

# EE-2003

# Computer Organization & Assembly Language

## INSTRUCTOR

Engr. Aashir Mahboob

Lecturer, Department of Computer Science

Office: Room No: 16, Opposite HoD (CS) Office

FAST NUCES (Karachi)

Aashir.mahboob@nu.edu.pk



**CHAPTER No: 8**

# **ADVANCE PROCEDURES**

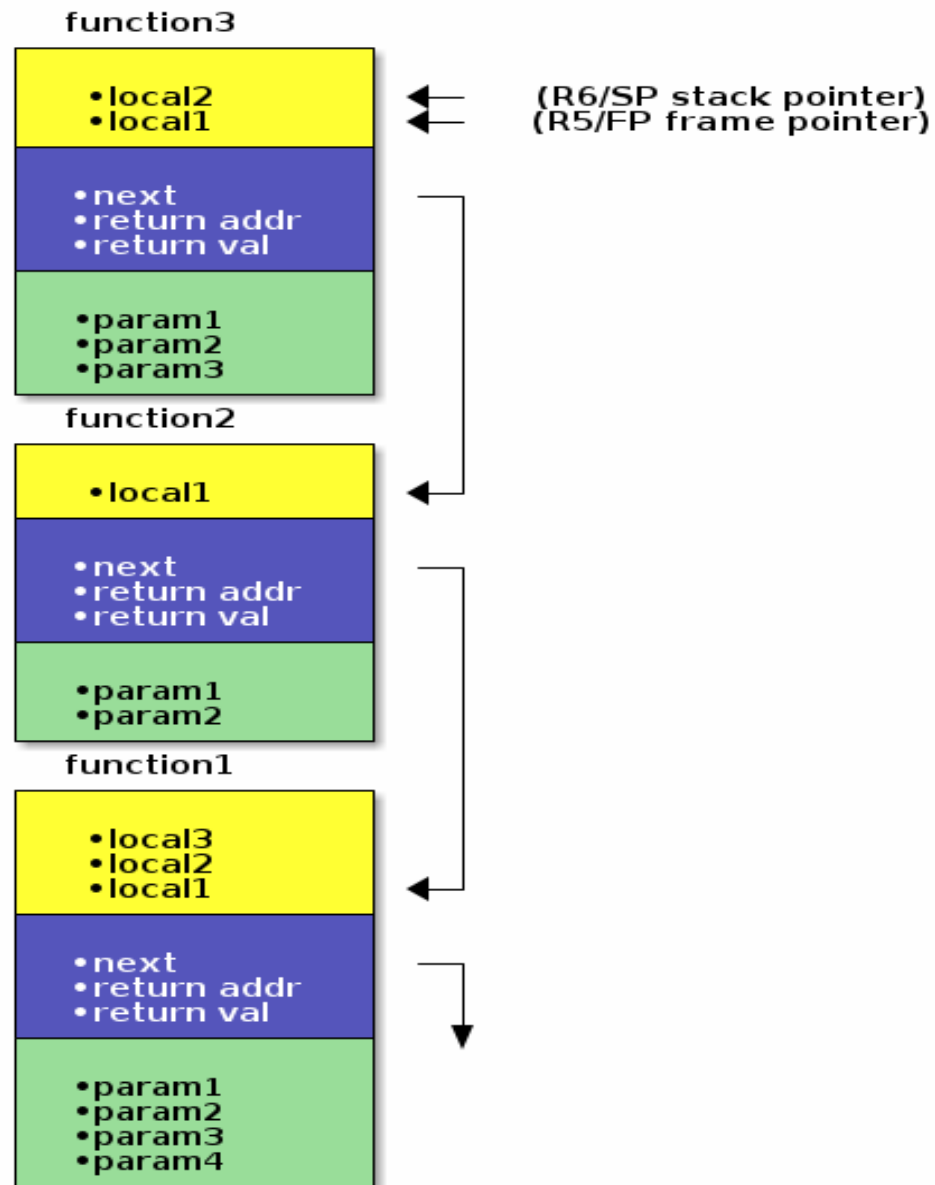
# OUTLINE

- ▶ Stack frames
- ▶ Parameters
- ▶ local variable

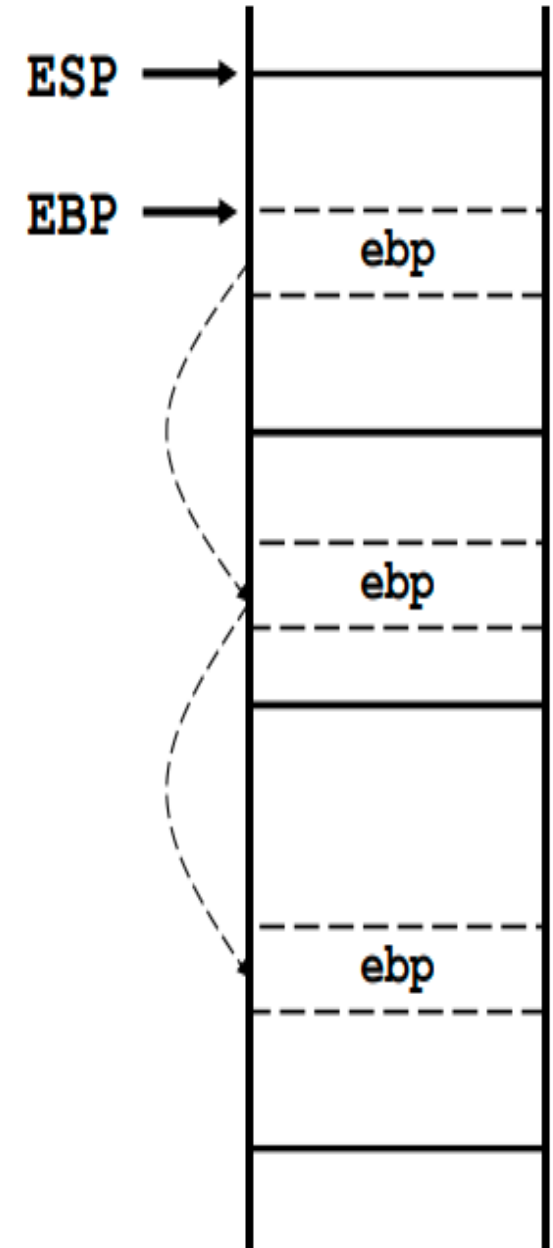
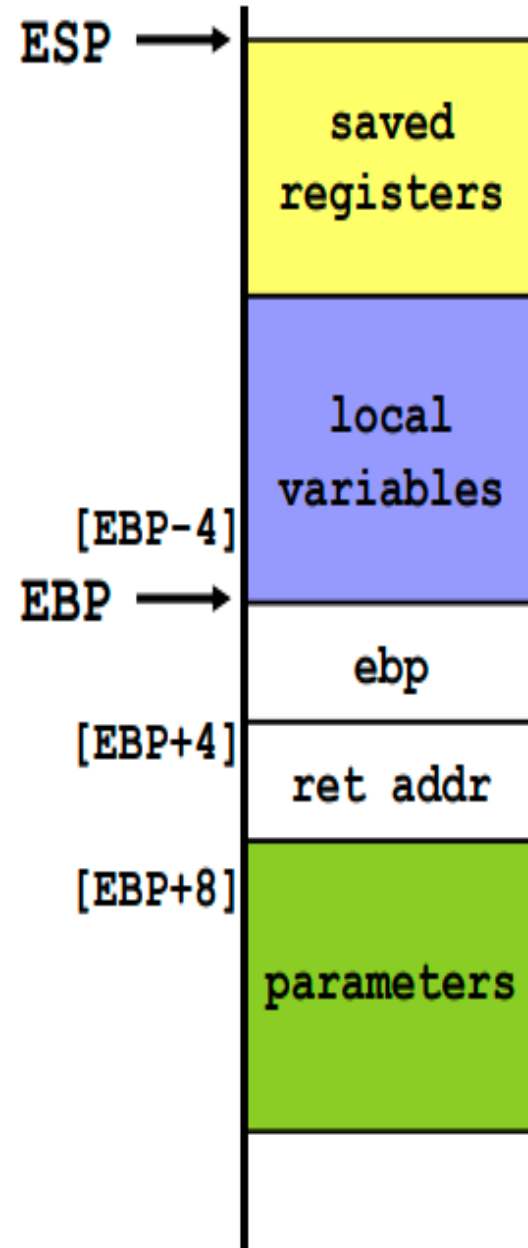
# WHAT IS STACK FRAME?

- ▶ The idea behind a stack frame is that each subroutine can act independently of its location on the stack, and each subroutine can act as if it is the top of the stack. When a function is called, a new stack frame is created at the current esp location. A stack frame acts like a partition on the stack.

↑  
lo



↓  
hi



# Parameter Passing

- ▶ Parameter passing in assembly language is different
  - ▶ More complicated than that used in a high-level language
- ▶ In assembly language
  - ▶ Place all required parameters in an accessible storage area
  - ▶ Then call the procedure
- ▶ Two types of storage areas used
  - ▶ Registers: general-purpose registers are used (**register method**)
  - ▶ Memory: stack is used (**stack method**)
- ▶ Two common mechanisms of parameter passing
  - ▶ Pass-by-value: parameter **value** is passed
  - ▶ Pass-by-reference: **address** of parameter is passed



# Stack Parameters

- Consider the following max procedure

```
int max ( int x, int y, int z ) {  
    int temp = x;  
    if (y > temp) temp = y;  
    if (z > temp) temp = z;  
    return temp;  
}
```

Calling procedure: `mx = max(num1, num2, num3)`

## Register Parameters

```
mov    eax, num1  
mov    ebx, num2  
mov    ecx, num3  
call   max  
mov    mx,  eax
```

## Stack Parameters

```
push   num3  
push   num2  
push   num1  
call   max  
mov    mx,  eax
```

} **Reverse  
Order**

# Parameters

- ▶ **Two types:** register parameters and stack parameters.
- ▶ Stack parameters are more convenient than register parameters.
- ▶ Example demonstrates calling DumpMem using register parameters and stack parameters

## ; Register Parameters

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

## ;Stack Parameters

```
push TYPE array  
push LENGTHOF array  
push OFFSET array  
call DumpMem
```



# Register versus Stack Parameters

## ▶ Passing Parameters in Registers

- ▶ Pros: Convenient, easier to use, and faster to access
- ▶ Cons: Only few parameters can be passed
  - ▶ A small number of registers are available
  - ▶ Often these registers are used and need to be saved on the stack
  - ▶ Pushing register values on stack negates their advantage

## ▶ Passing Parameters on the Stack

- ▶ Pros: Many parameters can be passed
  - ▶ Large data structures and arrays can be passed
- ▶ Cons: Accessing parameters is not simple
  - ▶ More overhead and slower access to parameters

# Arguments pushed on the stack

- ▶ 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)



## ► Passing by value:

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

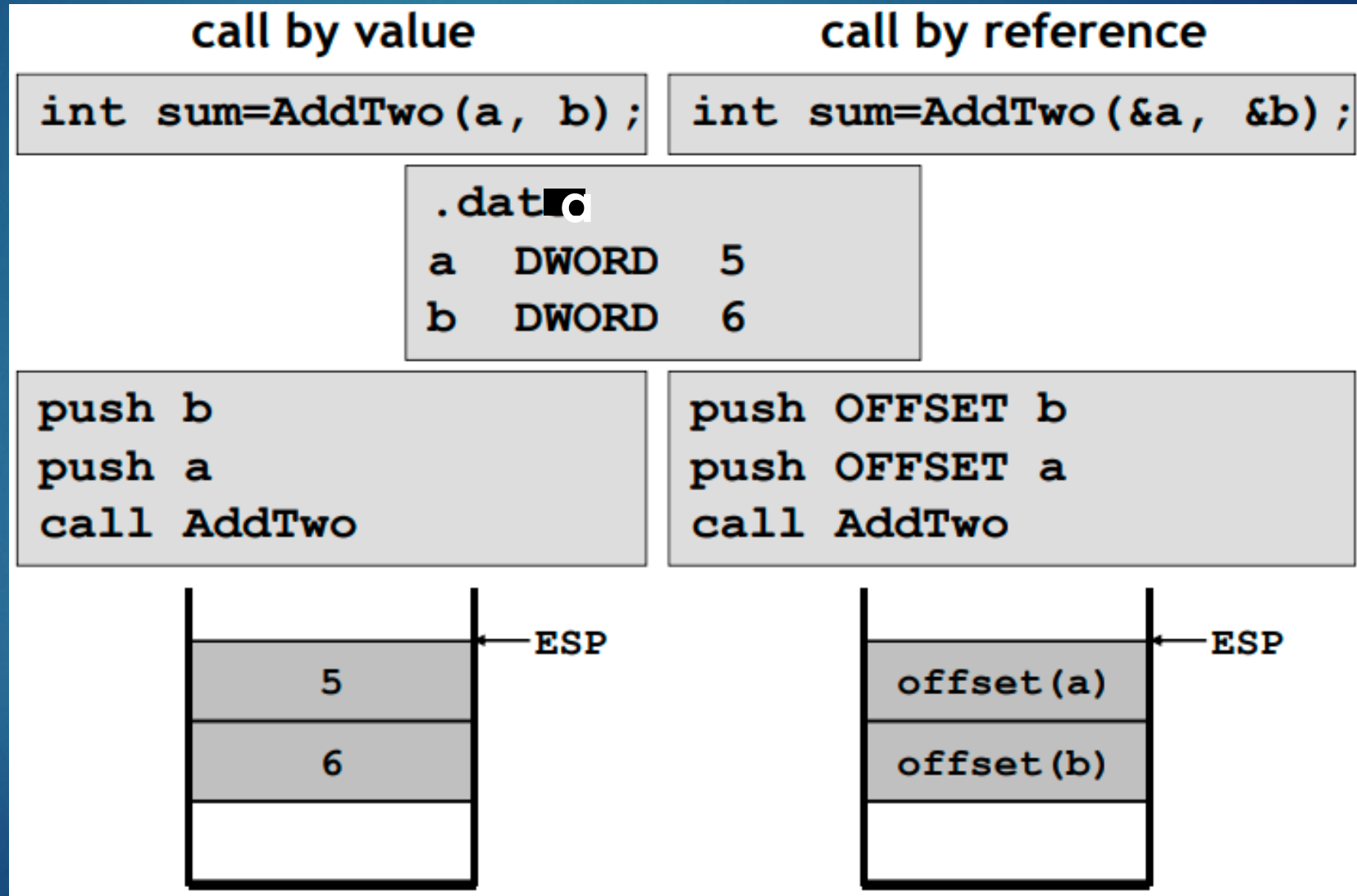
## ► Passing by Reference

An argument passed by reference consists of the address (offset) of an object.

## ► Passing Arrays

High-level languages always pass arrays to subroutines by reference. That is, they push the address of an array on the stack. one would not want to pass an array by value, because doing so would require each array element to be pushed on the stack separately. Such an operation would be very slow, and it would use up precious stack space.

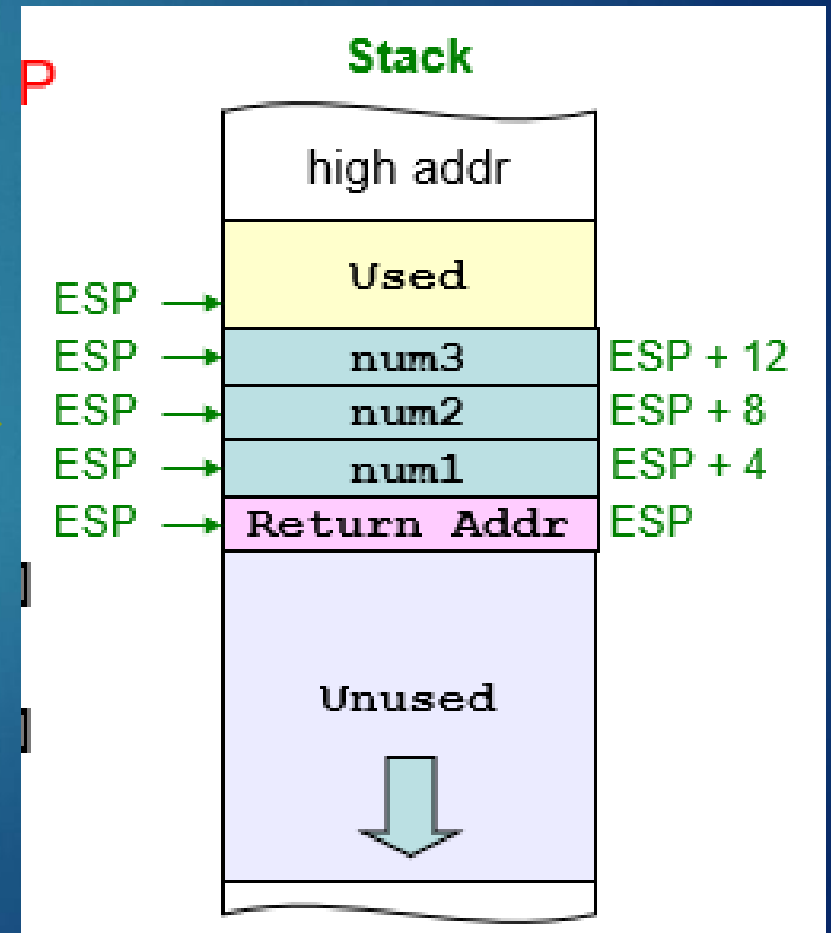
# Passing by value and passing by reference



# Passing Parameters on the Stack

- ▶ Calling procedure pushes **parameters on the stack**
- ▶ Procedure **max** receives parameters on the stack
  - ▶ Parameters are pushed in **reverse order**
  - ▶ Parameters are located **relative to ESP**

```
Passing  
Parameters  
on the stack  
  
push num3  
push num2  
push num1  
call max  
mov mx, EAX  
add ESP, 12  
  
max PROC  
    mov EAX, [ESP+4]  
    cmp EAX, [ESP+8]  
    jge @1  
    mov EAX, [ESP+8]  
    @1: cmp EAX, [ESP+12]  
    jge @2  
    mov EAX, [ESP+12]  
    @2: ret  
max ENDP
```



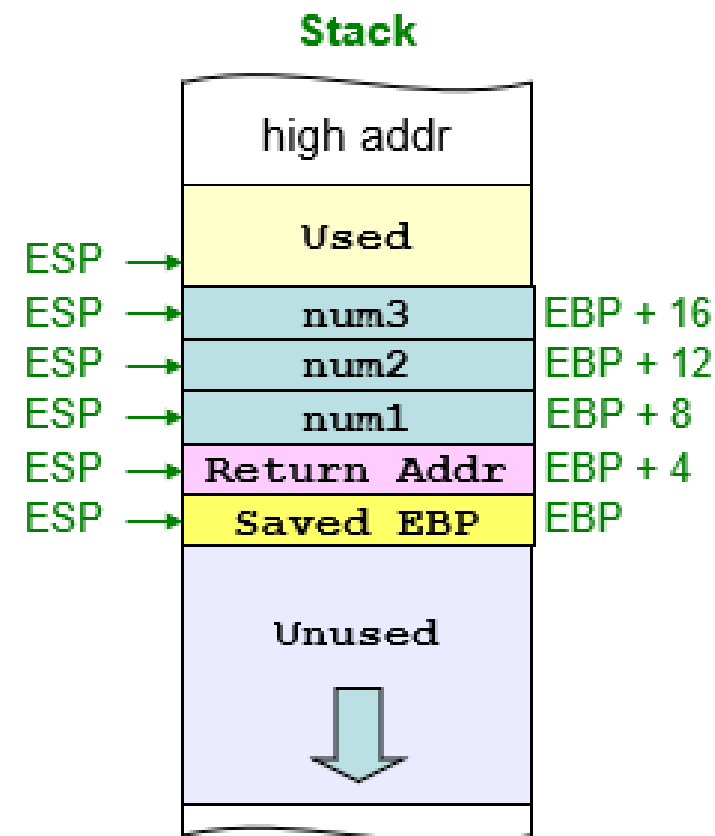
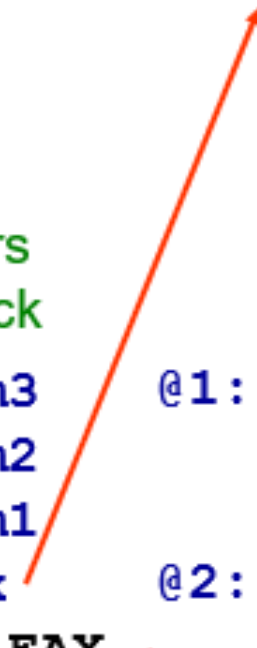


# Using the Base Pointer Register

- ▶ EBP is used to locate parameters on the stack
- ▶ Like any other register, EBP must be saved before use

Passing  
Parameters  
on the stack

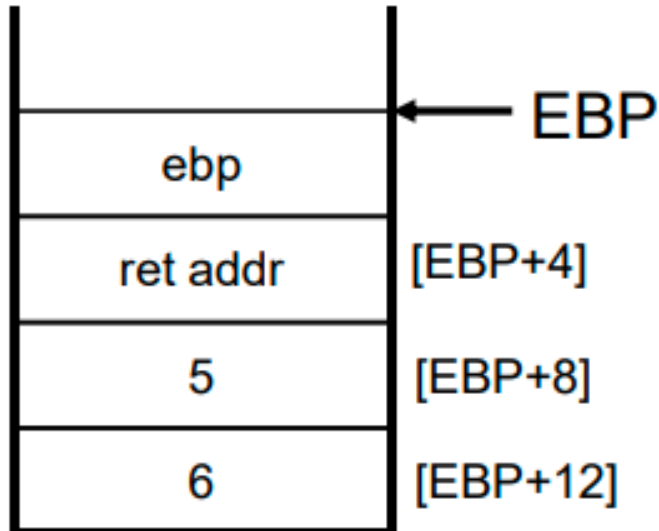
```
max PROC
    push EBP
    mov  EBP, ESP
    mov  EAX, [EBP+8]
    cmp  EAX, [EBP+12]
    jge  @1
    mov  EAX, [EBP+12]
    @1: cmp  EAX, [EBP+16]
    jge  @2
    mov  EAX, [EBP+16]
    @2: pop  EBP
    mov  mx, EAX
    add  ESP, 12
max ENDP
```



# Stack Frame Example:

```
.data
sum DWORD ?
.code
push 6          ; second argument
push 5          ; first argument
call AddTwo     ; EAX = sum
mov sum, eax    ; save the sum
```

```
AddTwo PROC
push ebp
mov ebp, esp
.
.
```

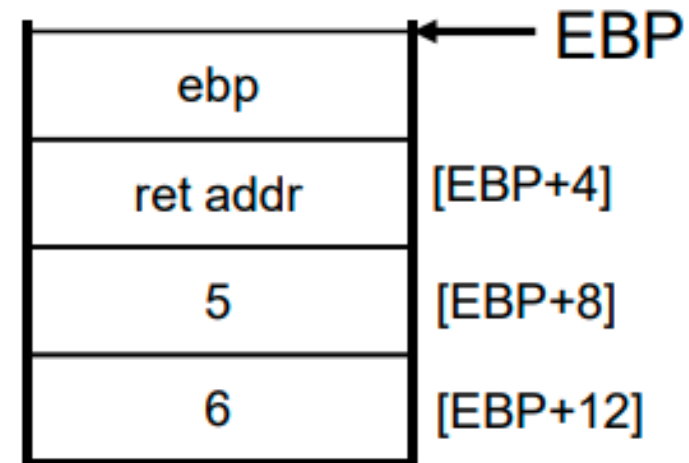


```
int AddTwo( int x, int y )
{ return x + y;
}
```

# Stack Frame Example:

```
AddTwo PROC
    push ebp
    mov ebp, esp          ; base of stack frame
    mov eax, [ebp + 12]   ; second argument (6)
    add eax, [ebp + 8]    ; first argument (5)
    pop ebp
    ret 8                 ; clean up the stack
AddTwo ENDP              ; EAX contains the sum
```

Who should be responsible to remove arguments? It depends on the language model.



# Base-Offset Addressing

- We will use base-offset addressing to access stack parameters. EBP is the base register and the offset is a constant. 32-bit values are usually returned in EAX. The following implementation of AddTwo adds the parameters and returns their sum in EAX:

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

# Explicit Stack Parameters

- ▶ When stack parameters are referenced with expressions such as `[ebp+8]` we call them explicit stack parameters. The reason for this term is that the assembly code explicitly states the offset of the parameter as a constant value.



# 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.

- ▶ Example:

main PROC

call Example1

exit

main ENDP

Example1 PROC

push 6

push 5

call AddTwo

ret ; stack is corrupted!

Example1 ENDP

# Who Should Clean up the Stack?

- ▶ When returning for a procedure call ...
  - ▶ Who should remove parameters and clean up the stack?
- ▶ Clean-up can be done by the calling procedure
  - ▶ `add ESP,12` ; will clean up stack
- ▶ Clean-up can be done also by the called procedure
  - ▶ We can specify an **optional integer** in the `ret` instruction
  - ▶ `ret 12` ; will return and clean up stack
- ▶ Return instruction is used to clean up stack
  - ▶ `ret n` ; *n* is an integer constant
  - ▶ Actions taken
    - ▶ `EIP = [ESP]`
    - ▶ `ESP = ESP + 4 + n`

# Ret Instruction

- ▶ **Return from subroutine**
- ▶ Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- ▶ **Syntax:**
  - ▶ RET
  - ▶ RET n
- ▶ Optional operand n causes n bytes to be added to the stack pointer after EIP (or IP) is assigned a value.

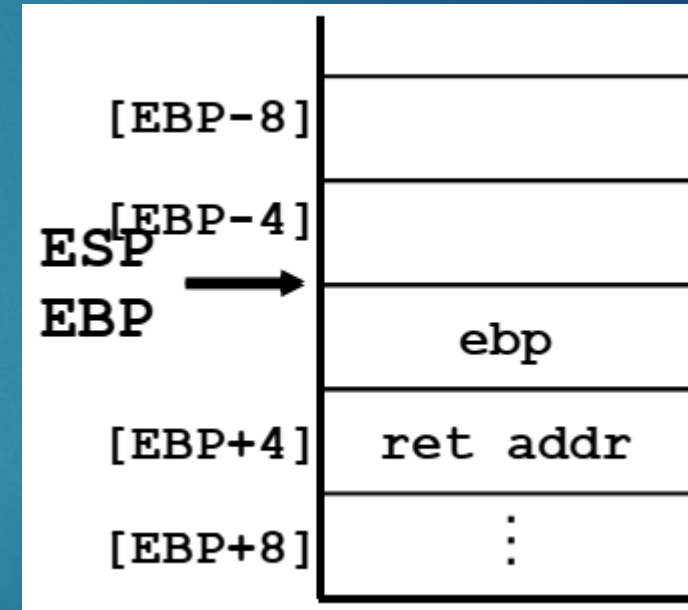
# LOCAL VARIABLES

- ▶ The variables defined in the data segment can be taken as static, global variables
  - ▶ Visibility
    - ▶ Static → program duration
    - ▶ Global → the whole program
- ▶ A local variable is created, used, and destroyed within a single procedure (block)
- ▶ Advantages of local variables:
  - ▶ Restricted access: easy to debug, less error prone
  - ▶ Efficient memory usage
  - ▶ Same names can be used in two different procedures
  - ▶ Essential for recursion

# CREATING LOCAL VARIABLE

- ▶ Local variables are created on the runtime stack, usually above EBP
- ▶ To explicitly create local variables, subtract their total size from ESP

```
MySub PROC
    push ebp
    mov  ebp, esp
    sub  esp, 8
    mov  [ebp-4], 123456h
    mov  [ebp-8], 0
    .
    .
```





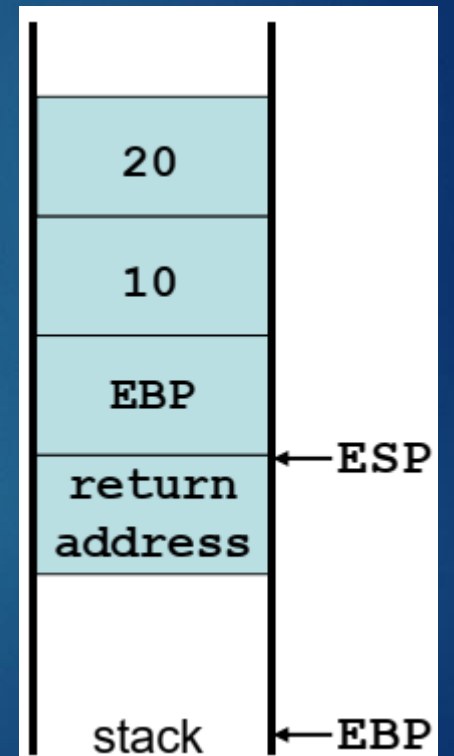
# LOCAL VARIABLE

- ▶ They can't be initialized at assembly time but can be assigned to default values at runtime

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



```
MySub PROC  
    push ebp  
    mov  ebp, esp  
    sub  esp, 8  
    mov  DWORD PTR [ebp-4], 10  
    mov  DWORD PTR [ebp-8], 20  
    ...  
    mov  esp, ebp  
    pop  ebp  
    ret  
MySub ENDP
```



# LOCAL VARIABLES

- Local variables are created on the runtime stack, usually below the base pointer (EBP).

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

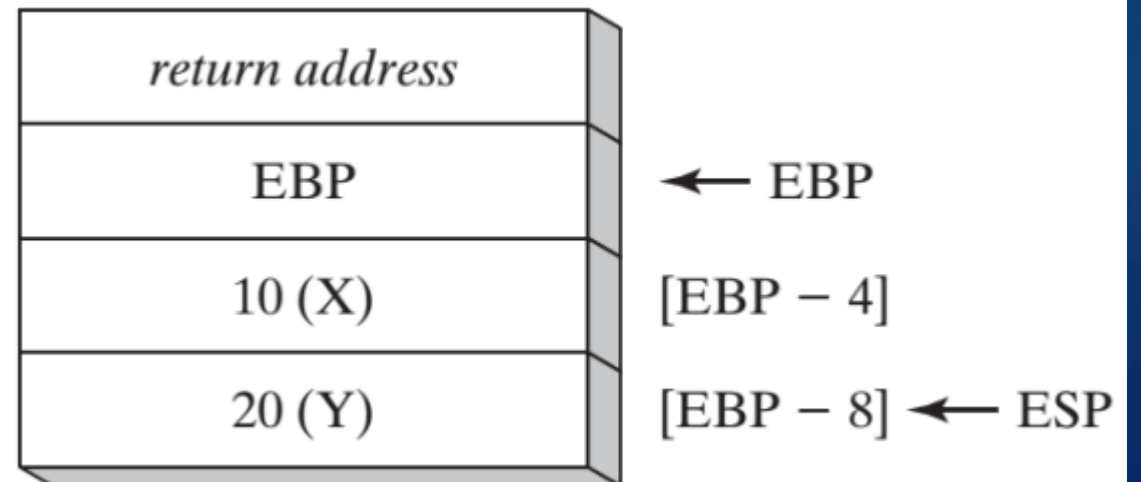
Variable	Bytes	Stack Offset
X	4	EBP - 4
Y	4	EBP - 8

# LOCAL VARIABLES

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
```



# ENTER AND LEAVE INSTRUCTIONS

- The ENTER instruction performs three operations:

1. Pushes EBP on the stack (push ebp)
2. Sets EBP to the base of the stack frame (mov ebp, esp)
3. Reserves space for local variables (sub esp, numbytes)

**ENTER numbytes, nestinglevel**

- Both the operands are immediate values,
- The first is a constant specifying the number of bytes of stack space to reserve for local variables.
- The second specifies the lexical nesting level of the procedure.

# ENTER AND LEAVE INSTRUCTIONS

E.g. a procedure with no local variables:

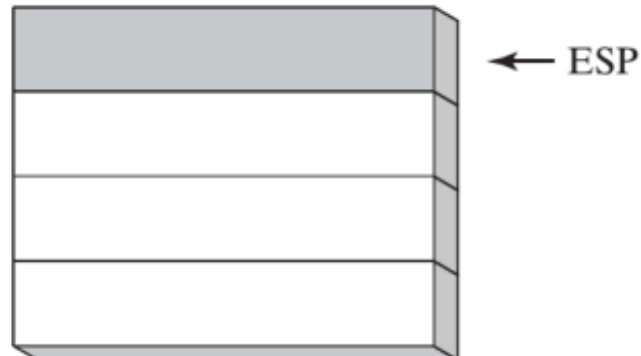
```
MySub PROC  
    ENTER 0,0
```

E.g. The ENTER instruction reserves 8 bytes of stack space for local variables.

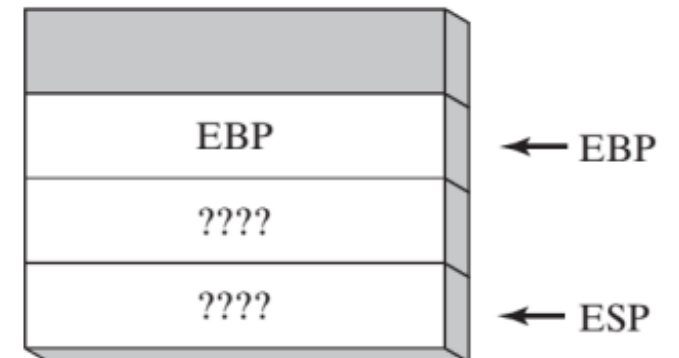
```
MySub PROC  
    ENTER 8,0
```

```
MySub PROC  
    push ebp  
    mov  ebp, esp  
    sub  esp, 8
```

Before



After executing ENTER 8,0





# ENTER AND LEAVE INSTRUCTIONS

- The LEAVE instruction terminates the stack frame for a procedure.
- It reverses the action of a previous ENTER instruction by restoring ESP and EBP to the values they were assigned when the procedure was called.

```
MySub PROC
    enter 8,0
    .
    .
    leave
    ret
MySub ENDP
```

```
mov esp, ebp
pop ebp
```

# LOCAL DIRECTIVE

LOCAL declares one or more local variables by name, assigning them size attributes.

ENTER, on the other hand, only reserves a single unnamed block of stack space for local variables.

If used, LOCAL must appear on the line immediately following the PROC directive.

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

```
MySub PROC  
    LOCAL var1:BYTE, var2:WORD, var3:SDWORD
```

# LOCAL DIRECTIVE (EXAMPLE)

```
BubbleSort PROC
    LOCAL temp:DWORD, SwapFlag:BYTE
    . . .
    ret
BubbleSort ENDP
```

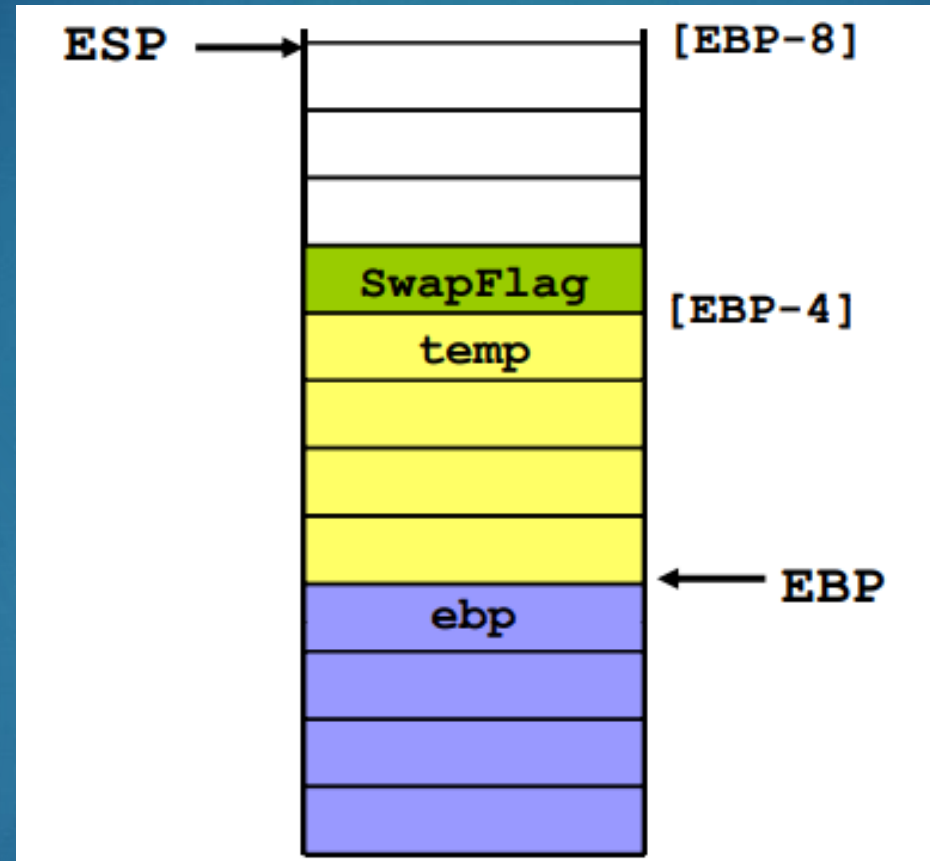
MASM generates the following code:

```
BubbleSort PROC
    push ebp
    mov  ebp,esp
    add  esp,0FFFFFFF8h ; add -8 to ESP
    . . .
    mov  esp,ebp
    pop  ebp
    ret
BubbleSort ENDP
```

# Non-Doubleword Local Variables

- ▶ • Local variables can be different sizes.
- ▶ • How are they created in the stack by LOCAL directive:
  - ▶ – 8-bit: assigned to next available byte
  - ▶ – 16-bit: assigned to next even (word) boundary
  - ▶ – 32-bit: assigned to next doubleword boundary

# LOCAL DIRECTIVE (EXAMPLE)



```
mov    eax, temp      →    mov    eax, [ebp-4]
mov    bl, SwapFlag    →    mov    bl, [ebp-5]
```



# INVOKE DIRECTIVE

- The INVOKE directive, only available in 32-bit mode, pushes arguments on the stack and calls a procedure.
- INVOKE is a convenient replacement for the CALL instruction because it lets you pass multiple arguments using a single line of code.

**INVOKE** *procedureName* [, *argumentList*]

# CALL VS INVOKE

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

The equivalent statement using INVOKE is reduced to a single line in which the arguments are listed in reverse order (assuming STDCALL is in effect).

**INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array**

**INVOKE** permits almost any number of arguments, and individual arguments can appear on separate source code lines.

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 <i>name</i>	ADDR myList
OFFSET <i>name</i>	OFFSET myList

# EXAMPLE

```
.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]
```

```
.data
val1 DWORD 12345h
val2 DWORD 23456h

.code
    INVOKE AddTwo, val1, val2

push val1
push val2
call AddTwo
```

# 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
- The ADDR operator can only be used in conjunction with INVOKE:

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

```
mov esi, ADDR myArray ; error
```

```
.data
Array DWORD 20 DUP(?)
.code
...
INVOKE Swap, ADDR Array, ADDR [Array+4]
```

```
push OFFSET Array+4
push OFFSET Array
Call Swap
```



# PROC DIRECTIVE

The PROC directive declares a procedure with an optional list of named parameters.

**label PROC, parameter\_list**

- The PROC directive permits you to declare a procedure with a comma-separated list of named parameters.

**label PROC, parameter\_1, parameter\_2, ..., parameter\_n**

- Your implementation code can refer to the parameters by name rather than by calculated stack offsets such as [ebp - 8].

# PROC example

```
label PROC [attributes] [USES reglist],  
    parameter_1,  
    parameter_2,  
    .  
    .  
    parameter_n
```

```
label PROC [attributes], parameter_1,  
parameter_2, . . . , parameter_n
```

A single parameter has the following syntax:

```
paramName:type
```

# PROC example

```
AddTwo PROC,  
    val1:DWORD,  
    val2:DWORD  
    mov     eax,val1  
    add     eax,val2  
    ret  
AddTwo ENDP
```



```
AddTwo PROC  
    push    ebp  
    mov     ebp, esp  
    mov     eax,dword ptr [ebp+8]  
    add     eax,dword ptr [ebp+0Ch]  
    leave  
    ret     8  
AddTwo ENDP
```

# RET Instruction Modified by PROC

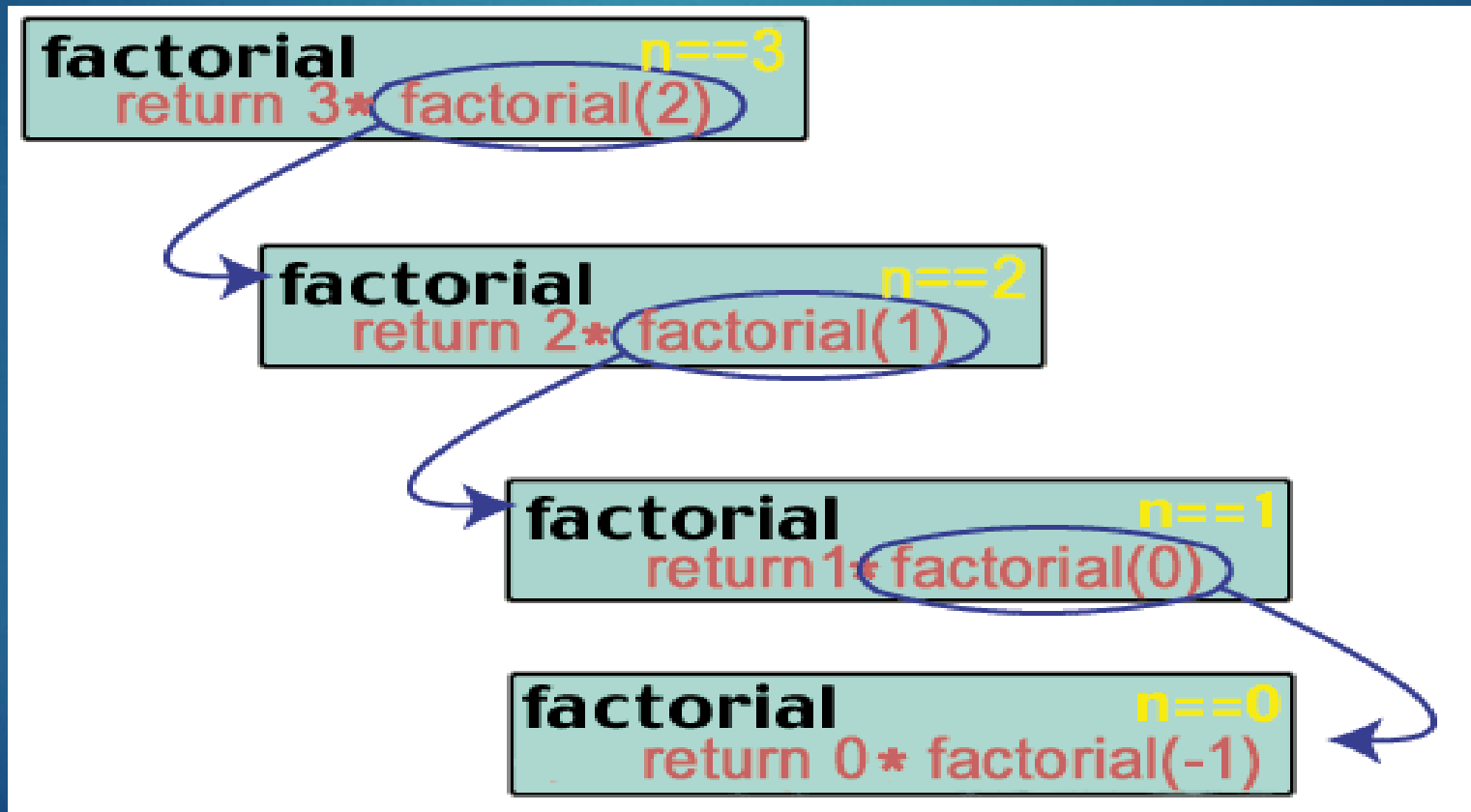
When PROC is used with one or more parameters and STDCALL is the default protocol, MASM generates the following entry and exit code, assuming PROC has  $n$  parameters:

```
push    ebp
mov     ebp, esp
.  
.  
leave  
ret     ( $n*4$ )
```

We can replace PUSH EBP and MOV EBP,ESP with ENTER 0,0 Instruction.

# Recursion

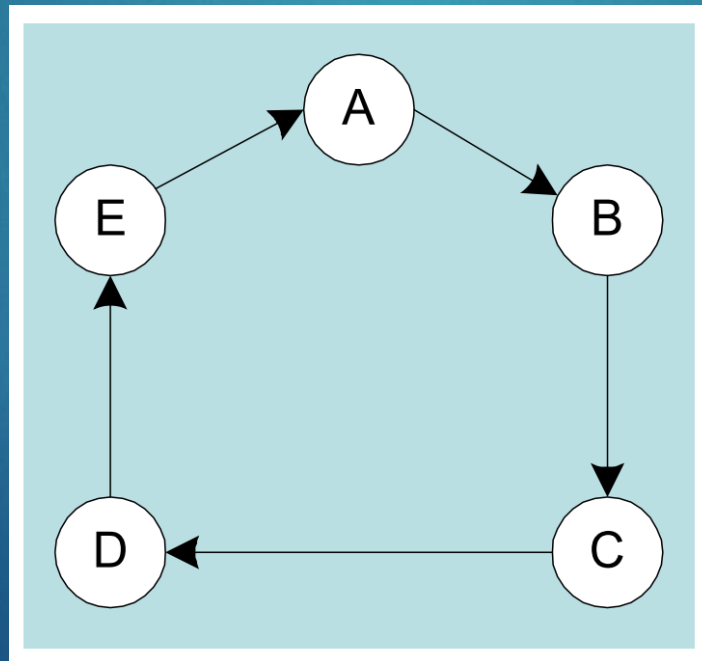
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.





# RECURSION

- ▶ 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



5

```
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
```

# RECURSION EXAMPLE

```
INCLUDE Irvine32.inc
.code
main PROC
    mov     ecx,5                ; count = 5
    mov     eax,0                ; holds the sum
    call    CalcSum              ; calculate sum
L1:  call    WriteDec             ; display EAX
    call    Crlf                 ; new line
    exit
main ENDP

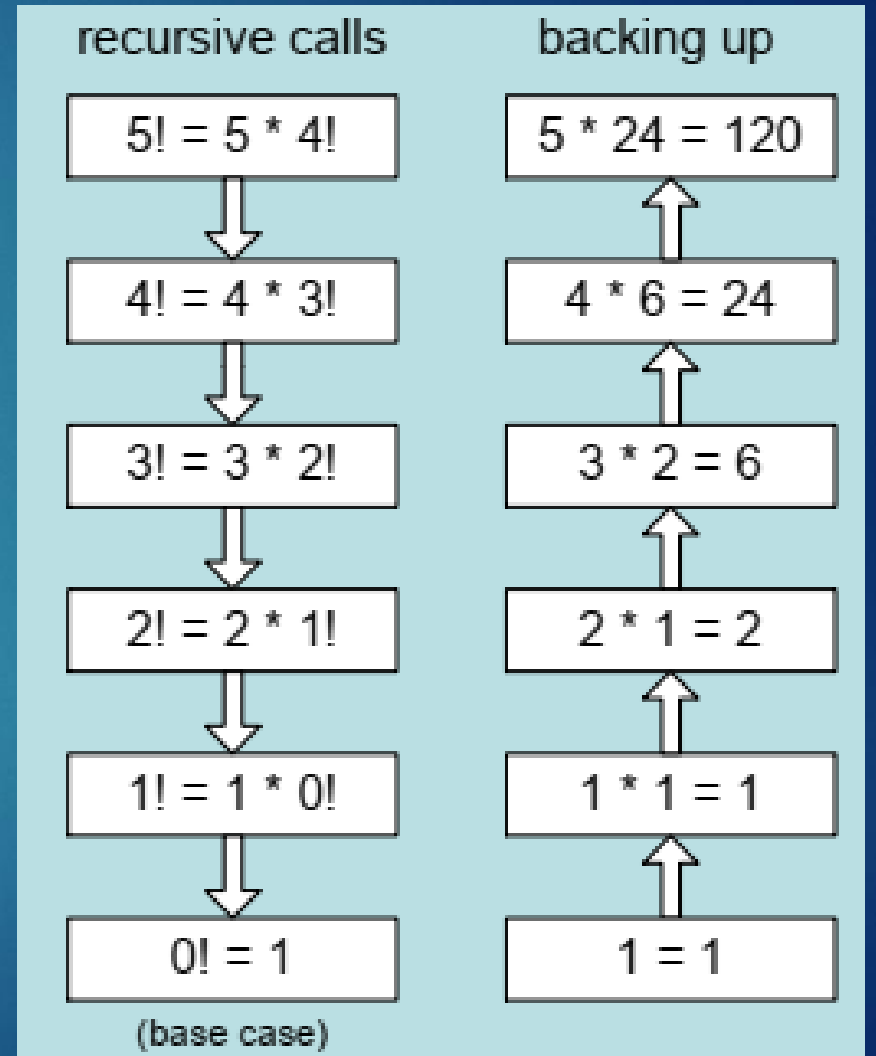
;-----
CalcSum PROC
; Calculates the sum of a list of integers
; Receives: ECX = count
; Returns: EAX = sum
;-----
    cmp     ecx,0                ; check counter value
    jz      L2                   ; quit if zero
    add     eax,ecx               ; otherwise, add to sum
    dec     ecx                  ; decrement counter
    call    CalcSum              ; recursive call
L2:  ret
CalcSum ENDP
```

# RECURSION

- ▶ This function calculates the factorial of integer n
- ▶ A new value of n is saved in each stack frame

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

factorial(5);



# RECURSION

```
; Calculating a Factorial (Fact.asm)
INCLUDE Irvine32.inc
.code
main PROC
    push    5                ; calc 5!
    call    Factorial        ; calculate factorial
(EAX)
    call    WriteDec         ; display it
    call    Crlf
    exit
main ENDP
```



# RECURSION

```
Factorial PROC
    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
    jmp  L2
L1: dec  eax
    push eax              ; Factorial(n-1)
    call Factorial

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
```

ebp
ret Factorial
0
⋮
ebp
ret Factorial
11
ebp
ret main
12

# PROTO DIRECTIVE

```
#include <stdio.h>
int addNumbers(int a, int b);           // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);           // function call
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)           // function definition
{
    int result;
    result = a+b;
    return result;                       // return statement
}
```

# PROTO DIRECTIVE

- Creates a procedure prototype

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

# EXAMPLE

```
MySub PROTO      ; procedure prototype
```

```
.code
```

```
INVOKE MySub     ; procedure call
```

```
MySub PROC       ; procedure implementation
```

```
.
```

```
.
```

```
MySub ENDP
```

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

```
ArraySum PROTO,
```

```
    ptrArray:PTR DWORD, ; points to the array
```

```
    szArray:DWORD       ; array size
```

```
ArraySum PROC USES esi, ecx,
```

```
    ptrArray:PTR DWORD, ; points to the array
```

```
    szArray:DWORD       ; array size
```

# LEA instruction (load effective address)

- The LEA instruction returns offsets of both direct and indirect operands at run time.
- OFFSET only returns constant offsets (assemble time).
- LEA is required when obtaining the offset of a stack parameter or local variable. For example:

```
CopyString PROC,  
    count:DWORD  
    LOCAL temp[20]:BYTE  
  
    mov edi,OFFSET count; invalid operand  
    mov esi,OFFSET temp ; invalid operand  
    lea edi,count        ; ok  
    lea esi,temp         ; ok
```



# Related directives

## **.MODEL directive**

---



- **.MODEL** directive specifies a program's memory model and model options (language-specifier).
- Syntax:  

```
.MODEL memorymodel [,modeloptions]
```
- *memorymodel* can be one of the following:
  - tiny, small, medium, compact, large, huge, or flat
- *modeloptions* includes the language specifier:
  - procedure naming scheme
  - parameter passing conventions
- **.MODEL flat, STDCALL**

# Memory models



- A program's memory model determines the number and sizes of code and data segments.
- Real-address mode supports tiny, small, medium, compact, large, and huge models.
- Protected mode supports only the flat model.

Small model: code < 64 KB, data (including stack) < 64 KB.  
All offsets are 16 bits.

Flat model: single segment for code and data, up to 4 GB.  
All offsets are 32 bits.

# Language specifiers



- STDCALL (used when calling Windows functions)
  - procedure arguments pushed on stack in reverse order (right to left)
  - called procedure cleans up the stack
  - `_name@nn` (for example, `_AddTwo@8`)
- C
  - procedure arguments pushed on stack in reverse order (right to left)
  - calling program cleans up the stack (variable number of parameters such as `printf`)
  - `_name` (for example, `_AddTwo`)
- PASCAL
  - arguments pushed in forward order (left to right)
  - called procedure cleans up the stack
- BASIC, FORTRAN, SYSCALL

