CSci 1113, Spring 2019
Lab Exercise 5 (Week 6): Reference Parameters and Basic File I/O

So far, we've been getting our program input from the console keyboard using `cin` and displaying results on the terminal display using `cout`. But limiting our program input to the keyboard is very restrictive. The power of modern digital computing allows us to process huge volumes of data at high speed, so we need to explore ways to input data from other sources such as data *files*. In this lab we will continue to practice constructing functions, and will also begin our exploration of file input and output.

It is natural, but incorrect, to think of `cin` and `cout` as *operations*. For example, you might hear others say "I'll just *cout* the value of x…" or, "you need to *cin* the person's last name…". `cin` and `cout` aren't operations, they're *objects*. We'll be learning much more about objects in later weeks, but for now you can think of an *object* as a sort of "uber variable" that also has functional behaviors. Each *object* is a specific instance (variable) of a *class* of objects. For example, `cin` and `cout` are objects of the *stream* class.

So when we write something like:

```
int   ivalue;
cout << "enter an integer value: ";
cin  >> ivalue;
```

the operations are *input* (>>) and *output* (<<). The *stream-object* named `cin` is the *source* of the data for the input operation and, conversely, the *stream-object* named `cout` is the *destination* for the output data.

"Stream" is short for "character data stream" or a *serial stream* of *characters*. The terminology is descriptive. A character stream is a temporal "flow" of data appearing one character at a time. This is easy to understand when visualizing input from a keyboard: the data appear as they are being typed. Consider the following:

```
int   ivalue1, ivalue2;
char  punctuation;
cin >> ivalue1 >> punctuation >> ivalue2;
```

Then examine what happens if we type the following as program input:

```
12/15<enter>
```

The I/O device (in this case the keyboard) will produce a *stream* of characters (temporally ordered from left-to-right):

```
'1'  '2'  '/'  '1'  '5'  '\n'
```

and the `cin` *object* will convert those characters to appropriate data representations based on the type of the other input operand(s). In this interesting case, the first operation requests an *integer*, so the `cin` object starts accumulating *digit* characters from the stream until it encounters the `'/'` (which is not a valid *integer* digit). At this point it completes the input operation by converting the digits it obtained from the stream (`'1'`, `'2'`) and returns the *integer value* 12. The next operation will cause `cin` to continue scanning the stream looking for any (single) character. The next element in the character stream is `'/'`, so the `cin` object returns it and the input operation stores it in the variable `punctuation`. The third input operation again asks for an integer, so `cin` will process the stream until it encounters the `'\n'` and convert `'1'` and `'5'` to the integer value 15.

From the viewpoint of a program, *files* are processed by reading them as serial data *streams*. Although the data

exists *statically* on the storage device, it is read *dynamically* (much like a cassette tape… one character after another starting with the first). This is beneficial to us, because all we need to do to input/output data from/to a file is to change the *stream-object*. If we want to *read* data from a file, we need to declare an *ifstream* object ("input file" stream) and, conversely, if we want to write data to a file we need to declare an *ofstream* object ("output file" steam). After we've done the necessary "housekeeping" (like connecting the stream to the file and checking that it is, indeed, connected), the input/output operations work exactly as they have since we started using them! You simply replace the `cin` and/or `cout` operands with the applicable file stream objects. For example,

```
#include <fstream>
ifstream  myInputFile;  // file stream object named myInputFile
ofstream  myOutputFile; // file stream object named myOutputFile
int       someInteger;
        .
        .
        .
ifstream >> someInteger; // read an integer from the file stream
ofstream << someInteger; //  and write it to the output file stream
```

Note that we don't "prompt" the file to give us data! There is no human available to ask for input, so we instead implement a loop to continue reading data elements from the file until we reach the end.

Mystery-Box Challenge

Here is your next mystery-box challenge. Consider the following code segment and determine the console output after `main()` is executed. Explain to one of your TAs how you arrived at your answer.

```
void onyx( int x, int& y)
{
   x = x*y;
   y = x;
}

int main()
{
   int y = 4;
   int x = 2;
   onyx(y,x);
   cout << x << ' ' << y << endl;
   return 0;
}
```

Warm-up

1) **Date Input/Output**
Write a short C++ program that will prompt the user and input three integer values (month, day, year) from the *console* in a "date" format that includes '/' separator characters. e.g., 8/21/2013. Your program should then output the date as 3 integer values in year, month, day order separated by commas. [Hint: your program will need to read an integer, followed by a character, followed by an integer, and so on.]

```
Enter a date in mm/dd/yyyy format: 8/21/2013
Year, month, day: 2013,8,21
```

When you complete this part, have a TA check your work.

Stretch

1) **Time Conversion**
Clock time is generally expressed in terms of separate hour and minute values, e.g., 3:12. Write a C++ program that contains a user-defined function named `timeToMinutes` that will take two separate integer arguments representing a time duration in *hours* and *minutes* and return the total number of minutes:

```
int timeToMinutes (int hours, int mins)
```

Your `main` program needs to do the following:

- Prompt the user to input a time duration in the following format (including the colon character):
  `hh:mm`
- Call `timeToMinutes` to compute the equivalent number of minutes and display the value on the terminal.
- Include a loop that will continue this process as long as the user wishes.

Example:

```
Enter a time duration (hh:mm)  0:21
Total minutes: 21
Continue? (y/n): y

Enter a time duration (hh:mm)  4:41
Total minutes: 281
Continue? (y/n): n
```

2) **Time Conversion, Part 2**
Now write a program to convert minutes to time (separate hours/minutes). Include a user-defined *void* function named `minutesToTime` that will take an integer number of minutes and convert it to two *separate* integer values that represent the equivalent number of hours and minutes. You must use reference parameters for the hours and minutes. (Before proceeding further, discuss with your partner why this function uses reference parameters.)

```
void minutesToTime(int minute_value, int& hours, int& mins)
```

Your `main` program needs to do the following:

- Prompt the user and take as input an integer number of minutes from the console.
- Call `minutesToTime` to compute the equivalent number of hours and minutes.

- Display the result on the terminal display using a 'colon' character to separate the hours/minutes. Moreover, if the number of minutes is less than 10, print it out with a leading zero (so, e.g. 8 minutes would be printed as 08 rather than 8).
- Include a loop that will continue this process as long as the user wishes.
- Example:

```
Enter a number of minutes: 60
Hours:minutes is 1:00
Continue? (y/n): y

Enter a number of minutes: 8
Hours:minutes is 0:08
Continue? (y/n): y

Enter a number of minutes: 337
Hours:minutes is 5:37
Continue? (y/n): n
```

3) **Time Intervals**

Write a C++ program that contains a user-defined *void* function named `elapsedTime` that will compute the interval between two time values (separate hours/minutes):

```
void elapsedTime(int h1, int m1, int h2, int m2, int& h, int& m)
```

Your function must determine the elapsed time regardless of the order of the values (i.e., the first input time may be greater, less-than, or equal to the second input time).

The elapsed time should be expressed in hours/minutes and returned via the reference arguments h and `m`. Note that the returned minutes value should not be larger than 59!

Your `main` program needs to do the following:

- Prompt the user and take as input two time values using the format hh:mm (including the colon).
- Call `elapsedTime` to compute the time interval.
- Display the result on the terminal display using a 'colon' character to separate the hours/minutes.
- Include a loop that will continue this process as long as the user wishes.

Examples:

```
Enter first time  (hours:minutes) : 3:32
Enter second time (hours:minutes) : 3:45
Elapsed time is: 0:13
Continue? (y/n): y

Enter first time  (hours:minutes) : 2:45
Enter second time (hours:minutes) : 2:32
Elapsed time is: 0:13
Continue? (y/n): y

Enter first time  (hours:minutes) : 1:15
Enter second time (hours:minutes) : 14:15
Elapsed time is: 13:00
Continue? (y/n): n
```

Once you have your program written you and your partner should individually think up two additional test cases for your program. Then share your test cases, and discuss why you think they are good test cases. (And use them to test your program further.)

When you complete this part, have a TA check your work on the stretch problems.

Workout

1) **Flight Data Collection**

A certain airline is required to report statistics regarding their on-time flight performance. You've been given the following data sample consisting of the flight number, the scheduled arrival time, and the actual arrival time for a few flights.

Flight Number	Scheduled Arrival Time	Actual Arrival Time
NW1735	12:03	12:15
NW1395	12:56	13:21
UA8863	2:19	2:20
NW2852	2:45	3:15
UA2740	3:10	4:00
NW1568	3:10	3:11
NW9886	14:21	19:36
DL2981	18:36	19:21
UA882	5:15	5:15
UA231	7:16	7:44

Write a C++ program to record this data in a file. Specifically, write a program that will do the following:

- Open an *output* file with the filename `flightData.dat` and determine if the file was opened successfully. If not, provide a suitable error message and exit the program.
- Prompt the user and input three *strings* for the flight number and the two arrival times. Include the colon character `':'` in the time values.
- Write the strings to the file. Each datum should be separated by a single space and the line should be delimited (terminated) with a newline (`'\n'`)
- Continue to input and write data to the file until the user inputs the sentinel string "`end`" (lowercase) for the flight number. **Do not write the sentinel value to the file!**
- Close the file before exiting the program.

Example:

```
Enter the flight number: NW1735
Enter the scheduled/actual arrival times: 12:03 12:15

enter flight number: NW1395
enter scheduled/actual arrival times: 12:56 13:21
      .
      .
enter flight number: UA231
enter scheduled/actual arrival times: 7:16 7:44

enter flight number: end
```

Now test your program by entering all the flight data above. When you have finished entering the data, use a text editor to view the `flightData.dat` file to ensure it is correct and properly formatted:

```
NW1735 12:03 12:15
NW1395 12:56 13:21
UA8863 2:19 2:20
NW2852 2:45 3:15
UA2740 3:10 4:00
NW1568 3:10 3:11
NW9886 14:21 19:36
DL2981 18:36 19:21
UA882  5:15 5:15
UA231  7:16 7:44
```

When you complete this part, have a TA check your work before you go on to the next problem.

2) **Flight Arrival Statistics**

(This is a longer problem. Even if you do not get it finished during the lab, do as much as you can since it involves many useful skills.)

Read in the data from the file you created in Workout Exercise (1), and determine the *average*, *maximum* and *minimum* flight delays for the flights recorded in the file. Specifically, your program should do the following:

- Solicit the name of a flight data file from the user and open it for reading.
- Determine if the file was opened successfully. If not, provide a suitable error message and exit the program.
- Read all the records of the file and determine:
  - The *average* flight delay
  - The flight number *and* delay with the *minimum* delay
  - The flight number *and* delay with the *maximum* delay
- Display all delay values using time format (hours:minutes)

Your program should use the functions you created in the Stretch exercises! (`elapsedTime, timeToMinutes, MinutesToTime`).

Program Note: The *ifstream*.open(*filename*) function will not accept a string variable as an argument without first being converted to a different form. After the user has input the filename, use the following in your program to allow a string variable as the *filename* in the open function:

ifstream`.open(` *stringvariable*`.c_str( ) );`

where *ifstream* is the name of your input file stream object and *stringvariable* is the name of the string variable that contains the filename you wish to open.

Hint: This is a longer and more complicated program than the previous ones. Before writing code, carefully outline what your program needs to do. Then write and test the program incrementally. Specifically, you might want to start by writing the portion of the program that gets the flight data file name, opens that file for reading, and counts the number of lines of flight data in it. Once you have that working add the portion for computing and displaying the average, etc.

When you are done with this workout problem, have a TA check your work.

Check

You and your partner should individually write down (i) one important thing you learned in lab today, and (ii) one question you still have. Then compare answers.

Challenge

Here is an extra challenge problem. You should be able to complete the warm-up, stretch and workout problems in the lab. Try this if you have extra time or would like additional practice outside of lab.

1) **Craps**

The casino game *Table Craps* is a dice game in which a number of betting options are available depending on the eventual roll of a pair of dice. Players may bet "for" or "against" certain outcomes in a dizzying array of combinations with varying odds. Each round of *Craps* begins with an initial roll of the dice (the so-called 'come-out' roll). The player 'wins' if the initial roll totals 7 or 11, and *'Craps out'* (loses) if the roll is 2, 3 or 12. Any other initial roll (4, 5, 6, 8, 9, 10) becomes the *point* value. The player continues rolling dice until either the *point* is matched or a 7 is rolled (whichever occurs first). If the *point* is matched the player wins, if a 7 is rolled the player loses.

Write a program to simulate the game of *Craps*. Your program must do the following:

- Include a void function, `diceRoll`, that will call the `rand()` function and provide 2 pseudo-random integer values in the range [1..6] representing a single roll of the dice. You need to use reference parameters to return the individual values of each die.
- Include a second void function, `displayRoll` that will output the values of two individual die: *n*, *m* to the console in the following format: [*n*,*m*] (include the brackets and comma).
- Simulate the initial (come-out) roll by calling `diceRoll` and displaying the outcome using the `displayRoll` function. Determine if the result is 'win', 'lose' or 'point' and display the result.
- If a *point* is established, output a message to indicate the value of the *point* (4, 5, 6, 8, 9, 10) and continue to simulate play until either a '7' is rolled or the *point* is matched. Display the result of each roll as you continue. Finally, indicate if the player wins or loses at the end.

Here's a sample output:

```
Initial roll is: [6,5] = 11
  You win!

Initial roll is: [5,2] = 7
  You win!

Initial roll is: [1,1] = 2
  Craps!  Sorry, you lose

Initial roll is: [4,4] = 8
  Point is 8. Roll again
  Roll is: [5,4] = 9
  Roll is: [5,1] = 6
  Roll is: [2,1] = 3
  Roll is: [4,2] = 6
  Roll is: [1,6] = 7  Sorry, you lose

Initial roll is: [5,1] = 6
  Point is 6. Roll again
  Roll is: [3,3] = 6  You win!
```