Assembly Programming

Chapter 8: Advanced Procedures

CSE3030

Prof. Youngjae Kim

Distributed Computing and Operating Systems Laboratory (DISCOS)

https://discos.sogang.ac.kr

8.1 Advanced Procedures

- This chapter will introduce the underlying structure of subroutine calls, focusing on the runtime stack.

- Modern languages push arguments on the stack before calling subroutines. The subroutines in turn store their local variables on the stack.

- In this chapter, we will learn how arguments are passed by value and by reference, how local variables are created and destroyed, and how recursion in implemented.

- Procedures in MASM
  - Subroutines (functions) in C and C++, methods in Java.

8.2 Stack Frames (Stack Parameters)

- In earlier chapters, subroutines received register parameters.
  - In this chapter, we will show how subroutines can receive parameters on the runtime stack.

- How will subroutines receive parameters on the runtime stack?
  - Windows functions receive a combination of register parameters and stack parameters.

- Stack Frame (Activation Record)
  - The area of the stack set aside for passed arguments (subroutine return address, local variables, and saved registers)

Stack Parameters

- The stack frame is created by the following steps:
  1. Passed arguments, if any, are pushed on the stack.
  2. The subroutine is called, causing the subroutine return address to be pushed on the stack.
  3. As the subroutine begins to execute, EBP is pushed on the stack.
  4. EBP is set equal to ESP. (EBP acts as a base reference for all of the subroutine parameters.)
  5. If there are local variables, ESP is decremented to reserve space for the variables on the stack.
  6. If any register need to be saved, they are pushed on the stack.

- Nearly all high-level languages use passing arguments on the stack.
  - For example, if you want to call functions the 32-bit Windows Application Interface (API), you must pass arguments on the stack.

Disadvantages of Register Parameters (1/2)

- The registers used for parameters (typically including EAX, EBX, ECX, EDX, etc) need to be pushed on the stack, executing more slowly.
  - These same registers are used to hold data values such as loop counters and operands in calculations.
  - Any registers used as parameters must first be pushed on the stack before procedure calls, assigned the value of procedure arguments, and later restored to their original vales after the procedure returns.

Example

```
push ebx
push ecx
push esi
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
pop esi
pop ecx
pop ebx
```

Disadvantages of Register Parameters (2/2)

- Bug Example
  - Programmers have to be very careful that each register's PUSH is matched by its appropriate POP.

```
1:  push ebx
2:  push ecx
3:  push esi
4:  mov esi,OFFSET array
5:  mov ecx,LENGTHOF array
6:  mov ebx,TYPE array
7:  call DumpMem
8:  cmp eax 1
9:  je error_exit
10:
11: pop esi
12: pop ecx
13: pop ebx
14: ret
15: error_exit:
16: mov edx, offset error_msg
17: ret
```

If eax is equal to 1, the procedure will not return to its caller on line 17.

Because three register values are left on the runtime stack.

Register Parameter vs. Stack Parameter

- Using stack parameters
  - More flexible than register parameters because it does not require register parameters before a subroutine call
  - Just need to push the arguments on the stack

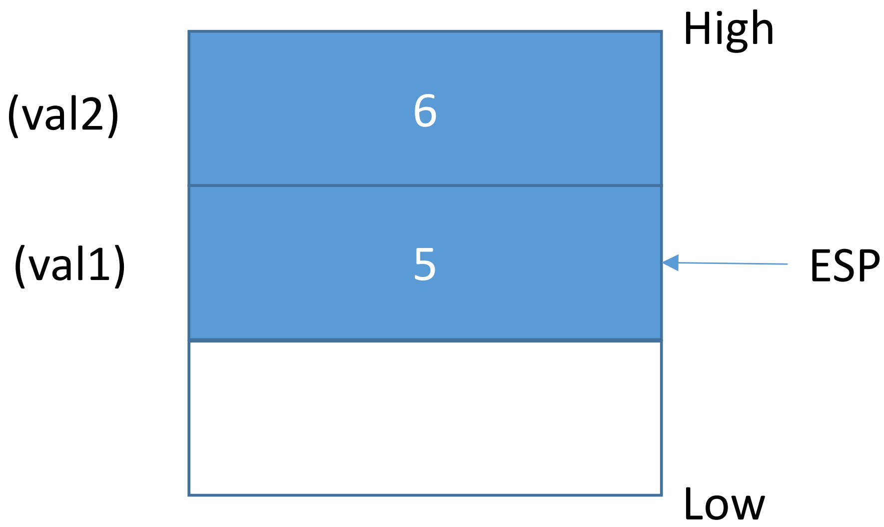
```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
popad
```

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

Passing by Value

- When an argument is passed by value, a copy of the value is pushed on the stack.
- Example (AddTwo)

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



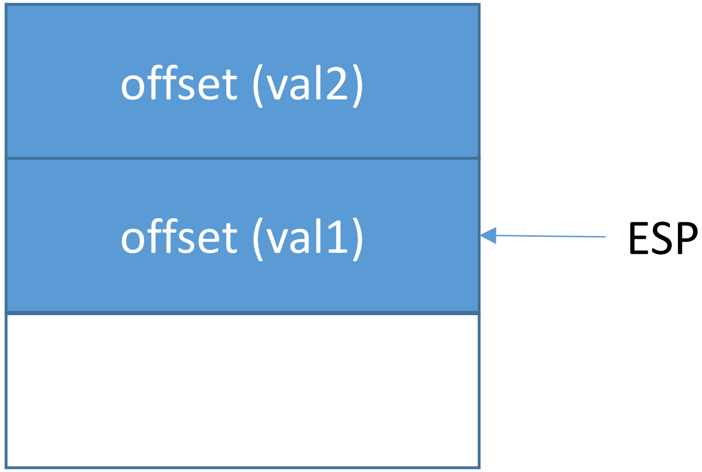
Int sum = AddTwo (val1, val2); in C++

- Arguments are pushed on the stack in reverse order, which is the norm for the C and C++ languages.

Passing by Reference

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

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



Swap (&val1, &val2);   in C++

Passing Arrays

- High-level languages always pass arrays to subroutines by references.
  - They push the address of an array on the stack.
  - Why don’t you want to pass an array by value?
- Example (ArrayFill)

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

Accessing Stack Parameters

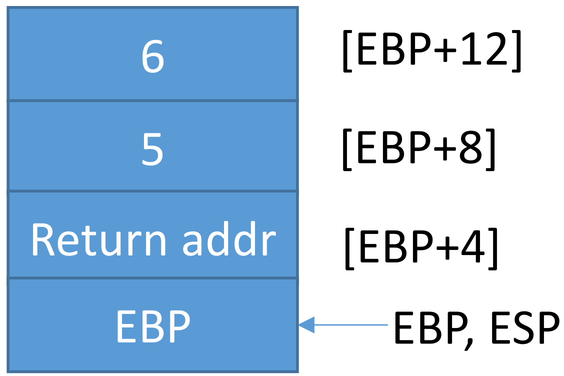
- Following the C and C++ language as an example,
  - They begin with a *prologue* consisting of statement that save the EBP register and point EBP to the top of the stack.
  - The end of the function of consists of an *epilogue* in which the EBP register is stored and the RET instruction returns to the caller.
- Example

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

Base of stack frame

```
AddTwo PROC
    push ebp
    mov ebp, esp
    mov eax, [ebp+12]
    add eax, [ebp+8]
    pop ebp
    ret
AddTwo ENDP
```

AddTwo (5, 6)



Base-Offset Addressing: EBP is the base register and the offset is a constant.

Explicit Stack Parameters

- Explicit Stack Parameters
  - When stack parameters are referenced with expressions such as [ebp+8], we call them *explicit stack parameters*.

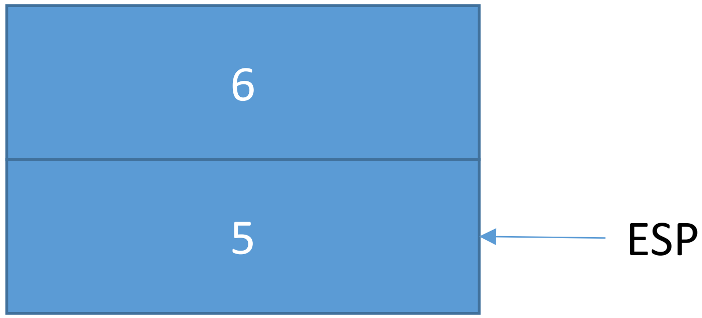
- Symbolic Constants

```
y_param EQU [ebp+12]
x_param EQU [ebp+8]
AddTwo PROC
    push ebp
    mov ebp, esp
    mov eax, y_param
    add eax, x_param
    pop ebp
ret
AddTwo ENDP
```

Cleaning Up the Stack

- How to remove parameters from the stack when a subroutine returns?
  - If the are not removed, a memory leak would result, and the stack would become corrupted.

```
push 6
push 5
call AddTwo
```



It shows the stack after returning from the call.

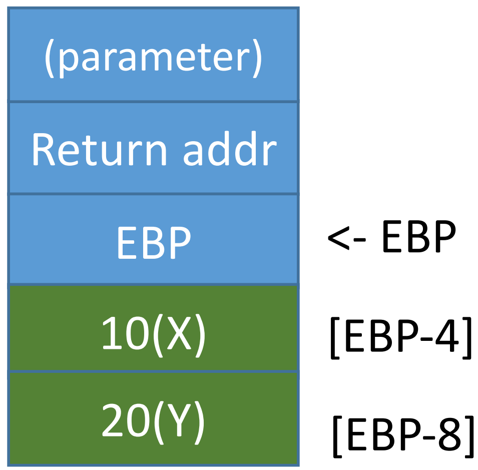
Local Variables

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

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

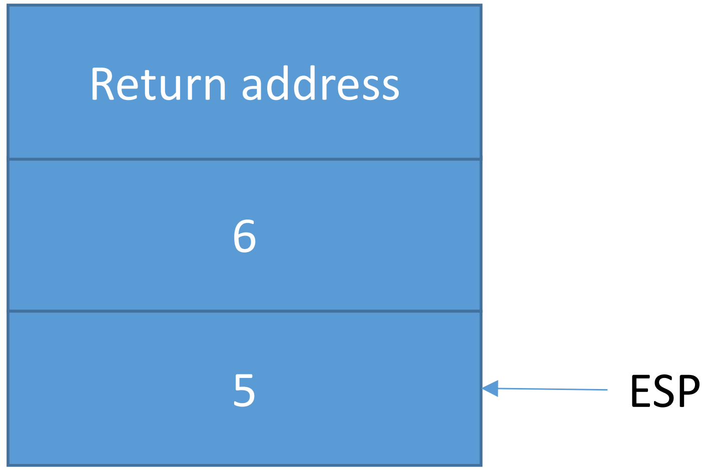
지역변수

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



Stack is corrupted!

```
main PROC
   call Example1
   exit
main ENDP

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



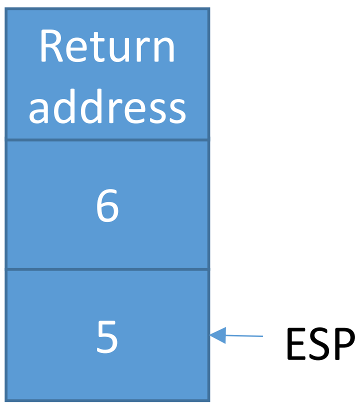
Stack is corrupted!

When the RET instruction in Example1 is about to execute, ESP points to the integer 5 rather than the return address that would take it back to main.

The RET instruction will load the value 5 into the instruction pointer and attempts to transfer control to memory address 5. Assuming the address (5) is outside the program's code boundary, the processor issues a runtime exception, which tells the OS to terminate the program.

32-Bit Calling Conventions

- C Calling Convention
  - When a program calls a subroutine, it follows the CALL instruction with a statement that adds a value to the stack pointer (ESP) equal to the combined sizes of the subroutine parameters.




```
main PROC
   call Example1
   exit
main ENDP

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

Remove arguments from the stack

Here, parameters are 6 and 5.
Each will take up 4 bytes and there are two parameters, which are 2 x 4 bytes so 8 bytes.
Thus, add esp, 8

32-Bit Calling Conventions

- STDCALL Calling Convention

```
AddTwo PROC
        push ebp
        mov ebp, esp
        mov eax, [ebp+12]
        mov eax, [ebp+8]
        pop ebp
        ret 8          ; Clean up the stack
AddTwo ENDP
```

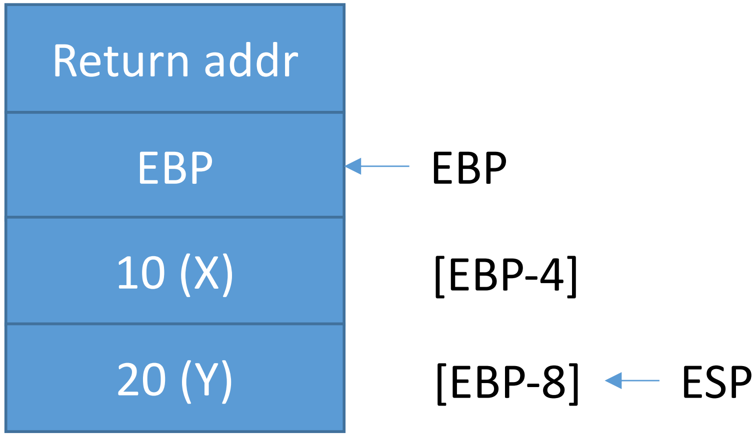
STDCALL will reduce the amount of code generated for subroutine calls (by one Instruction (ret 8 in this example) and ensures that calling programs will never forget to clean up the stack.

Irvine32 library uses the STDCALL calling convention.

Local Variables

- What's wrong with the following code?

```
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

   pop ebp
   ret
MySub ENDP
```



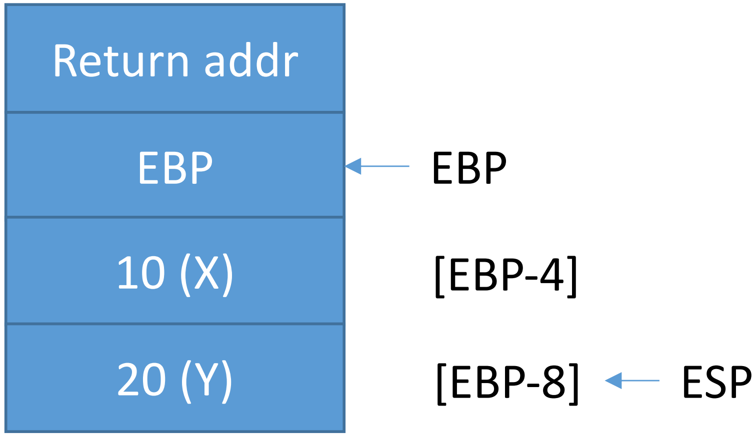
mov esp, ebp; remove locals from stack

If this step is omitted, the POP EBP instruction would set EBP to 20 and the RET instruction would branch to memory location 10, causing the program to halt with a processor exception.

Local Variables

- Correct Codes

```
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
   pop ebp
   ret
MySub ENDP
```



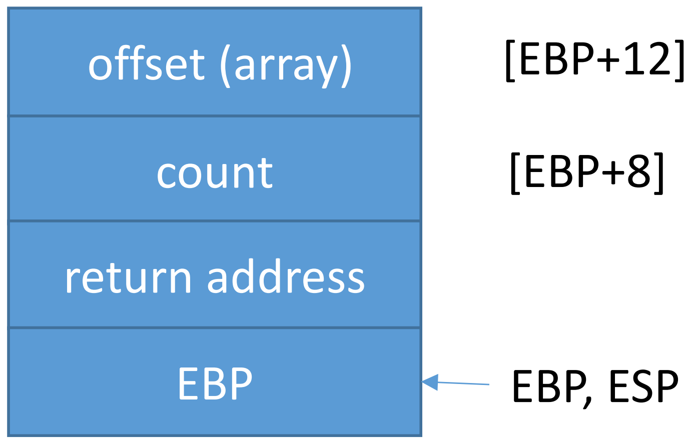
mov esp, ebp; remove locals from stack

If this step is omitted, the POP EBP instruction would set EBP to 20 and the RET instruction would branch to memory location 10, causing the program to halt with a processor exception.

Reference Parameters

- ArrayFill example

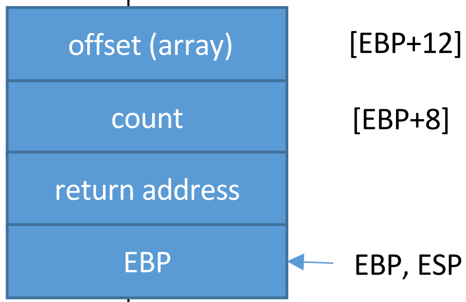
```
.data
count = 100
array DWORD count DUP(?)

.code
push OFFSET array
push count
call ArrayFill

ArrayFill PROC
   push ebp
   mov ebp, esp
   …
```



Reference Parameters

ArrayFill saves the general-purpose registers, retrieves the parameters, and fills the array.

```
ArrayFill PROC
        push ebp
        mov ebp,esp
        pushad

        mov esi,[ebp+12]      ; offset of array
        mov ecx,[ebp+8]       ; array size
        cmp ecx,0             ; ECX < 0?
        jle L2                ; yes: skip over loop
L1:
        mov eax,10000h        ; get random 0 - FFFFh
        call RandomRange      ; from the link library
        mov [esi],eax
        add esi,TYPE DWORD
        loop L1

L2:     popad
        pop ebp
        ret 8  ; clean up the stack
ArrayFill ENDP
```



- *ret 8*: when *ret* instruction is to run, ESP points to return address at the stack, and it will return to calling procedure, then, it will clean up parameters (8 bytes) from the stack.

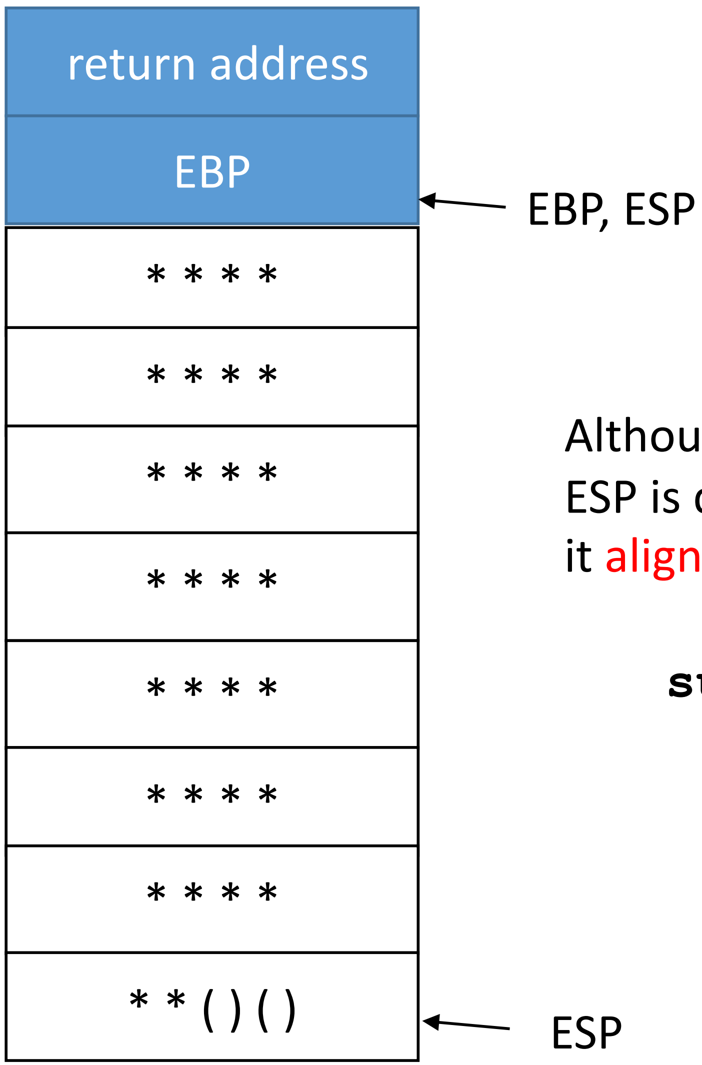
LEA Instruction

- The LEA instruction returns the address of an indirect operand.
  - Because indirect operands contain one or more registers, their offsets are calculated at runtime.
- Example

```
void MakeArray( )
{
        char myString[30];
        for ( int i=0; I<30; i++)
                myString[i] = `*';
}
```

```
 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
```

Load address of myString



Although the array is only 30 bytes ESP is decremented by 32 to keep it aligned on a double word boundary.

```
sub esp, 32
```

LEA Instruction (Cont')

- OFFSET
  - It's not possible to use OFFSET to get the address of a stack parameter because OFFSET only works with addresses known at compile time.

```
mov esi, OFFSET [ebp-30]  ;error
```

- 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
```

ENTER and LEAVE Instructions

- The ENTER instruction automatically creates a stack frame for a called procedure.
  - It reserves stack space for local variables and save EBP on the stack.
- Three major actions
  - Push EBP on the stack (*push ebp*)
  - Set EBP to the base of the stack frame (*mov ebp, esp*)
  - Reserve space for local variables (*sub esp, numbytes*)
- Usages

```
ENTER numbytes, nestinglevel
```

Numbytes is a multiple of 4. nestinglevel is set to be zero always in our program.

- Example1

```
MySub PROC
    enter 0, 0
```

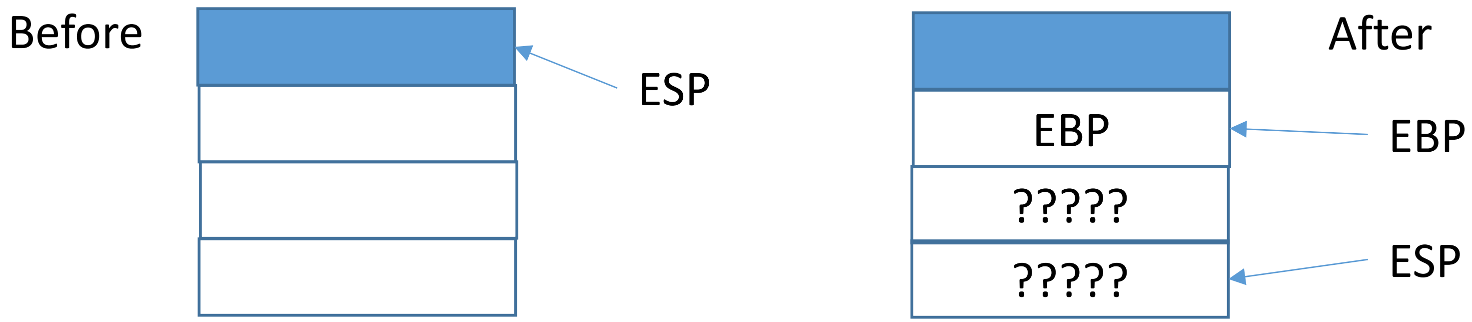
```
MySub PROC
    push ebp
    mov ebp, esp
```

- Example2

```
MySub PROC
    enter 8, 0
```

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

Stack frame before and after ENTER has executed.



If you use the ENTER instruction, it is strongly advised that you also use the LEAVE instruction at the end of the same procedure. Otherwise, the stack space you create for local variables might not be released<u>. This would create the RET instruction to pop the wrong return address off the stack</u>.

LEAVE Instruction

- The LEAVE instruction terminates the stack from 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
```

```
MySub PROC
   push ebp
   mov ebp, esp
   sub esp, 8
   …
   mov esp, ebp
   pop ebp
   ret
```

LOCAL Directive

- Microsoft created the LOCAL directive as a high-level substitute for the ENTER instruction.
  - LOCAL declares one or more local variables by name, assigning them size attribute.
- On the other hand, ENTER only reserves a single unnamed block of stack space for local variables.
- Syntax

```
LOCAL varlist
```

Varlist is a list of variable definitions, separated by commas.
Each variable definition has the following form: *label:type*

The label may be any valid identifier.
Type is a standard type (WORD, DWORD, etc).

LOCAL Directive (Cont’)

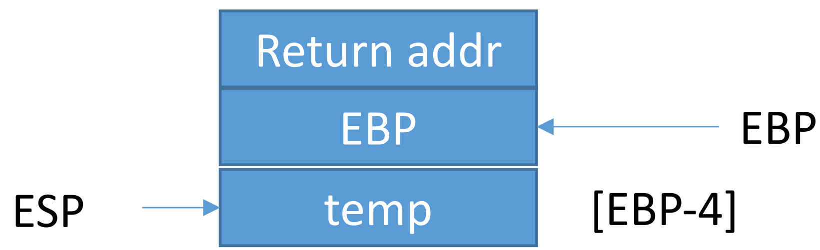
- Example

```
Example1 PROC
   LOCAL temp:DWORD

   mov eax, temp
   ret
Examle1 ENDP
```

MASM Code Generation

```
push ebp
mov ebp, esp
add esp, 0FFFFFFFCh ; substract -4
mov eax, [ebp-4]; add -4 to ESP
leave
ret
```

- Stack Frame’s diagram



8.3 Recursion

- Recursive Subroutine
  - Is one that calls itself, either directly or indirectly.
    - Proc A calls proc B, which in turn calls proc A
  - Be a powerful tool when working with data structures that have repeating patterns.
- Examples
  - Linked list, Connected graphs (where a program must retrace its path)

Endless Recursion

- A procedure calls itself repeatedly without every stopping.

```
; Endless Recursion
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
Endless ENDP
END main
```

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

Each time the proc calls itself, it uses up 4 bytes of stack space (return address) when the CALL instruction pushes the return address.

Never executes!

Recursively Calculating a Sum

- CalcSum recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum.

```
; Sum of Integers
 main PROC
     mov ecx, 5                  ; count = 5
     mov eax, 0                  ; holds the sum
     call CalSum                 ; calculate sum
L1:  call WriteDec               ; display EAX
     call Crlf
     exit
 main ENDP
```

```
CalcSum PROC
    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
```

Recursively Calculating a Sum

- CalcSum recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum.

```
CalcSum PROC
      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
```

Stack frame:

Pushed On Stack	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

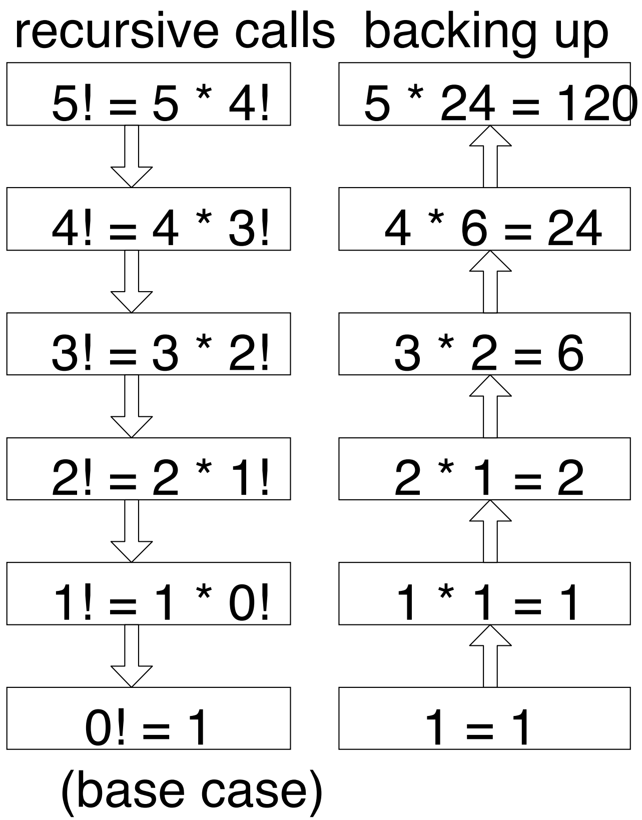
- Even a simple recursive procedure makes ample use of the stack.
- ** Four bytes ** of stack space are used up each time a procedure call take place because the return address must be saved on the stack.

Calculating a Factorial (1)

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

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

As each call instance returns, the product it returns is multiplied by the previous value of n.

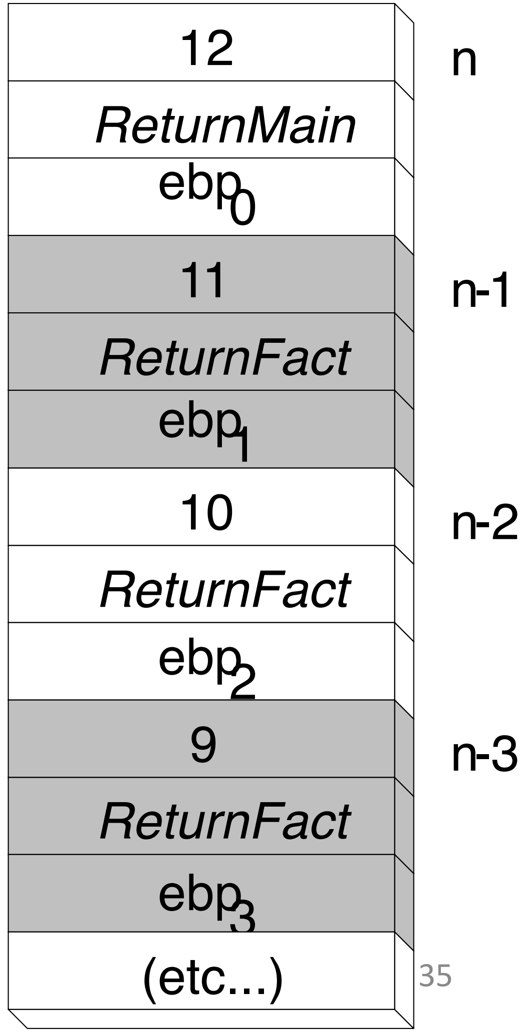


Calculating a Factorial (2)

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

```asm
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

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

ReturnFact:
    mov  ebx,[ebp+8]    ; get n
    mul  ebx            ; ax = ax * bx

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

Stack frame for calculating 12!



Each call uses 12 bytes of stack

Multimodule Programs

- A multimodule program is a program whose source code has been divided up into separate ASM files.

- Each ASM file(module) is assembled into a separate OBJ file.

- All OBJ files belonging to the same program are linked using the link utility into a single EXE file.
  - This process is called static linking.

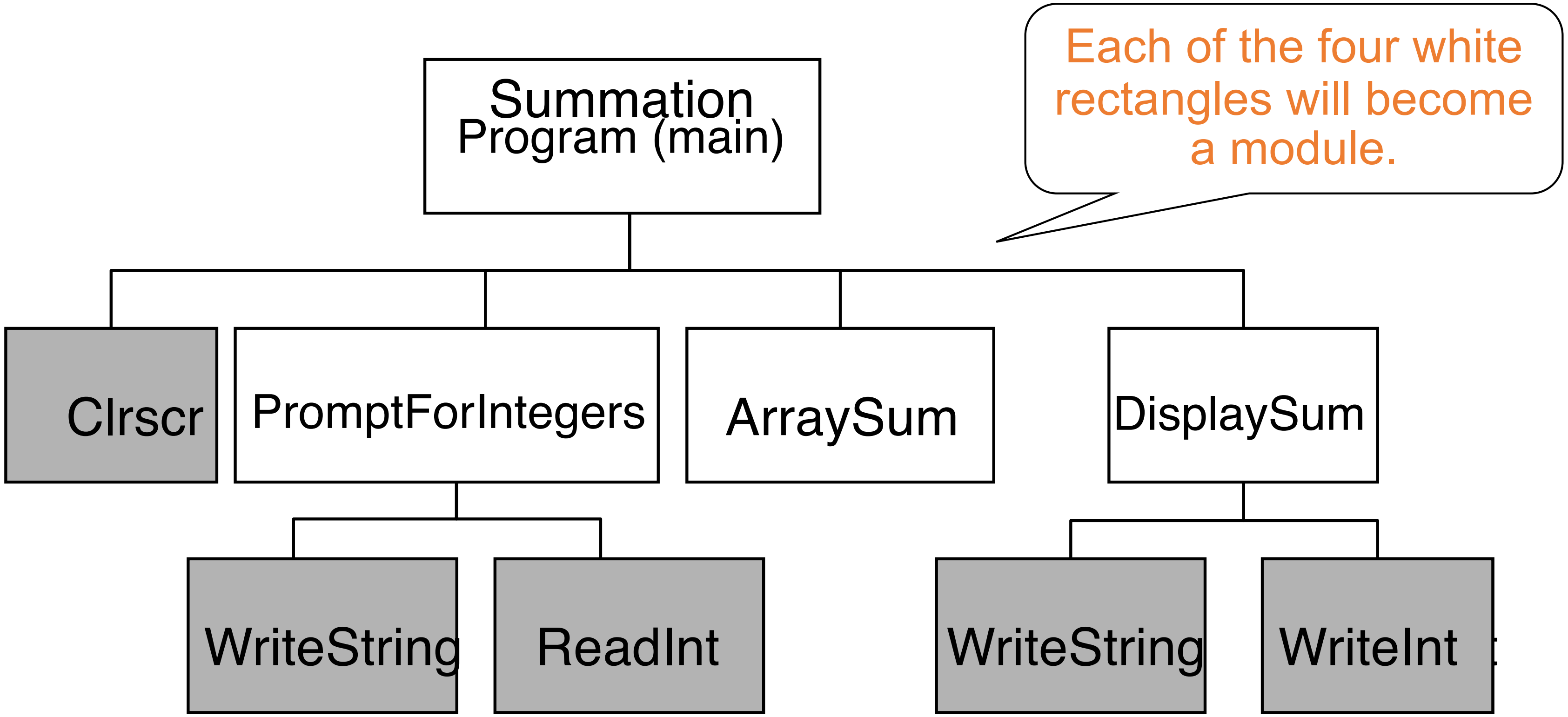
Multimodule Programs (Advantages)

- Large programs are easier to write, maintain, and debug when divided into separate source code modules.
- When changing a line of code, only its enclosing module needs to be assembled again. Linking assembled modules requires little time.
- A module can be a container for logically related code and data (think object-oriented here...)
  - encapsulation: procedures and variables are automatically hidden in a module unless you declare them public.

Creating a Multimodule Program

- Here are some basic steps to follow when creating a multimodule program:
  - Create the main module.
  - Create a separate source code module for each procedure or set of related procedures.
  - Create an include file that contains procedure prototypes for external procedures (ones that are called between modules).
  - Use the INCLUDE directive to make your procedure prototypes available to each module.

Example: ArraySum Program



sample program output

```
Enter a signed integer: -25

Enter a signed integer: 36

Enter a signed integer: 42

The sum of the integers is: +53
```

- INCLUDE File
  - The sum.inc file contains prototypes for external functions that are not in the Irvine32 library:

```
INCLUDE Irvine32.inc

PromptForIntegers PROTO,
    ptrPrompt:PTR BYTE,          ; prompt string
    ptrArray:PTR DWORD,          ; points to the array
    arraySize:DWORD              ; size of the array

ArraySum PROTO,
    ptrArray:PTR DWORD,          ; points to the array
    count:DWORD                  ; size of the array

DisplaySum PROTO,
    ptrPrompt:PTR BYTE,          ; prompt string
    theSum:DWORD                 ; sum of the array
```

- Individual Modules
  - Main

```
TITLE Integer Summation Program   (Sum_main.asm)

INCLUDE sum.inc

; modify Count to change the size of the array:
Count = 3

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

- Main

```
.code
main PROC
        call Clrscr
        INVOKE PromptForIntegers,    ; input the array
          ADDR prompt1,
          ADDR array,
          Count

        INVOKE ArraySum,             ; sum the array
          ADDR array,         ; (returns sum in EAX)
          Count

        mov sum,eax           ; save the sum
        INVOKE DisplaySum,           ; display the sum
          ADDR prompt2,
          sum
        call Crlf
        INVOKE ExitProcess,0
main ENDP
END main
```

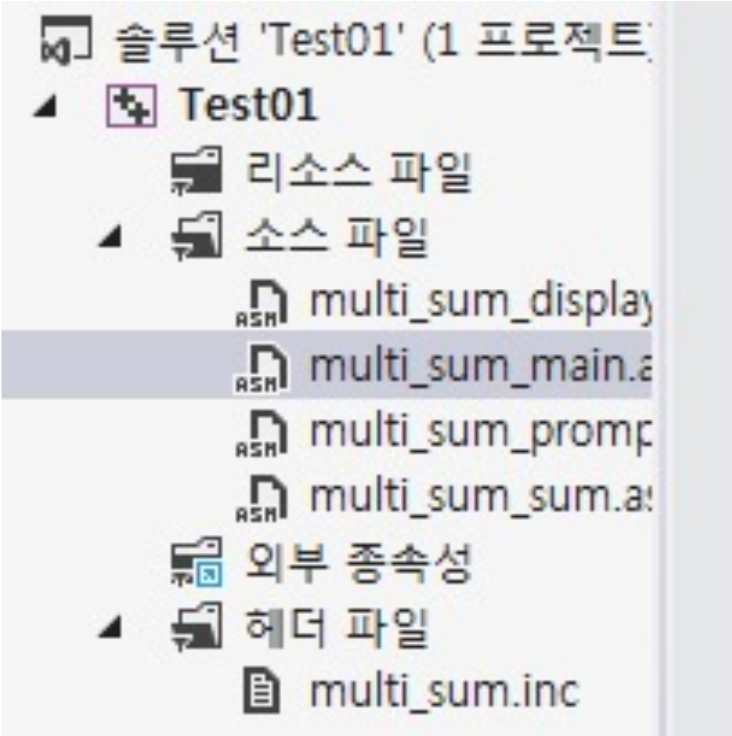
```
TITLE Prompt For Integers          (_prompt.asm)
INCLUDE sum.inc
.code
;-----------------------------------------------------
PromptForIntegers PROC,
  ptrPrompt:PTR BYTE,  ; prompt string
  ptrArray:PTR DWORD,  ; pointer to array
  arraySize:DWORD             ; size of the array
;
; Prompts the user for an array of integers and fills
; the array with the user's input.
; Returns:  nothing
;-----------------------------------------------------
       pushad            ; save all registers
       mov  ecx,arraySize
       cmp  ecx,0        ; array size <= 0?
       jle  L2           ; yes: quit
       mov  edx,ptrPrompt       ; address of the prompt
       mov  esi,ptrArray
L1:    call WriteString         ; display string
       call ReadInt             ; read integer into EAX
       call Crlf                ; go to next output line
       mov  [esi],eax           ; store in array
       add  esi,4               ; next integer
       loop L1
L2:    popad             ; restore all registers
       ret
PromptForIntegers ENDP
END
```

```
TITLE ArraySum Procedure                    (_arrysum.asm)
INCLUDE sum.inc
.code
;-----------------------------------------------------
ArraySum PROC,
        ptrArray:PTR DWORD,     ; pointer to array
        arraySize:DWORD         ; size of array
;
; Calculates the sum of an array of 32-bit integers.
; Returns:  EAX = sum
;-----------------------------------------------------
        push ecx        ; don't push EAX
        push esi
        mov  eax,0      ; set the sum to zero
        mov  esi,ptrArray
        mov  ecx,arraySize
        cmp  ecx,0      ; array size <= 0?
        jle  AS2        ; yes: quit
AS1:
        add  eax,[esi] ; add each integer to sum
        add  esi,4      ; point to next integer
        loop AS1        ; repeat for array size
AS2:
        pop esi
        pop ecx ; return sum in EAX
        ret
ArraySum ENDP
END
```

```
TITLE DisplaySum Procedure              (_display.asm)

INCLUDE Sum.inc

.code
;-----------------------------------------------------
DisplaySum PROC,
       ptrPrompt:PTR BYTE,    ; prompt string
       theSum:DWORD    ; the array sum
;
; Displays the sum on the console.
; Returns:  nothing
;-----------------------------------------------------
       push eax
       push edx

       mov  edx,ptrPrompt     ; pointer to prompt
       call WriteString
       mov  eax,theSum
       call WriteInt   ; display EAX
       call Crlf

       pop  edx
       pop  eax
       ret
DisplaySum ENDP

END
```

Assemble and Link

- In Visual Studio : add files to the project, and build them.




```
; modify Count to change the s
Count = 3

.data
prompt1 BYTE   "Enter a signed
prompt2 BYTE   "The sum of the
array   DWORD  Count DUP(?)
sum     DWORD  ?

.code
main PROC
    call Clrscr
```