

CHAPTER 8: ADVANCED PROCEDURES

Chapter Overview

- **Stack Frames**
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Local Variables

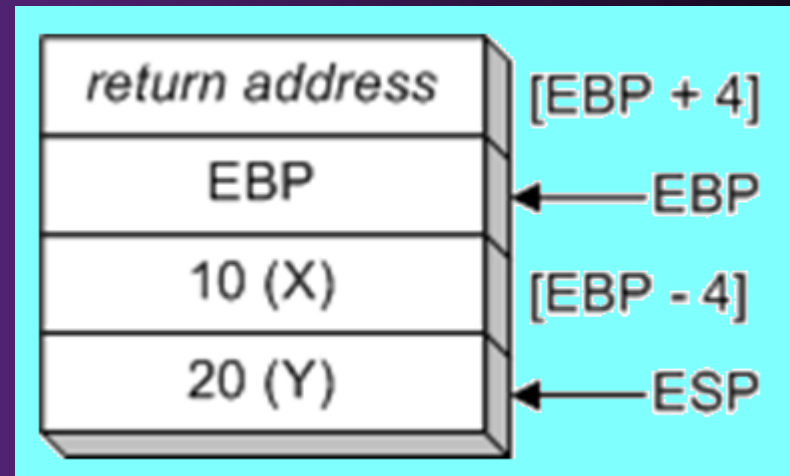
- Only statements within subroutine can view or modify local variables
- Storage used by local variables is released when subroutine ends
- local variable name can have the same name as a local variable in another function without creating a name clash
- Essential when writing recursive procedures, as well as procedures executed by multiple execution threads

Local Variables

To explicitly create local variables, subtract total size from ESP.

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

```
MySub PROC  
    push    ebp  
    mov     ebp,esp  
    sub     esp,8                ; create variables  
    mov     DWORD PTR [ebp-4],10 ; X  
    mov     DWORD PTR [ebp-8],20 ; Y  
    ; ... Do something  
    mov     esp,ebp            ; remove locals from stack  
    pop     ebp  
    ret  
MySub ENDP
```



LocalVars.asm

ENTER and LEAVE

- ENTER instruction creates stack frame for a called procedure
 - pushes EBP on the stack (*push ebp*)
 - sets EBP to the base of the stack frame (*mov ebp, esp*)
 - reserves space for local variables (*sub esp, n*)
 - Syntax: **ENTER numBytesReserved**, nestingLevel (=0)
- LEAVE instruction terminates the stack frame for a called procedure
 - restores ESP to release local variables (*mov esp, ebp*)
 - pops EBP for the caller (*pop ebp*)

LEAVE Instruction

Terminates the stack frame for a procedure.

Equivalent operations

MySub PROC
 enter 8,0

...

...

...

 leave

 ret

MySub ENDP

push ebp
mov ebp,esp
sub esp,8 ; 2 local DWORDs

mov esp,ebp ; free local space
pop ebp

LOCAL Directive

- The LOCAL directive declares a list of local variables
 - immediately follows the PROC directive
 - each variable is assigned a type
- Syntax:

LOCAL *varlist*

Example:

MySub PROC

LOCAL var1:BYTE, var2:WORD, var3:SDWORD

Using LOCAL

Examples:

```
LOCAL flagVals[20]:BYTE      ; array of bytes
```

```
LOCAL pArray:PTR WORD       ; pointer to an array
```

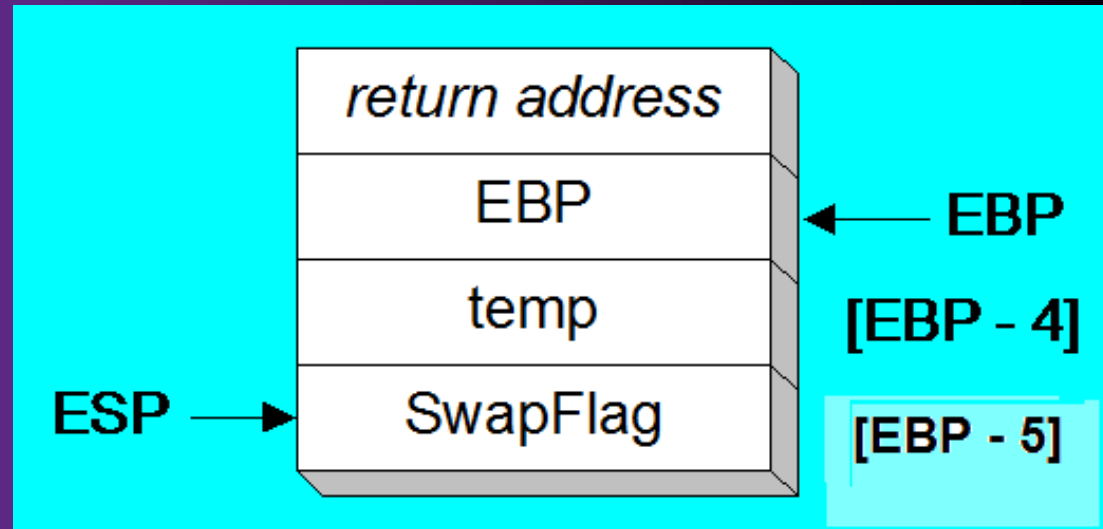
```
myProc PROC                  ; procedure  
    LOCAL t1:BYTE,           ; local variables
```


LOCAL Example

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

MASM generates:

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



See LocalExample.asm

LEA Instruction

- LEA returns offsets of direct and indirect operands
 - OFFSET operator only returns constant offsets
- LEA required when obtaining offsets of stack parameters & local variables
- 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
```

LEA Example

Suppose you have a Local variable at [ebp-8]

And you need the address of that local variable in ESI

You cannot use this:

```
mov esi, OFFSET [ebp-8] ; error
```

Use this instead:

```
lea esi, [ebp-8]
```

What's Next

- Stack Frames
- **Recursion**
 - **Reading material**
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

What's Next

- Stack Frames
- Recursion
- **INVOKE, ADDR, PROC, and PROTO**
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- PROC Directive
- PROTO Directive
- Parameter Classifications
- Example: Exchanging Two Integers
- Debugging Tips

Not in 64-bit
mode!

INVOKE Directive

- In 32-bit mode, the INVOKE directive is a powerful replacement for Intel's CALL instruction that lets you pass multiple arguments

- Syntax:

INVOKE *procedureName* [, *argumentList*]

- *ArgumentList* is an optional comma-delimited list of procedure arguments
- Arguments can be:
 - immediate values and integer expressions
 - variable names
 - address and ADDR expressions
 - register names

INVOKE Examples

```
.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]
```


Not in 64-bit
mode!

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
- Simple example:

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

Not in 64-bit
mode!

PROC Directive (1 of 2)

- The PROC directive declares a procedure
 - Syntax:
label PROC [attributes] [USES regList], paramList
- The USES clause must be on the same line as PROC.
- Attributes: distance, language type, visibility
- ParamList is a list of parameters separated by commas.
label PROC, parameter1, parameter2, ..., parameterN
 - Each parameter has the following syntax:
paramName : type
type must either be one of the standard ASM types (BYTE, SBYTE, WORD, etc.), or it can be a pointer to one of these types.

PROC Directive (2 of 2)

- Alternate format permits parameter list to be on one or more separate lines:

label PROC, ← comma required
 paramList

- The parameters can be on the same line . . .

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

- Or they can be on separate lines:

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

AddTwo Procedure

- The AddTwo procedure receives two integers and returns their sum in EAX.

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

PROTO Directive

- Creates a procedure prototype
- Syntax:
 - *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

PROTO Directive

- 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:

```
MySub PROTO                ; procedure prototype

.code
INVOKE MySub                ; procedure call

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

PROTO Example

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

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

Parameter Classifications

- An **input parameter** is data passed by a calling program to a procedure.
 - The called procedure is not expected to modify the corresponding parameter variable, and even if it does, the modification is confined to the procedure itself.
- An **output parameter** is created by passing a pointer to a variable when a procedure is called.
 - The procedure does not use any existing data from the variable, but it fills in a new value before it returns.
- An **input-output parameter** is a pointer to a variable containing input that will be both used and modified by the procedure.
 - The variable passed by the calling program is modified.

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**

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**

CHAPTER 9:

STRINGS AND ARRAYS

Chapter Overview

- **String Primitive Instructions**
- Selected String Procedures
- Two-Dimensional Arrays
- Searching and Sorting Integer Arrays
- Java Bytecodes: String Processing (optional topic)

String Primitive Instructions

- MOVSB, MOVSW, and MOVSD
- CMPSB, CMPSW, and CMPSD
- SCASB, SCASW, and SCASD
- STOSB, STOSW, and STOSD
- LODSB, LODSW, and LODSD

MOVSb, MOVSw, and MOVSD (1 of 2)

- The MOVSb, MOVSw, and MOVSD instructions copy data from the memory location pointed to by ESI to the memory location pointed to by EDI.

```
.data
source DWORD 0FFFFFFFFh
target DWORD ?
.code
mov esi,OFFSET source
mov edi,OFFSET target
movsd
```

MOVSb, MOVSW, and MOVSD (2 of 2)

- ESI and EDI are **automatically** incremented or decremented:
 - MOVSb increments/decrements by 1
 - MOVSW increments/decrements by 2
 - MOVSD increments/decrements by 4

Direction Flag

- The Direction flag controls the incrementing or decrementing of ESI and EDI.
 - DF = clear (0): increment ESI and EDI
 - DF = set (1): decrement ESI and EDI

The Direction flag can be explicitly changed using the CLD and STD instructions:

```
CLD                ; clear Direction flag
STD                ; set Direction flag
```


Using a Repeat Prefix

- REP (a repeat prefix) can be inserted just before MOVSB, MOVSW, or MOVSD.
- ECX controls the number of repetitions
- Example: Copy 20 doublewords from source to target

```
.data
source DWORD 20 DUP('z')
target DWORD 20 DUP(?)
.code
cld                      ; direction = forward
mov ecx,LENGTHOF source  ; set REP counter
mov esi,OFFSET source
mov edi,OFFSET target
rep movsd
```

CMPSB, CMPSW, and CMPSD

- The CMPSB, CMPSW, and CMPSD instructions each compare a memory operand pointed to by ESI to a memory operand pointed to by EDI.
 - CMPSB compares bytes
 - CMPSW compares words
 - CMPSD compares doublewords
- Repeat prefix often used
 - REPE (REPZ)
 - REPNE (REPNZ)

Comparing a Pair of Doublewords

If source > target, the code jumps to label L1; otherwise, it jumps to label L2

```
.data
source DWORD 1234h
target DWORD 5678h

.code
mov esi,OFFSET source
mov edi,OFFSET target
cmpsd                ; compare doublewords
ja L1                ; jump if source > target
jmp L2                ; jump if source <= target
```

Comparing Arrays

Use a REPE (repeat while equal) prefix to compare corresponding elements of two arrays.

```
.data
source DWORD COUNT DUP(?)
target DWORD COUNT DUP(?)
.code
mov ecx,COUNT                ; repetition count
mov esi,OFFSET source
mov edi,OFFSET target
cld                          ; direction = forward
repe cmpsd                   ; repeat while equal
```

SCASB, SCASW, and SCASD

- The SCASB, SCASW, and SCASD instructions compare a value in **AL/AX/EAX** to a byte, word, or doubleword, respectively, addressed by **EDI**.
- Useful types of searches:
 - Search for a specific element in a long string or array.
 - Search for the first element that does not match a given value.

SCASB Example

Search for the letter 'F' in a string named **alpha**:

```
.data
alpha BYTE "ABCDEFGH",0
.code
mov edi,OFFSET alpha
mov al,'F' ; search for 'F'
mov ecx,LENGTHOF alpha
cld
repne scasb ; repeat while not equal
jnz quit
dec edi ; EDI points to 'F'
```

What is the purpose of the JNZ instruction?

STOSB, STOSW, and STOSD

- The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by **EDI**.
- Example: fill an array with 0FFh

```
.data
Count = 100
string1 BYTE Count DUP(?)
.code
mov al,0FFh           ; value to be stored
mov edi,OFFSET string1 ; ES:DI points to target
mov ecx,Count          ; character count
cld                    ; direction = forward
rep stosb              ; fill with contents of AL
```

LODSB, LODSW, and LODSD

- LODSB, LODSW, and LODSD load a byte or word from memory at **ESI** into AL/AX/EAX, respectively.
- Example:

```
.data
array BYTE 1,2,3,4,5,6,7,8,9
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
    cld
L1:  lodsb                ; load byte into AL
    or al,30h            ; convert to ASCII
    call WriteChar       ; display it
    loop L1
```


What's Next

- String Primitive Instructions
- **Selected String Procedures**
- Two-Dimensional Arrays
- Searching and Sorting Integer Arrays
- Java Bytecodes: String Processing (optional topic)

Selected String Procedures

The following string procedures may be found in the Irvine32 and Irvine16 libraries:

- Str_compare Procedure
- Str_length Procedure
- Str_copy Procedure
- Str_trim Procedure
- Str_ucase Procedure

What's Next

- String Primitive Instructions
- Selected String Procedures
- **Two-Dimensional Arrays**
- Searching and Sorting Integer Arrays
- Java Bytecodes: String Processing (optional topic)

Two-Dimensional Arrays

- Base-Index Operands
- Base-Index Displacement

Base-Index Operand

- A **base-index** (基址变址) operand adds the values of two registers (called base and index), producing an **effective address**. Any two 32-bit general-purpose registers may be used.
- Base-index operands are great for accessing arrays of structures. (A structure groups together data under a single name.)

Structure Application

A common application of base-index addressing has to do with addressing arrays of structures (Chapter 10). The following defines a structure named COORD containing X and Y screen coordinates:

```
COORD STRUCT
    X WORD ?           ; offset 00
    Y WORD ?           ; offset 02
COORD ENDS
```

Then we can define an array of COORD objects:

```
.data
setOfCoordinates COORD 10 DUP(<>)
```

Structure Application

The following code loops through the array and displays each Y-coordinate:

```
mov     ebx,OFFSET setOfCoordinates
mov     esi,2                      ; offset of Y value
mov     eax,0
mov     ecx,lengthof setOfCoordinates
L1:mov   ax,[ebx+esi]
      call WriteDec
      add  ebx,SIZEOF COORD
      loop L1
```

Base-Index-Displacement Operand

- A **base-index-displacement** (相对基址变址) operand adds base and index registers to a constant, producing an **effective address**. Any two 32-bit general-purpose registers may be used.
- Common formats:

$[\textit{base} + \textit{index} + \textit{displacement}]$
 $\textit{displacement} [\textit{base} + \textit{index}]$

64-bit Base-Index-Displacement Operand

- A 64-bit **base-index-displacement** operand adds base and index registers to a constant, producing a 64-bit **effective address**. Any two 64-bit general-purpose registers can be used.
- Common formats:

$[\textit{base} + \textit{index} + \textit{displacement}]$
 $\textit{displacement} [\textit{base} + \textit{index}]$

Two-Dimensional Table Example

Imagine a table with three rows and five columns. The data can be arranged in any format on the page:

```
table  BYTE  10h,  20h,  30h,  40h,  50h
        BYTE  60h,  70h,  80h,  90h,  0A0h
        BYTE  0B0h, 0C0h, 0D0h, 0E0h, 0F0h
NumCols = 5
```

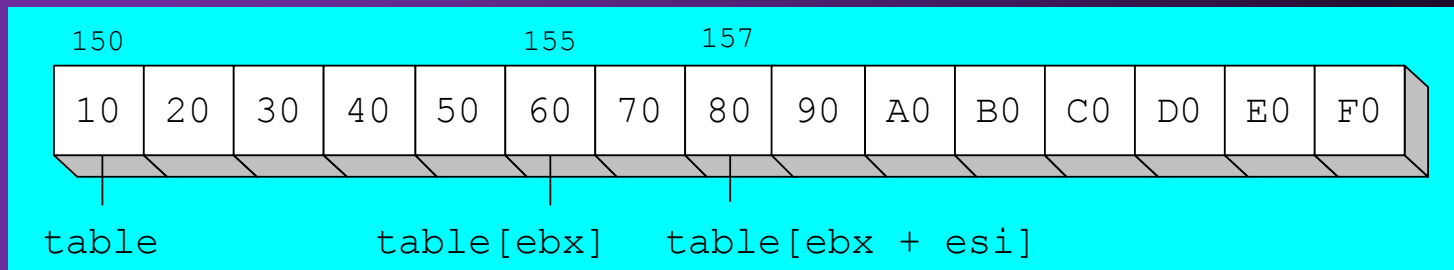
Alternative format:

```
table  BYTE  10h,20h,30h,40h,50h,60h,70h,
        80h,90h,0A0h,
        0B0h,0C0h,0D0h,
        0E0h,0F0h
NumCols = 5
```

Two-Dimensional Table Example

The following code loads the table element stored in row 1, column 2:

```
RowNumber = 1  
ColumnNumber = 2  
  
mov ebx, NumCols * RowNumber  
mov esi, ColumnNumber  
mov al, table[ebx + esi]
```



Two-Dimensional Table Example (64-bit)

The following 64-bit code loads the table element stored in row 1, column 2:

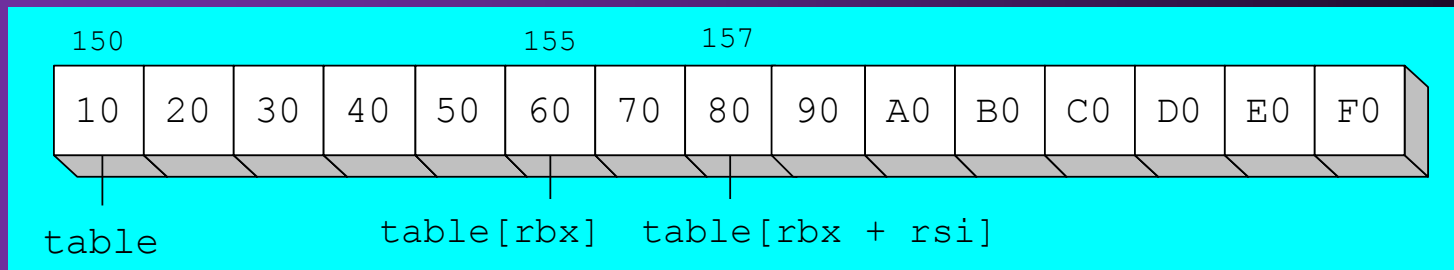
```
RowNumber = 1
```

```
ColumnNumber = 2
```

```
mov  rbx, NumCols * RowNumber
```

```
mov  rsi, ColumnNumber
```

```
mov  al, table[rbx + rsi]
```



What's Next

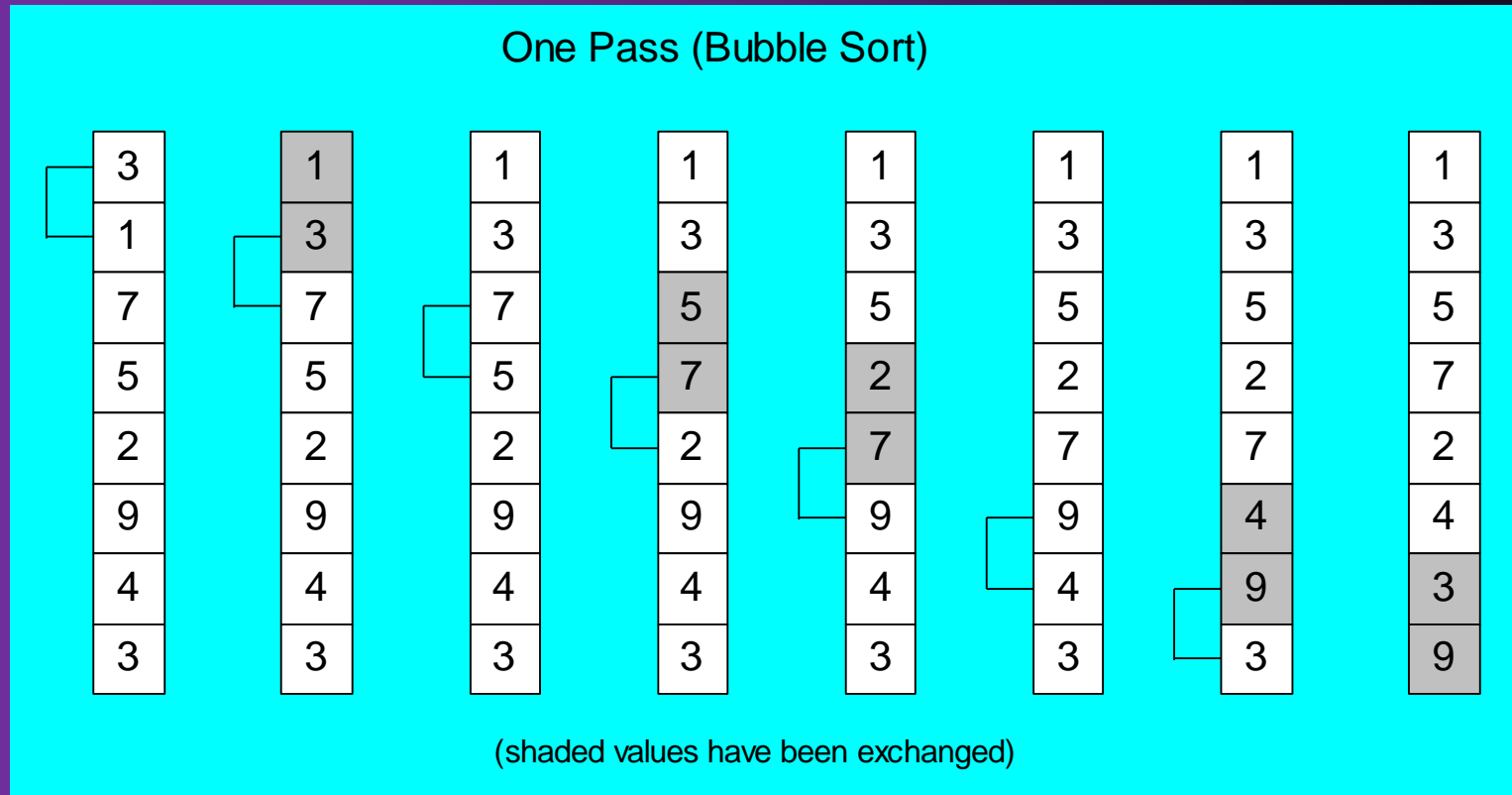
- String Primitive Instructions
- Selected String Procedures
- Two-Dimensional Arrays
- **Searching and Sorting Integer Arrays**
- Java Bytecodes: String Processing (optional topic)

Searching and Sorting Integer Arrays

- Bubble Sort
 - A simple sorting algorithm that works well for small arrays
- Binary Search
 - A simple searching algorithm that works well for large arrays of values that have been placed in either ascending or descending order

Bubble Sort

Each pair of adjacent values is compared, and exchanged if the values are not ordered correctly:



Bubble Sort Pseudocode

N = array size, cx1 = outer loop counter, cx2 = inner loop counter:

```
cx1 = N - 1
while( cx1 > 0 )
{
    esi = addr(array)
    cx2 = cx1
    while( cx2 > 0 )
    {
        if( array[esi] < array[esi+4] )
            exchange( array[esi], array[esi+4] )
        add esi, 4
        dec cx2
    }
    dec cx1
}
```


Bubble Sort Implementation

```
BubbleSort PROC USES eax ecx esi,  
    pArray:PTR DWORD,Count:DWORD  
    mov    ecx,Count  
    dec    ecx                ; decrement count by 1  
L1:  push  ecx                ; save outer loop count  
    mov    esi,pArray        ; point to first value  
L2:  mov    eax,[esi]         ; get array value  
    cmp    [esi+4],eax        ; compare a pair of values  
    jge    L3                ; if [esi] <= [edi], skip  
    xchg   eax,[esi+4]        ; else exchange the pair  
    mov    [esi],eax  
L3:  add    esi,4             ; move both pointers forward  
    loop   L2                ; inner loop  
    pop    ecx                ; retrieve outer loop count  
    loop   L1                ; else repeat outer loop  
L4:  ret  
BubbleSort ENDP
```

CHAPTER 10:

STRUCTURES AND MACROS

Chapter Overview

- **Structures**
- Macros
- Conditional-Assembly Directives
- Defining Repeat Blocks

Structures - Overview

- Defining Structures
- Declaring Structure Variables
- Referencing Structure Variables
- Example: Displaying the System Time
- Nested Structures
- Example: Drunkard's Walk
- Declaring and Using Unions

Structure

- A template or pattern given to a logically related group of variables.
- **field** - structure member containing data
- Program access to a structure:
 - entire structure as a complete unit
 - individual fields
- Useful way to pass multiple related arguments to a procedure
 - example: file directory information

Using a Structure

Using a structure involve ? sequential steps:

1. Define the structure.
2. Declare one or more variables of the structure type, called **structure variables**.
3. Write runtime instructions that access the structure.

Structure Definition Syntax

```
name STRUCT  
    field-declarations  
name ENDS
```

- Field-declarations are identical to variable declarations

COORD Structure

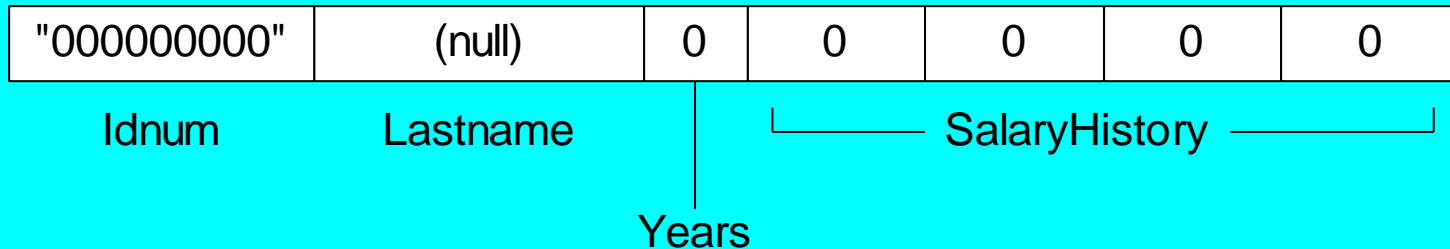
- The COORD structure used by the MS-Windows programming library identifies X and Y screen coordinates

```
COORD STRUCT
    X WORD ?           ; offset 00
    Y WORD ?           ; offset 02
COORD ENDS
```


Employee Structure

A structure is ideal for combining fields of different types:

```
Employee STRUCT
    IdNum BYTE "0000000000"
    LastName BYTE 30 DUP(0)
    Years WORD 0
    SalaryHistory DWORD 0,0,0,0
Employee ENDS
```



Declaring Structure Variables

- Structure name is a user-defined type
- Insert replacement initializers between brackets:

`<...>`

- Empty brackets `<>` retain the structure's default field initializers
- Examples:

```
.data
point1 COORD <5,10>
point2 COORD <>
worker Employee <>
```

Initializing Array Fields

- Use the DUP operator to initialize one or more elements of an array field:

```
.data  
emp Employee <,,,2 DUP(20000)>
```

Array of Structures

- An array of structure objects can be defined using the DUP operator.
- Initializers can be used

```
NumPoints = 3
AllPoints COORD NumPoints DUP(<0,0>)

RD_Dept Employee 20 DUP(<>)

accounting Employee 10 DUP(<,,,4 DUP(20000) >)
```

Referencing Structure Variables

```
Employee STRUCT                                ; bytes
    IdNum BYTE "0000000000"                    ; 9
    LastName BYTE 30 DUP(0)                     ; 30
    Years WORD 0                                ; 2
    SalaryHistory DWORD 0,0,0,0                 ; 16
Employee ENDS                                  ; 57
```

.data

worker Employee <>

```
mov eax,TYPE Employee                        ; 57
mov eax,SIZEOF Employee                      ; 57
mov eax,SIZEOF worker                        ; 57
mov eax,TYPE Employee.SalaryHistory          ; 4
mov eax,LENGTHOF Employee.SalaryHistory      ; 4
mov eax,SIZEOF Employee.SalaryHistory        ; 16
```

. . . continued

```
mov dx,worker.Years
mov worker.SalaryHistory,20000      ; first salary
mov worker.SalaryHistory+4,30000    ; second salary
mov edx,OFFSET worker.LastName

mov esi,OFFSET worker
mov ax,(Employee PTR [esi]).Years

mov ax,[esi].Years      ; invalid operand (ambiguous)
```

Nested Structures (1 of 2)

- Define a structure that contains other structures.
- Used nested braces (or brackets) to initialize each COORD structure.

```
Rectangle STRUCT
    UpperLeft COORD <>
    LowerRight COORD <>
Rectangle ENDS
```

```
.data
rect1 Rectangle { {10,10}, {50,20} }
rect2 Rectangle < <10,10>, <50,20> >
```

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS
```

Nested Structures (2 of 2)

- Use the dot (.) qualifier to access nested fields.
- Use indirect addressing to access the overall structure or one of its fields

```
mov rect1.UpperLeft.X, 10
mov esi,OFFSET rect1
mov (Rectangle PTR [esi]).UpperLeft.Y, 10

// use the OFFSET operator
mov edi,OFFSET rect2.LowerRight
mov (COORD PTR [edi]).X, 50
mov edi,OFFSET rect2.LowerRight.X
mov WORD PTR [edi], 50
```


Declaring and Using Unions

- A union is similar to a structure in that it contains multiple fields
- All of the fields in a union begin at the same offset
 - (differs from a structure)
- Provides alternate ways to access the same data
- Syntax:

```
unionname UNION  
  
    union-fields  
  
unionname ENDS
```

Integer Union Example

The Integer union consumes 4 bytes (equal to the largest field)

```
Integer UNION
    D DWORD 0
    W WORD 0
    B BYTE 0
Integer ENDS
```

D, W, and B are often called **variant fields**.

Integer can be used to define data:

```
.data
val1 Integer <12345678h>
val2 Integer <100h>
val3 Integer <>
```

What's Next

- Structures
- **Macros**
- Conditional-Assembly Directives
- Defining Repeat Blocks

Macros

- Introducing Macros
- Defining Macros
- Invoking Macros
- Macro Examples
- Nested Macros
- Example Program: Wrappers

Introducing Macros

- A **macro**¹ is a named block of assembly language statements.
- Once defined, it can be invoked (called) one or more times.
- During the assembler's **preprocessing step**, each macro call is expanded into a copy of the macro.
- The expanded code is passed to the **assembly step**, where it is checked for correctness.

¹Also called a **macro procedure**.

Defining Macros

- A macro must be defined before it can be used.
- Parameters are optional.
- Each parameter follows the rules for identifiers. It is a string that is assigned a value when the macro is invoked.
- Syntax:

```
macroname MACRO [parameter-1, parameter-2,...]  
    statement-list  
ENDM
```

mNewLine Macro Example

This is how you define and invoke a simple macro.

```
mNewLine MACRO                ; define the macro
    call CrLf
ENDM
.data

.code
mNewLine                      ; invoke the macro
```

The assembler will substitute "call crlf" for "mNewLine".

mPutChar Macro

Writes a single character to standard output.

Definition:

```
mPutchar MACRO char
    push eax
    mov al,char
    call WriteChar
    pop eax
ENDM
```

Invocation:

```
.code
mPutchar 'A'
```

Expansion:

```
1      push eax
1      mov al,'A'
1      call WriteChar
1      pop eax
```

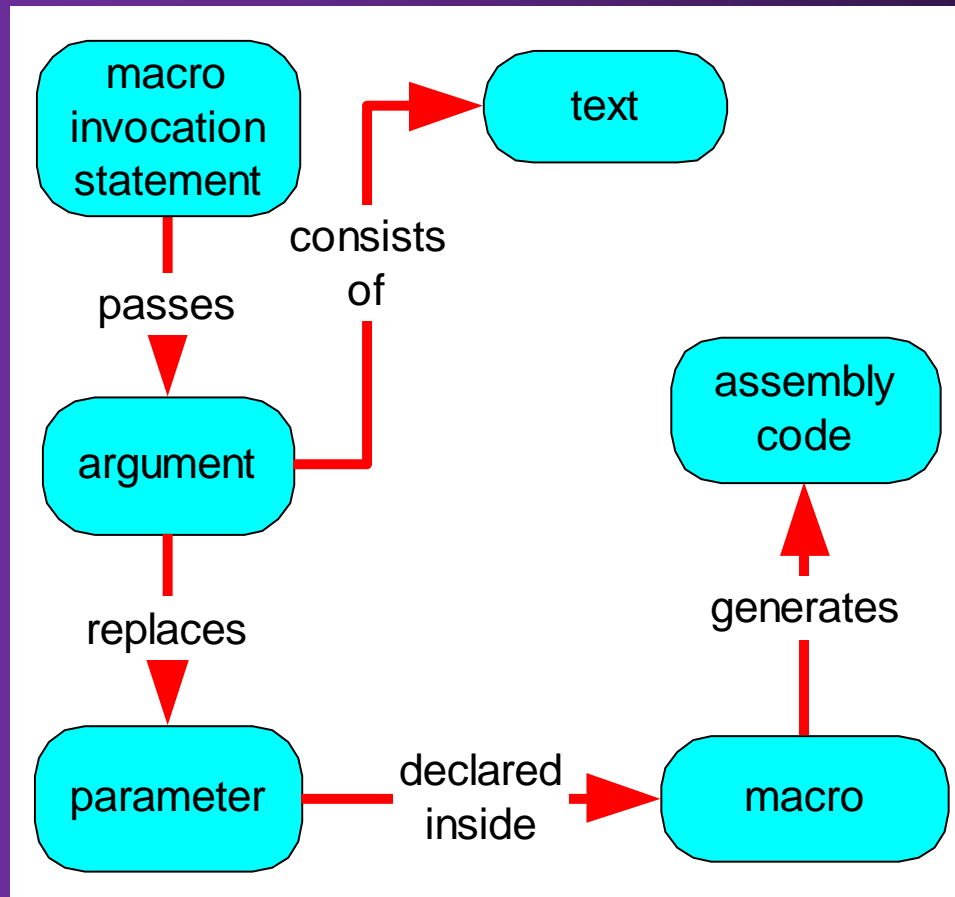
viewed in the
listing file

Invoking Macros (1 of 2)

- When you invoke a macro, each argument you pass matches a declared parameter.
- Each parameter is replaced by its corresponding argument when the macro is expanded.
- When a macro expands, it generates assembly language source code.
- Arguments are treated as simple text by the preprocessor.

Invoking Macros (2 of 2)

Relationships between macros, arguments, and parameters:



mWriteStr Macro (1 of 2)

Provides a convenient way to display a string, by passing the string name as an argument.


```
mWriteStr MACRO buffer
    push edx
    mov  edx,OFFSET buffer
    call WriteString
    pop  edx
ENDM

.data
str1 BYTE "Welcome!",0
.code
mWriteStr str1
```

mWriteStr Macro (2 of 2)

The expanded code shows how the **str1** argument replaced the parameter named **buffer**:

```
mWriteStr MACRO buffer
    push edx
    mov  edx,OFFSET buffer
    call WriteString
    pop  edx
ENDM
```



```
1    push  edx
1    mov   edx,OFFSET str1
1    call  WriteString
1    pop   edx
```

Invalid Argument

- If you pass an invalid argument, the error is caught when the expanded code is assembled.
- Example:

```
.code  
mPuchar 1234h
```

```
1      push eax  
1      mov al,1234h           ; error!  
1      call WriteChar  
1      pop  eax
```

Blank Argument

- If you pass a blank argument, the error is also caught when the expanded code is assembled.
- Example:

```
.code  
mPuchar
```

```
1      push eax  
1      mov al,  
1      call WriteChar  
1      pop eax
```

What's Next

- Structures
- Macros
- **Conditional-Assembly Directives**
- Defining Repeat Blocks

Conditional-Assembly Directives

- Checking for Missing Arguments
- Default Argument Initializers
- Boolean Expressions
- IF, ELSE, and ENDIF Directives
- The IFIDN and IFIDNI Directives
- Special Operators
- Macro Functions

Checking for Missing Arguments

- The **IFB** directive returns true if its argument is blank.
For example:

```
IFB <row>                ;; if row is blank,  
    EXITM                 ;; exit the macro  
ENDIF
```

mWriteString Example

Display a message during assembly if the string parameter is empty:

```
mWriteStr MACRO string
    IFB <string>
        ECHO -----
        ECHO * Error: parameter missing in mWriteStr
        ECHO * (no code generated)
        ECHO -----
        EXITM
    ENDIF
    push edx
    mov edx,OFFSET string
    call WriteString
    pop edx
ENDM
```

Default Argument Initializers

- A **default argument initializer** automatically assigns a value to a parameter when a macro argument is left blank. For example, **mWriteLn** can be invoked either with or without a string argument:

```
mWriteLn MACRO text:=<" ">
    mWrite text
    call Crlf
ENDM
.code
mWriteLn "Line one"
mWriteLn
mWriteLn "Line three"
```

Sample output:

```
Line one
Line three
```

Boolean Expressions

A boolean expression can be formed using the following operators:

- LT - Less than
- GT - Greater than
- EQ - Equal to
- NE - Not equal to
- LE - Less than or equal to
- GE - Greater than or equal to

Only assembly-time constants may be compared using these operators.

IF, ELSE, and ENDIF Directives

A block of statements is assembled if the boolean expression evaluates to **true**. An alternate block of statements can be assembled if the expression is false.

```
IF boolean-expression  
    statements  
[ELSE  
    statements]  
ENDIF
```

Simple Example

The following IF directive permits two MOV instructions to be assembled if a constant named **RealMode** is equal to 1:

```
IF RealMode EQ 1
    mov ax,@data
    mov ds,ax
ENDIF
```

RealMode can be defined in the source code any of the following ways:

```
RealMode = 1
```

```
RealMode EQU 1
```

```
RealMode TEXTEQU 1
```

The IFIDN and IFIDNI Directives

- IFIDN compares two symbols and returns true if they are equal (case-sensitive)
- IFIDNI also compares two symbols, using a case-insensitive comparison
- Syntax:

```
IFIDNI <symbol>, <symbol>  
    statements  
ENDIF
```

Can be used to prevent the caller of a macro from passing an argument that would conflict with register usage inside the macro.

IFIDNI Example

Prevents the user from passing EDX as the second argument to the mReadBuf macro:

```
mReadBuf MACRO bufferPtr, maxChars
    IFIDNI <maxChars>,<EDX>
        ECHO Warning: Second argument cannot be EDX
        ECHO *****
        EXITM
    ENDIF
    .
    .
ENDM
```


Special Operators

- The **substitution** (&) operator resolves ambiguous references to parameter names within a macro.
- The **expansion** (%) operator expands text macros or converts constant expressions into their text representations.
- The **literal-text** (<>) operator groups one or more characters and symbols into a single text literal. It prevents the preprocessor from interpreting members of the list as separate arguments.
- The **literal-character** (!) operator forces the preprocessor to treat a predefined operator as an ordinary character.

替换操作符 Substitution (&)

Text passed as **regName** is substituted into the literal string definition:

```
ShowRegister MACRO regName
.data
tempStr BYTE " &regName=",0
.
.
.code
ShowRegister EDX          ; invoke the macro
```

Macro expansion:

```
tempStr BYTE " EDX=",0
```

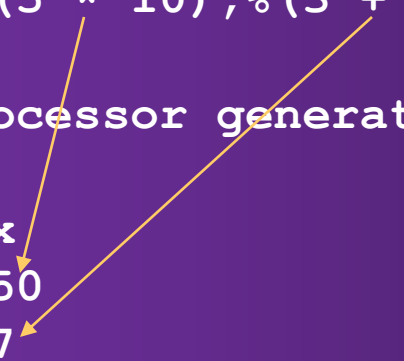
展开操作符 Expansion (%)

Forces the evaluation of an integer expression. After the expression has been evaluated, its value is passed as a macro argument:

```
mGotoXY %(5 * 10),%(3 + 4)
```

The preprocessor generates the following code:

```
1 push edx
1 mov dl,50
1 mov dh,7
1 call Gotoxy
1 pop edx
```



文本操作符 Literal-Text (<>)

The first macro call passes three arguments. The second call passes a single argument:

```
mWrite "Line three", 0dh, 0ah
```

```
mWrite <"Line three", 0dh, 0ah>
```

特殊文本字符操作符 Literal-Character (!)

The following declaration prematurely ends the text definition when the first > character is reached.

```
BadYValue TEXT$QU Warning: <Y-coordinate is > 24>
```

The following declaration continues the text definition until the final > character is reached.

```
BadYValue TEXT$QU <Warning: Y-coordinate is !> 24>
```

Macro Functions (1 of 2)

- A **macro function** returns an integer or string constant
- The value is returned by the EXITM directive
- Example: The **IsDefined** macro acts as a wrapper for the IFDEF directive.

```
IsDefined MACRO symbol
    IFDEF symbol
        EXITM <-1>                ;; True
    ELSE
        EXITM <0>                  ;; False
    ENDIF
ENDM
```

Notice how the assembler defines True and False.

Macro Functions (2 of 2)

- When calling a macro function, the argument(s) must be enclosed in parentheses
- The following code permits the two MOV statements to be assembled only if the **RealMode** symbol has been defined:

```
IF IsDefined( RealMode )  
    mov ax,@data  
    mov ds,ax  
ENDIF
```

What's Next

- Structures
- Macros
- Conditional-Assembly Directives
- **Defining Repeat Blocks**

Defining Repeat Blocks

- WHILE Directive
- REPEAT Directive
- FOR Directive
- FORC Directive
- Example: Linked List

WHILE Directive

- The WHILE directive repeats a statement block as long as a particular constant expression is true.
- Syntax:

```
WHILE constExpression  
    statements  
ENDM
```

WHILE Example

Generates Fibonacci integers between 1 and F0000000h at assembly time:

```
.data
val1 = 1
val2 = 1
DWORD val1                ; first two values
DWORD val2
val3 = val1 + val2

WHILE val3 LT 0F0000000h
    DWORD val3
    val1 = val2
    val2 = val3
    val3 = val1 + val2
ENDM
```

REPEAT Directive

- The REPEAT directive repeats a statement block a **fixed** number of times.
- Syntax:

```
REPEAT constExpression  
    statements  
ENDM
```

ConstExpression, an unsigned constant integer expression, determines the number of repetitions.

REPEAT Example

The following code generates 100 integer data definitions in the sequence 10, 20, 30, ...

```
iVal = 10  
REPEAT 100  
    DWORD iVal  
    iVal = iVal + 10  
ENDM
```

FOR Directive

- The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols.
- Each symbol in the list causes one iteration of the loop.
- Syntax:

```
FOR parameter,<arg1,arg2,arg3,...>  
    statements  
ENDM
```

FOR Example

The following Window structure contains frame, title bar, background, and foreground colors. The field definitions are created using a FOR directive:

```
Window STRUCT
    FOR color,<frame,titlebar,background,foreground>
        color DWORD ?
    ENDM
Window ENDS
```

Generated code:

```
Window STRUCT
    frame DWORD ?
    titlebar DWORD ?
    background DWORD ?
    foreground DWORD ?
Window ENDS
```

FORC Directive

- The FORC directive repeats a statement block by iterating over a string of characters. Each character in the string causes one iteration of the loop.
- Syntax:

```
FORC parameter, <string>  
statements  
ENDM
```


FORC Example

Suppose we need to accumulate seven sets of integer data for an experiment. Their label names are to be Group_A, Group_B, Group_C, and so on. The FORC directive creates the variables:

```
FORC code,<ABCDEFG>  
    Group_&code WORD ?  
ENDM
```

Generated code:

```
Group_A WORD ?  
Group_B WORD ?  
Group_C WORD ?  
Group_D WORD ?  
Group_E WORD ?  
Group_F WORD ?  
Group_G WORD ?
```

Summary (Chap 8)

- Stack parameters
 - more convenient than register parameters
 - passed by value or reference
 - ENTER and LEAVE instructions
- Local variables
 - created on the stack below stack pointer
 - LOCAL directive
- Recursive procedure calls itself
- Calling conventions (C, stdcall)
- MASM procedure-related directives
 - INVOKE, PROC, PROTO

Summary (Chap 9)

- String primitives are optimized for efficiency
- Strings and arrays are essentially the same
- Keep code inside loops simple
- Use base-index operands with two-dimensional arrays
- Avoid the bubble sort for large arrays
- Use binary search for large sequentially ordered arrays

Summary (Chap 10)

- Use a structure to define complex types
 - contains fields of different types
- Macro – named block of statements
 - substituted by the assembler preprocessor
- Conditional assembly directives
 - IF, IFNB, IFIDNI, ...
- Operators: &, %, <>, !
- Repeat block directives (assembly time)
 - WHILE, REPEAT, FOR, FORC

Homework

- Reading Chap 8 -- 10
- Exercise
- Homework

Thanks!