

EE 1301

Lab 5: 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`. Limiting our program input to the keyboard, however, is very restrictive. The power of modern digital computing allows us to process **huge** volumes of data at high speed, and we don't want to wait for someone at the keyboard to enter all the data manually! 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 we 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 someone say, "I'll just `cout` the value of `x`...", or, "You need to `cin` the person's last name...". `cin` and `cout` actually aren't operations, they're *objects*. We'll be learning much more about objects in the weeks to come, but for now you can think of an *object* as a sort of "super variable" or container that describes all the information and actions relevant to a particular class of objects. Each *object* is a specific instance (variable) of a *class* of objects. For example, `cin` and `cout` are objects of the *stream* class. `cout` is an object that describes an output terminal (e.g., a screen). The data associated with the object can be a message that should be displayed, and an action associated with a `cout` object is to send a message to screen; the `<<` is the action or function that sends the message. Likewise, `cin` is an object that interfaces with an input device, e.g., a keyboard, and `>>` is an operation that gets data from the input device.

So when we write something like:

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

the operations performed are *input* (`>>`) and *output* (`<<`). The *stream-object* named `cin` is the *source* of the data for the input operation, and 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 serial character stream is a temporal "flow" of data arriving one character at a time. This is easy to understand when visualizing input from a keyboard: the data appear one at a time in the order they are typed. Consider the following lines of code.

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

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

12/15<enter>

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

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

The `cin` *object* converts those input 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 digits in the stream until it encounters the '\n' and convert '1' and '5' to the integer value 15.

How does this relate to file I/O? From the viewpoint of a program, *files* are processed by reading them as serial data *streams*. In other words, data are read from the file one character at a time, just like we read characters from the keyboard with `cin`. This is convenient, because all we need to do to input data from a file or output data to a file is to change the type of *stream-object* we use. If we want to *read* data from a file instead of the keyboard, instead of using the `cin` object, we need to declare an *ifstream* object (input file stream). Likewise, if we want to write data to a file, instead of using the `cout` object, we need to declare an *ofstream* object (output file stream). After we've done the necessary initialization, like connecting the stream to the correct file and making sure it is connected correctly, the input (>>) and output (<<) operations work exactly the same way as they do for `cin` and `cout`! You simply replace the `cin` and/or `cout` operands with the applicable file stream objects, as in the following example.

```
#include <fstream>
ifstream InputFile; // input file stream object named InputFile
ofstream OutputFile; // output file stream object named OutputFile
int someInteger;

.
.
.

InputFile >> someInteger; // read an integer from the input file stream
OutputFile << someInteger; // write the integer to the output file stream
```

Note that this is all performed by the computer without the need for a human to feed or view data. We can simply implement a loop that continues 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 your TA 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 prompts the user to input three integer values (month, day, year) from the *console* in a "date" format that includes '/' separator characters, e.g., 7/4/1776. 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.]

Example:

```
Enter a date in mm/dd/yyyy format: 7/4/1776
Year, month, day: 1776,7,4
```

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 takes 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 from the console in hh:mm format (including the colon character).
- 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 takes an integer number of minutes and converts 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 a neighbor or TA 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 to 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 hours and minutes. Moreover, if the number of minutes is less than 10, print it out with a leading zero (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 computes 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 than, 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 to input two time values using the format hh:mm (including the colon).
- Call `elapsedTime` to compute the time interval between the two input times.
- 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, think up two additional test cases for your program. Discuss your test cases with a neighbor and explain why you think they are good test cases. Finally, perform your test.

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 numbers, the scheduled arrival times, and the actual arrival times for a few flights.

<u>Flight Number</u>	<u>Scheduled Arrival Time</u>	<u>Actual Arrival Time</u>
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 to input three *strings* for the flight number and the two arrival times (scheduled and actual). 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!**
- Be sure to 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 in the table 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, as shown below.

```

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

```

2) Flight Arrival Statistics

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, write a C++ program that will do the following:

- Request 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 with the *minimum* delay and its delay. The flight number with the *maximum* delay and its 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 unless the string is converted to a different form (called a C-string). After the user has entered the filename, use the following line of code in your program to perform the conversion, allowing a string variable to be used as the *filename* in the open function.

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

In the line of code, *ifstream* should be replaced by the name of your input file stream object and *stringvariable* should be replaced by 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.

Debugging time!

Get some practice with gdb while testing your program. Step through the code and print data items as they are read in from the file to make sure that file reading is working correctly.

Check

Write down (i) one important thing you learned in this lab, and (ii) one question you still have. Share with a neighbor or TA what you have written.

Challenge

Are you looking for a challenge? Try this fun challenge problem in which you create a simple computer game.

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 (called the "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 the dice until either the *point* value is rolled again or a 7 is rolled (whichever occurs first). If the *point* value is rolled again before getting a 7, the player wins, but if a 7 is rolled before matching the point value, 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 dice, *n*, *m*, to the console in the following format: `[n,m]` (include the brackets and comma).
- Simulate the initial (come-out) roll by calling the `diceRoll` function and then display 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!
```