Foundation for Advancement of Science & Technology

ADVANCE PROCEDURES
CHAPTER # 08

Stack Frame: Stack frame, Stack ka ek esa area jaha hum parameters, function calls k return addresses, local variables & saved register store kar sakte hai.

STACK PARAMETERS:

Abhi tak hum registers ko as parameter in a function use kr rahe the. Which makes a code clutter. Often we need to save the existing values of register before using it as parameters.

```
pushad                  ; pushes all 32-bit general register
mov esi, OFFSET arr
mov ecx, LENGTHOF arr
mov ebx, TYPEOF arr
~~popad~~                  ~~; rest~~
call DumpMem
popad                   ; restoring values
```

If we use stack parameters then our code would be

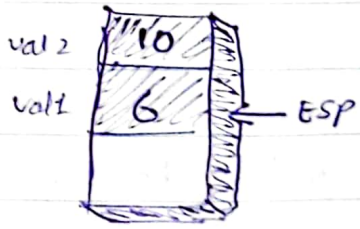
```
PUSH   OFFSET arr
PUSH   LENGTHOF arr
PUSH   TYPEOF arr
call   dumpMem
```

TYPES OF ARGUMENTS:

1. Value Arg. (values of variables & constants)
2. Reference Arg. (addresses of variables)

PASS BY VALUE:

```
.data
val2 DWORD 6
val2 DWORD 10
.code
push val2
push val1
call Add
```



In C++ : int sum = Add(val1, val2);

→

Pass By Reference :

```
push OFFSET val2
push OFFSET val1
call Add
```


```
Add ( &val1, &val2)
```

By array as an arg.

```
push OFFSET arr
call ArrayFill
```

Extended Base Pointer (EBP) :

→ It helps in stack frame managment during function execution

→ EBP hum stack k parameters, or local variables waghera ko access krne k liye use krte hai.

→ EBP hamara ek fixed reference point bn jata hai baaqi cheezon ko access krne k liye within a function.

WORKING :

① Sabse phele jab bhi function call hoga to EBP ki current value to store krenge (we will push it on stack).

② Ab hum EBP mein ESP ki value store krenge taake baaqi variables ko access kr sakein.

③ Hum EBP mein (+) krke baaqi variables ko access krenge.

④ Phir function k end mein EBP ki previous value store krenge jisse hamara stack frame clean hojayega.

```
AddTwo PROC
    ; val1 & val2 ko push kr chuke hai
    ; stack par
    push EBP
    mov  EBP, ESP    ; base of stack fram
```

32-bit register hai or values bhi 32 bit hai to is wajase +4 krshe hai

```
    mov  eax, [EBP+12]   ; second parame
    add  eax, [EBP+8]    ; 1st parameter
    pop  EBP
    ret
AddTwo ENDP
```

FAST Foundation for Advancement of Science & Technology

Explicit Stack Parameters:

→ Humein khud phele se pta hai k
{EBP + 12} par 2nd param. &
[EBP + 8] " 1st " store
honge to humne indono ko ek
naam dediya for better code
readablity

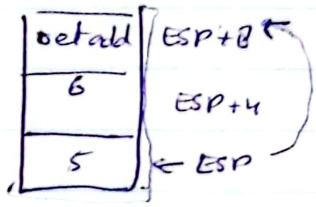
```
x_para  EQU  [EBP + 12]
y_para  EQU  [EBP + 8]

AddTwo PROC
    {
    mov eax, y_para
    add eax, x_para
    }
AddTwo ENDP
```

Cleaning Stack:

→ If we donot clean a stack frame before returning from procedure it can lead to crash & throws run time exception.

⇒ A simple way to clean a stack is to add value to ESP equal to the combined size of parameters.
Then ESP will point to the stack location that contain's subroutine (function) return address.

```
Example1 PROC
    push 6
    push 5
    call Add
    add esp, 8     ; remove arguments
    ret
E~ ENDP
```



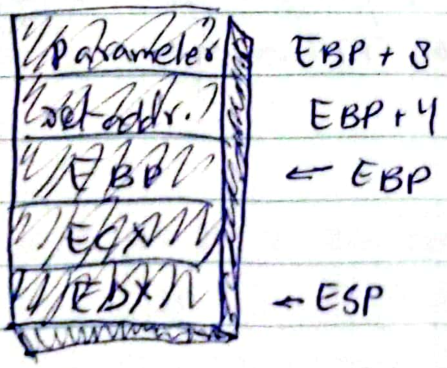
→ An 8-bit (BYTE) value cannot be used as an operand for PUSH so we need to 1st movzx the value in EAX & then PUSH it on stack.

→ Hum woh size k arguments stack mein paas krenge jo function humse expect krega ~~warna syntax~~ warna error ayega.

Saving & Restoring Registers:

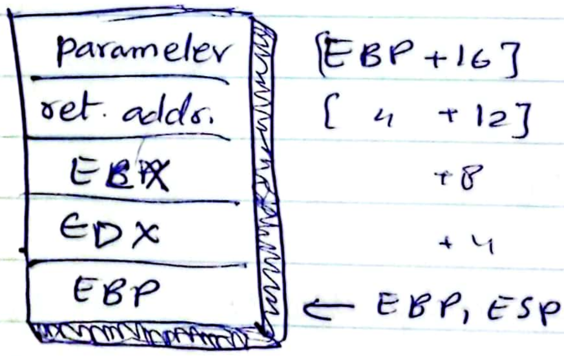
The registers to be saved should be pushed on the stack just ~~After~~ setting EBP to ESP and just be fore reserving space for local variables. This helps us to avoid changing ~~pas~~ offsets of existing parameters.

```
MyPro PROC
    push ebp
    mov ebp, esp
    push ecx;        ; save ecx
    push edx;        ; save edx
    mov eax, [ebp+8]
    ⋮
    pop edx          ; restore
    pop ecx          ; restore
    ret
MyProENDP
```



Procedures using explicit stack parameters should avoid the use of USES operator

```
MySub2 PRO USES ecx, edx
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]   ; error
    pop ebp
    ret 4
MySub ENDP
```



humein +16 move krna hoga +8 par error ayega.

↳ k +8 explicit define krdiya hoga

LOCAL VARIABLES:

Local variable are create in a runtime stack, usually below EBP.

```
MyPro PROC
  push EBP
  mov  EBP,ESP
  sub  ESP, 8          ; humein ebp k neeche bnaane ha local var to is waja se -8 kra take ESP neeche chala jaye
  mov DWORD PTR [EBP-4], 10   ; do var bnaane hai is waja se -8 kra the hota to -4 krte.
  mov DWORD PTR [EBP-8], 20   ; (20) -> initializing value
  mov esp,EBP
  pop ebp
  ret
MyPro ENDP
```

set.add	
EBP	← EBP
10 (X)	EBP-4
20 (Y)	EBP-8 ← ESP

Before finishing, the function resets the ESP to EBP to release the local variables agr value set na krte to return instruction to "pop ebp" instr. would set EBP to 20 and bcz ESP is pointing to 20 and the RET instruction would branch to memory location 10, causing the program to halt

Local Variables Symbols:

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

we can use these as our local variable

Accessing Reference Parameters:

Passing an array as a refrence.

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

.code
  push OFFSET array
  push count
  call ArrayFill

ArrayFill PROC
  push ebp
  mov ebp, esp
  pushad              ; as we will use esi & ecx so saving their prev values
  mov esi, [ebp+12]
  mov ecx, [ebp+4]
  cmp ecx, 0          ; if ecx=0 donot enter loop
  je L2
```

OFFSET arr	ebp+…
count	+8
ret.add	+4
EBP	← ES…

Foundation for Advancement of Science & Technology

11130

1110

```
L1:
    mov eax, 10000h
    call RandomRange
    mov [esi], ax
    add esi, TYPE WORD
    Loop L1

L2: popad        ; restoring register
    pop ebp
    ret 8        ; cleaning stack

ArrayFill ENDP
```

LEA Instruction:

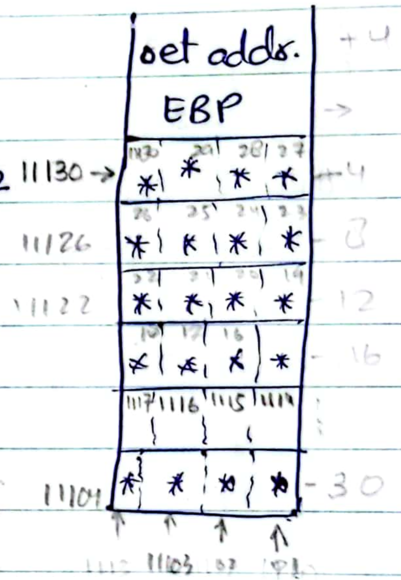
```
char string[30];
for( ________ ){
    string[i] = "*";
}
```

Assembly code;

```
MakeArray PROC
    push ebp
    mov ebp, esp
    sub esp, 32
```

humare array ka size 30 bytes tha to humne ESP ko 30 decre kr taake stack mein jaga bna sakein lekn 32 k liye kyun q k doubleword mein → memory aligned the.

```
lea esi, [ebp-30]   ; load address of mystring
mov ecx, 30

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

add esp, 32
pop ebp
ret

MyArray ENDP
```



ENTER INSTR:

→ Automatically creates a stack frame for called procedure.

↳ Pushes EBP on stack (push ebp)

↳ Sets EBP to base of stack frame (mov ebp, esp)

↳ Reserves space for local var. (sub esp, num bytes)

It has two operands both imm val.

Enter numbytes, nesting level.

[enter reserves a single unnamed block of stack space for local variables]

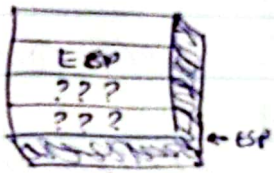
Foundation for Advancement of Science & Technology

Numbyte → is the no. of bytes of stack space to reserve for local variable

nestinglevel → always 0.

MySub PROC

Enter 8,0 ≡ push ebp
mov ebp, esp
sub esp, 8



LEAVE INSTR:

→ Terminates the stack frame for a procedure.

Leave ≡ mov esp, ebp
pop ebp
ret.

When we use ENTER then LEAVE should also be used at the end of a program otherwise correct return address won't popout.

LOCAL DIRECTIVE

→ A high level substitute for ENTER

→ LOCAL allows us to declare one or more variables by name & assigning them size attributes.

→ It must be written right after PROC directive.

Hello PROC

LOCAL temp : BYTE
name size

BubbleSort PROC

LOCAL temp : DWORD , SwapFlag : BYTE

or

LOCAL pArray : PTR WORD

pointer to a 16 bit integer

or

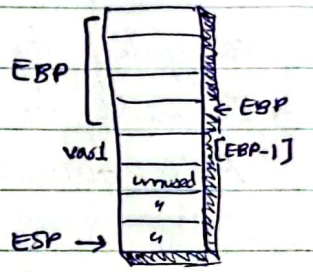
LOCAL TempArray[10] : DWORD

an array of 10 double words.

Foundation for Advancement of Science & Technology

— RECURSION —

Example PROC
LOCAL var1 : BYTE
. mov al, var1 ; [EBP-1]
ret
ENDP



▲ each block represent single byte

Example PROC
LOCAL Temp:DWORD, SwapFlag:BYTE
i
ret

Example ENDP
or
push ebp
mov ebp, esp
add esp, 0FFFFFFFBh ; add -8
mov eax, [EBP-4] ; temp
mov bl, [EBP-5] ; SwapFlag
leave
ret
ENDP

Though SwapFlag is only one byte but ESP is rounded to next double word location

each address consumes 4-bytes at each recursive call.

Recursively Calc. Sum to N integers.
.code
main PROC
mov ecx, 05 ; Sum till 5
mov eax, 0 ; stores S =
call CalcSum
L1: call WriteDec
exit
ret

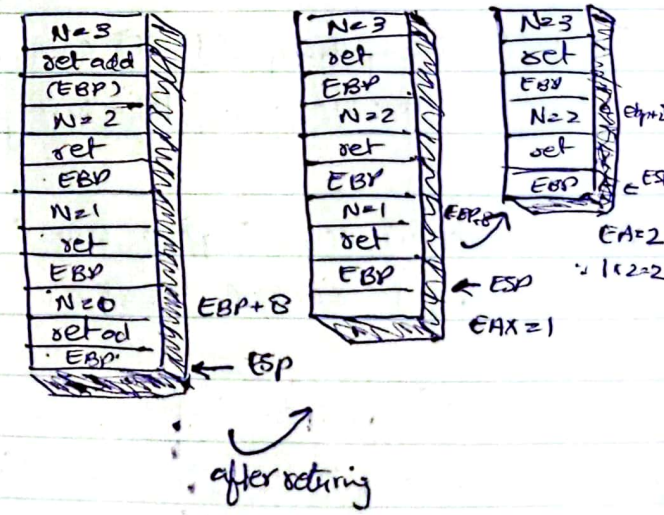
CalcSum PROC
; recieves ecx & eax
cmp ecx, 0 ; base condition if ecx = 0 then terminate
jz L2
add eax, ecx
dec ecx
call CalcSum
L2:
ret
CalcSum ENDP
end Main

Foundation for Advancement of Science & Technology

Calculating Factorial:

```
.code
main PROC                    ; to start
    push 5
    call Fact
    call WriteDec
    call WriteDec
main ENDP
Fact PROC
; Receives: [EBP+8] = n, the no. to calc
; Return: eax = the fact of n

    push ebp
    mov ebp, esp

    mov eax, [EBP+8]         ; get n
    cmp eax, 0               ; n>0
    ja L1                    ; if yes, continue
    mov eax, 1               ; mean eax=1
    jmp L2                   ; return 1

L1: dec eax                  ; ye mene new function call k liye eax push kiya hua
    push eax                 ; Fact(n-1)
    call Factorial
```

```
; Now these will execute at return of
; each function call

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

L2:
    pop ebp                  ; return eax
    ret 4                    ; clean up stack
Fact ENDP
END Main
```



Foundation for Advancement of Science & Technology

INVOKE, ADDR, PROC & PROTO

INVOKE :

→ INVOKE ko hum "call" ki jaga use kr sakte hai. Is mein hum parameters bhi bhej sakte hai.

```
push Type array
push LENGTHOF array
push OFFSET array
call DumpArray
```

OR

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

phele function name phir unke parameters in reverse order.

Argument Types :

1. Immediate value → 10, 300h, OFFSET arr, TYPE arr
2. Integer Expression → (10 * 20), COUNT
3. Variable → myList, array, mWord, myDWord
4. Address Expression → [myList+2], [ebx + esi]
5. Register → eax, bl, edi
6. ADDR name → ADDR myList
7. OFFSET name → OFFSET myList

EAX & EDX are overwritten if we ~~dont~~ pass any argument of less than 32-bits in INVOKE. We can avoid this by always passing 32-bit arg. or we can save values of eax, edx before function call.

ADDR Operator :

→ used to pass a pointer argument to INVOKE. Can only be used with INVOKE

```
INVOKE FillArray, ADDR myArray ✓
```

```
INVOK mySub, ADDR [ebp+127] X
```

The arg. passed to addr must be an assembly time constant.

```
INVOK Swap,
    ADDR Array,        ≡   push OFFSET Array+4
    ADDR [Array+4],        push OFFSET Array
                           call swap
```

passing 1st two elements of an array in SWAP

→ After DLD portion

Foundation for Advancement of Science & Technology

PROC :

Recieves a pointer to an Array of BYTE

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

NameofReg PROC [attributes] [USES reglist], param_list

see list of attributes from pg# 201

Parameter List:

```
ReadFile PROC USES eax ebx,
    pBuffer : PTR BYTE → recieves a byte ptr
    LOCAL fileHandle :DWORD
                ↓ local variable
    mov esi, pBuffer
    mov fileHandle, eax
```

Label PROC [attributes] [USES Reglist],
    param_1,
    param_2,
    :
    param_n

Parameter Syntax : paraName : type

Parameter Name is an arbitary Name and has a scope within that function (Local Scope). But it cannot be a name of Global variable or a code label.

RET Instruction :

When PROC is used with 1 or more parameters & STDCALL is the default protocol, MASM generates following code. Assuming that PROC has n parameters.

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

(OR)

```
push ebp          ] or ENTER 0,0
move ebp, esp     ]
mov eax, DWORD PTR [ebp+8]
add eax, 4  4 [ebp+12]
leave
ret 8
```

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

↳ no. of para (x) by 4 (bcz each parameter is a double word).

Foundation for Advancement of Science & Technology

PROTO :

→ Creates a prototype for an existing ~~Integer~~ Procedure.

→ MASM req. a prototype for each procedure called by INVOKE. PROTO must appear before INVOKE.

```
MySub PROTO        ; prototype

INVOKE MySub       ; procedure call
  {

MySub ENDP
```

⇒ Or we can write the implementation of procedure before calling INVOKE.

⇒ You can easily create a PROTOTYPE by copying PROC statement & change word PROC to PROTO, remove USES & reglist.

↳ written before ".data"

```
.data
  array DWORD 10000h, 20000h, ... - 8000h
  theSum DWORD ?
.code
  main PROC          ← this will come before .data
  ArraySum PROTO ptrArray: PTR DWORD,
                 Sz1Array: DWORD

INVOKE ArraySum, ADDR Array, LENGTHOF array
                 passing arguments
mov theSum, eax

main ENDP

ArraySum PROC USES esi, ecx
        ptrA ——          ; points to array
  Sz1Array : DWORD ; array size

mov esi, ptrArray  ; address of array
mov ecx, Sz1Array
mov eax, 0         ; set sum = 0
cmp ecx, 0         ; length = 0 ?
je L2              ; if yes : quit.
L1: add eax, [esi]
    add esi, 4
    loop L1
L2: ret
ArraySum ENDP
```

If a function is set to recieve a reference parameter & if we pass an argument which is an immediate value it will give Error.

MULTI MODULE PROGRAMMING

→ Large ASM files hard to manage
→ You can divide programs into multiple modules.
  ↳ Each module is a seperate ASM file.
  ↳ Each module is assembled into seperate OBJ file
↳ The Linkers combine all OBJ to form EXE file.

ADVANTAGES:
→ Easy to debug
→ Easy to write & maintain
→ A module can be a container for logically related codes
→ Can be reused in diff programs.