

## Macros and Structure and Conditional Assembly Directives

### Structures:

- Defining

A *structure* is a template or pattern given to a logically related group of variables. The variables in a structure are called *fields*. Program statements can access the structure as a single entity, or they can access individual fields. Structures often contain fields of different types.

Suppose input parameters for a procedure consisted of 20 different units of data relating to a disk drive. Calling such a procedure would be error-prone, since one might mix up the order of arguments, or pass the incorrect number of arguments. Instead, you could place all of the input data in a structure and pass the address of the structure to the procedure. Minimal stack space would be used (one address), and the called procedure could modify the contents of the structure.

Using a structure involves three 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 fields.
    - Defined using *struc* and *ends* directives
    - Inside the structure, you define fields using the same syntax as for ordinary variables.
- Basic Syntax
    - name STRUCT
    - *data declaration*
    - name ENDS

### Field Initializer:

- **Undefined:** The ? operator leaves the field contents undefined.
- **String literals:** Character strings enclosed in quotation marks.
- **Integers:** Integer constants and integer expressions.
- **Arrays:** The DUP operator can initialize array elements.

**Example:**

```
.data
ALIGN DWORD
myVar DWORD ?

Employee STRUCT
IdNum BYTE "0000000000" ; 9
LastName BYTE 30 DUP(0) ; 30
ALIGN WORD ; 1 byte added
Years WORD 0 ; 2
ALIGN DWORD ; 2 bytes added
SalaryHistory DWORD 0,0,0,0 ; 16
Employee ENDS ; 60 total
```

**Example**

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

.data
point1 COORD <5,10> ; X = 5, Y = 10
point2 COORD <20> ; X = 20, Y = ?
point3 COORD <> ; X = ?, Y = ?
worker Employee <> ; (default initializers)
```

**Example:**

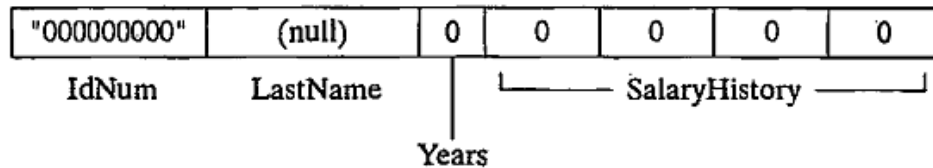
- ☐ Defining a Structure Employee information with different fields as below

```

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

```

The following figure shows a linear representation of the structure:



### Declaring Structure Variables:

- Under Data Segment
- .data
  - worker Employee <>

### Field Referencing:

- To Access Structure Variable Fields, Under Code Segment
- .code
  - mov dx, worker.years
  - mov worker.salaryhistory,20000

### Referencing Structure Variables:

```

.data
worker Employee <>
.code
mov dx,worker.Years
mov worker.SalaryHistory,20000 ; first salary
mov [worker.SalaryHistory+4],30000 ; second salary

```

### Using the OFFSET Operator:

```

mov dx,OFFSET worker.LastName

```

### Indirect and Indexed Operands:

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

### ***Indexed Operands:***

```
.data
department Employee 5 DUP(<>)
.code
mov si,TYPE Employee ; index = 1
mov department[si].Years, 4
```

### **Macros:**

- Macros are just like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions.
- When you *invoke* a macro procedure, a copy of its code is inserted directly into the program at the location where it was invoked. This type of automatic code insertion is also known as *inline expansion*.
- Macros are expanded during the assembler's *preprocessing* step. In this step, the preprocessor reads a macro definition and scans the remaining source code in the program. At every point where the macro is called, the assembler inserts a copy of the macro's source code into the program
- *Parameters* Macro parameters are named placeholders for text arguments passed to the caller. The arguments may in fact be integers, variable names, or other values, but the preprocessor treats them as text. Parameters are not typed, so the preprocessor does not check argument types to see whether they are correct.
- Macro Definition
  - name MACRO [parameters,...]
  - <instructions>
  - ENDM

### **Using Macros:**

- When you want to use a macro, you can just type its name. For example:
  - MyMacro

- Macro is expanded directly in program's code. So if you use the same macro 100 times, the compiler expands the macro 100 times, making the output executable file larger and larger, each time all instructions of a macro are inserted.
- In general, macros execute more quickly than procedures because procedures have the extra overhead of CALL and RET instructions. There is, however, one disadvantage to using macros: repeated use of large macros tends to increase a program's size because each call to a macro inserts a new copy of the macro's statements in the program.

### **Passing Arguments to Macro:**

- To pass parameters to macro, you can just type them after the macro name. For example:
  - MyMacro 1, 2, 3
- To mark the end of the macro ENDM directive is enough

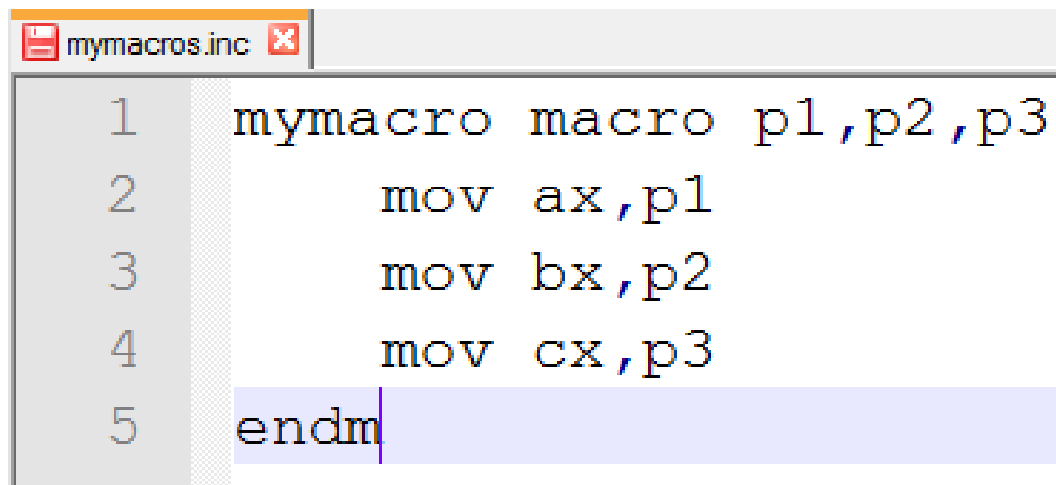
Example:

- Unlike procedures, macros should be defined above the code that uses it.
- For Example
  - .code
  - mymacro macro p1,p2,p3
  - mov ax,p1
  - mov bx,p2
  - mov cx,p3
  - endm
  - main proc
  - mymacro 1,2,3
  - mov ah,4ch
  - int 21h
  - main endp
  - end

### **Defining Macros in Separate file:**

- To define Macros in Separate file;
  - Open your assembler Source Directory

- C:\masm611\include\
- C:\emu8086\inc\
- Create a File named “mymacros.inc”
- Write your Macro in this file and Save. Make sure your file have extension .inc
- Include this file in your source program (\*.asm), by writing below line on top of your code
  - include mymacros.asm
- Compile your Code.



```
1  mymacro macro p1,p2,p3
2      mov ax,p1
3      mov bx,p2
4      mov cx,p3
5  endm
```

```
MUS.ASM x
1  include \include\mymacros.inc
2  .model small
3  .stack
4  .data
5  .code
6  start:
7  main proc
8  mymacro 1,2,3
9  mov ah,4ch
10 int 21h
11 main endp
12 end start
13 end
```

### Conditional Assembly Directives:

The Microsoft assembler (MASM) provides .IF, a high-level directive that makes programming compound IF statements much easier than if you were to code using CMP and conditional jump instructions. Here is the syntax:

```
.IF condition1
    statements
[.ELSEIF condition2
    statements ]
[.ELSE
    statements ]
.ENDIF
```

Operator	Description
<i>expr1</i> == <i>expr2</i>	Returns true when <i>expr1</i> is equal to <i>expr2</i> .
<i>expr1</i> != <i>expr2</i>	Returns true when <i>expr1</i> is not equal to <i>expr2</i> .
<i>expr1</i> > <i>expr2</i>	Returns true when <i>expr1</i> is greater than <i>expr2</i> .
<i>expr1</i> >= <i>expr2</i>	Returns true when <i>expr1</i> is greater than or equal to <i>expr2</i> .
<i>expr1</i> < <i>expr2</i>	Returns true when <i>expr1</i> is less than <i>expr2</i> .
<i>expr1</i> <= <i>expr2</i>	Returns true when <i>expr1</i> is less than or equal to <i>expr2</i> .
! <i>expr</i>	Returns true when <i>expr</i> is false.
<i>expr1</i> && <i>expr2</i>	Performs logical AND between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i>    <i>expr2</i>	Performs logical OR between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i> & <i>expr2</i>	Performs bitwise AND between <i>expr1</i> and <i>expr2</i> .
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

.While Directives:



### 6.7.3 .REPEAT and .WHILE Directives

The .REPEAT and .WHILE directives offer alternatives to writing your own loops with CMP and conditional jump instructions. They permit the conditional expressions listed earlier in Table 6-6.

The .REPEAT directive executes the loop body before testing the runtime condition following the .UNTIL directive:

```
.REPEAT
    statements
.UNTIL condition
```

The .WHILE directive tests the condition before executing the loop:

```
.WHILE condition
    statements
.ENDW
```

**Examples:** The following statements display the values 1 through 10 using the .WHILE directive:

```
mov eax,0
.WHILE eax < 10
    inc eax
    call WriteDec
    call Crlf
.ENDW
```

**.Repeat directive :**

The following statements display the values 1 through 10 using the .REPEAT directive:

```
mov eax,0
.REPEAT
    inc eax
    call WriteDec
    call Crlf
.UNTIL eax == 10
```