplaySum() procedure.

This procedure displays the sum of the integers in an array on the console. The procedure takes the following parameters:

- • • **ptrPrompt:** A pointer to the prompt string.

- • • **theSum:** The sum of the integers in the array.

The DisplaySum() procedure works as follows:

It displays the prompt string on the console. It displays the sum of the integers in the array on the console. It moves the cursor to the next line on the console.

The integer summation program is a good example of how to use multiple modules to write a program. By dividing the program into modules, we can make the program more modular, reusable, and maintainable.

==========

Creating Modules using INVOKE and PROTO

==========

Creating the Modules Using INVOKE and PROTO section are the use of the INVOKE, PROTO, and PROC directives. These directives are used to create multimodule programs in 32-bit mode.

The INVOKE directive is used to call a procedure in another module. The PROTO directive is used to declare a prototype for a procedure. The PROC directive is used to define a procedure.

The following table shows the differences between the traditional use of CALL and EXTERN and the use of INVOKE, PROTO, and PROC:

Traditional Method:

Traditional Method	Advanced Method
CALL is used to call a procedure.	INVOKE is used to call a procedure in another module.
EXTERN is used to declare a symbol that is defined in another module.	PROTO is used to declare a prototype for a procedure.
PROC is used to define a procedure.	PROC is used to define a procedure, but it can also be used to declare parameters for a procedure.

The **main advantage of using the INVOKE, PROTO, and PROC directives** is that they allow the assembler to match up the argument lists passed by INVOKE to the

corresponding parameter lists declared by PROC. This helps ensure that the program is correct and that it does not crash.

Example Using INVOKE, PROTO, and PROC:

```
; Include the necessary include file
INCLUDE sum.inc

; Define data and code sections
.data
Count = 3
prompt1 BYTE "Enter a signed integer: ",0
prompt2 BYTE "The sum of the integers is: ",0
array DWORD Count DUP(?)
sum DWORD ?

.code
main PROC
    call Clrscr

    ; Call PromptForIntegers using INVOKE with argument lists
    INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count

    ; Call ArraySum using INVOKE with argument lists
    INVOKE ArraySum, ADDR array, Count
    mov sum, eax

    ; Call DisplaySum using INVOKE with argument lists
    INVOKE DisplaySum, ADDR prompt2, sum

    call Crlf
    exit
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
```

```
; display.asm
INCLUDE sum.inc

.code
DisplaySum PROC,
    ptrPrompt:PTR BYTE,
    theSum:DWORD

    push eax
    push edx
    mov edx, ptrPrompt
    call WriteString
    mov eax, theSum
    call WriteInt
    call Crlf
    pop edx
    pop eax
    ret
DisplaySum ENDP
END
```

```
; sum_main.asm
INCLUDE sum.inc

Count = 3

.data
    prompt1 BYTE "Enter a signed integer: ", 0
    prompt2 BYTE "The sum of the integers is: ", 0
    array DWORD Count DUP(?)
    sum DWORD ?
.code
    main PROC
        call Clrscr
        INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count
        INVOKE ArraySum, ADDR array, Count
        mov sum, eax
        INVOKE DisplaySum, ADDR prompt2, sum
        call Crlf
        exit
    main ENDP
    END
```

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

Traditional method:

Use the EXTERN directive to declare symbols that are defined in another module. Use the CALL directive to call procedures in other modules.

Advanced method:

Use the PROTO directive to declare prototypes for procedures in other modules. Use the INVOKE directive to call procedures in other modules. Use the PROC directive to define procedures, and declare parameters for procedures. The advanced method is simpler to use and more efficient, but it is only available in 32-bit mode.

Conclusion:

The advanced method is the preferred method for creating multimodule programs in 32-bit mode. It is simpler to use and more efficient than the traditional method. However, the traditional method is still supported, and it may be necessary for some programs.

ADVANCED OPTIONAL TOPIC 1 – USES OPERATOR

The USES operator is a powerful tool that can be used to save and restore registers at the beginning and end of a procedure.

However, it should not be used when declaring procedures that access their stack parameters using constant offsets such as [ebp + 8].

The following MySub1 procedure employs the USES operator to save and restore ECX and EDX:

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

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:

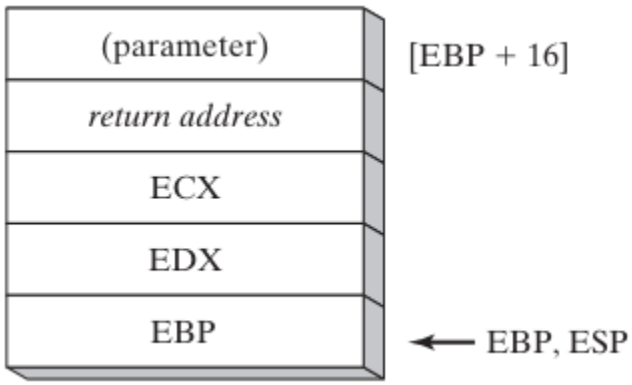
```
1620 MySub2 PROC USES ecx edx
1621 push ebp
1622 mov ebp,esp
1623 mov eax,[ebp+8]
1624 ; this is wrong!
1625 pop ebp
1626 ret 4
1627 MySub2 ENDP
```

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.



This is why it is important to avoid using the USES operator when declaring procedures that access their stack parameters using constant offsets.

If you need to save and restore registers in such a procedure, you can use the PUSH and POP instructions explicitly.

Here is a more in-depth explanation of why the USES operator causes problems in this case:

When a procedure is called, the caller pushes the procedure's arguments onto the stack.

The procedure then saves its caller-saved registers (EBP, ESI, EDI, and EBX) onto the stack. The procedure's base pointer (EBP) is set to point to the top of the stack frame.

The USES operator tells the assembler to save and restore the specified registers at the beginning and end of the procedure.

When the USES operator is used in a procedure that accesses its stack parameters using constant offsets, the assembler inserts the PUSH and POP instructions for the specified registers at the beginning and end of the procedure.

This corrupts the standard prologue code commonly used for subroutines, which relies on the stack pointer (ESP) to be pointing to the top of the stack frame.

In the example of MySub2, the USES operator is used to save and restore ECX and EDX. When MySub2 is called, its argument is pushed onto the stack.

The USES operator then causes the assembler to push ECX and EDX onto the stack. This corrupts the stack frame, because the stack pointer is now pointing to the wrong location.

To avoid this problem, you should not use the USES operator in procedures that access their stack parameters using constant offsets.

If you need to save and restore registers in such a procedure, you can use the PUSH and POP instructions explicitly.

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

Passing 8-bit Arguments

When passing stack arguments to procedures in 32-bit mode, it is best to push 32-bit operands.

This is because the stack pointer (ESP) must be aligned on a doubleword boundary.

If 16-bit operands are pushed onto the stack, ESP will not be aligned and a page fault may occur. Additionally, runtime performance may be degraded.

If you need to pass a 16-bit operand to a procedure in 32-bit mode, you can use the MOVZX instruction to expand the operand to 32 bits before pushing it onto the stack.

For example, the following Uppercase procedure receives a character argument and returns its 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 will automatically expand the character to 32 bits:

```
1667 push 'x'
1668 call Uppercase
```

However, if we pass a character variable to Uppercase, the PUSH instruction will not allow us to push an 8-bit operand onto the stack.

To work around this, we can use the MOVZX instruction to expand the character into 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 will ensure that ESP is aligned on a doubleword boundary and that the call to Uppercase is successful.

Passing 16-bit Arguments.

The AddTwo procedure expects two 32-bit integer arguments (the two integers to be added). However, the word1 and word2 variables are 16-bit integers.

Therefore, if we push word1 and word2 onto the stack and call AddTwo, the procedure will not be able to correctly add the two integers.

To fix this, we can zero-extend each argument before pushing it onto the stack. Zero-extension means that the high-order 16 bits of the argument are set to zero.

This will effectively convert the 16-bit argument to a 32-bit argument.

The following code correctly calls AddTwo by zero-extending each argument 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.

Once the arguments have been zero-extended, they are pushed onto the stack in reverse order (word2 first, then word1).

When AddTwo is called, it will pop the two arguments off the stack and add them together. The sum of the two integers will be returned in the EAX register.

It is important to note that the caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

In the case of stack parameters, the order and size of the parameters are important.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

Passing 64-bit Arguments

To pass 64-bit integer arguments to procedures in 32-bit mode, we must push the high-order doubleword of the argument first, followed by the low-order doubleword.

This is because the stack grows downwards, so the lower order doubleword of the argument will be at the lower address on the stack.

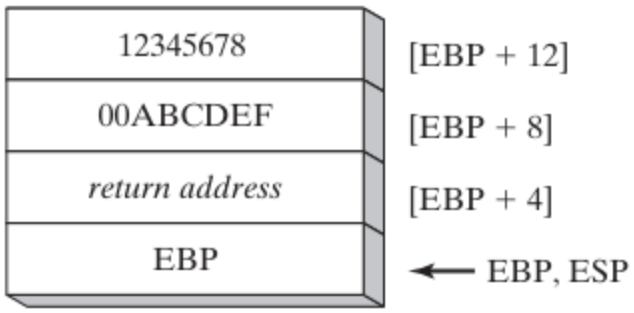
The following WriteHex64 procedure displays a 64-bit integer in hexadecimal:

```
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 then easily retrieve the high and low doublewords of the argument from the stack and display them in hexadecimal.

It is important to note that the caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

In the case of stack parameters, the order and size of the parameters are important.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

Here is a more in-depth explanation of why we must push the high-order doubleword of a 64-bit integer first when passing it to a procedure in 32-bit mode:

In 32-bit mode, the stack grows downwards. This means that when we push a value onto the stack, the stack pointer (ESP) is decremented.

When we pop a value off the stack, ESP is incremented.

When we pass a 64-bit integer to a procedure in 32-bit mode, we must push the high-order doubleword of the integer first, followed by the low-order doubleword.

This is because we want the integer to be stored on the stack in little-endian order.

In **little-endian order**, the low-order byte of the integer is stored at the lowest address on the stack.

If we were to push the low-order doubleword of the integer first, followed by the high-order doubleword, the integer would be stored on the stack in big-endian order.

In **big-endian order,** the high-order byte of the integer is stored at the lowest address on 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 then easily retrieve the high and low doublewords of the integer from the stack and display them in hexadecimal.

Why is it important to ensure that the arguments passed to a procedure are consistent with the parameters expected by the procedure?

The caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

This is because the procedure is expecting certain values to be passed to it in a certain order.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

For example, if the WriteHex64 procedure expects one 64-bit integer argument, and the caller passes two 64-bit integer arguments, the procedure will not be able to correctly display the two integers.

Or, if the caller passes a 32-bit integer argument instead of a 64-bit integer argument, the procedure will also not be able to correctly display the integer.

It is important to note that the compiler will not check to make sure that the caller is passing the correct number of arguments to a procedure, or that the arguments are in the correct order or have the wrong size. This is the responsibility of the programmer.

Non-Doubleword Local Variables

In 32-bit mode, the stack grows downwards. This means that when we push a value onto the stack, the stack pointer (ESP) is decremented.

When we pop a value off the stack, ESP is incremented.

When we declare a local variable in a procedure, MASM will allocate space for it on the stack. The size of the space allocated will depend on the size of the variable.

For example, if we declare a byte variable, MASM will allocate one byte of space on the stack. If we declare a doubleword variable, MASM will allocate four bytes of space on the stack.

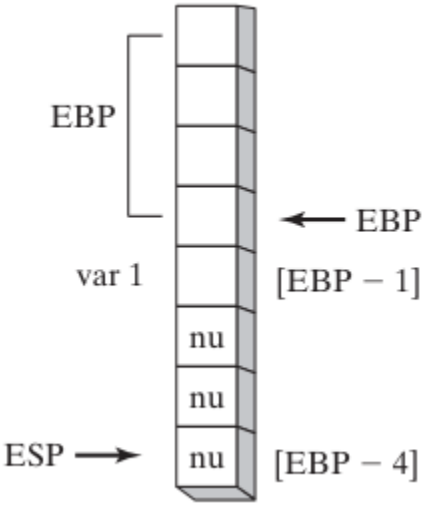
If we declare a local variable of a size that is not a multiple of four bytes (such as a byte or a word), MASM will round the size of the variable up to the next multiple of four bytes.

This is because the stack is aligned on a doubleword boundary. This means that all addresses on the stack must be divisible by four.

For example, if we declare a byte variable named var1 in the Example1 procedure, MASM will allocate four bytes of space for it on the stack, even though the variable is only one byte in size. The remaining three bytes will be unused.

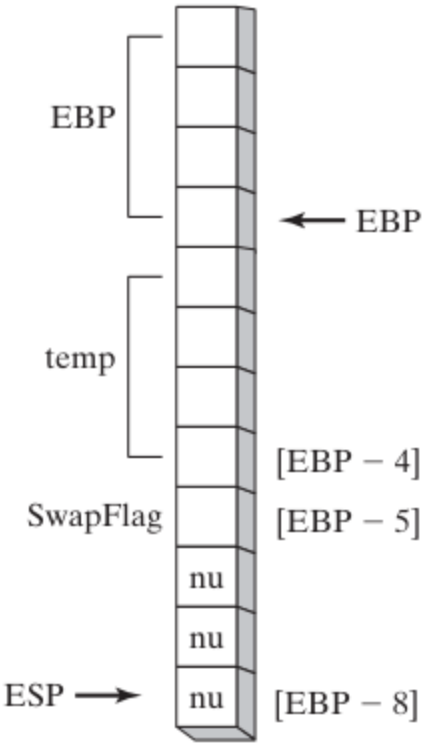
The following diagram shows how the stack looks after the Example1 procedure has been compiled and assembled:

Creating space for local variables (*Example1* Procedure).



The nu blocks represent unused bytes. The following diagram shows how the stack looks after the Example2 procedure has been compiled and assembled:

Creating space in Example 2 for local variables.



The temp variable is a doubleword variable, so it is aligned on a doubleword boundary.

The SwapFlag variable is a byte variable, but it is still allocated four bytes of space on the stack because the stack is aligned on a doubleword boundary.

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 is used to store the local variables and return addresses for all active procedure calls.

For example, in the following code:

```
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
```

The stack size required for Sub1 is 200 bytes, the stack size required for Sub2 is 160 bytes, and the stack size required for Sub3 is 1200 bytes. Therefore, the total stack size required for this code is 1560 bytes.

This stack size is the minimum amount of stack space that must be available in order for this code to execute correctly. If there is not enough stack space available, the program will crash.

Recursive procedure calls

If a procedure is **called recursively**, the stack space it uses will be approximately the size of its local variables and parameters multiplied by the estimated depth of the recursion.

For example, if a procedure has 100 bytes of local variables and parameters, and it is called recursively to a depth of 10, then the procedure will use approximately 1000 bytes of stack space.

Stack overflow

If the stack space required for a program exceeds the amount of stack space available, the program will crash. This is called a **stack overflow.**

To avoid stack overflows, it is important to be aware of the s**tack space requirements of your program.** You can use the **STACK directive** to reserve additional stack space if necessary.

The stack size required for nested procedure calls is the sum of the stack sizes required for each individual procedure call.

If a procedure is called recursively, the stack space it uses will be approximately the size of its local variables and parameters multiplied by the estimated depth of the recursion.

To avoid stack overflows, it is important to be aware of the stack space requirements of your program and to reserve additional stack space, if necessary.

Here is a summary of the key points from the chapter:

- There are two types of procedure parameters: register parameters (faster, used by Irvine libraries) and stack parameters (more flexible).
- A stack frame is the region of stack used by a procedure for its parameters, local variables, saved registers, and return address.
- Parameters can be passed by value (copied) or by reference (address passed). Arrays should be passed by reference.
- Stack parameters are accessed using EBP offset addressing like [EBP-8]. LEA is good for getting stack parameter addresses.
- ENTER/LEAVE instructions manage the stack frame set up/teardown.
- Recursive procedures call themselves directly or indirectly. Recursion works well with repeating data structures.
- Local variables have restricted scope, lifetime tied to the procedure, don't cause naming clashes, and enable recursion.
- INVOKE directive calls procedures with multiple arguments. ADDR passes pointers.
- PROC declares procedures, PROTO prototypes existing procedures.
- Large programs should be split into multiple source code modules for manageability.
- Java bytecode is the machine language in compiled Java programs. The JVM executes it. Bytecodes use a stack-oriented model.