

**EL2003**  
**Computer**  
**Organization &**  
**Assembly Language**

**Lab 10**  
Advanced  
Procedures

**Instructor: Mr. Mubashir**

# LAB 10

## Learning Objectives

- Implementing procedures using stack frame
- Using stack parameters in procedures
- Passing value type and reference type parameters

## Stack Applications

There are several important uses of runtime stacks in programs:

- A stack makes a convenient temporary save area for registers when they are used for more than one purpose. After they are modified, they *can* be restored to their original values.
- When the CALL instruction executes, the CPU saves the current subroutine's return address on the stack.
- When calling a subroutine, you pass input values called arguments by pushing them on the stack.
- The stack provides temporary storage for local variables inside subroutines.

## Stack Parameters

- **Passing by value**

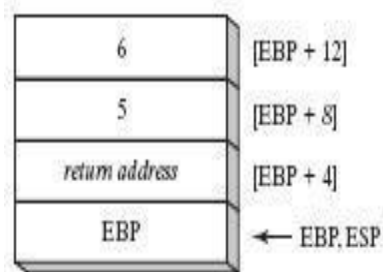
When an argument is passed by value, a copy of the value is pushed on the stack.

### EXAMPLE # 01:

```
.data
    var1    DWORD    5
    var2    DWORD    6
```

```
.code
    push var2
    push var1
    call AddTwo
    exit
```

```
AddTwo PROC
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp + 12]
    add     eax, [ebp + 8]
    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.

**EXAMPLE # 02:**

```
.data
    var1    DWORD    5
    var2    DWORD    6

    y_param    EQU    [ebp + 12]
    x_param    EQU    [ebp+ 8]

.code
    push var2
    push var1
    call AddTwo
    exit

AddTwo PROC
    push     ebp
    mov ebp, esp
    mov  eax, y_param
    add  eax, x_param
    pop  ebp
    ret
AddTwo ENDP
```

- **Passing by reference**

An argument passed by reference consists of the offset of an object to be passed.

**EXAMPLE # 03:**

```
.data
    count = 10
    arr    WORD    count DUP (?)

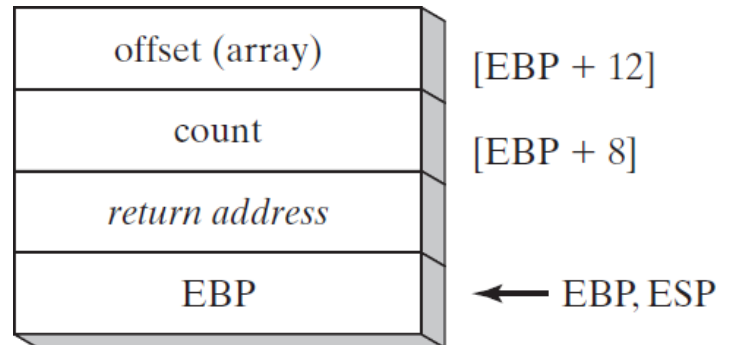
.code
    push  OFFSET arr
    push  count
    call  ArrayFill
```

exit

```
ArrayFill    PROC
    push    ebp
    mov     ebp, esp
    pushad
    mov     esi, [ebp + 12]
    mov     ecx, [ebp + 8]
    cmp     ecx, 0
    je      L2

L1:
    mov     eax, 100h
    call    RandomRange
    mov     [esi], ax
    add     esi, TYPE WORD
    loop    L1

L2:
    popad
    pop     ebp
    ret     8
ArrayFill    ENDP
```



## LEA Instruction

LEA instruction returns the effective address of an indirect operand. Offsets of indirect operands are calculated at runtime.

### EXAMPLE # 04:

```
.code
    call    makeArray
    exit

makeArray    PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 32
    lea     esi, [ebp - 30]
    mov     ecx, 30

L1:
    mov     BYTE PTR [esi], '*'
    inc     esi
```

```

        loop    L1
        add     esp, 32
        pop     ebp
        ret
makeArray ENDP

```

## ENTER & LEAVE Instructions

Enter instruction automatically creates stack frame for a called Procedure. Leave instruction reverses the effect of enter instruction.

### EXAMPLE # 06:

```

.data
    var1    DWORD    5
    var2    DWORD    6

.code
    push var2
    push var1
    call AddTwo
    exit

```

```

AddTwo PROC
    enter 0, 0
    mov  eax, [ebp + 12]
    add  eax, [ebp + 8]
    pop  ebp
    leave
    ret
AddTwo ENDP

```



## Local Variables

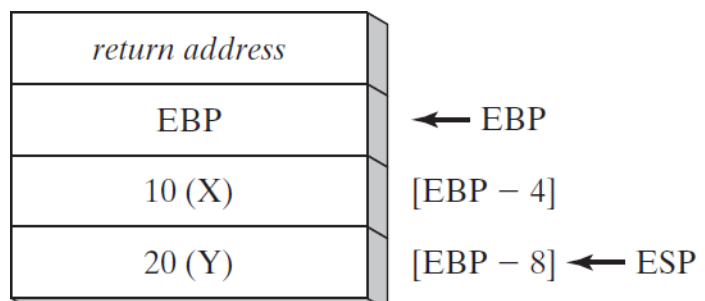
In MASM Assembly Language, local variables are created at runtime stack, below the base pointer (EBP).

### EXAMPLE # 05:

```

.code
    call  MySub
    exit

```



```

MySub      PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8
    mov     DWORD PTR [ebp - 4], 10    ; first parameter
    mov     DWORD PTR [ebp - 8], 20    ; second parameter
    mov     esp, ebp
    pop     ebp
    ret
MySub      ENDP

```

## LOCAL Directive

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

### EXAMPLE # 07:

```

.code
    call LocalProc
    exit

LocalProc  PROC
    LOCAL  temp : DWORD
    mov    temp, 5
    mov    eax, temp
    ret
LocalProc  ENDP

```

## Recursive Procedures

Recursive procedures are those that call themselves to perform some task.

### EXAMPLE # 08:

```

.code
    mov     ecx, 5
    mov     eax, 0
    call    CalcSum
L1:

```

```

        call    WriteDec
        call    crlf
        exit

CalcSum    PROC
        cmp     ecx, 0
        jz      L2
        add     eax, ecx
        dec     ecx
        call    CalcSum
L2:
        ret
CalcSum    ENDP

```

- **INVOKE Directive**

The INVOKE directive 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.

Here is the general syntax:

INVOKE procedureName [, argumentList]

Using the CALL instruction, for example, we could call a procedure named DumpArray after executing several PUSH instructions:

```

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.

INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array

- **ADDR Operator**

The ADDR operator can be used to pass a pointer argument when calling a procedure using INVOKE. The following INVOKE statement, for example, passes the address of myArray to the FillArray procedure:

```
INVOKE FillArray, ADDR myArray
```

- **PROC Directive**

Syntax of the PROC Directive

The PROC directive has the following basic syntax:

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

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

Example: The FillArray procedure receives a pointer to an array of bytes:

```
FillArray PROC,  
    pArray:PTR BYTE  
  
    . . .  
  
FillArray ENDP
```

- **PROTO Directive**

The PROTO directive creates 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.

```
MySub PROTO      ; procedure prototype  
.  
INVOKE MySub     ; procedure call  
.  
MySub PROC       ; procedure implementation  
.  
.  
MySub ENDP
```



## ACTIVITIES:

1. Write a program which contains a procedure named **ThreeProd** that displays the product of three numeric parameters passed through a stack.
2. Write a program which contains a procedure named **TakeInput** which takes input numbers from user and call a procedure named **GCD** which calculates their GCD and display the answer on console by calling another function **Display**. (Also show ESP values during nested function calls)
2. Write a program which contains a procedure named **MinMaxArray** that displays the minimum & maximum values in an array. Pass a size-20 array by reference to this procedure.
3. Write a program which contains a procedure named **LocalSquare** . The procedure must declare a local variable. Initialize this variable by taking an input value from the user and then display its square. Use **ENTER & LEAVE** instructions to allocate and de-allocate the local variable.
4. Write a program that calculates factorial of a given number *n*. Make a recursive procedure named **Fact** that takes n as an input parameter.
5. Write a non-recursive version of the procedure **Fact** that uses a loop to calculate factorial of given number n. Compare efficiency of both versions of the **Fact** procedure using **GetMSeconds**.
6. Write a program to take 4 input numbers from the users. Then make two procedures **CheckPrime** and **LargestPrime**. The program should first check if a given number is a prime number or not. If all of the input numbers are prime numbers then the program should call the procedure LargestPrime.  
  
CheckPrime: This procedure tests if a number is prime or not  
LargestPrime: This procedure finds and displays the largest of the four prime numbers.
7. Write a program which contains a procedure named BubbleSort that sorts an array which is passed through a stack using indirect addressing.
8. Write a program which contains a procedure named TakeInput which takes input numbers from user and call a procedure named Armstrong which checks either a number is an Armstrong number or not and display the answer on console by calling another function Display. (Also show ESP values during nested function calls)