C COMPUTER PROGRAMMING PRACTICE LABORATORY (II) (1042)：Final

Department of Computer Science and Engineering
National Sun Yat-sen University

June 23, 2016, 13：30 ~ 16：00 (Total 2.5 hours)

Name：_____	**Student ID number：_____**
Instructor：_____	

General instructions：

1. This exam has 8 pages including this cover.
2. There are 5 questions.
3. No cheating！Cheating that score will be calculated zero！
4. No calculator or translators can be used.
5. Please turn off all 3C products and remove all headphones.
6. Please name your files according to the question number.

   EX：Q1.cpp, Q2.cpp , …… and so on.
7. Please compress all your archives (code files) , and named as your student ID number.

   EX：B0430400XX.ZIP
8. Make sure that your compressed file has been submitted correctly.

1. Write a program that inputs a time from the console. The time should be in the format "HH:MM AM" or "HH:MM PM". Hours may be one or two digits, for example, "1:10 AM" or "11:30 PM". Your program should include a function that takes a string parameter containing the time. This function should convert the time into a four-digit military time based on a 24-hour clock. For example, "1:10 AM" would output "0110 hours", "11:30 PM" would output "2330 hours", and "12:15 AM" would output "0015 hours". The function may either write the time to the console or return a string to be written to the console by the main function.

Result:

*Noted that red text is entered by user.
/ run 1 */* Enter a time in 'HH:MM AM' or 'HH:MM PM' format: 1:10 AM In military format, '1:10 AM' is '0110 hours'. */* run 2 */* Enter a time in 'HH:MM AM' or 'HH:MM PM' format: 11:30 PM In military format, '11:30 PM' is '2330 hours'. */* run 3 */* Enter a time in 'HH:MM AM' or 'HH:MM PM' format: 12:15 AM In military format, '12:15 AM' is '0015 hours'.

2. One problem with dynamic arrays is that once the array is created using the new operator, the size cannot be changed. For example, you might want to add or delete entries from the array as you can with a vector. This project asks you to create functions that use dynamic arrays to emulate the behavior of a vector.

   First, write a program that creates a dynamic array of five strings. Store five names of your choice into the dynamic array. Next, complete the following two functions:

   string addEntry(string *dynamicArray, int &size, string newEntry);*

   This function should create a new dynamic array one element larger than *dynamicArray*, copy all elements from *dynamicArray* into the new array, add the new entry onto the end of the new array, increment size, delete *dynamicArray*, and return the new dynamic array.

   string deleteEntry(string *dynamicArray, int &size, string entryToDelete);*

   This function should search *dynamicArray* for *entryToDelete*. If not found, the request should be ignored and the unmodified *dynamicArray* returned. If found, create a new dynamic array one element smaller than *dynamicArray*. Copy all elements except *entryToDelete* into the new array, delete *dynamicArray*, decrement size, and return the

new dynamic array.

Test your functions by adding and deleting several names to the array while outputting the contents of the array. You will have to assign the array returned by *addEntry* or *deleteEntry* back to the dynamic array variable in your *main* function.

Result:

```
Initial list:
  0: Frink
  1: Wiggum
  2: Nahasapeemapetilon
  3: Quimby
  4: Flanders

After adding a name:
  0: Frink
  1: Wiggum
  2: Nahasapeemapetilon
  3: Quimby
  4: Flanders
  5: Spuckler

After removing a name:
  0: Frink
  1: Wiggum
  2: Quimby
  3: Flanders
  4: Spuckler

After removing a name that isn't on the list:
  0: Frink
  1: Wiggum
  2: Quimby
  3: Flanders
  4: Spuckler

After adding another name:
  0: Frink
  1: Wiggum
  2: Quimby
  3: Flanders
  4: Spuckler
```

```
 5: Muntz

After removing all of the names:

After adding a name:
 0: Burns
```

3. Write a program that merges the numbers in two files and writes all the numbers into a third file. Your program takes input from two different files and writes its output to a third file. Each input file contains a list of numbers of type *int* in sorted order from the smallest to the largest. After the program is run, the output file will contain all the numbers in the two input files in one longer list in sorted order from smallest to largest. Your program should define a function that is called with the two input-file streams and the output-file stream as three arguments.

Result:

```
*Noted that red text is entered by user.
```

```
Enter the first input file name:
a
Enter the second input file name:
b
Enter the output file name:
c
```

```
/* Contents of file 1 */
22
44
55
77
99

/* Contents of file 2 */
11
55
101

/* Contents of merged file */
11
22
44
55
55
77
```

99 101

4. **The Blocks Problem**

Background

Many areas of Computer Science use simple, abstract domains for both analytical and empirical studies. For example, an early AI study of planning and robotics (STRIPS) used a block world in which a robot arm performed tasks involving the manipulation of blocks. In this problem you will model a simple block world under certain rules and constraints. Rather than determine how to achieve a specified state, you will ``program" a robotic arm to respond to a limited set of commands.

The Problem

The problem is to parse a series of commands that instruct a robot arm in how to manipulate blocks that lie on a flat table. Initially there are *n* blocks on the table (numbered from 0 to *n*-1) with block b_i adjacent to block b_{i+1} for all $0 \leq i < n-1$ as shown in the diagram below:

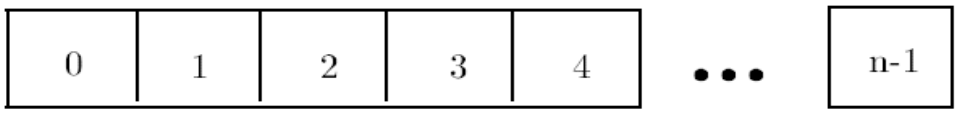

Figure 1: Initial Blocks World

The valid commands for the robot arm that manipulates blocks are:

- move *a* onto *b*
  where *a* and *b* are block numbers, puts block *a* onto block *b* after returning any blocks that are stacked on top of blocks *a* and *b*to their initial positions.
- move *a* over *b*
  where *a* and *b* are block numbers, puts block *a* onto the top of the stack containing block *b*, after returning any blocks that are stacked on top of block *a* to their initial positions.
- pile *a* onto *b*
  where *a* and *b* are block numbers, moves the pile of blocks consisting of block *a*, and any blocks that are stacked above block *a*, onto block *b*. All blocks on top of block *b* are moved to their initial positions prior to the pile taking place. The blocks stacked above block *a* retain their order when moved.
- pile *a* over *b*
  where *a* and *b* are block numbers, puts the pile of blocks consisting of block *a*, and any blocks that are stacked above block *a*, onto the top of the stack containing block *b*. The blocks stacked above block *a* retain their original order when moved.
- quit
  terminates manipulations in the block world.

Any command in which *a* = *b* or in which *a* and *b* are in the same stack of blocks is an illegal command. All illegal commands should be ignored and should have no affect on the configuration of blocks.

The Input

The input begins with an integer *n* on a line by itself representing the number of blocks in the block world. You may assume that 0 < *n* < 25.

The number of blocks is followed by a sequence of block commands, one command per line. Your program should process all commands until the `quit` command is encountered. You may assume that all commands will be of the form specified above. There will be no syntactically incorrect commands.

The Output

The output should consist of the final state of the blocks world. Each original block position numbered *i* (0 ≤ i < *n* where *n* is the number of blocks) should appear followed immediately by a colon. If there is at least a block on it, the colon must be followed by one space, followed by a list of blocks that appear stacked in that position with each block number separated from other block numbers by a space. Don't put any trailing spaces on a line. There should be one line of output for each block position (i.e., *n* lines of output where *n* is the integer on the first line of input).

Result:

*Noted that red text is entered by user.

Sample Input

```
10
move 9 onto 1
move 8 over 1
move 7 over 1
move 6 over 1
pile 8 over 6
pile 8 over 5
move 2 over 1
move 4 over 9
quit
```

Sample Output

```
0: 0
1: 1 9 2 4
2:
3: 3
4:
5: 5 8 7 6
6:
7:
8:
```

9:

5. **Poker Solitaire Evaluator**

Input

The input will contain several test cases. First line of the input is the an integer which indicate the number of test case followed by a blank line. Each consecutive test case will also separated by a blank line.

Each test case gets 25 cards, 5 cards per line. Each card consists of two characters. The first represents the rank of the card: `A', `2', `3', `4', `5', `6', `7', `8', `9', `X', `J', `Q', `K'. The second represents the suit of the card: `S', `H', `D', `C'.

The cards are dealt into a 5×5 square. Each row and column is evaluated to determine the highest hand type for which its 5 cards qualify. The hand types, from low to high, are Nothing, Pair, Two Pair, Three of a Kind, Straight, Flush, Full House, Four of a Kind, Straight Flush. A hand qualifies only once, and then only for its highest type. For example, a Four of a Kind does not count as two pair or three of a kind.

Output

For each test case output a list of 9 integers, telling how many hands of each handtype were found. from lowest to highest, being:

1. **Nothing**: does not qualify as any of the following. Example: AC, 3H, QS, JD, 7D.
2. **One Pair**: contains two cards of the same rank and does not qualify for any of the following. Example: 2C, 3H, 4H, 2H, KD.
3. **Two Pair**: contains two cards of one rank and two cards of another and does not qualify for any of the following. Example: 2C, 3H, 4H, 2H, 4D.
4. **Three of a Kind**: contains three cards of the same rank and does not qualify for any of the following. Example: QS, KH, 2C, QD, QC.
5. **Straight**: the five cards of the hand may be sorted on rank so that an unbroken sequence of 5 ranks is formed and the hand does not qualify for any of the following. There can be cycle through Ace. That is AC, 2H, 4D, 3H, 5S forms a straight, as does JH, XD,QC, KD, AS and QC, KD, AS, 2H, 3D.
6. **Flush**: the five cards are all of the same suit and the hand does not qualify for any of the following. Example: 5D, AD, KD, 7D, QD.
7. **Full House**: the hand contains three cards of one rank and two cards of another. Example: 3C, QS, QD, 3H, 3S.
8. **Four of a kind**: the hand contains four cards of the same rank. Example: AS, AD, AH, 7C, AC.
9. **Straight Flush**: the hand meets the criteria for being both a straight and a flush.

Two consecutive output will separated by a blank line.

For the example below, the five rows evaluate to Straight Flush, Straight, Pair, Flush, Three of a Kind. The Five columns evaluate to Four of A Kind, Full House, Two Pair, Nothing, and Two Pair.

Result:

*Noted that red text is entered by user.
Sample Input 2 AS 2S 3S 4S 5S AC 2H 3H 5C 4C AH 2D KC KH 5D AD 3D KD 9D 8D XH 3C XC XS 8C AS 2S 3S 4S 6S AC 2H 3H 5C 4C AH 2D KC KH 5D AD 3D KD 9D 8D XH 3C XC XS 8C
Sample Output 1, 1, 2, 1, 1, 1, 1, 1, 1 1, 2, 1, 1, 1, 2, 1, 1, 0