# CS278 Study Guide 14: Ch. 10 Structures and Macros

# ASSIGNMENT #14

## NAME: DUE: Nov 13 Received: .

**GRADE:**

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| EX14\_01 |  | 50 |
| EX14\_02 |  | 50 |
| Extra Credit |  | 20 |
| **TOTAL** |  | 100 |

## EXERCISES:

**EX14\_01 –** Do problem 3 on page 416. This problem has you write your own MOV instruction so that it can move memory to memory. Of course, inside the macro you will need to use a register, so be sure to push and pop that working register. Put your macro inside a program and use it several times in the main program using different parameters each time to make sure it works. Take a look at your listing file to see how each different macro expansion was handled. EXTRA CREDIT: Modify your macro so that it can handle a “B” Byte, “W” Word, or “D” Doubleword move by requiring that value to be passed in as the third parameter. Name this new macro mMov. *Hint: Use the macro IF statements.* Here is a possible use:

mMov X, Y, B – This would move one byte from Y to X

**EX14\_02 –** Do problem 10 on page 417. This problem has you build handy three operand versions of ADD, SUB, MUL, DIV. You can assume that the results will stay within 32 bits and don’t need to check for it (a real world implementation would want to check). You are told not to worry about pushing EAX, that anyone using these macros should know that EAX will be used. Be careful about EDX. For the DIV and MUL versions, you need to think about the impact on EDX and for DIV it should start out as zero. The best way to protect the prior contents of EDX is just to push it and then pop it back at the end of the macro. Your macros should require all three operands. Define all of your macros in a program and then call them to test them out. Take a look at the expanded macro code in your listing file.

*Note that I have only assigned 2 programs for this assignment. This is so you will have time to work on your projects!*

# CS278 Study Guide 12: Structures and Macros

## Reading Assignment for Next Class Period

* Read Chapter 11 of Irvine

## Background Material for this Study Guide

* Chapter 10 in Irvine

## Copyright

* Assembly Language for Intel-Based Computers, 4th Edition , Kip R. Irvine
* © Pearson Education, 2002. All rights reserved.
* Modified extensively by Kent Jones

## Learning Goals for this Study Guide

* Understand what a structure is and how to define, declare and access it in a program.
* Understand what a field within a structure is and how to access fields.
* Know how to generate and use the COORD structure.
* Understand how to next structures.
* Understand how the drunkard’s walk algorithm works.
* Know what a union is and how to use it.
* Know what a MASM macro is and how to define one.
* Understand how MASM macros are invoked.
* Study and understand the MASM macro examples.
* Understand how the conditional assembly directives can generate values at compile time.
* Know how to specify repeat blocks.
* Know how to use macros to generate data before compile time.
* Understand how to implement a linked list in assembly language.

# Structures

## What is a Structure?

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

## How do we Use Structures?

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.

## How do we Define a Structure?

*name* **STRUCT**

*field-declarations*

*name* **ENDS**

* Field-declarations are identical to variable declarations

### Example: The COORD Structure Used by MS-Windows

* 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

### Example: Employee Structure

Employee STRUCT

IdNum BYTE "000000000"

LastName BYTE 30 DUP(0)

Years WORD 0

SalaryHistory DWORD 0,0,0,0

Employee ENDS

"000000000"

(null)

0

0

0

0

0

SalaryHistory

Lastname

Years

Idnum

## 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> ; This coord gets X=5, Y=10

point2 COORD <> ; This gets the defaults

worker Employee <> ; This gets the defaults

### 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 "000000000" ; 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

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)

### Looping Through an Array of Points

* Sets the X and Y coordinates of the AllPoints array to sequentially increasing values (1,1), (2,2), ...

.data

NumPoints = 3

AllPoints COORD NumPoints DUP(<0,0>)

.code

mov edi,0 ; array index

mov ecx,NumPoints ; loop counter

mov ax,1 ; starting X, Y values

L1:

mov (COORD PTR AllPoints[edi]).X, ax

mov (COORD PTR AllPoints[edi]).Y, ax

add edi,TYPE COORD

inc ax

Loop L1

### Example: Displaying the System Time

* Retrieves and displays the system time at a selected screen location.
* Uses COORD and SYSTEMTIME structures:

SYSTEMTIME STRUCT

wYear WORD ?

wMonth WORD ?

wDayOfWeek WORD ?

wDay WORD ?

wHour WORD?

wMinute WORD ?

wSecond WORD ?

wMilliseconds WORD ?

SYSTEMTIME ENDS

* Uses a Windows API call to get the standard console output handle. SetConsoleCursorPosition positions the cursor. GetLocalTime gets the current time of day:

**.**data

XYPos COORD <10,5>

consoleHandle DWORD ?

.code

INVOKE GetStdHandle, STD\_OUTPUT\_HANDLE

mov consoleHandle,eax.code

INVOKE SetConsoleCursorPosition,

consoleHandle, XYPos

INVOKE GetLocalTime, ADDR sysTime

* Display the time using library calls:

mov edx,OFFSET TheTimeIs ; "The time is "

call WriteString

movzx eax,sysTime.wHour ; hours

call WriteDec

mov edx,offset colonStr ; ":"

call WriteString

movzx eax,sysTime.wMinute ; minutes

call WriteDec

mov edx,offset colonStr ; ":"

call WriteString

movzx eax,sysTime.wSecond ; seconds

call WriteDec

### Nested Structures

**COORD STRUCT**

**X WORD ?**

**Y WORD ?**

**COORD ENDS**

* 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

.code

rect1 Rectangle { {10,10}, {50,20} }

rect2 Rectangle < <10,10>, <50,20> >

* 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

### Example: Drunkard's Walk

* Random-path simulation
* Uses a nested structure to accumulate path data as the simulation is running
* Uses a multiple branch structure to choose the direction

TITLE Drunkard's Walk (Walk.asm)

; Drunkard's walk program. This is a simple version that

; starts at coordinate 50,50 and wanders around the immediate area.

INCLUDE Irvine32.inc

WalkMax = 30

StartX = 25

StartY = 25

DrunkardWalk STRUCT

path COORD WalkMax DUP(<0,0>)

pathsUsed WORD 0

DrunkardWalk ENDS

DisplayPosition PROTO currX:WORD, currY:WORD

.data

aWalk DrunkardWalk <>

.code

main PROC

mov esi,offset aWalk

call TakeDrunkenWalk

exit

main ENDP

;-------------------------------------------------------

TakeDrunkenWalk PROC

LOCAL currX:WORD, currY:WORD

;

; Take a walk in random directions (north, south, east, west).

; Receives: ESI points to a DrunkardWalk structure

; Returns: the structure is initialized with random values

;-------------------------------------------------------

pushad

; Point EDI to the array of COORD objects.

mov edi,esi

add edi,OFFSET DrunkardWalk.path

mov ecx,WalkMax ; loop counter

mov currX,StartX ; current X-location

mov currY,StartY ; current Y-location

Again:

; Insert current location in array.

mov ax,currX

mov (COORD PTR [edi]).X,ax

mov ax,currY

mov (COORD PTR [edi]).Y,ax

INVOKE DisplayPosition, currX, currY

mov eax,4 ; choose a direction (0-3)

call RandomRange

.IF eax == 0 ; North

inc currY

.ELSEIF eax == 1 ; South

dec currY

.ELSEIF eax == 2 ; West

dec currX

.ELSE ; East (EAX = 3)

inc currX

.ENDIF

next:

add edi,TYPE COORD ; point to next COORD

loop Again

finish:

mov ax,WalkMax ; count the steps taken

sub ax,cx

mov (DrunkardWalk PTR [esi]).pathsUsed, ax

popad

ret

TakeDrunkenWalk ENDP

;-------------------------------------------------------

DisplayPosition PROC currX:WORD, currY:WORD

;

; Display the current X and Y positions.

; Optional: used for debugging.

;-------------------------------------------------------

.data

commaStr BYTE ",",0

.code

pushad

movzx eax,currX ; current X position

call WriteDec

mov edx,OFFSET commaStr ; "," string

call WriteString

movzx eax,currY ; current Y position

call WriteDec

call Crlf

popad

ret

DisplayPosition ENDP

END main

## 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 <>

* The variant field name is required when accessing the union:

mov val3.B, al

mov ax,val3.W

add val3.D, eax

## Union Inside a Structure

* An Integer union can be enclosed inside a FileInfo structure:

Integer UNION

D DWORD 0

W WORD 0

B BYTE 0

Integer ENDS

FileInfo STRUCT

FileID Integer <>

FileName BYTE 64 DUP(?)

FileInfo ENDS

.data

myFile FileInfo <>

.code

mov myFile.FileID.W, ax

# Macros

## Introducing Macros

* A macro (also called a macro procedure) 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.

## 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

**Definition**

mPutchar MACRO char

push eax

mov al,char

call WriteChar

pop eax

ENDM

**Invocation**

.code

mPutchar 'A'

**Expansion**

From the listing files generated by MASM

push eax

mov al,'A'

call WriteChar

pop eax

### Invoking Macros

* 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.

parameter

macro

text

argument

replaces

declared

inside

Consists

of

passes

macro

invocation

statement

generates

assembly

code

### mWriteStr Macro

* 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

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

mWriteStr MACRO buffer

push edx

mov edx,OFFSET buffer

MASM

Generated

Code

call WriteString

pop edx

ENDM

push edx

mov edx,OFFSET str1

call WriteString

pop edx

### Invalid and Blank Arguments

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

.code

MASM

Generated

Code

mPutchar 1234h

push eax

**mov al,1234h ; error!**

call WriteChar

pop eax

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

**.**code

MASM

Generated

Code

mPutchar

push eax

mov al,

call WriteChar

pop eax

## Macro Examples

* **mReadStr** - reads string from standard input
* **mGotoXY** - locates the cursor on screen
* **mDumpMem** - dumps a range of memory
* **mWrite** – writes as string literal to the standard output

### mReadStr

* The mReadStr macro provides a convenient wrapper around ReadString procedure calls.

mReadStr MACRO varName

push ecx

push edx

mov edx,OFFSET varName

mov ecx,(SIZEOF varName) - 1

call ReadString

pop edx

pop ecx

ENDM

.data

firstName BYTE 30 DUP(?)

.code

mReadStr firstName

### mGotoXY

* The mGotoXY macro ets the console cursor position by calling the Gotoxy library procedure.

mGotoxy MACRO X:REQ, Y:REQ

push edx

mov dh,Y

mov dl,X

call Gotoxy

pop edx

ENDM

* The REQ next to X and Y identifies them as required parameters.

### mDumpMem

* The mDumpMem macro streamlines calls to the link library's DumpMem procedure.

mDumpMem MACRO address, itemCount, componentSize

push ebx

push ecx

push esi

mov esi,address

mov ecx,itemCount

mov ebx,componentSize

call DumpMem

pop esi

pop ecx

pop ebx

ENDM

### mWrite

* The mWrite macro writes a string literal to standard output. It is a good example of a macro that contains both code and data.

mWrite MACRO text

LOCAL string

.data ;; data segment

string BYTE text,0 ;; define local string

.code ;; code segment

push edx

mov edx,OFFSET string

call Writestring

pop edx

ENDM

* The LOCAL directive prevents string from becoming a global label.

## Nested Macros

* The mWriteLn macro contains a nested macro (a macro invoked by another macro).

mWriteLn "My Sample Macro Program"

mWriteLn MACRO text

mWrite text

call Crlf

MASM

Generated

Code

ENDM

.data

??0002 BYTE "My Sample Macro Program",0

.code

push edx

mov edx,OFFSET ??0002

call Writestring

pop edx

call Crlf

### Your turn . . .

* Write a nested macro that clears the screen, locates the cursor at a given row and column, asks the user to enter an account number, and inputs the account number. Use any macros shown so far.
* Use the following data to test your macro:

.data

acctNum BYTE 30 DUP(?)

.code

main proc

mAskForString 5,10,"Input Account Number: ",acctNum

### . . . Solution

mAskForString MACRO row,col,prompt,inbuf

call Clrscr

mGotoXY col,row

mWrite prompt

mReadStr inbuf

ENDM

## Example Program: Wrappers

* Demonstrates various macros from this chapter
* Shows how macros can simplify argument passing
* View the [source code](file:///\\fs1\macs\SHARE\kjones\CS278_2004_Fall\Wraps.asm)

TITLE Procedure Wrapper Macros (Wraps.ASM)

; This program demonstrates macros as wrappers

; for library procedures. Contents: mGotoxy, mWrite,

; mWriteLn, mWriteStr, mReadStr, and mDumpMem.

; Last update: 3/16/02

INCLUDE Irvine32.inc

INCLUDE Macros.inc ; macro definitions

.data

array DWORD 1,2,3,4,5,6,7,8

firstName BYTE 31 DUP(?)

lastName BYTE 31 DUP(?)

.code

main PROC

mGotoXY 20,0

mWriteLn "Sample Macro Program"

mGotoXY 0,5

mWrite "Please enter your first name: "

mReadStr firstName

call Crlf

mWrite "Please enter your last name: "

mReadStr lastName

call Crlf

; Display the person's complete name:

mWrite "Your name is "

mWriteStr firstName

mWrite " "

mWriteStr lastName

; Display a dump of the array:

call Crlf

mDumpMem OFFSET array,LENGTHOF array, TYPE array

exit

main ENDP

END main

# Conditional-Assembly Directives

## 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:=<" ">

Sample output:

mWrite text

call Crlf

**Line one**

**Line three**

ENDM

.code

mWriteln "Line one"

mWriteln

mWriteln "Line three"

## Boolean Expressions

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

* 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

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

IFIDNI <symbol>, <symbol>

statements

ENDIF

### 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

### Substitution (&)

* The substitution (&) operator resolves ambiguous references to parameter names within a macro.
* 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

### Expansion (%)

* The expansion operator (%) expands text macros or converts constant expressions into their text representations.
* 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)

MASM

Generated

Code

push edx

mov dl,50

mov dh,7

call Gotoxy

pop edx

### Literal-Text (<>)

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

mWrite "Line three", 0dh, 0ah

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

### Literal-Character (!)

* The literal-character operator (!) forces the preprocessor to treat a predefined operator as an ordinary character.
* The following declaration prematurely ends the text definition when the first > character is reached.

BadYValue TEXTEQU Warning: <Y-coordinate is > 24>

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

BadYValue TEXTEQU <Warning: Y-coordinate is !> 24>

## Macro Functions

* 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.
* Notice how the assembler defines True and False.

IsDefined MACRO symbol

IFDEF symbol

EXITM <-1> ;; True

ELSE

EXITM <0> ;; False

ENDIF

ENDM

* 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

## Defining Repeat Blocks

### WHILE Directive

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

WHILE constExpression

statements

### 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

iVal = 10

REPEAT 100

DWORD iVal

iVal = iVal + 10

ENDM

* How might we assign a data name to this list of integers?

### Your turn . . .

What will be the last integer to be generated by the following loop?

rows = 10

columns = 5

.data

iVal = 10

REPEAT rows \* columns

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

MASM

Generated

Code

Window ENDS

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 ?

MASM

Generated

Code

ENDM

Group\_A WORD ?

Group\_B WORD ?

Group\_C WORD ?

Group\_D WORD ?

Group\_E WORD ?

Group\_F WORD ?

Group\_G WORD ?

### Example: Linked List

* We can use the REPT directive to create a singly linked list at assembly time.
* Each node contains a pointer to the next node.
* A null pointer in the last node marks the end of the list

data

link

data

link

data

link

null

* Each node in the list is defined by a ListNode structure:

ListNode STRUCT

NodeData DWORD ? ; the node's data

NextPtr DWORD ? ; pointer to next node

ListNode ENDS

TotalNodeCount = 15

NULL = 0

Counter = 0

* The REPEAT directive generates the nodes.
* Each ListNode is initialized with a counter and an address that points 8 bytes beyond the current node's location:

.data

LinkedList LABEL DWORD

REPEAT TotalNodeCount

Counter = Counter + 1

ListNode <Counter, ($ + Counter \* SIZEOF ListNode)>

ENDM

* The value of $ does not change—it remains fixed at the location of the LinkedList label.
* The following hexadecimal values in each node show how each NextPtr field contains the address of its following node.

NextPtr

**Offset Contents**

00000000 00000001

00000008

00000008 00000002

00000010

00000010 00000003

00000018

00000018 (etc.)

TITLE Creating a Linked List (List.asm)

; This program shows how the STRUC directive and the REPT directive can be

; combined to create a linked list at assembly time.

INCLUDE Irvine32.inc

ListNode STRUCT

NodeData DWORD ?

NextPtr DWORD ?

ListNode ENDS

TotalNodeCount = 15

NULL = 0

Counter = 0

.data

LinkedList LABEL DWORD

REPT TotalNodeCount

Counter = Counter + 1

ListNode <Counter, ($ + Counter \* SIZEOF ListNode)>

ENDM

ListNode <0,0> ; tail node

.code

main PROC

mov esi,OFFSET LinkedList

; Display the integers in the NodeData members.

NextNode:

; Check for the tail node.

mov eax,(ListNode PTR [esi]).NextPtr

cmp eax,NULL

je quit

; Display the node data.

mov eax,(ListNode PTR [esi]).NodeData

call WriteDec

call Crlf

; Get pointer to next node.

mov esi,(ListNode PTR [esi]).NextPtr

jmp NextNode

quit:

exit

main ENDP

END main