CSCI 2160 Project 2: Pep/9 Matrix Manipulation

Due Monday, November 9th, 2020 7:30 AM Eastern
30% of Projects Grade

Tools

You need to have the Pep/9 application installed on your computer to complete this project. You may download the Pep/9 simulator at: http://computersystemsbook.com/5th-edition/pep9/.

Goal

Write a ***menu-driven*** application in Pep/9 Assembly (.pep file) that accepts the input of matrices, checks to see what kind of arithmetic may be performed on those matrices (addition and multiplication), performs what arithmetic the user specifies, and displays the resulting matrix to the user. The application shall continue to display the menu after every step until the user chooses to exit.

Specifications

Your program will consist of several functions and procedures that implement the above goals of the program.

1. `extractMatrix(char *lpInput, short *matrix):void`

   This procedure accepts a pointer to a string of input characters and a pointer to an array to store the matrix parsed from the string input in. Assume the user input valid data (see below in `main()` for a list of input assumptions).

2. `displayMatrix(short *matrix, byte rows, byte cols):void`

   This procedure accepts a pointer to an array of shorts that represents the matrix, the number of rows in the matrix, and the number of columns in the matrix. As in the lab, display the matrix like the below (with appropriate prompt):

   ```
   | 2  4  6|
   | 8 10 12|
   |14 16 18|
   ```

3. `addMatrices(short *matrixA, byte rowsA, byte colsA, short *matrixB, byte rowsB, byte colsB):short*`

   This function accepts two pointers to arrays of shorts (represented as matrices), along with associated rows and columns for each matrix. After verifying the addition may occur (display an error message and return a NULL pointer if not), the function performs the matrix addition and returns a pointer (allocated from the "heap"—memory after your program) to the resulting summed matrix.

4. `multMatrices(short *matrixA, byte rowsA, byte colsA, short *matrixB, byte rowsB, byte colsB):short*`

   This function accepts two pointers to arrays of shorts (represented as matrices), along with associated rows and columns for each matrix. After verifying the multiplication may occur (display an appropriate error

message and return a NULL pointer if not), the function performs the matrix multiplication and returns a pointer (allocated from the "heap"—memory after your program) to the resulting product matrix.

- You may wish to use helper procedures and functions here to help streamline your code. I strongly encourage you do use helper procedures to help you with debugging and organization.
    - You will need helper procedures (from the Pep/9 heap lab) that will simulate `malloc()` and `free()` from glibc.
- You will earn **partial credit** for this function if you implement the **naïve** way of doing multiplication—repeated addition. You will earn more credit (up to **full credit)** if you successfully implement an algorithm that accomplishes multiplication in fewer instructions, e.g., shifting and adding or a lookup table of operations.

5. `main():void`

   This procedure implements your driver logic, including your menu. On program start, your main procedure will display your menu in a nicely aligned format and present options (selectable by lowercase letters) to operate the program. At a minimum, you should present options to input the matrix operands (separate options for matrix A and matrix B), add the matrices together, and multiply the matrices—these options, along with an exit option, make five options total. If you think more options appropriate, feel free to add them.

   Note: you may make the following assumptions regarding user input (which should be handled here or in a helper procedure of your own making):

   - The user will always input valid data in the range [-255, 255] per element in the matrix
       - The user will separate each matrix element with whitespace (spaces or tabs)
   - The user will always input between 1 and 25 elements
   - The user will always specify between 1 to 5 rows / 1 to 5 columns per matrix
       - Matrices may be any combination of those two numbers: 1x1, 2x5, 4x3, …
   - The user will always end input of a matrix by pressing enter

   An example input: `-5 -2  89  31 0  9 9  6 -33 4  <enter>`

Code Requirements

1. If your assembly code does not assemble in the Pep/9 simulator, you **will** earn a grade of 0 on this project!

2. You **must** comment **every single instruction** you write with a **why** comment, i.e., **why** did you write this instruction (not **what** the instruction is—that's apparent from the mnemonic.)

3. You **must** properly align your code:
    a. Each code block should align with the same indentation of instruction mnemonics and pseudo-ops
    b. Each operand within that code block should align by indentation
    c. Each comment for each instruction should align by indentation

   Failure to adhere to the alignment standards **will** result in lost points! Assembly is difficult enough to read and understand as-is without adding the additional challenge of trying to parse disorganized code.

4. You **must** include a header block at the top of your submission that contains pertinent program information. **To reiterate**: assembly programs are difficult to read and understand—help your fellow programmers out by writing a plain-English synopsis of what your program does at the head.

An example comment header:

```
;*************************************************************************
;* Program Name: <program name here>
;* Programmer:   <your name here>
;* Class:        CSCI 2160-001
;* Lab:          <lab/homework/project name here>
;* Date:         <creation date here>
;* Purpose:      <write a few descriptive sentences here - answer what, how, why>
;*************************************************************************
```

Deliverables

Rename your source code to `proj2.pep`. Submit your ***source code only*** (the .pep file—***no binaries***) to the Project 2 Dropbox by the posted deadline.

Rubric

- **15 points**: proper commenting and code style
  - Comment **each** instruction with why you wrote it, as well as each procedure, function, and file
  - Align code properly (symbols, mnemonics, operands, and comments)
  - Write consistent code: indentation, spacing, symbol naming
- **10 points**: `extractMatrix`
  - Handles specified input values, whitespace, and format restrictions
  - Correctly parses input strings
- **10 points**: `displayMatrix`
  - Displays matrix (passed by reference) to the standard output
  - Nicely aligns and formats matrix during display
- **20 points**: `addMatrices`
  - Checks row and column arguments for each matrix for compatibility before performing addition
    - Displays error message and returns NULL pointer in case of incompatible matrix sizes
  - Correctly performs addition
  - Stores summed matrix to memory allocated from the heap
- **20 points**: `multMatrices`
  - Checks row and column arguments for each matrix for compatibility before performing addition
    - Displays error message and returns NULL pointer in case of incompatible matrix sizes
  - Correctly performs multiplication
  - Stores product matrix to memory allocated from the heap
- **10 points**: multiplication in fewer steps
  - Implements some scheme for multiplying matrices in fewer operations than brute-force addition
    - **Note**: points awarded will depend on quality of implementation and number of operations reduced in total
- **15 points**: menu and driver
  - Displays a nicely-formatted menu with requested options
  - Handles input with nice formatting and correct results
  - Controls execution of program without error