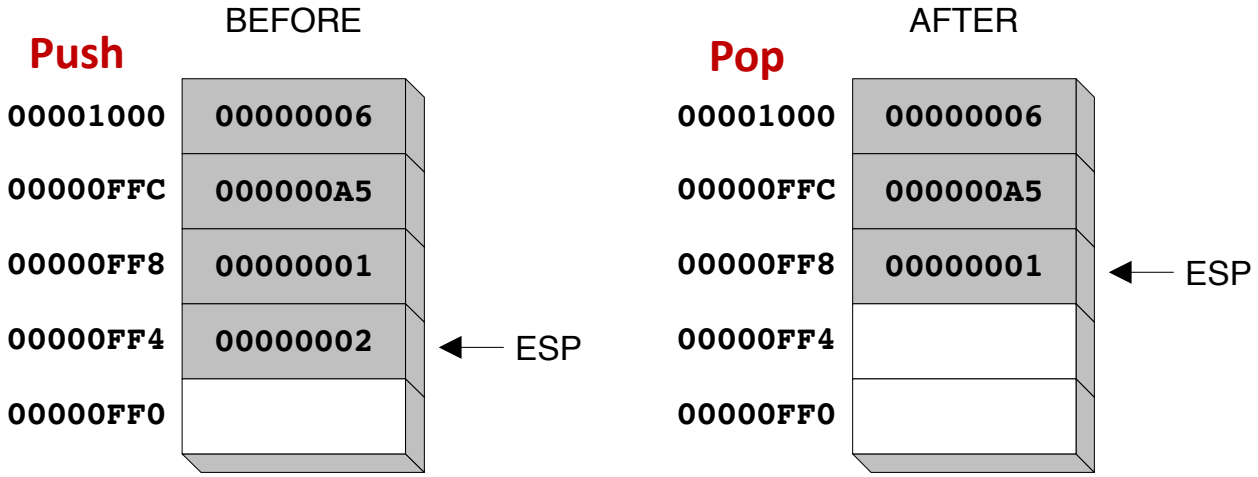
CSC 3210 Computer Organization and Programming

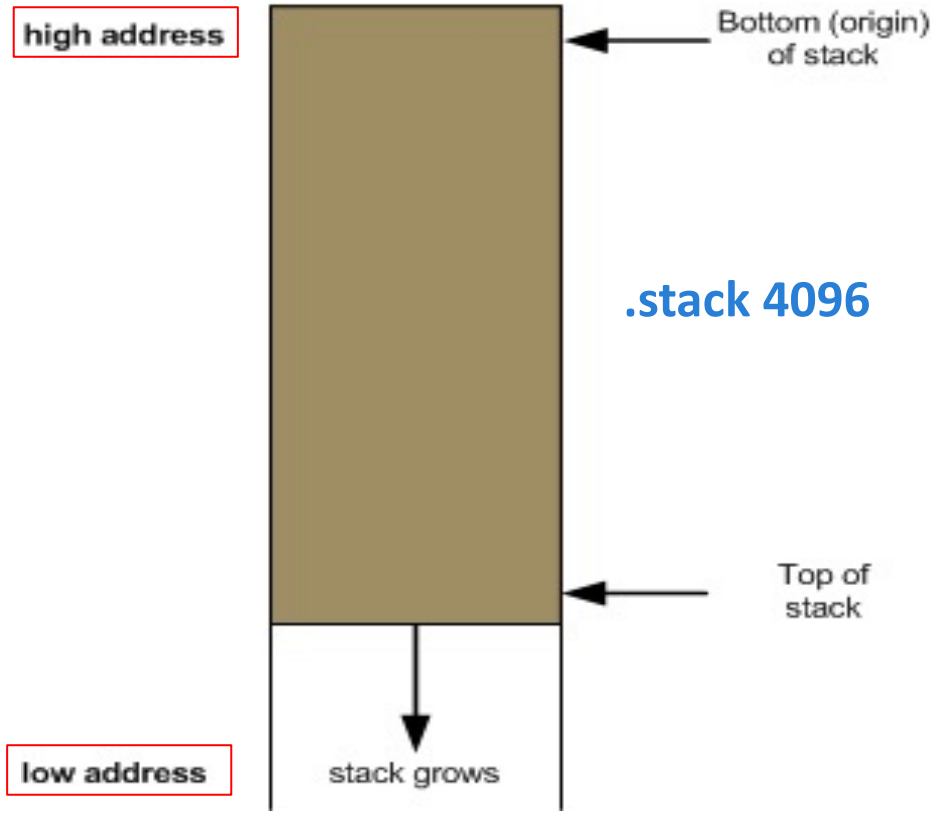
CHAPTER 8: ADVANCED PROCEDURES (STACK FRAME)

Outline

- **Stack Overview**
- Stack Frame
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - Recursion
- INVOKE, ADDR, PROC, and PROTO

Stack Overview



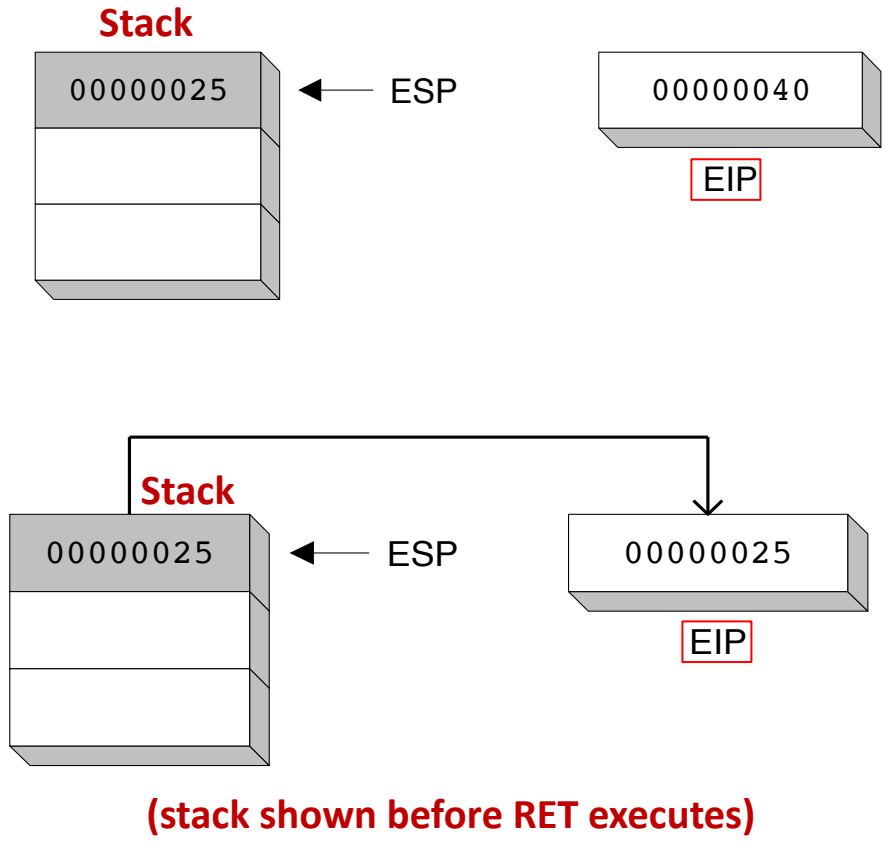


Stack Overview: CALL-RET

The **CALL** instruction **pushes 00000025** onto the **stack**, and **loads 00000040** into **EIP**

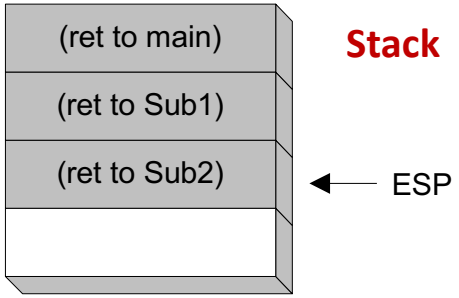
The **RET** instruction **pops 00000025** from the **stack** into **EIP**

```
main PROC
     00000020 call MySub
     00000025 mov eax,ebx
     .
     .
main ENDP

MySub PROC
     00000040 mov eax,edx
     .
     .
     ret
MySub ENDP
```

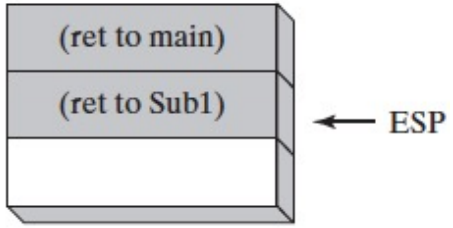


Stack Overview: Nested Calls

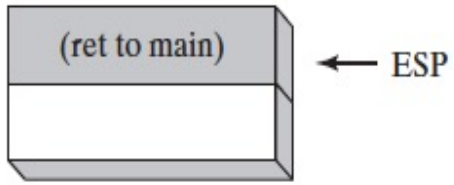
1. By the time **Sub3** is called, **the stack** contains all **three** return addresses (**main, sub1, sub2**):



2. After the **return**,
   - **ESP** points to the next-highest stack entry.



3. Finally, when **Sub1** **returns**, **stack[ESP]** is **popped** into the instruction pointer, and execution resumes in **main**:




```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

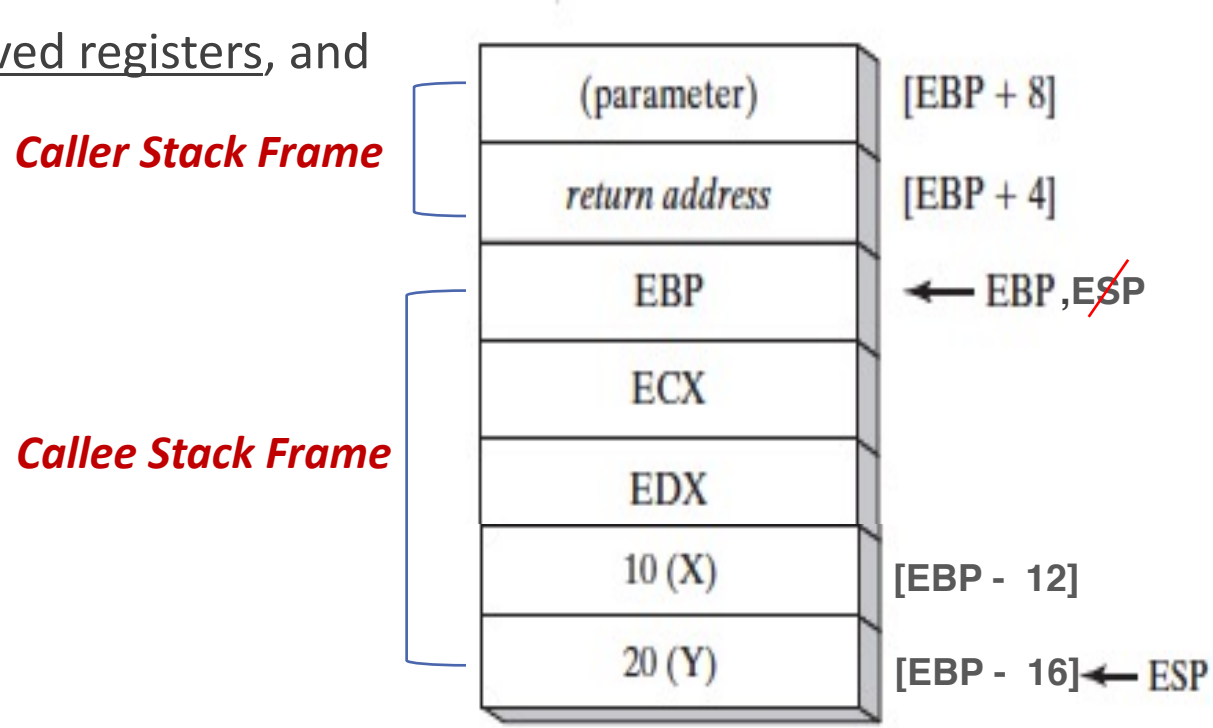
Outline

- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - Recursion
- INVOKE, ADDR, PROC, and PROTO

Stack Frame

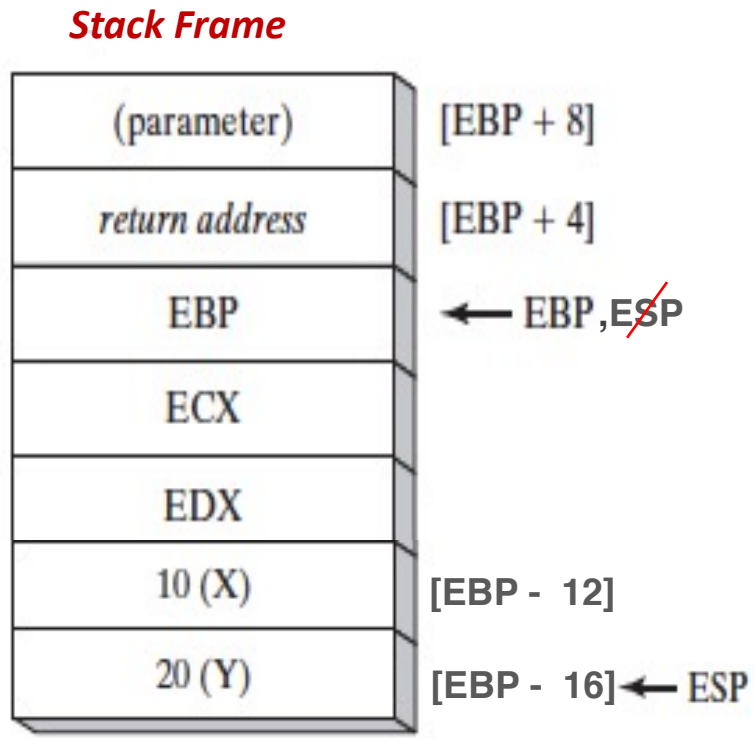
- Stack ***Frame***, also known as an ***activation record***
  - **Area of the stack** set aside for a procedure's:
    - passed parameters, return address, saved registers, and local variables

- The **structure of a stack frame** is directly affected by a **program's memory model** & its **argument passing convention**



Stack Frame: ESP and EBP

- **ESP**: a register that contains the address of **the top of the stack**
- **EBP:** a register that points at the top of the stack **when** the function is first called
- These two registers are **the heart of the x86 function-call mechanism**
- Why not use just the **ESP**?

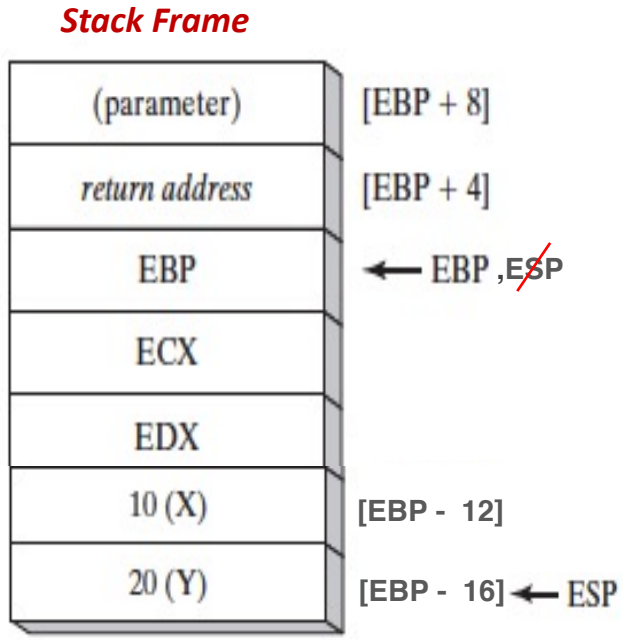


Stack Frame: EBP (Why EBP?)

- Function **arguments** and **local variables** are **stored** at **an offset** from where the **ESP** is when the function starts

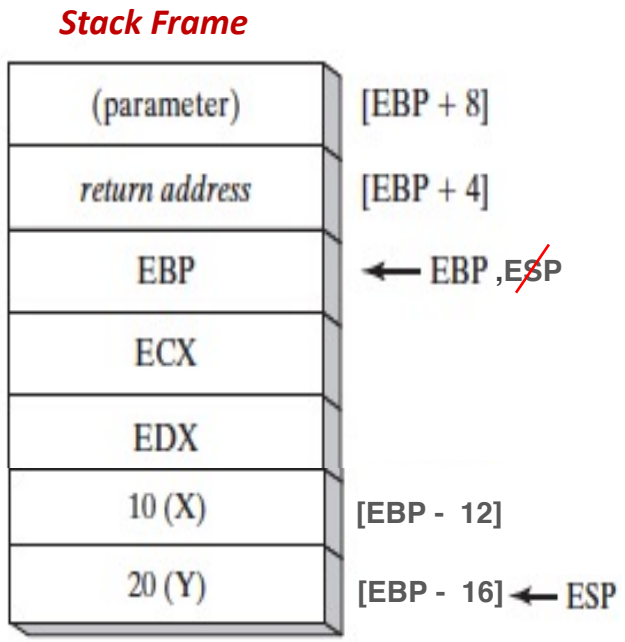
- As **ESP** grows and shrinks during a function calls:
  - the **offset of local variables** and **function parameters** relative to **ESP** **changes**

- To simplify things, x86 uses a **base pointer** (sometimes called a **frame pointer**) that is stored in **EBP**.



Stack Frame: EBP (Why EBP?)

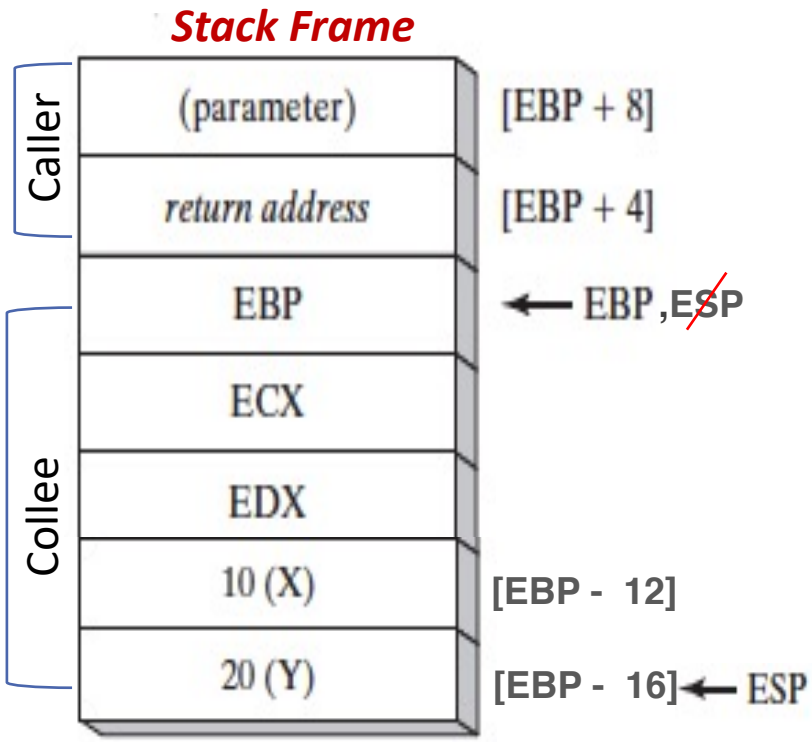
- **EBP** was designed to provide a "Base Pointer" for the **current function** so **parameters** and **local variables** would be **at a fixed offset from the base pointer even if ESP moved** with **push** and **pop** instructions



Stack Frame: Creation

- **Created** by the following steps:

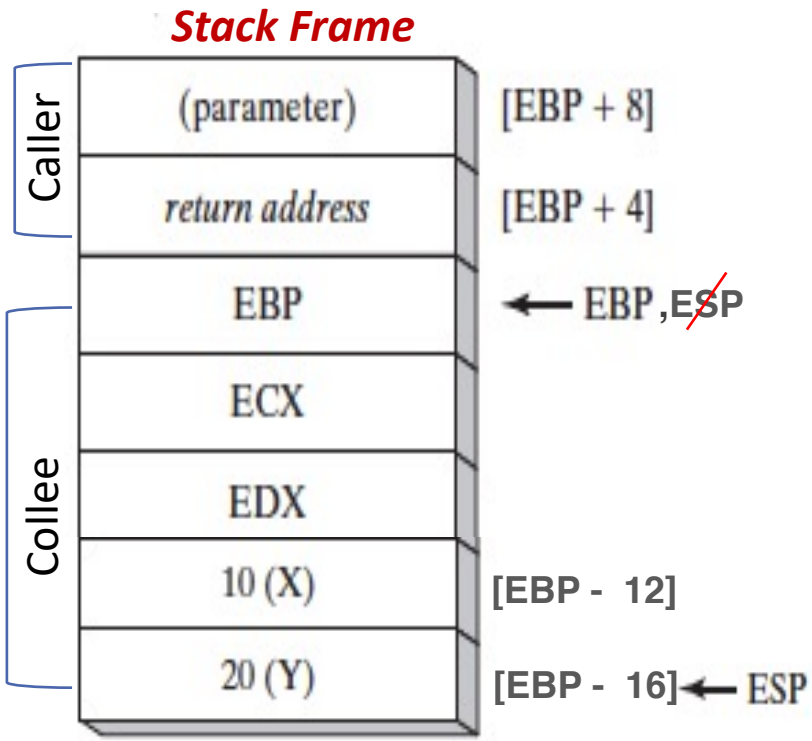
  1. **Calling program pushes**
     - ***arguments*, if any, and**
     - ***return address*** on the stack,
     - and **calls** the procedure
  2. The **called procedure pushes**
     - **EBP** on the stack, and
     - **sets EBP** to **ESP** [from this point on, EBP acts as a **base reference** for all the subroutine parameters]



Stack Frame: Creation

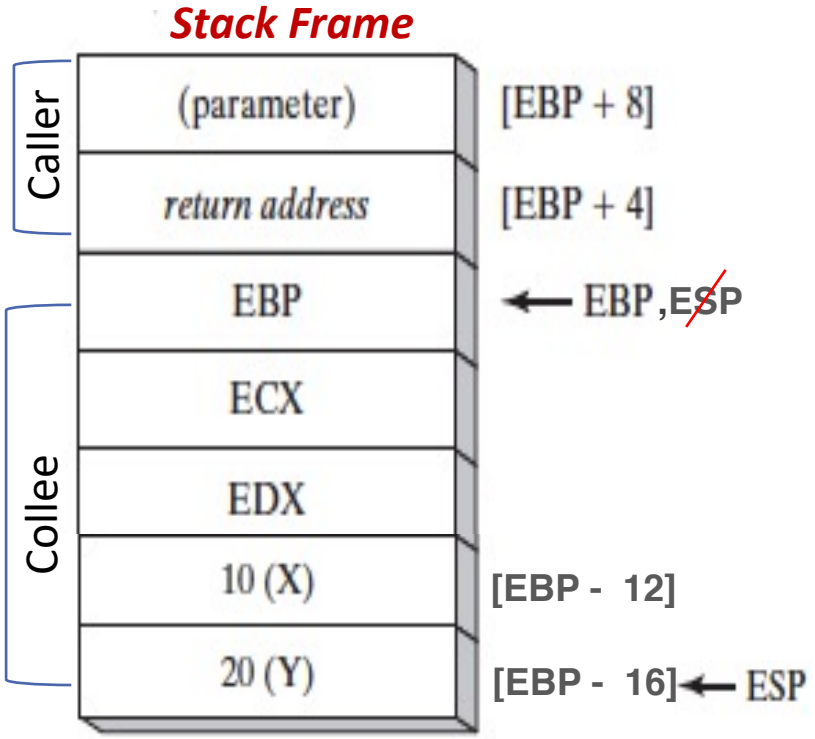
- **Created** by the following steps:

3. If any **registers** are needed,
   - they are **pushed** on the stack
4. If **local variables** are needed,
   - a constant is **subtracted** from **ESP** to make room on the stack



Outline

- Stack Overview
- **Stack Frame**
  - **Stack Parameters**
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - Recursion



Stack Frame: Stack Parameters (Why?)

- The procedures we have used so far **incorporate** the **general purpose registers** to pass **parameters** to one another:

```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
popad
```

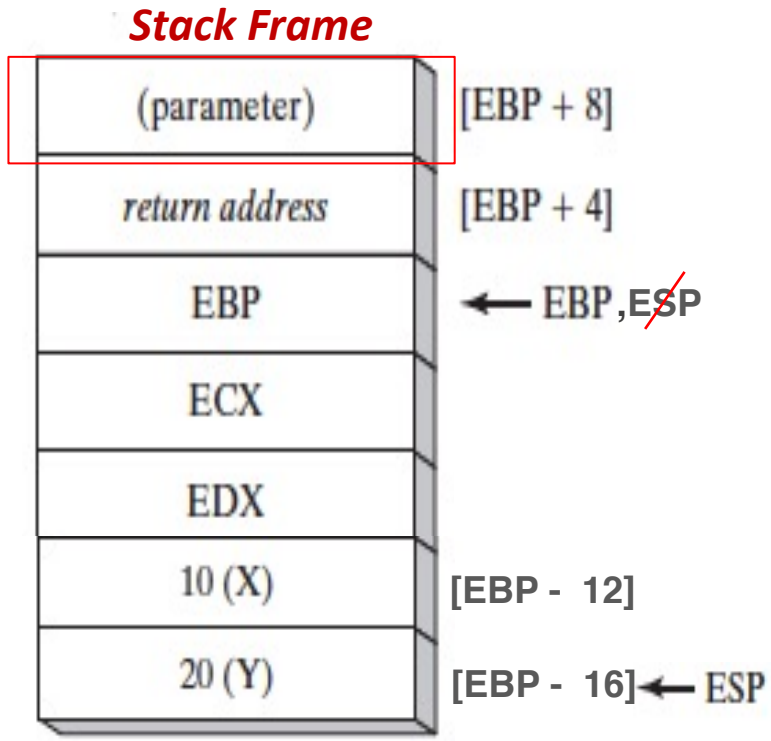
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

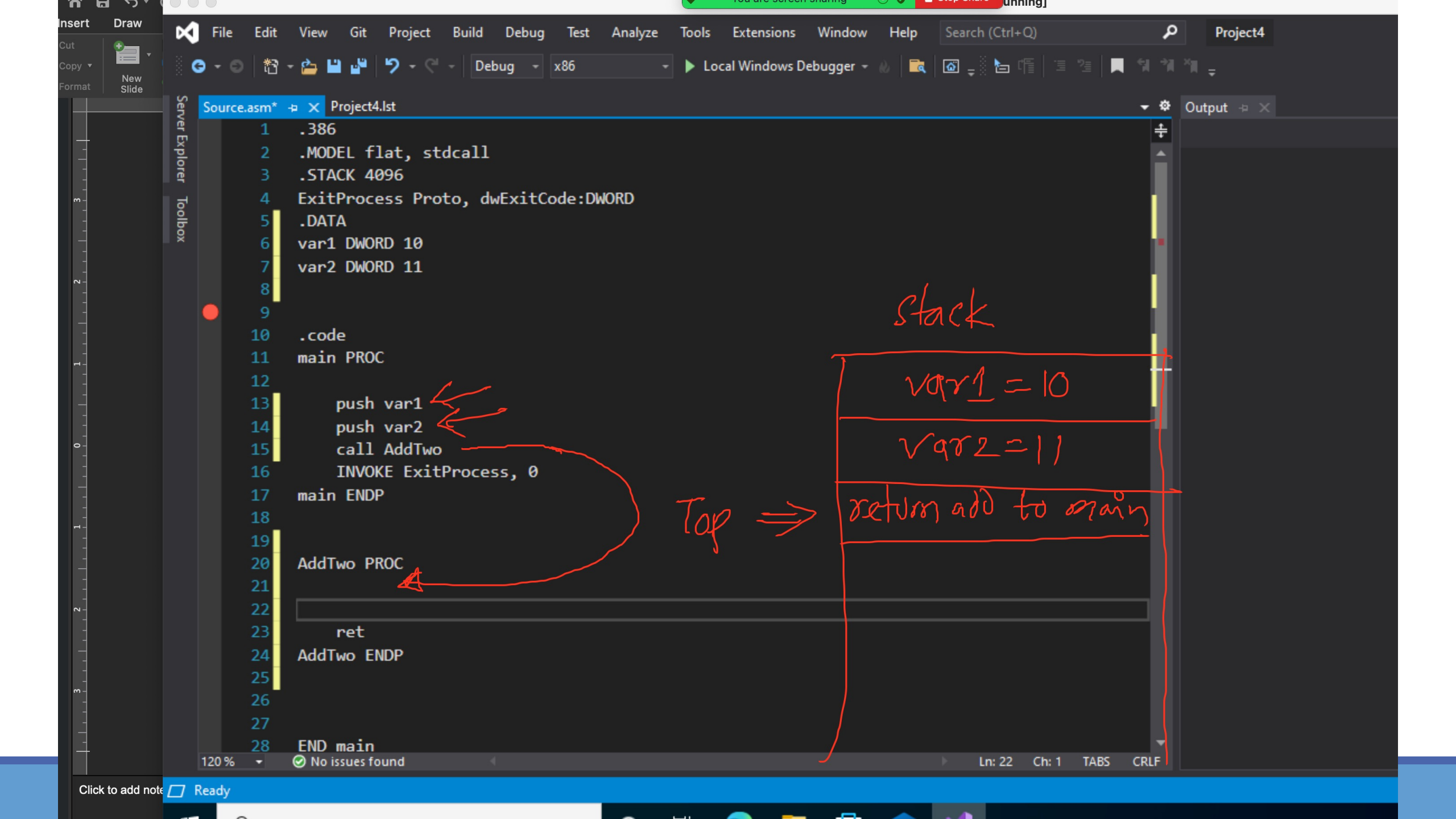
- This is okay **except** when you create subroutines that require **several** (**> 8,** 😟) parameters

- In this case you will need to **use the stack to pass the parameters**

Outline

- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - **Passing Arguments**
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - Recursion




File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Project4
Debug
x86
Local Windows Debugger
Source.asm*
Project4.lst
Output
.386
.MODEL flat, stdcall
.STACK 4096
ExitProcess Proto, dwExitCode:DWORD
.DATA
var1 DWORD 10
var2 DWORD 11
.code
main PROC
push var1
push var2
call AddTwo
INVOKE ExitProcess, 0
main ENDP
AddTwo PROC
ret
AddTwo ENDP
END main
stack
var1 = 10
var2 = 11
return add to main
Top
120 %
No issues found
Ln: 22
Ch: 1
TABS
CRLF
Ready

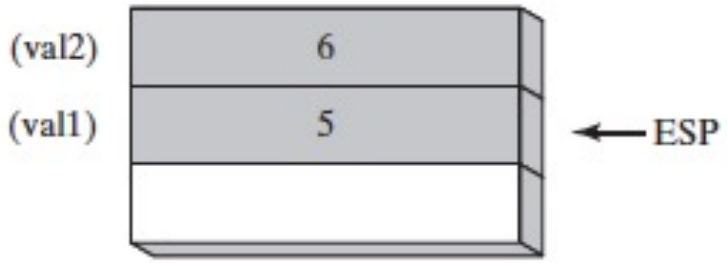
Stack Frame: Passing Arguments

- **Two general types of arguments** are pushed on the stack during subroutine calls:
  - **Value arguments** (values of variables and constants),
  - **Reference arguments** (addresses of variables)

1) Passing by Value

- When an argument is passed by value, **a copy of the value is pushed on the stack**

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

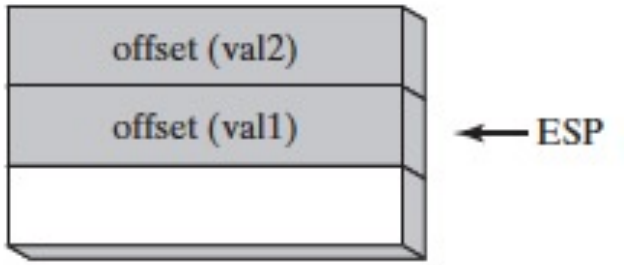


Stack Frame: Passing Arguments

2) Passing by Reference

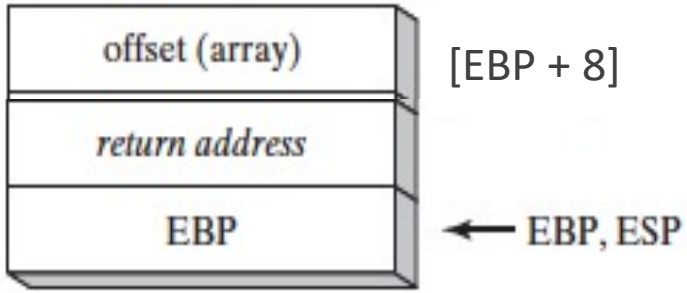
- An argument passed by reference **consists of the address (offset)** of an object
- The following statements call **Swap**, passing the two arguments by reference:

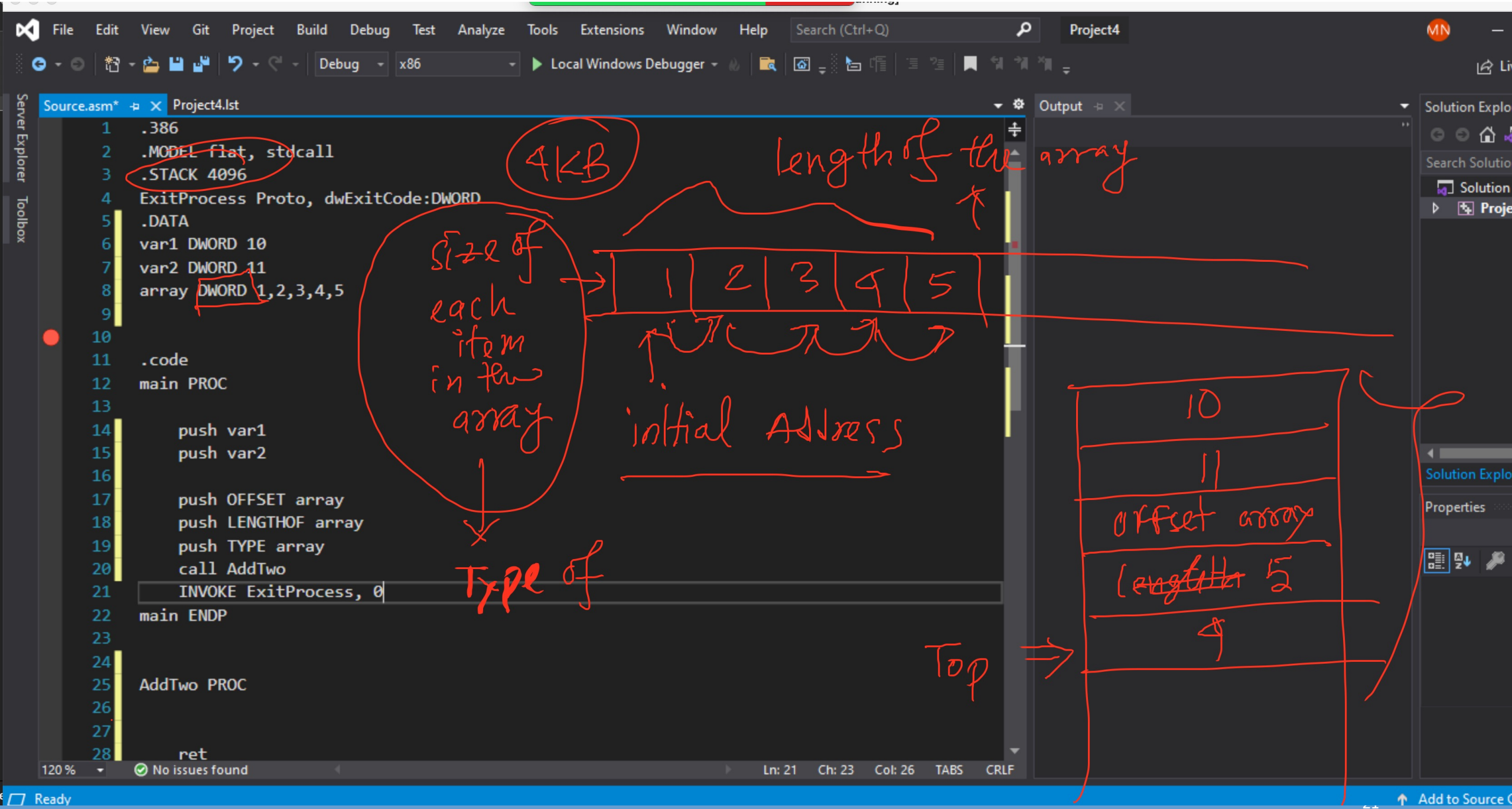
```
push   OFFSET val2
push   OFFSET val1
call   Swap
```



- **Passing Arrays:**

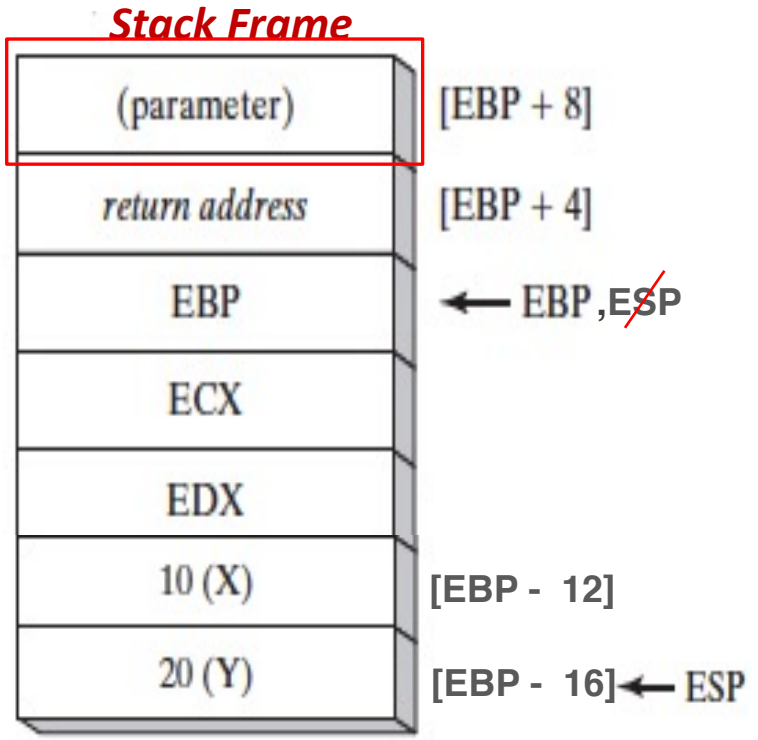
```
.data
array  DWORD 50 DUP(?)
.code
push   OFFSET array
call   ArrayFill
```




File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Project4
Debug
x86
Local Windows Debugger
Source.asm*
Project4.lst
Output
Solution Explorer
Toolbox
.386
.MODEL flat, stdcall
.STACK 4096
ExitProcess Proto, dwExitCode:DWORD
.DATA
var1 DWORD 10
var2 DWORD 11
array DWORD 1,2,3,4,5
.code
main PROC
push var1
push var2
push OFFSET array
push LENGTHOF array
push TYPE array
call AddTwo
INVOKE ExitProcess, 0
main ENDP
AddTwo PROC
ret
4KB
length of the array
size of each item in the array
initial Address
Type of
Top
10
11
offset array
length 5
4
120 %
No issues found
Ln: 21
Ch: 23
Col: 26
TABS
CRLF
Ready
Add to Source

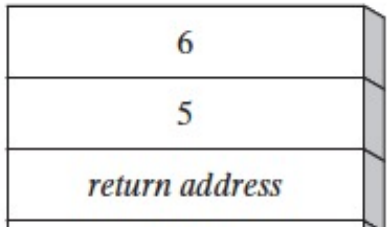
Outline

- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - Passing Arguments
  - **Accessing Stack Parameters**
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - Recursion

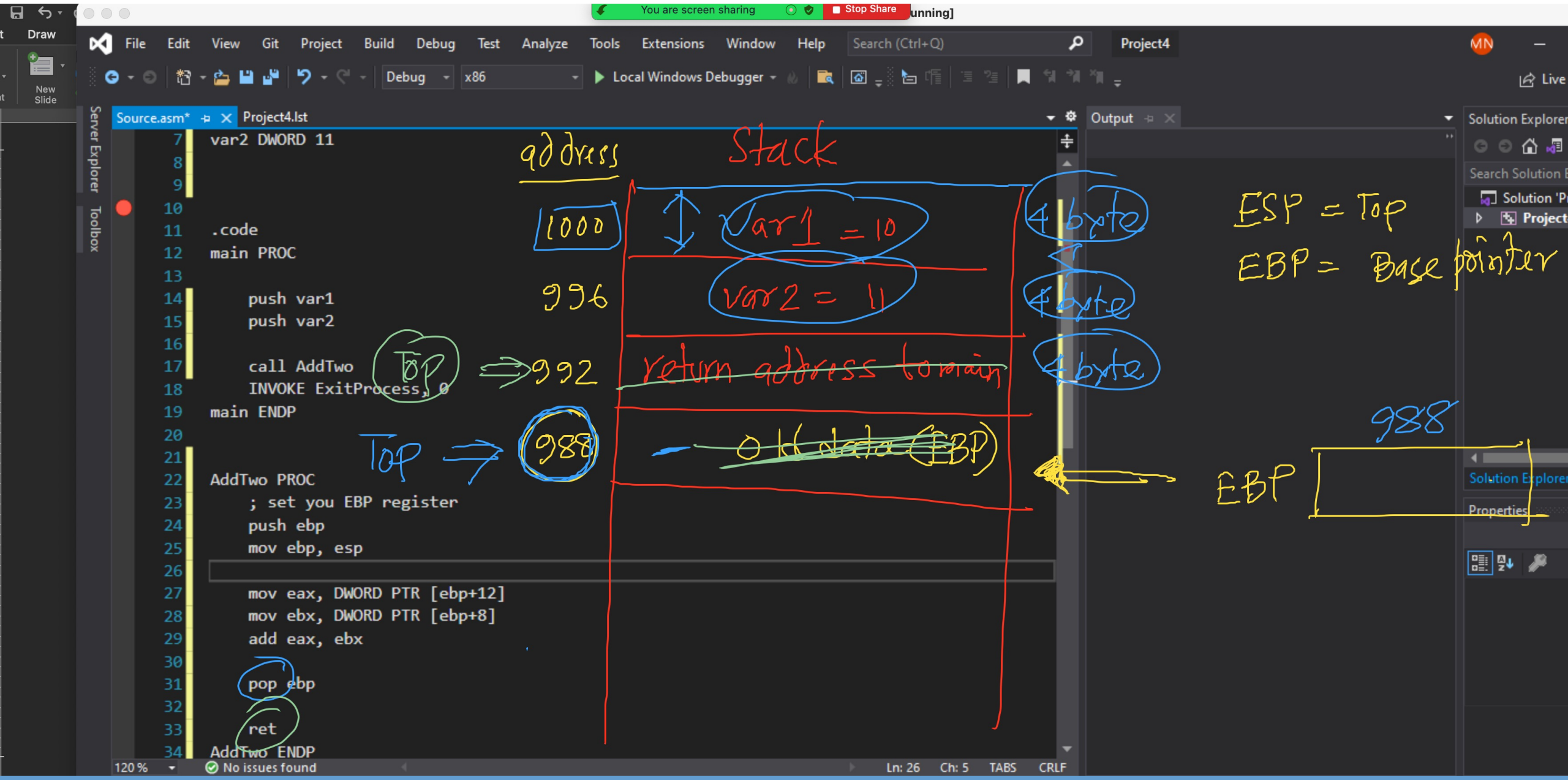


Stack Frame: Accessing Stack Parameters

- **Base-Offset Addressing**
  - **EBP** is the base register and **the offset is a constant**.
  - A **function call** such as **AddTwo(5, 6)** would cause the **second parameter** to be pushed on the stack, followed by the **first parameter**:



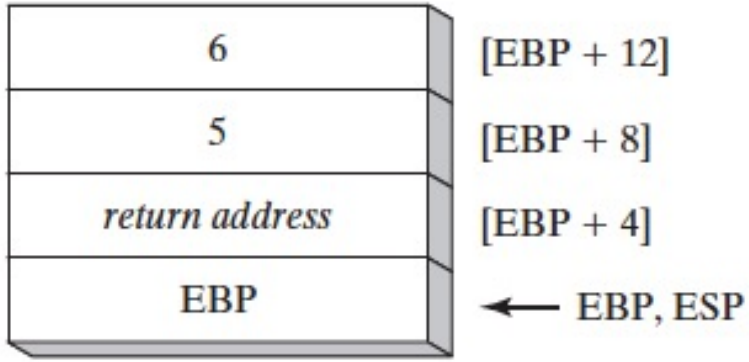
  - 32-bit values are usually returned in **EAX**


You are screen sharing
Stop Share
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Project4
Debug
x86
Local Windows Debugger
Source.asm*
Project4.lst
Output
Solution Explorer
Server Explorer
Toolbox
var2 DWORD 11
.code
main PROC
push var1
push var2
call AddTwo
INVOKE ExitProcess, 0
main ENDP
AddTwo PROC
; set you EBP register
push ebp
mov ebp, esp
mov eax, DWORD PTR [ebp+12]
mov ebx, DWORD PTR [ebp+8]
add eax, ebx
pop ebp
ret
AddTwo ENDP
address
Stack
1000
var1 = 10
996
var2 = 11
Top
992
return address to main
Top
988
old data (EBP)
4 byte
4 byte
4 byte
ESP = Top
EBP = Base pointer
988
EBP
120 %
No issues found
Ln: 26
Ch: 5
TABS
CRLF

Stack Frame: Accessing Stack Parameters

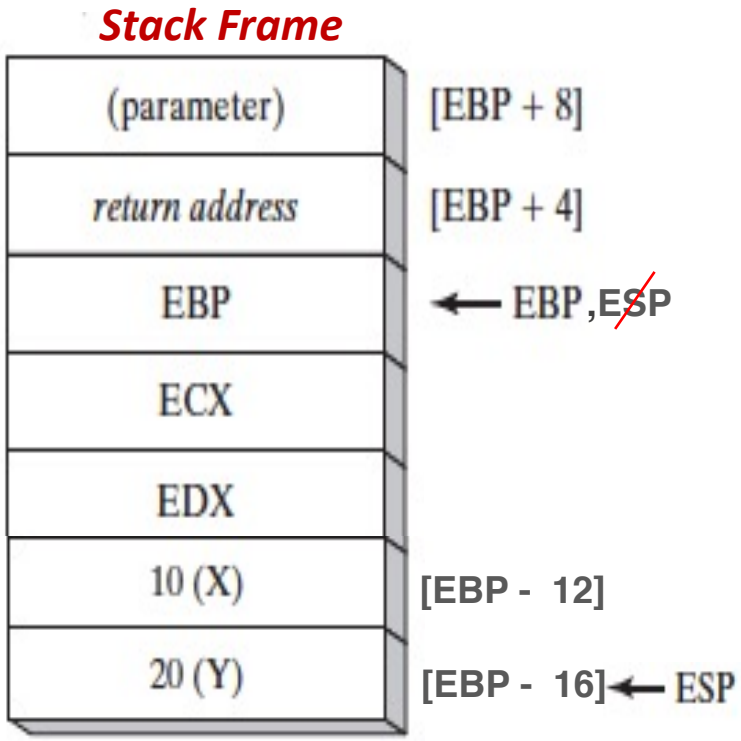
- **Base-Offset Addressing**

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

AddTwo PROC
       push  ebp
       mov   ebp,esp                ; base of stack frame
       mov   eax,[ebp + 12]         ; second parameter
       add   eax,[ebp + 8]          ; first parameter
       pop   ebp
       ret
AddTwo ENDP
```



Outline

- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - **Cleaning Up the Stack**
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - Recursion

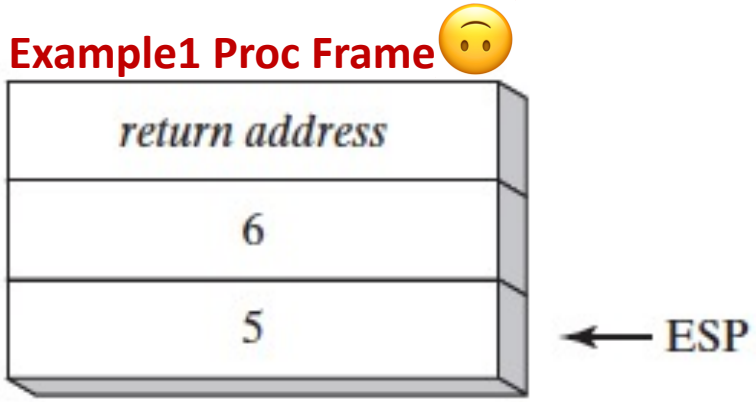


Stack Frame: Cleaning Up the Stack

- There must be **a way for parameters to be removed from the stack** when a subroutine returns
  - Otherwise, **a memory leak would result**, and the **stack would become corrupted**.

```
main PROC
        call Example1
        exit
main ENDP

Example1 PROC
        push 6
        push 5
        call AddTwo
        ret
Example1 ENDP
```

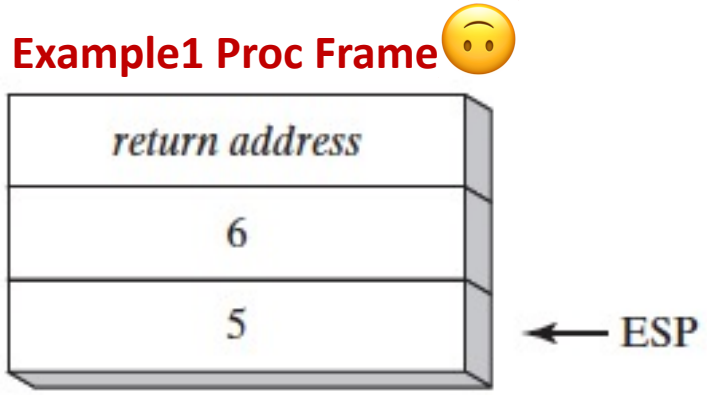
```
AddTwo PROC
    push  ebp
    mov   ebp,esp            ; base of stack frame
    mov   eax,[ebp + 12]     ; second parameter
    add   eax,[ebp + 8]      ; first parameter
    pop   ebp
    ret
AddTwo ENDP
```



Stack Frame: Cleaning Up the Stack

- There must be **a way for parameters to be removed from the stack** when a subroutine returns
  - Otherwise, **a memory leak would result**, and the **stack would become corrupted**.

```
main PROC
        call Example1
        exit
main ENDP

Example1 PROC
        push 6
        push 5
        call AddTwo
        ret             ; stack is corrupted!
Example1 ENDP
```



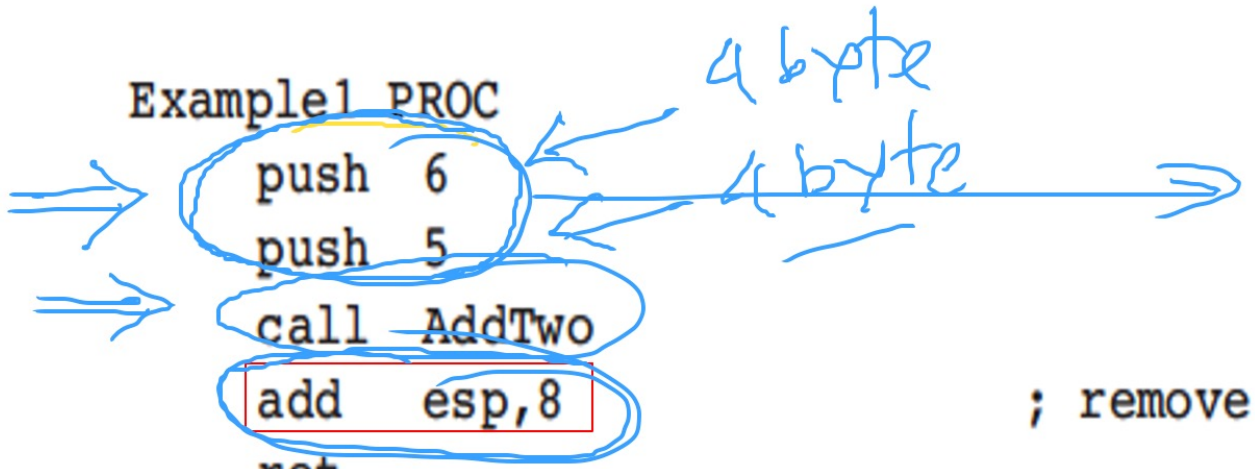
Stack Frame: Cleaning Up the Stack

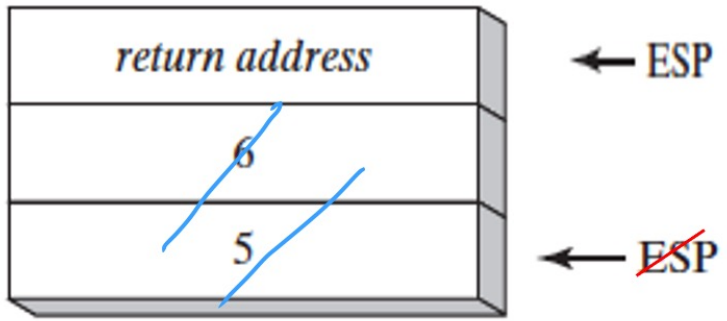
- The **solution** is to use one of the **calling conventions**:
  - **The C Calling Convention**
  - **STDCALL Calling Convention**

Stack Frame: Cleaning Up the Stack

1) **The C Calling Convention:**
   - Follow **CALL** instruction with a statement that **adds** a value to the ESP equal to the combined sizes of the subroutine **parameters**

```
Example1 PROC
    push  6
    push  5
    call  AddTwo
    add   esp,8
    ret
Example1 ENDP
```






```
; remove arguments from the stack
```

Stack Frame: Cleaning Up the Stack

2) STDCALL Calling Convention

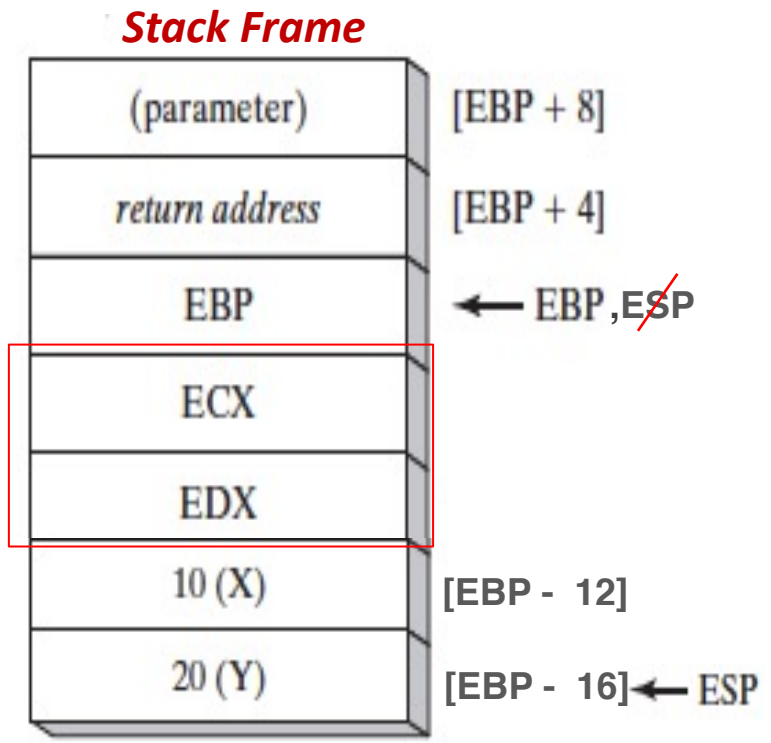
- It supplies an **integer parameter** to the RET instruction
- The **integer** must equal the number of bytes of stack space consumed by the procedure's parameters:

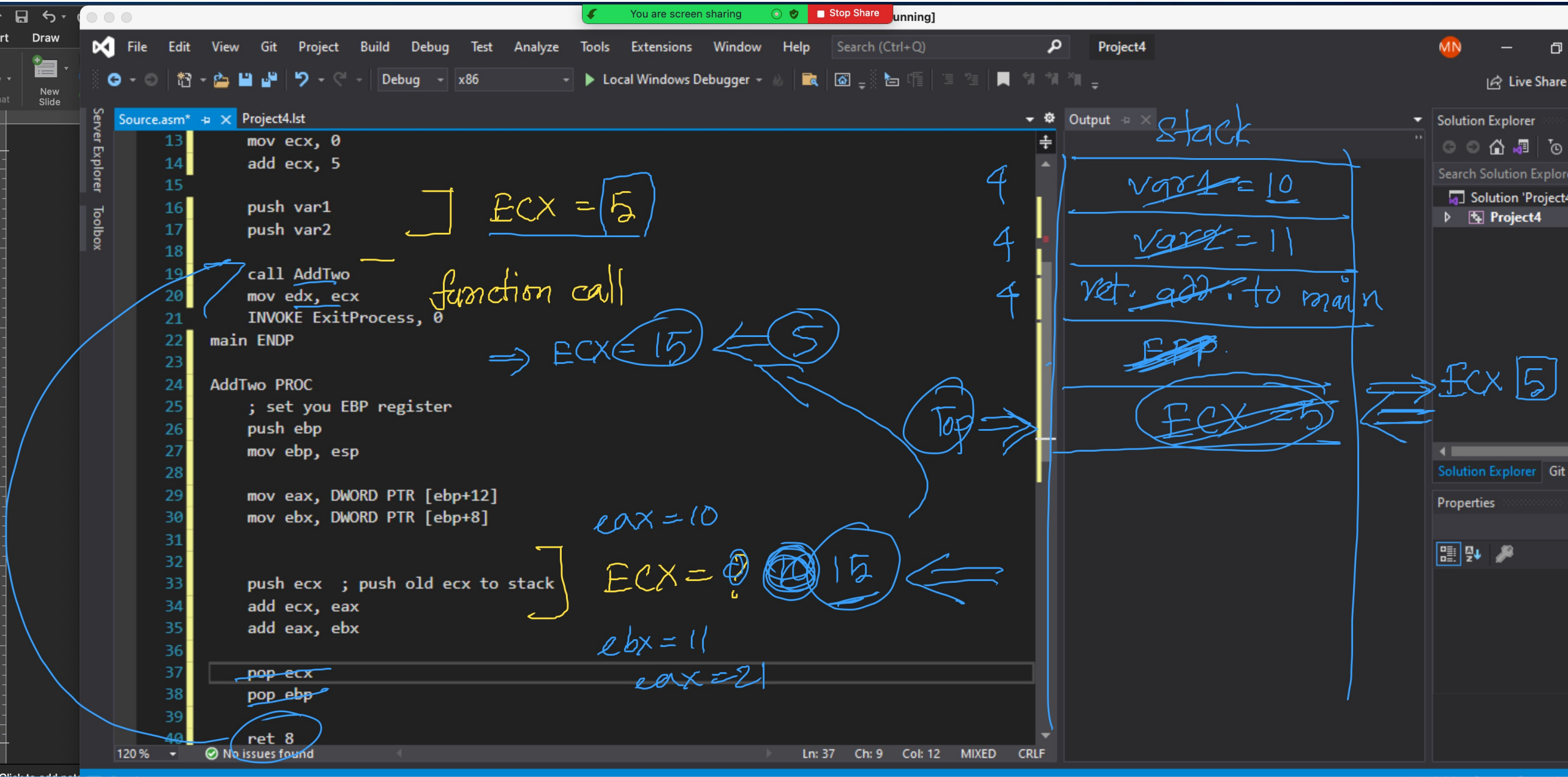
```
AddTwo PROC
     push  ebp
     mov   ebp,esp                ; base of stack frame
     mov   eax,[ebp + 12]         ; second parameter
     add   eax,[ebp + 8]          ; first parameter
     pop   ebp
     ret   8                      ; clean up the stack
AddTwo ENDP
```

STDCALL, like C, pushes arguments onto the stack **in reverse order**

Outline

- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - **Saving and Restoring Registers**
  - Local Variables
  - Reference Parameters
  - Recursion

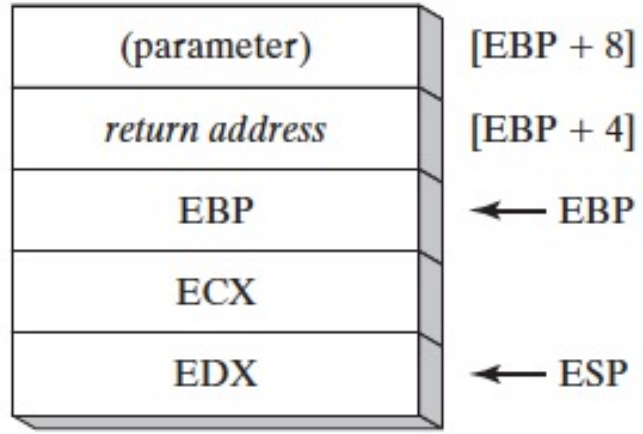



You are screen sharing
Stop Share
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
Project4
Debug
x86
Local Windows Debugger
Live Share
Source.asm*
Project4.lst
Output
Solution Explorer
Server Explorer
Toolbox
13 mov ecx, 0
14 add ecx, 5
15
16 push var1
17 push var2
18
19 call AddTwo
20 mov edx, ecx
21 INVOKE ExitProcess, 0
22 main ENDP
23
24 AddTwo PROC
25 ; set you EBP register
26 push ebp
27 mov ebp, esp
28
29 mov eax, DWORD PTR [ebp+12]
30 mov ebx, DWORD PTR [ebp+8]
31
32
33 push ecx ; push old ecx to stack
34 add ecx, eax
35 add eax, ebx
36
37 pop ecx
38 pop ebp
39
40 ret 8
ECX = 5
function call
ECX = 15
5
Top
eax = 10
ebx = 11
eax = 21
stack
var1 = 10
var2 = 11
ret. add. to main
EBP
ECX = 5
4
4
4
ECX 5
120 %
No issues found
Ln: 37 Ch: 9 Col: 12 MIXED CRLF

Stack Frame: Saving and Restoring Registers

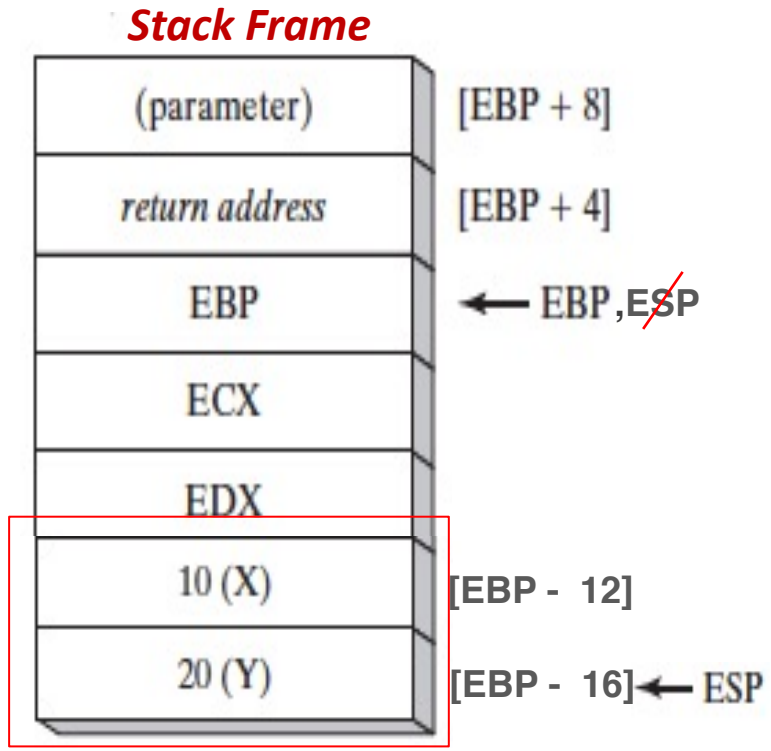
- Registers in question **should be pushed on the stack (Rules)**
  - **just after setting EBP to ESP, and just before reserving space for local variables**
  - This helps to avoid changing offsets of existing stack parameters.

```
MySub PROC
     push   ebp                  ; save base pointer
     mov    ebp,esp              ; base of stack frame
     push   ecx
     push   edx                  ; save EDX
     mov    eax,[ebp+8]          ; get the stack parameter
     .
     .
     pop    edx                  ; restore saved registers
     pop    ecx
     pop    ebp                  ; restore base pointer
     ret                         ; clean up the stack
MySub ENDP
```



Outline

- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - **Local Variables**
  - Reference Parameters
  - Recursion



Stack Frame: Local Variables

- **Local variables** are **created** on the runtime stack, usually **below the base pointer (EBP)**
- **They cannot be assigned default values at assembly time**, BUT can be **initialized at runtime**
- **EXAMPLE1:**

```
void MySub()
{
    int X = 10;
    int Y = 20;
}
```

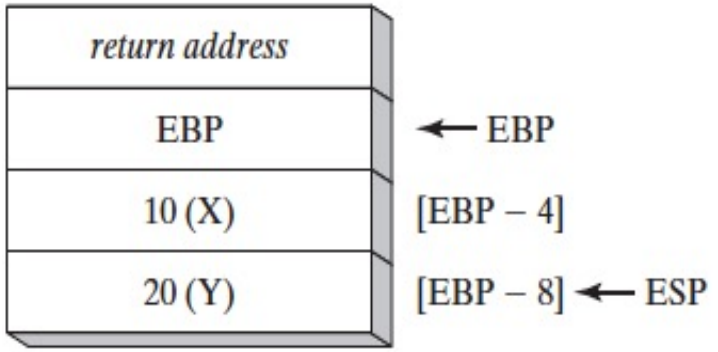
- **Cannot be assigned default values at assembly time**
- **initialized at runtime**

Variable	Bytes	Stack Offset
X	4	EBP – 4
Y	4	EBP – 8

Stack Frame: Local Variables

- **EXAMPLE2:**

```
MySub PROC
     push  ebp
     mov   ebp,esp
     sub   esp,8                        ; create locals

     mov   DWORD PTR [ebp-4],10         ; X
     mov   DWORD PTR [ebp-8],20         ; Y
     mov   esp,ebp                      ; remove locals from stack
     pop   ebp
     ret
MySub ENDP
```

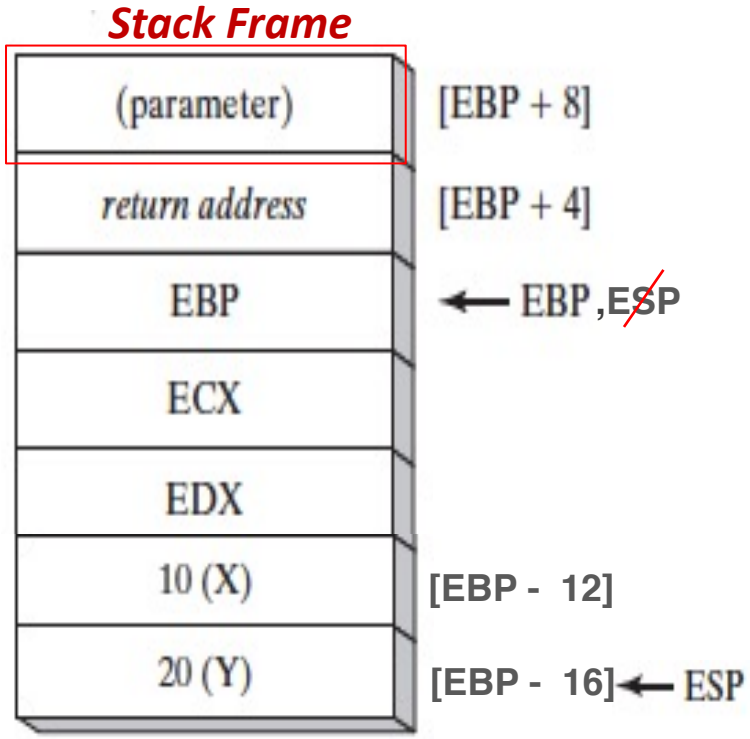


What if you did not have this

Attendance

Outline

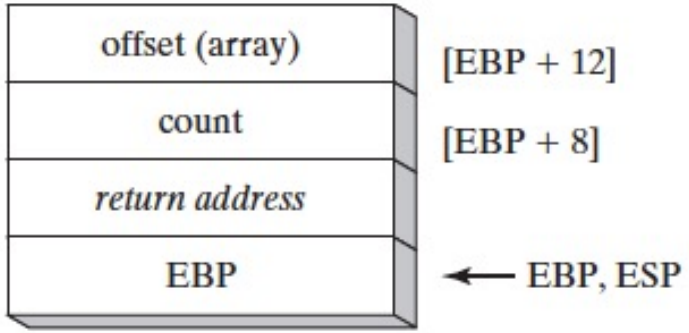
- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - **Reference Parameters**
  - Recursion



Stack Frame: Reference Parameters

- **Reference parameters** are usually accessed by procedures **using base-offset addressing** (from **EBP**)
- Because each reference parameter is a pointer, it is usually loaded into a register as **an indirect operand**
- **EXAMPLE1**:
  - Suppose that **a pointer to an array** is located at **stack address [ebp12]**
  - The following statement copies the pointer into **ESI**:

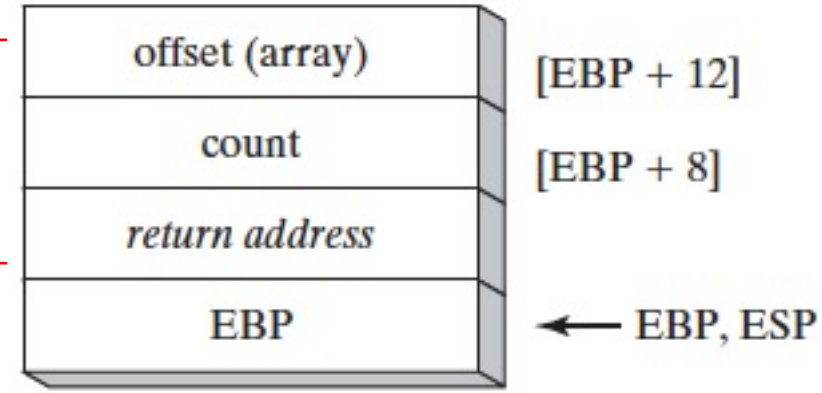
```
mov esi,[ebp+12]        ; points to the array
```



Stack Frame: Reference Parameters

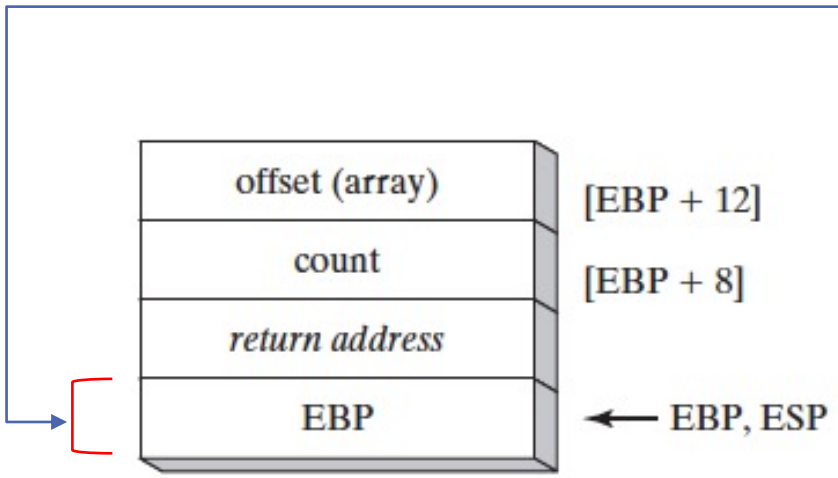
- **EXAMPLE2**:

```
.data
count = 100
array WORD count DUP(?)

.code
push   OFFSET array
push   count
call   ArrayFill
```



Stack Frame: Reference Parameters

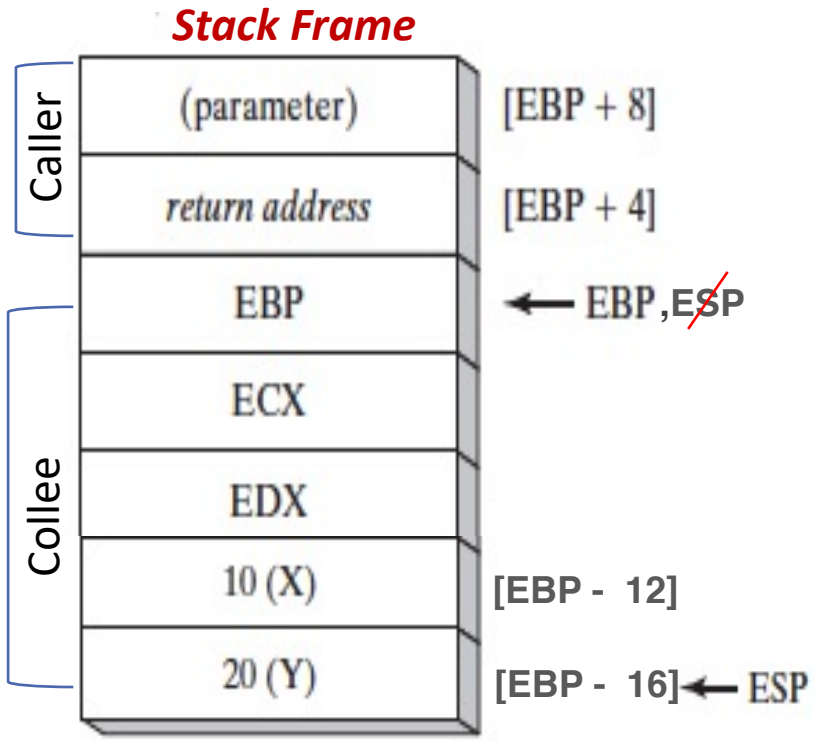
- **EXAMPLE2**:




```
ArrayFill PROC
    push   ebp
    mov    ebp,esp
    pushad                          ; save registers
    mov    esi,[ebp+12]             ; offset of array
    mov    ecx,[ebp+8]              ; array length
    cmp    ecx,0                    ; ECX == 0?
    je     L2                       ; yes: skip over loop
L1:
    mov    eax,10000h               ; get random 0 - FFFFh
    call   RandomRange              ; from the link library
    mov    [esi],ax                 ; insert value in array
    add    esi,TYPE WORD            ; move to next element
    loop   L1
L2: popad                           ; restore registers
    pop    ebp
    ret    8                        ; clean up the stack
ArrayFill ENDP
```

Outline

- Stack Overview
- **Stack Frame**
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - **Recursion**

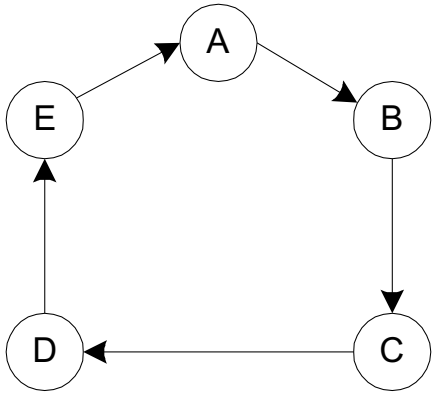


Recursion

- **A recursive** subroutine is **one that calls itself**, either directly or indirectly.

  The process created when . . .
  - A procedure calls itself
  - Procedure A calls procedure B, which in turn calls procedure A

- Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a cycle:



Recursion

- **Endless Recursion:**

```
; Endless Recursion                    (Endless.asm)

INCLUDE Irvine32.inc
.data
endlessStr BYTE "This recursion never stops",0
.code
main PROC
     call  Endless
     exit
main ENDP

Endless PROC
     mov   edx,OFFSET endlessStr
     call  WriteString
     call  Endless
     ret                              ; never executes
Endless ENDP
END main
```

- The **RET** instruction **is never executed**
- The program halts when the **stack overflows**

Recursion with break condition

Recursion: Recursively Calculating a Factorial

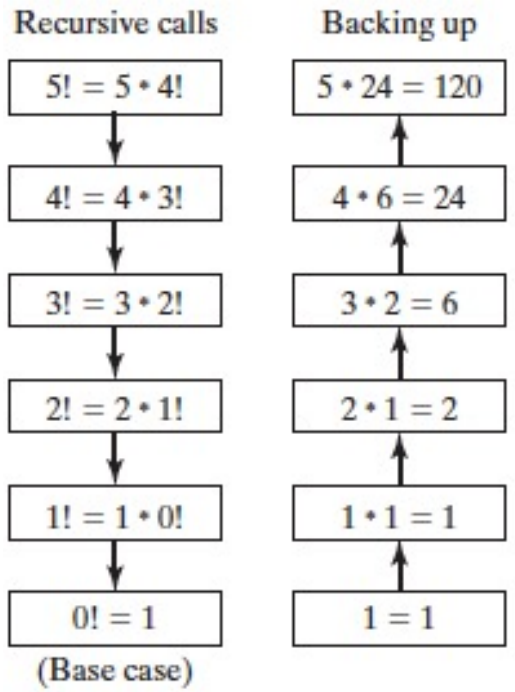
- Recursive subroutines often **store temporary data in stack parameters**
- When the **recursive calls unwind**, the **data saved** on **the stack** can be useful
- **EXAMPLE:**
    - **calculate 5!**

```
int function factorial(int n)
{
      if(n == 0)
        return 1;
      else
        return n * factorial(n-1);
}
```

Recursion: Recursively Calculating a Factorial

- Recursive subroutines often **store temporary data in stack parameters**
- When the **recursive calls unwind**, the **data saved** on **the stack** can be useful
- **EXAMPLE:**
  - **calculate 5!**

```
int function factorial(int n)
{
      if(n == 0)
        return 1;
      else
        return n * factorial(n−1);
}
```



Recursion: Recursively Calculating a Factorial

- **Let's examine** the Factorial procedure more closely by tracking a call to it with an initial value of N = 3.

```
; Calculating a Factorial (Fact.asm)

INCLUDE Irvine32.inc
.code
main PROC
    push  3                    ; calc 3!
    call  Factorial            ; calculate factorial (EAX)
    call  WriteDec             ; display it
    call  Crlf
    exit
main ENDP
```

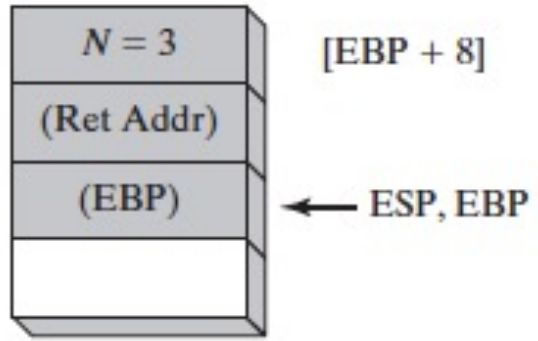
```
int function factorial(int n)
{
     if(n == 0)
       return 1;
     else
       return n * factorial(n-1);
}
```

```
;-----------------------------------------------------
Factorial PROC
; Calculates a factorial.
; Receives: [ebp+8] = n, the number to calculate
; Returns: eax = the factorial of n
;-----------------------------------------------------
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+8]          ; get n
    cmp   eax,0                ; n > 0?
    ja    L1                   ; yes: continue
    mov   eax,1                ; no: return 1 as the value of 0!
    jmp   L2                   ; and return to the caller

L1: dec   eax
    push  eax                  ; Factorial(n-1)
    call  Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov   ebx,[ebp+8]          ; get n
    mul   ebx                  ; EDX:EAX = EAX * EBX

L2: pop   ebp                  ; return EAX
    ret   4                    ; clean up stack
Factorial ENDP
END main
```

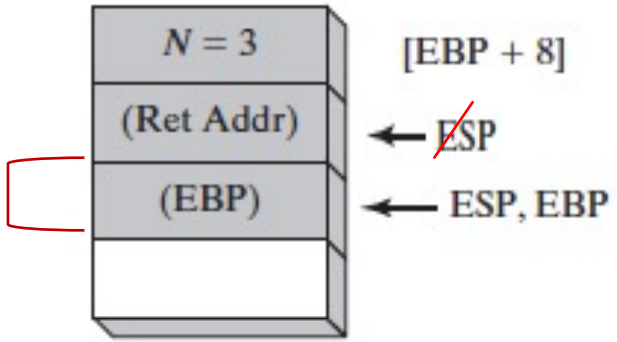
Recursion: Recursively Calculating a Factorial

- **Let's examine** the Factorial procedure more closely by tracking a call to it with an initial value of N = 3.

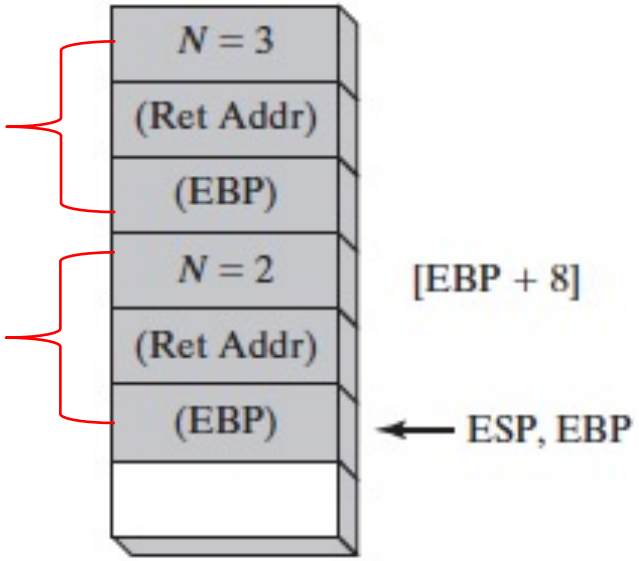
```
; Calculating a Factorial (Fact.asm)

INCLUDE Irvine32.inc
.code
main PROC
   push  3                    ; calc 3!
   call  Factorial            ; calculate factorial (EAX)
   call  WriteDec             ; display it
   call  Crlf
   exit
main ENDP
```



1

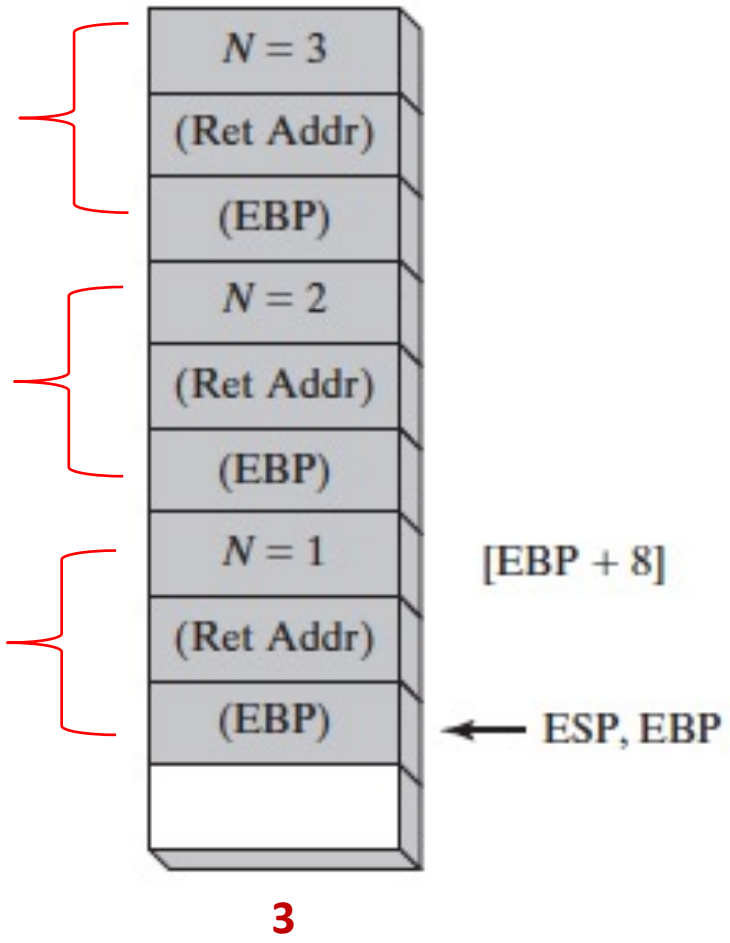


1



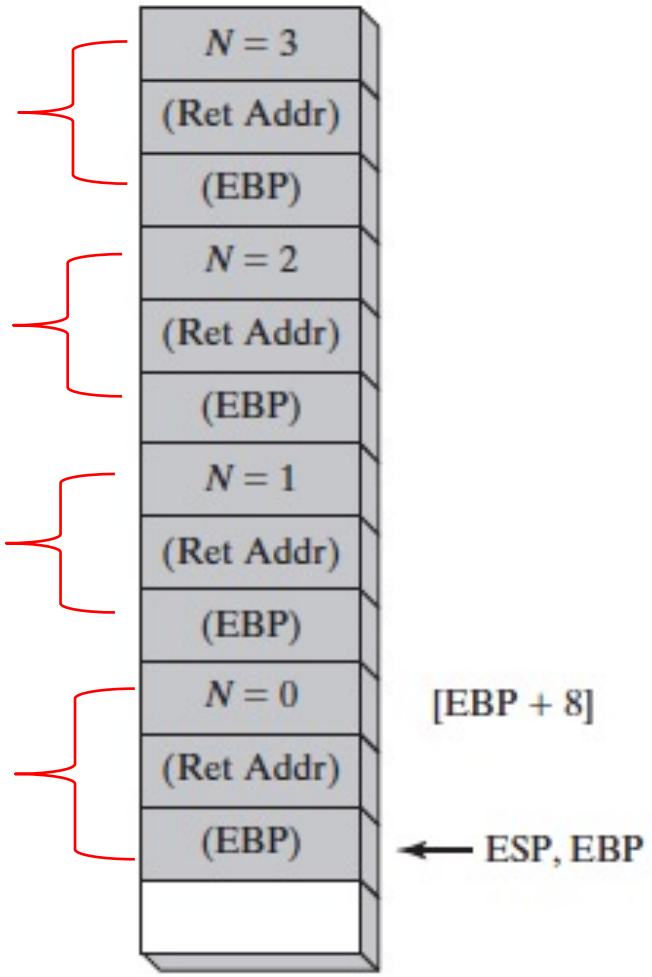
2

```
;-----------------------------------------------------
Factorial PROC
; Calculates a factorial.
; Receives: [ebp+8] = n, the number to calculate
; Returns: eax = the factorial of n
;-----------------------------------------------------
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+8]          ; get n
    cmp   eax,0                ; n > 0?
    ja    L1                   ; yes: continue
    mov   eax,1                ; no: return 1 as the value of 0!
    jmp   L2                   ; and return to the caller

L1: dec   eax
    push  eax                  ; Factorial(n-1)
    call  Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov   ebx,[ebp+8]          ; get n
    mul   ebx                  ; EDX:EAX = EAX * EBX

L2: pop   ebp                  ; return EAX
    ret   4                    ; clean up stack
Factorial ENDP
END main
```



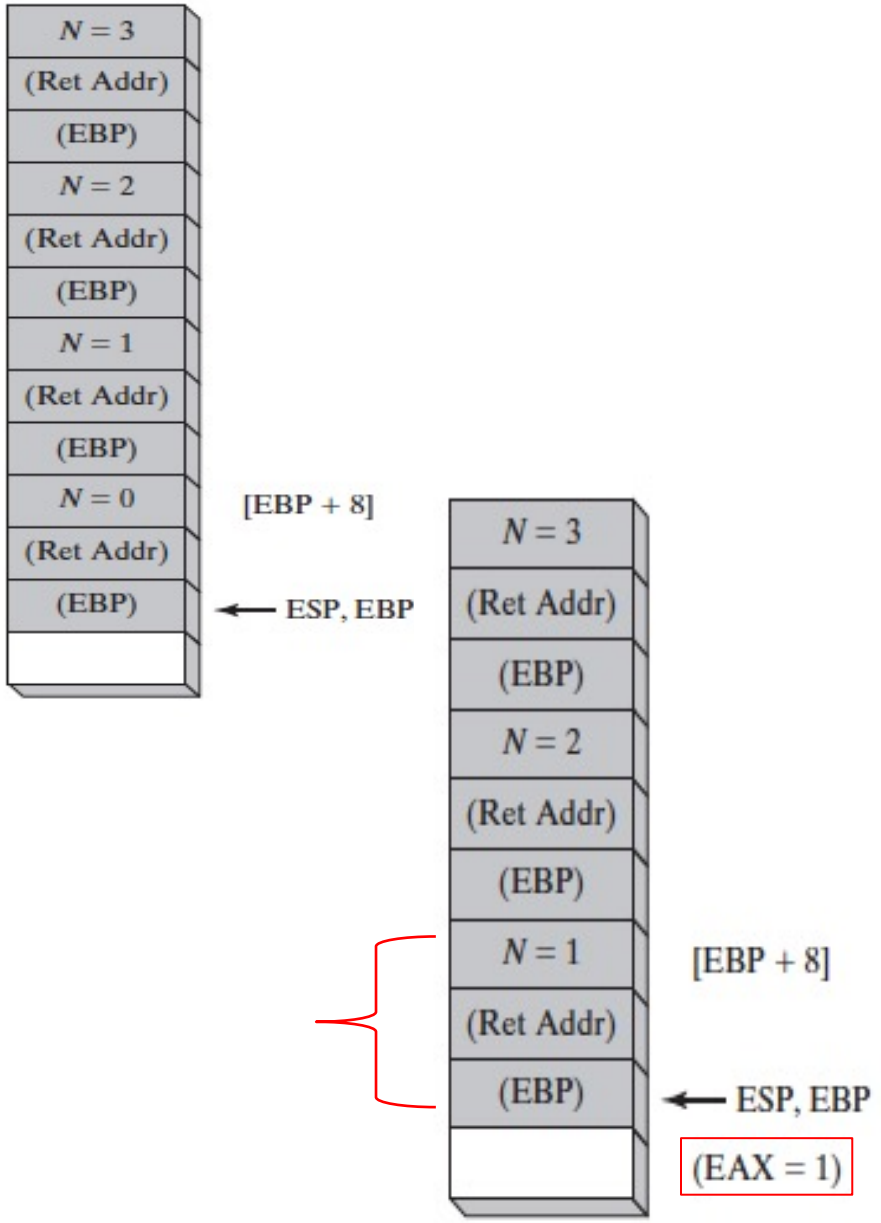
3

```
;-----------------------------------------------------
Factorial PROC
; Calculates a factorial.
; Receives: [ebp+8] = n, the number to calculate
; Returns: eax = the factorial of n
;-----------------------------------------------------
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+8]          ; get n
    cmp   eax,0                ; n > 0?
    ja    L1                   ; yes: continue
    mov   eax,1                ; no: return 1 as the value of 0!
    jmp   L2                   ; and return to the caller

L1: dec   eax
    push  eax                  ; Factorial(n-1)
    call  Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov   ebx,[ebp+8]          ; get n
    mul   ebx                  ; EDX:EAX = EAX * EBX

L2: pop   ebp                  ; return EAX
    ret   4                    ; clean up stack
Factorial ENDP
END main
```

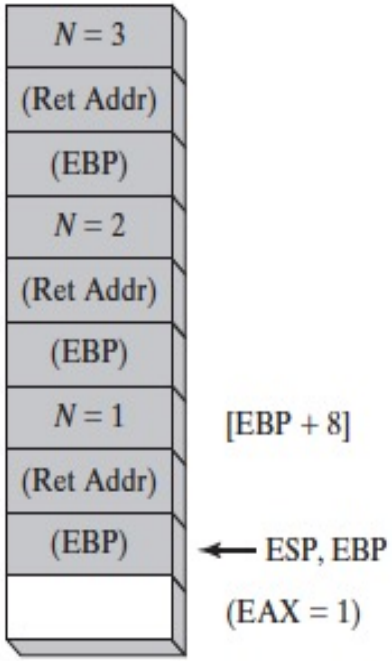


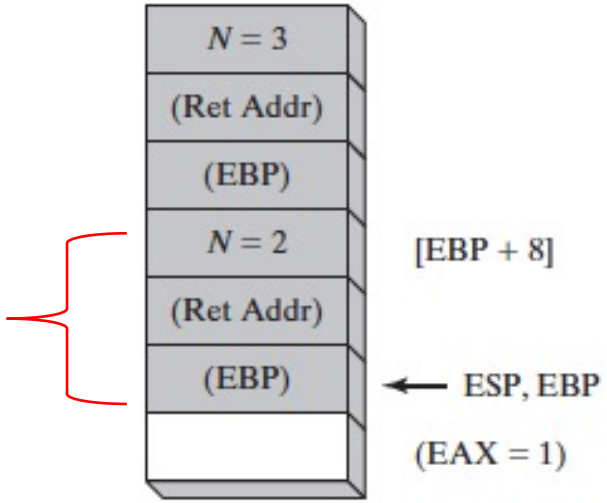
4

```
;-----------------------------------------------------
Factorial PROC
; Calculates a factorial.
; Receives: [ebp+8] = n, the number to calculate
; Returns: eax = the factorial of n
;-----------------------------------------------------
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+8]          ; get n
    cmp   eax,0                ; n > 0?
    ja    L1                   ; yes: continue
    mov   eax,1                ; no: return 1 as the value of 0!
    jmp   L2                   ; and return to the caller

L1: dec   eax
    push  eax                  ; Factorial(n-1)
    call  Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov   ebx,[ebp+8]          ; get n
    mul   ebx                  ; EDX:EAX = EAX * EBX

L2: pop   ebp                  ; return EAX
    ret   4                    ; clean up stack
Factorial ENDP
END main
```

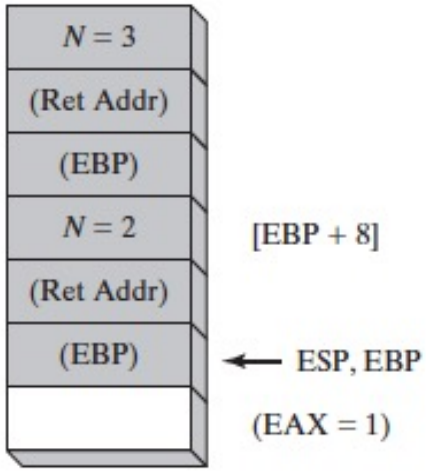


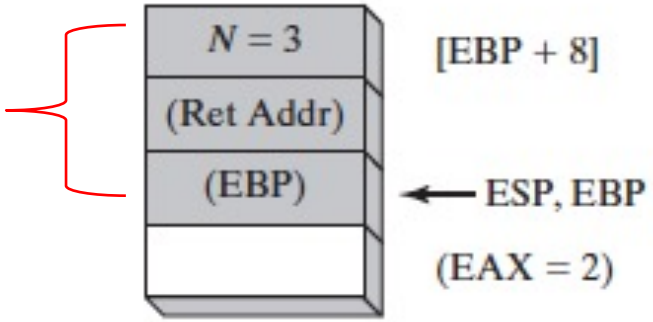

```
;-----------------------------------------------------
Factorial PROC
; Calculates a factorial.
; Receives: [ebp+8] = n, the number to calculate
; Returns: eax = the factorial of n
;-----------------------------------------------------
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+8]          ; get n
    cmp   eax,0                ; n > 0?
    ja    L1                   ; yes: continue
    mov   eax,1                ; no: return 1 as the value of 0!
    jmp   L2                   ; and return to the caller

L1: dec   eax
    push  eax                  ; Factorial(n-1)
    call  Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov   ebx,[ebp+8]          ; get n
    mul   ebx                  ; EDX:EAX = EAX * EBX

L2: pop   ebp                  ; return EAX
    ret   4                    ; clean up stack
Factorial ENDP
END main
```






```
;-----------------------------------------------------
Factorial PROC
; Calculates a factorial.
; Receives: [ebp+8] = n, the number to calculate
; Returns: eax = the factorial of n
;-----------------------------------------------------
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+8]          ; get n
    cmp   eax,0                ; n > 0?
    ja    L1                   ; yes: continue
    mov   eax,1                ; no: return 1 as the value of 0!
    jmp   L2                   ; and return to the caller

L1: dec   eax
    push  eax                  ; Factorial(n-1)
    call  Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov   ebx,[ebp+8]          ; get n
    mul   ebx                  ; EDX:EAX = EAX * EBX

L2: pop   ebp                  ; return EAX
    ret   4                    ; clean up stack
Factorial ENDP
END main
```





The return value in **EAX, 6**, is the computed value of 3 factorial.

```
;-----------------------------------------------------
Factorial PROC
; Calculates a factorial.
; Receives: [ebp+8] = n, the number to calculate
; Returns: eax = the factorial of n
;-----------------------------------------------------
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+8]          ; get n
    cmp   eax,0                ; n > 0?
    ja    L1                   ; yes: continue
    mov   eax,1                ; no: return 1 as the value of 0!
    jmp   L2                   ; and return to the caller

L1: dec   eax
    push  eax                  ; Factorial(n-1)
    call  Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov   ebx,[ebp+8]          ; get n
    mul   ebx                  ; EDX:EAX = EAX * EBX

L2: pop   ebp                  ; return EAX
    ret   4                    ; clean up stack
Factorial ENDP
END main
```

Outline

- Stack Overview
- Stack Frame
  - Stack Parameters
  - Passing Arguments
  - Accessing Stack Parameters
  - Cleaning Up the Stack
  - Saving and Restoring Registers
  - Local Variables
  - Reference Parameters
  - Recursion
- **INVOKE, ADDR, PROC, and PROTO**

INVOKE, ADDR, PROC, and PROTO

- **INVOKE Directive**
- ADDR Operator
- PROC Directive
- PROTO Directive
- Parameter Classifications
  - Example: Exchanging Two Integers
  - Debugging Tips

Not in 64-bit mode!

INVOKE Directive

- In 32-bit mode, the **INVOKE** directive is a powerful

  replacement for Intel's CALL instruction that **lets you pass multiple arguments**

- **Syntax:**
  - `INVOKE procedureName [, argumentList]`
- ***ArgumentList*** is an optional comma-delimited list of procedure arguments

- **Arguments** can be:
  - **immediate** values and integer expressions
  - **variable names**
  - **address** and **ADDR** expressions
  - **register** names

Table 8-2 Argument types used with INVOKE.

Type	Examples
Immediate value	10, 3000h, OFFSET mylist, TYPE array
Integer expression	(10 * 20), COUNT
Variable	myList, array, myWord, myDword
Address expression	[myList+2], [ebx + esi]
Register	eax, bl, edi
ADDR *name*	ADDR myList
OFFSET *name*	OFFSET myList

INVOKE Examples

```
.data
byteVal BYTE 10
wordVal WORD 1000h
.code
        ; direct operands:
        INVOKE Sub1,byteVal,wordVal

        ; address of variable:
        INVOKE Sub2,ADDR byteVal

        ; register name, integer expression:
        INVOKE Sub3,eax,(10 * 20)

        ; address expression (indirect operand):
        INVOKE Sub4,[ebx]
```

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- **ADDR Operator**
- PROC Directive
- PROTO Directive
- Parameter Classifications
  - Example: Exchanging Two Integers
  - Debugging Tips

Not in 64-bit mode!

ADDR Operator

- **Returns** a near or far **pointer to a variable**, depending on which memory model your program uses:
  - Small model: returns 16-bit offset
  - Large model: returns 32-bit segment/offset
  - **Flat model**: returns 32-bit offset
- **Simple example:**

```
.data
myWord WORD ?
.code
INVOKE mySub,ADDR myWord
```

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- **PROC Directive**
- PROTO Directive
- Parameter Classifications
  - Example: Exchanging Two Integers
  - Debugging Tips

Not in 64-bit mode!

PROC Directive

- The **PROC directive** declares a procedure **with an optional list of named parameters**.
- **Syntax**:

  ***label* PROC paramList**

  - ***paramList*** is a list of parameters separated by commas.
  - **Each parameter** has the following syntax:

    paramName* : *type

  - ***type*** must either be one of the standard ASM **types** (**BYTE, SBYTE, WORD, etc**.),
  - or it can be **a pointer** to one of these types.

You can r**efer to parameter by name** instead of **calculating the stack offsets (such as [ebp+8])**

PROC Directive

- Alternate format permits parameter list to be on one or more separate lines:

  ***label* PROC,** ← comma required

  paramList

  - The parameters can be **on the same line** . . .

    param-1:type-1*, *param-2:type-2*, . . ., *param-n:type-n

  - Or they can be on separate lines:

    ***param-1:type-1*,**

    ***param-2:type-2*,**

    ***. . .*,**

    param-n:type-n

AddTwo Procedure

- The AddTwo procedure receives two integers and returns their sum in EAX.

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

   mov eax,val1
   add eax,val2

   ret
AddTwo ENDP
```

PROC Examples

- FillArray receives a pointer to an array of bytes, a single byte fill value that will be copied to each element of the array, and the size of the array.

```
FillArray PROC,
     pArray:PTR BYTE, fillVal:BYTE,
     arraySize:DWORD

     mov ecx,arraySize
     mov esi,pArray
     mov al,fillVal
L1:  mov [esi],al
     inc esi
     loop L1
     ret
FillArray ENDP
```

Proc c,
Proc STDCALL,

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- PROC Directive
- **PROTO Directive**
- Parameter Classifications
  - Example: Exchanging Two Integers
  - Debugging Tips

PROTO Directive

- Creates a procedure prototype
- Syntax:
  - ***label*** **PROTO** *paramList*
- Every procedure called by the **INVOKE** directive **must have a prototype**
- A complete procedure definition can also serve as its own prototype
- Parameter list **not permitted in 64-bit mode**

PROTO Directive

- Standard configuration:
  - **PROTO** appears at top of the program listing,
  - **INVOKE** appears in the code segment,
  - **Procedure** implementation occurs later in the program:

```
MySub PROTO             ; procedure prototype

.code
INVOKE MySub            ; procedure call

MySub PROC              ; procedure implementation
    .
    .
MySub ENDP
```

PROTO Example

- Prototype for the ArraySum procedure, showing its parameter list:

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

Parameters are not permitted in 64-bit mode.

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- PROC Directive
- PROTO Directive
- **Parameter Classifications**
  - Example: Exchanging Two Integers
  - Debugging Tips

Parameter Classifications

- **An input parameter** is **data passed by a calling program** to a procedure.
  - The called procedure is **not expected to modify the corresponding parameter variable**, and
  - even if it does, the modification is confined to the procedure itself.

- **An output** parameter is created by **passing a pointer to a variable when a procedure is called**.
  - The procedure **does not use any existing data from the variable**, but it fills in a new value before it returns.

- **An input-output** parameter **is a pointer to a variable containing input** that will be both used and modified by the procedure.
  - The variable passed by the calling program is modified.

Trouble-Shooting Tips

- **Save and restore registers** when they are modified by a procedure.
  - **Except a register that returns a function result**

- When using **INVOKE**, **be careful to pass a pointer to the correct data type.**
  - For example, **MASM cannot distinguish between a DWORD argument and a PTR BYTE argument.**

- **Do not pass an immediate value** to a procedure **that expects a reference parameter**.
  - Dereferencing its address will likely cause a general-protection fault.