

# Assembly Language for Intel-Based Computers, 5<sup>th</sup> Edition

Kip R. Irvine

## Chapter 8: Procedimentos Avançados

*Slides prepared by Kip R. Irvine*

*Revision date: June 4, 2006*

(c) Pearson Education, 2006-2007. All rights reserved. You may modify and copy this slide show for your personal use, or for use in the classroom, as long as this copyright statement, the author's name, and the title are not changed.

# índice

- **Stack Frames**
- Recursão
- INVOKE, ADDR, PROC e PROTO
- Diretiva Model / Convenção de Chamadas

# Parâmetros de pilha

- Mais conveniente que parâmetros de registradores
- Duas possíveis formas de chamar DumpMem. Qual é a mais fácil?

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

OU

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

# Stack Frame

- Também conhecido como *activation record*
- Área da pilha destinada para o endereço de retorno, passagem de parâmetros, registradores salvos, e variáveis locais

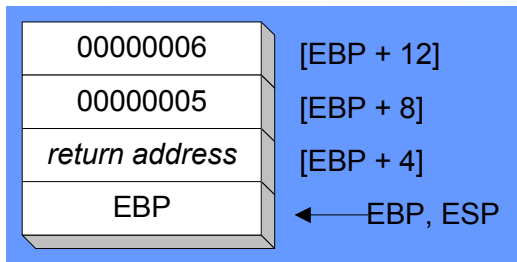
*Por que fazer isso ? Não está funcionando tudo ok ?*

*Resposta: Para organizar a chamada e retorno de funções, passagem de parâmetros e variáveis locais de modo independente, em termos de endereçamento de memória.*

*Raiz do problema: cada chamada de função implica em uma alocação de memória diferente, impossível de ser determinada em tempo de compilação.*

# Stack Frame

- Stack Frame: Criado pelos seguintes passos:
  - O programa chamador salva os argumentos na pilha e chama o procedimento.
  - O procedimento chamado salva EBP na pilha e carrega o valor de ESP em EBP.
    - **EBP passa a ser base para os parâmetros do procedimento chamado.**
  - Se variáveis locais são necessárias, uma constante é subtraída de ESP para abrir espaço na pilha.



**EBP: Extended BASE Pointer**

**ESP: Extended STACK Pointer**

*Específico da função em execução*

*Geral a todo o programa em execução*

# Acesso explícito a parâmetros de pilha

- Um procedimento pode acessar explicitamente parâmetros de pilha usando offsets a partir de EBP.
  - Exemplo: `[ebp + 8]`
- EBP é chamado de base pointer ou frame pointer porque contém o endereço da base do stack frame.
- EBP não muda de valor durante a execução do procedimento.
- EBP deve ser restaurado com seu valor original quando o procedimento termina (1 EBP por função ativa).

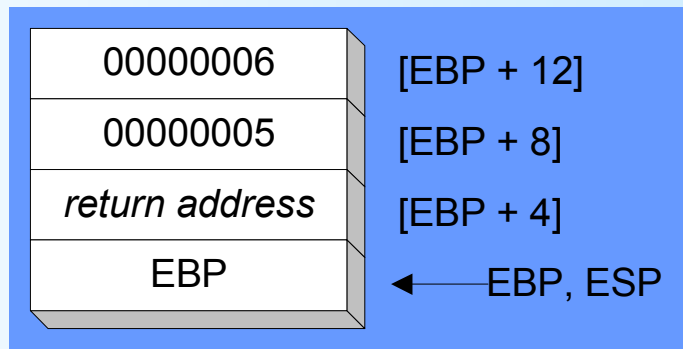
# Instrução RET

- *Retorna da subrotina*
- Recupera (Pop) o ponteiro de instrução (EIP ou IP). O controle é transferido para o endereço recuperado.
- Sintaxe:
  - **RET**
  - **RET *n***
- Operando opcional *n* causa **pop** de *n* bytes da pilha após EIP (ou IP) receber o endereço de retorno.

# Exemplo de Stack Frame

```
.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
    .
    .
```





# AddTwo Procedure (1 de 2)

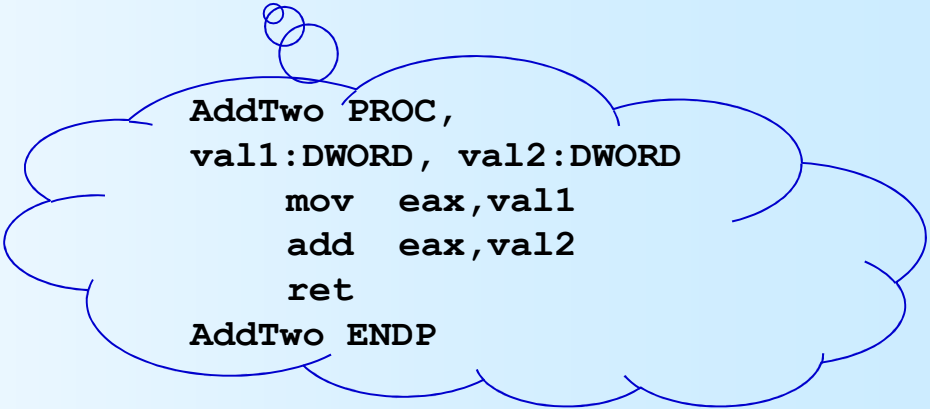
- Reescrevendo AddTwo Procedure

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

# AddTwo Procedure (2 de 2)

- MASM gera o seguinte código quando é montado AddTwo (do slide anterior):

```
AddTwo PROC,  
val1:DWORD, val2:DWORD  
    push ebp  
    mov  ebp, esp  
    mov  eax, val1  
    add  eax, val2  
    mov  esp, ebp  
    pop  ebp  
    ret  8  
AddTwo ENDP
```



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

# Passando Argumentos por Referência (1 de 2)

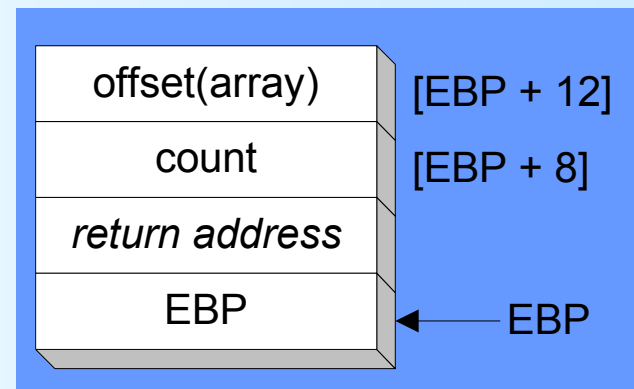
- O procedimento `ArrayFill` preenche um vetor com inteiros aleatórios de 16-bits
- O programa chamador passa o endereço do vetor, e o número de elementos:

```
.data
count = 100
array WORD count DUP(?)
.code
    push OFFSET array
    push COUNT
    call ArrayFill
```

# Passando Argumentos por Referência (2 de 2)

ArrayFill pode referenciar um vetor sem saber o nome :

```
ArrayFill PROC
    push ebp
    mov  ebp, esp
    pushad
    mov  esi, [ebp+12]
    mov  ecx, [ebp+8]
    .
    .
```



ESI aponta para o início do vetor, para facilitar o uso de um loop para acessar cada elemento.

*Argumentos por Referência: “EBP + N”*

# Variáveis locais

- Para criar variáveis locais explicitamente, subtrair o seu tamanho de ESP.
- O seguinte exemplo cria e inicializa duas variáveis locais de 32-bits( locA e locB):

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

*Variáveis Locais: “EBP - N”*

# Instrução LEA

- A instrução LEA retorna os offsets dos operandos
  - Mais genérico que o operador OFFSET que pode retornar somente offsets de constantes.
- LEA é requerida para a obtenção de offset de um parâmetro de pilha ou variável local. Por exemplo:

```
CopyString PROC,  
    count:DWORD  
    LOCAL temp:BYTE
```

```
    mov edi,OFFSET count        ; invalid operand  
    mov esi,OFFSET temp         ; invalid operand  
    lea edi,count               ; ok  
    lea esi,temp                ; ok
```

# Sua vez . . .

- Criar um procedimento **Difference** que subtrai o primeiro argumento do segundo. Exemplo:

```
push 14                ; first argument
push 30                ; second argument
call Difference        ; EAX = 16
```

```
Difference PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 8]   ; second argument
    sub  eax,[ebp + 12]  ; first argument
    pop  ebp
    ret  8
Difference ENDP
```

# Classificação de parâmetros

- Parâmetro de entrada é um dado passado do programa chamador para o procedimento.
  - O procedimento chamado não modifica o parâmetro. A modificação é confinada somente dentro do procedimento.
- Um parâmetro de saída é criado passando um ponteiro a uma variável quando um procedimento é chamado.
  - O procedimento não usa o valor existente na variável, mas coloca um novo valor antes do retorno.
- Um parâmetro de entrada-saída é um ponteiro para uma variável contendo uma entrada que será usada e modificada pelo procedimento.
  - A variável passada pelo programa chamador é modificada.



# Exemplo: trocando dois inteiros

O procedimento Swap troca valores de dois inteiros de 32 bits. pValX e pValY não trocam valores, mas os inteiros que eles apontam são modificados.

```
Swap PROC USES eax esi edi,  
    pValX:PTR DWORD,          ; pointer to first integer  
    pValY:PTR DWORD           ; pointer to second integer  
  
    mov esi,pValX              ; get pointers  
    mov edi,pValY  
    mov eax,[esi]              ; get first integer  
    xchg eax,[edi]             ; exchange with second  
    mov [esi],eax              ; replace first integer  
    ret  
Swap ENDP
```

# ENTER e LEAVE

- Instrução ENTER cria um stack frame para um procedimento chamado: tem dois operandos sendo o primeiro o número de bytes para variáveis locais, e o segundo o nível de “aninhamento” (nesting level)
  - salva EBP na pilha e novo EBP aponta para a base do stack frame
- Reserva espaço para variáveis locais

- Exemplo:

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

- Equivale a:

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

A instrução LEAVE desfaz o stack frame, resumindo-se em:

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

# Diretiva LOCAL

- Uma variável local é criada, usada, e destruída dentro de um procedimento
- A diretiva LOCAL declara uma lista de variáveis locais
  - Segue imediatamente à diretiva PROC
  - É atribuído um tipo a cada variável

- Sintaxe:

**LOCAL *varlist***

Exemplo:

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

LOCAL substitui ENTER, atribuindo nomes às variáveis locais

# Exemplo de LOCAL (1 de 2)

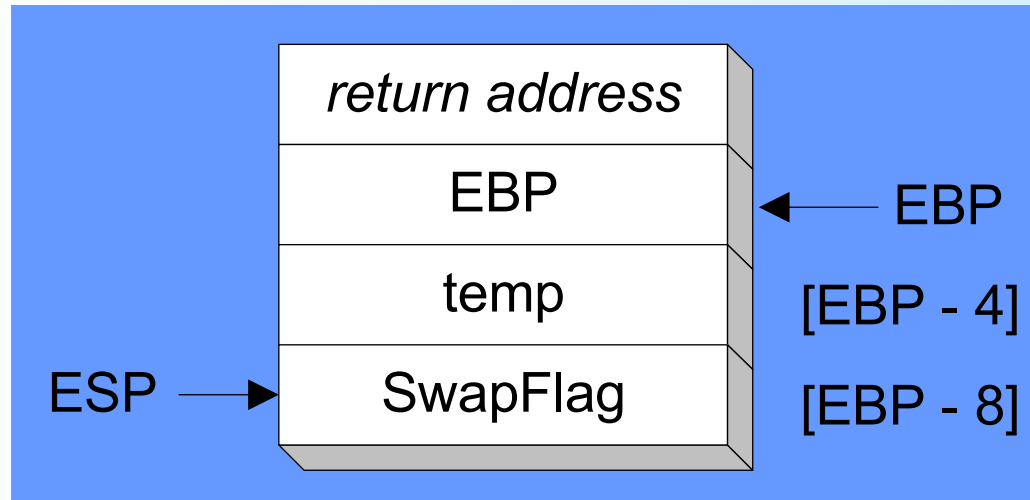
```
BubbleSort PROC
    LOCAL temp:DWORD, SwapFlag:BYTE
    . . .
    ret
BubbleSort ENDP
```

MASM gera o seguinte código:

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

## Exemplo de LOCAL (2 de 2)

Diagrama do **stack frame** para o procedimento BubbleSort:



# Variáveis locais menores que Doubleword

- Variáveis locais podem ser de diferentes tamanhos
- Como são criadas numa pilha com a diretiva LOCAL:
  - 8-bit: no próximo byte da pilha
  - 16-bit: no próximo word da pilha
  - 32-bit: no próximo doubleword da pilha

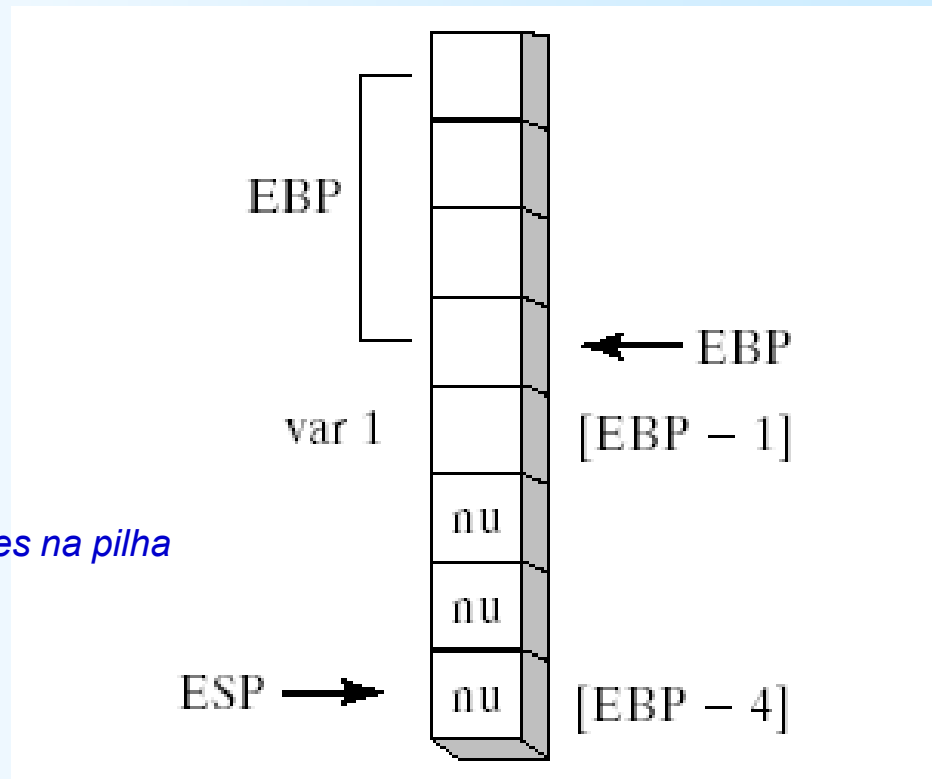
# Local Byte Variable

```
Example1 PROC  
    LOCAL var1:BYTE  
    mov al,var1                ; [EBP - 1]  
    ret  
Example1 ENDP
```

Particularidade do MASM:

*var1 = 1 byte → EBP - 1*

*... mas o default p/ armazenar valores na pilha  
é 4 bytes → ESP - 4*



# Procedimento WriteStackFrame

- Procedimento da biblioteca Irvine.
- Mostra o conteúdo do **stack frame** corrente
- Protótipo:

```
WriteStackFrame PROTO,  
    numParam:DWORD,      ; number of passed parameters  
    numLocalVal: DWORD,  ; number of DWordLocal variables  
    numSavedReg: DWORD   ; number of saved registers
```



# Exemplo de WriteStackFrame

```
main PROC
    mov eax, 0EAEAEAEAh
    mov ebx, 0EBEBEBEBh
    INVOKE aProc, 1111h, 2222h
    exit
main ENDP
```

```
aProc PROC USES eax ebx,
    x: DWORD, y: DWORD
    LOCAL a:DWORD, b:DWORD
    PARAMS = 2
    LOCALS = 2
    SAVED_REGS = 2
    mov a, 0AAAAh
    mov b, 0BBBBh
    INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS
```

# Próximo Tópico

- Stack Frames
- **Recursão**
- INVOKE, ADDR, PROC e PROTO
- Diretiva Model / Convenção de Chamadas



*Intervalo ?*

# Recursão

- O que é recursão?
- Calcular recursivamente uma soma
- Cálculo de um Fatorial

# O que é recursão?

- Um processo criado quando . . .
  - Um procedimento chama a si próprio
    - ou
  - Procedimento A chama procedimento B, que por sua vez chama o procedimento A

# Calcular recursivamente uma soma

O procedimento CalcSum calcula recursivamente a soma dos inteiros de 1 a N. Recebe: ECX = N. Retorna: EAX = soma

```
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
```

*Importante em toda função recursiva:  
Condição de Parada.*

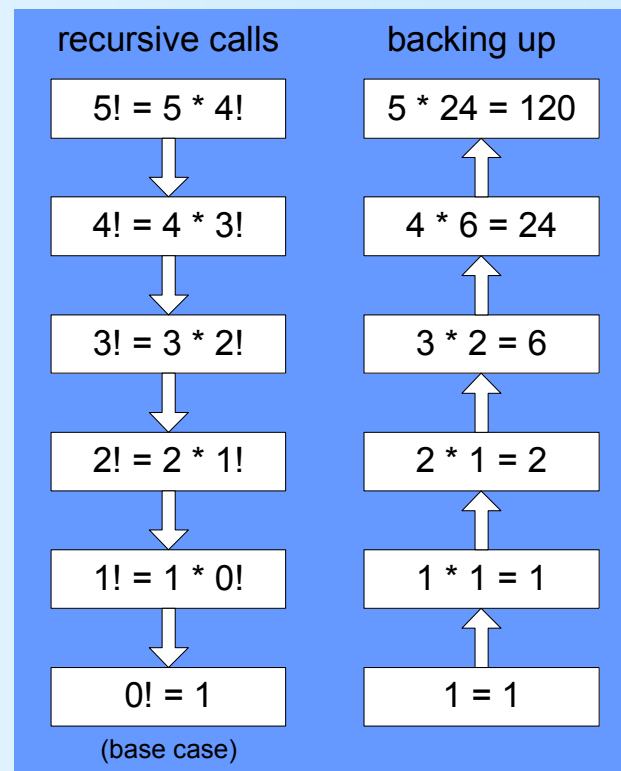
*Obs: este exemplo não usa o stack frame,  
tudo é feito via registradores.*

# Calculando um fatorial (1 de 3)

Esta função calcula o fatorial do inteiro  $n$ . Um novo valor de  $n$  é salvo em cada stack frame:

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

A cada retorno de uma chamada recursiva, o produto que retorna é multiplicado pelo valor prévio de  $n$ .



# Calculando um fatorial (2 de 3)

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                    ; eax = eax * ebx
```

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

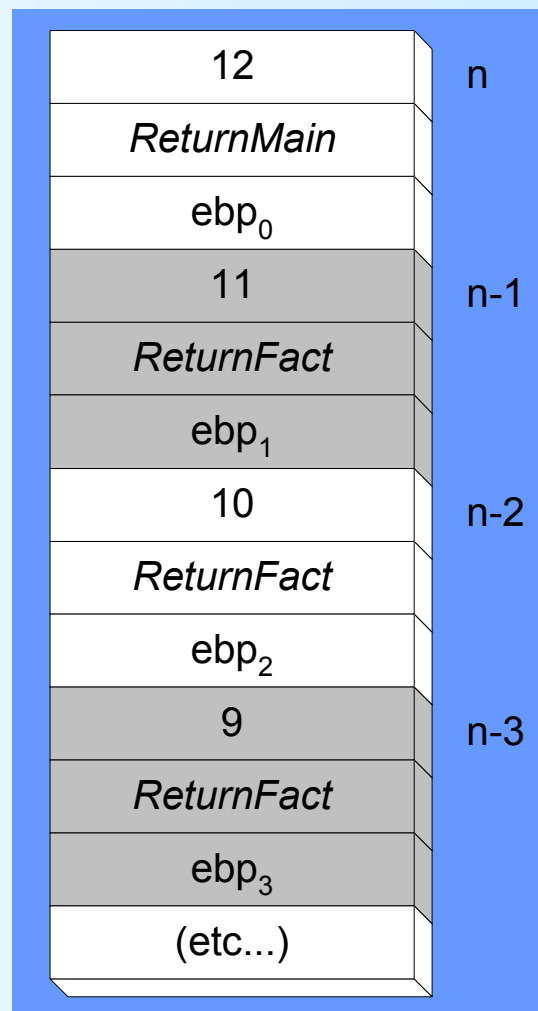
Factorial ENDP

# Calculando um fatorial (3 de 3)

Supondo o cálculo de 12!

O diagrama mostra os primeiros stack frames criados por chamadas recursivas a Factorial

Cada chamada recursiva usa 12 bytes do espaço de pilha.





# Próximo Tópico

- Stack Frames
- Recursão
- **INVOKE, ADDR, PROC e PROTO**
- Diretiva Model / Convenção de Chamadas

# Diretiva INVOKE

- A diretiva INVOKE é uma poderosa substituição para a instrução CALL que permite a passagem de múltiplos argumentos
- Sintaxe:  
`INVOKE procedureName [, argumentList]`
- *ArgumentList* é uma lista opcional delimitada por vírgula de argumentos de procedimento
- Argumentos podem ser:
  - Valores imediatos e expressões inteiras
  - Nomes de variáveis
  - Endereços e expressões ADDR
  - Nomes de registradores

# Exemplos de INVOKE

```
.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]
```

# Operador ADDR

- Retorna um ponteiro de uma variável:
- Exemplo:

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

# Diretiva PROC (1 de 2)

- A diretiva PROC declara um procedimento com uma lista opcional de parâmetros.

- Sintaxe:

*label* PROC paramList

- *paramList* é uma lista de parâmetros separados por vírgulas. Cada parâmetro tem a seguinte sintaxe:

*paramName* : *type*

*type* deve ser um dos tipos padrões (BYTE, SBYTE, WORD, etc.), ou um ponteiro a um desses tipos.

## Diretiva PROC (2 de 2)

- Os formatos alternativos permitem lista de parâmetros serem em uma ou mais linhas separadas:

*label* PROC,      ←————— vírgula obrigatória  
                 paramList

- Os parâmetros podem ser na mesma linha . . .

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

- Ou podem ser em linhas separadas:

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

# Exemplos de PROC (1 de 3)

- O procedimento AddTwo recebe dois inteiros e retorna a soma em EAX.

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

## Exemplos de PROC (2 de 3)

FillArray recebe um ponteiro a um vetor de bytes, um único byte que é copiado a cada elemento do vetor, e o tamanho do vetor.

```
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
```



# Exemplos de PROC (3 de 3)

```
Swap PROC,  
    pValX:PTR DWORD,  
    pValY:PTR DWORD  
    . . .  
Swap ENDP
```

```
ReadFile PROC,  
    pBuffer:PTR BYTE  
    LOCAL fileHandle:DWORD  
    . . .  
ReadFile ENDP
```

# Diretiva PROTO

- Cria um protótipo de procedimento
- Sintaxe:
  - *label* PROTO *paramList*
- Todo procedimento chamado pela diretiva INVOKE deve ter um protótipo
- Uma definição de procedimento completa pode também servir como o próprio protótipo

# Diretiva PROTO

- Configuração padrão: PROTO aparece no topo da listagem do programa, INVOKE aparece no segmento de código, e a implementação do procedimento ocorre depois.

```
MySub PROTO                ; procedure prototype

.code
INVOKE MySub                ; procedure call

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

# Exemplo de PROTO

- Protótipo para o procedimento ArraySum , mostrando a sua lista de parâmetros:

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

# Exemplo de PROTO

AddTwo PROTO, val1:DWORD, val2:DWORD

.data

.code

main PROC

mov eax, 0

INVOKE AddTwo, 1,2

exit

main ENDP

AddTwo PROC,  
val1:DWORD, val2:DWORD

mov eax,val1

add eax,val2

ret

AddTwo ENDP

END main

# Próximo Tópico

- Stack Frames
- Recursão
- INVOKE, ADDR, PROC e PROTO
- **Diretiva Model / Convenção de Chamadas**

# Diretiva .MODEL

- A diretiva .MODEL especifica o modelo de memória de um programa, e algumas opções relacionadas. Syntax:
  - `.MODEL memorymodel [,modeloptions]`
- *memorymodel* pode ser um dos seguintes:
  - tiny, small, medium, compact, large, huge, or flat
- *modeloptions* inclui **especificadores de linguagem**:
  - Esquema p/ passagem de parâmetros (calling convention)
  - Esquema p/ nome de procedures (naming convention)

# Memory Models

- Modo Real suporta: tiny, small, medium, compact, large, and huge models.
- Modo Protegido suporta apenas o modelo **flat**.

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

Flat model: segmento único p/ código e dados, até 4 GB.  
Todos os offsets são de 32 bits.



# Especificadores de Linguagem

## Convenção de Chamadas:

- **C:** *Usado em programas C/C++*
  - Argumentos de procedures são armazenados na pilha em ordem reversa (direita p/ esquerda).
  - A procedure **que chama** é responsável por limpar a pilha.
- **STDCALL:** *Usado c/ APIs do Windows*
  - Argumentos de procedures são armazenados na pilha em ordem reversa (direita p/ esquerda).
  - A procedure **chamada** é responsável por limpar a pilha.

# Especificadores de Linguagem

## Convenção de Nomes:

- Define como os nomes de funções são “decorados” p/ exportação ao linker.
- **C**
  - Ex: `int soma (int a, int b);` → exportado como `_soma`
- **STDCALL**
  - Ex: `int soma (int a, int b);` → exportado como `_soma@8`

*The End*