Lab #8: Refactored Database Program

Overview:

In this lab you will re-factor the simple data entry program that you implemented on Lab #7. The refactoring work consists on modifying the code to use procedures that you will create.

Requirements

The program should support the following features:

- allow user to enter personal information of 4 people
- store entered information in memory
- after entering the information of all people, the program should display the full database
- personal information should include full name (max 30 chars), address (max 60 chars), city (max 16 chars), state (max 2 chars) and zip code (max 5 chars)

The following example shows how the screen should look after a program run (user input shown in bold):

Full-Name: **John Doe I**
Address: **5555 NW Anywhere St.**
City: **Portland**
State: **OR**
Zip: **97000**

------------------------------------
Full-Name: **Jane Doe I**
Address: **6666 SE Lost St.**
City: **Hillsboro**
State: **OR**
Zip: **97124**

------------------------------------
Full-Name: **John Doe II**
Address: **7777 NW Anywhere St.**
City: **Portland**
State: **OR**
Zip: **97000**

------------------------------------
Full-Name: **Jane Doe II**
Address: **8888 SE Lost St.**
City: **Hillsboro**
State: **OR**
Zip: **97124**
Directions

1. Create an assembly language code file named c:\lab8.asm using Notepad++.

2. Type in (or copy & paste) the following code skeleton:

```assembly
.model small ;specify small memory model
extrn read_string: far
extrn print_string: far
extrn print_crlf: far

info struc
fullname db 31 dup(?)
address db 61 dup(?)
city db 17 dup(?)
state db 3 dup(?)
zip db 6 dup(?)
ends

.stack 200h ;specify a stack size of 512 bytes
.data
separator_msg db '-------------------------------',0
fullname_msg db 'Full-Name: ',0
address_msg db 'Address: ',0
city_msg db 'City: ',0
```
state_msg          db 'State: ',0
zip_msg           db 'Zip Code: ',0

database Info 4 dup(<>)

.code

.start:
  mov ax,@data ; set-up ds to be able to access our data
  mov ds,ax

  ; set-up index & loop counter
  mov si,0
  mov cx,length database

read:
  ; print full-name prompt message
  lea bx,fullname_msg
  call print_string

  ; read full-name from user (bx = buffer pointer, dx = max count)
  lea bx,database[si].fullname
  mov dx,31
  call read_string

  ; TODO; Step #4 here

  ; print separator
  lea bx,separator_msg
  call print_string
  call print_crlf

  ; adjust index & loop
  add si,type database
  loop read

  ; TODO; Step #5 & #6 here

.exit:
  ; Use DOS interrupt 21h, Service 4ch
  ; to exit program
  mov ax,4c00h
  int 21h

.end start ; tell assembler to finish

3. Take a close look at the above code and pay attention to the comments. Make sure that you understand what the given pieces of code do before proceeding any further.

4. Inside the read loop, add code to print prompt messages and read user input for address, city, state and zip. 
   Hint: look how this is done for fullname.
5. Create a loop with a label named print. The loop should use a counter and index similar to the read loop on the skeleton.

6. Inside the print loop, print the fullname, address, city, state and zipcode. For example, for fullname:

   ```
   lea bx, database[si].fullname
   call print_string
   call print_crlf
   ```

7. Inside the print loop, after the code you added on Step #6, print the separator. Hint: take a look at how this is done inside the read loop.

8. Create an assembly language code file named c:\lab8utl.asm using Notepad++.

9. Type in (or copy & paste) the following code skeleton:

   ```
   ;ASCI codes
   CR equ 0dh
   LF equ 0ah

   .model small                ;specify small memory model

   .data
   crlf_msg db CR, LF, 0

   .code

   public read_string, print_string, print_crlf

   read_string proc near
   ;TODO: Step #11 here
   read_string endp

   print_string proc near
   ;TODO: Step #12 here
   print_string endp

   print_crlf proc near
   ;TODO: Step #13 here
   ```
print_crlf endp
end

10. Take a close look at the above code and pay attention to the comments. Make sure that you understand what the given pieces of code do before proceeding any further.

11. Implement the read_string procedure. It should read characters from the keyboard storing on the buffer passed on the bx register up to a maximum count passed on the dx register. A ‘0’ should be added at the end of the buffer. Use the following algorithm:

   a. Preserve any registers used (using push instruction)
   b. Initialize index (si register) to zero.
   c. Initialize counter (cx register) to one less than the maximum count (cx = dx - 1).
   d. Use Interrupt 16h, Service 00h to read keyboard.
   e. If ENTER (CR) is pressed, go to Step j.
   f. Print character using Interrupt 10h, Service 0eh.
   g. Store read character into current buffer location ([bx][si] to al).
   h. Increment index
   i. Loop back to Step d
   j. Move a string terminator (0) into the current buffer location.
   k. Print a CR-LF pair by calling the print_crlf procedure.
   l. Restore registers saved on step a (using pop instruction).
   m. Return (use ret instruction).

12. Implement the print_string procedure. It should print the ‘0’ terminated string passed on the bx register. Use the following algorithm:

   a. Preserve any registers used (using push instruction)
   b. Initialize index (si register) to zero.
   c. Read next character ([bx][si]) into register al
   d. If al is zero goto step h.
   e. Print character using Interrupt 10h, Service 0eh.
   f. Increment index
   g. Go back to Step c
   h. Restore registers saved on step a (using pop instruction).
   i. Return (use ret instruction).
13. Implement the `print_crlf` procedure. It should simply print a CR-LF pair.

    a. Preserve any registers used (using `push` instruction)
    b. Set-up the `bx` register with the effective address of the `crlf_msg` (use `lea` instruction).
    c. Call the `print_string` procedure.
    d. Restore registers saved on step a (using `pop` instruction).
    e. Return (use `ret` instruction).

14. Assemble the code using TASM. At the DOS prompt type:

```plaintext
  tasm /l /zi lab8utl.asm
  tasm /l /zi lab8.asm
```

The “/l” switch tells the assembler to generate an output listing file (named `lab8.lst`) while the “/zi” tells it to generate symbolic information needed for the linker.

Note: MASM32 equivalent command is:

```plaintext
  ml /c /Fl lab8utl.asm
  ml /c /Fl lab8.asm
```

15. If you get any error on your code, take a look at `lab8.lst` and/or `lab8utl.lst` (you can use notepad). It shows the generated machine code side-by-side with your assembly language. Always pay attention to the first error as subsequent errors may be caused by the first one. Correct your error by editing and then go back to step #14.

16. Link the program by using TLINK. At the DOS prompt type:

```plaintext
  tlink /v lab8.obj lab8utl.obj
```

The “/v” switch tells the linker to generate full symbolic information needed for the debugger.

Note: MASM32 equivalent command is:

```plaintext
  link16 lab8.obj lab8utl.obj, , , , , , ,
```

17. Run the program. At the DOS prompt type:

```plaintext
  lab8
```
18. Verify that it operates correctly (as described on the Requirements section).

19. If the program doesn’t operate correctly, debug it using the Turbo Debugger tool. At the DOS prompt type:
   `td lab8.exe`

   Note: For MASM32 use DEBUG. The invocation command is:
   `debug lab8.exe`

20. If necessary, use the Turbo Debugger help (from the menu) for reference on how to use it (or for DEBUG, use the ? command).

21. If necessary, make corrections to your program by editing `lab8.asm` and/or `lab8utl.asm`, and then going back to step #14.

Lab Report

To complete this lab you should provide the instructor with the following files:

`lab8.asm`
`lab8.lst`
`lab8utl.asm`
`lab8utl.lst`
`lab8.exe`

The files should be archived on a zip file and send via e-mail. The zip file should be named as follows: `lab8_<your name>.zip` (e.g., `lab8_WalterLara.zip`).