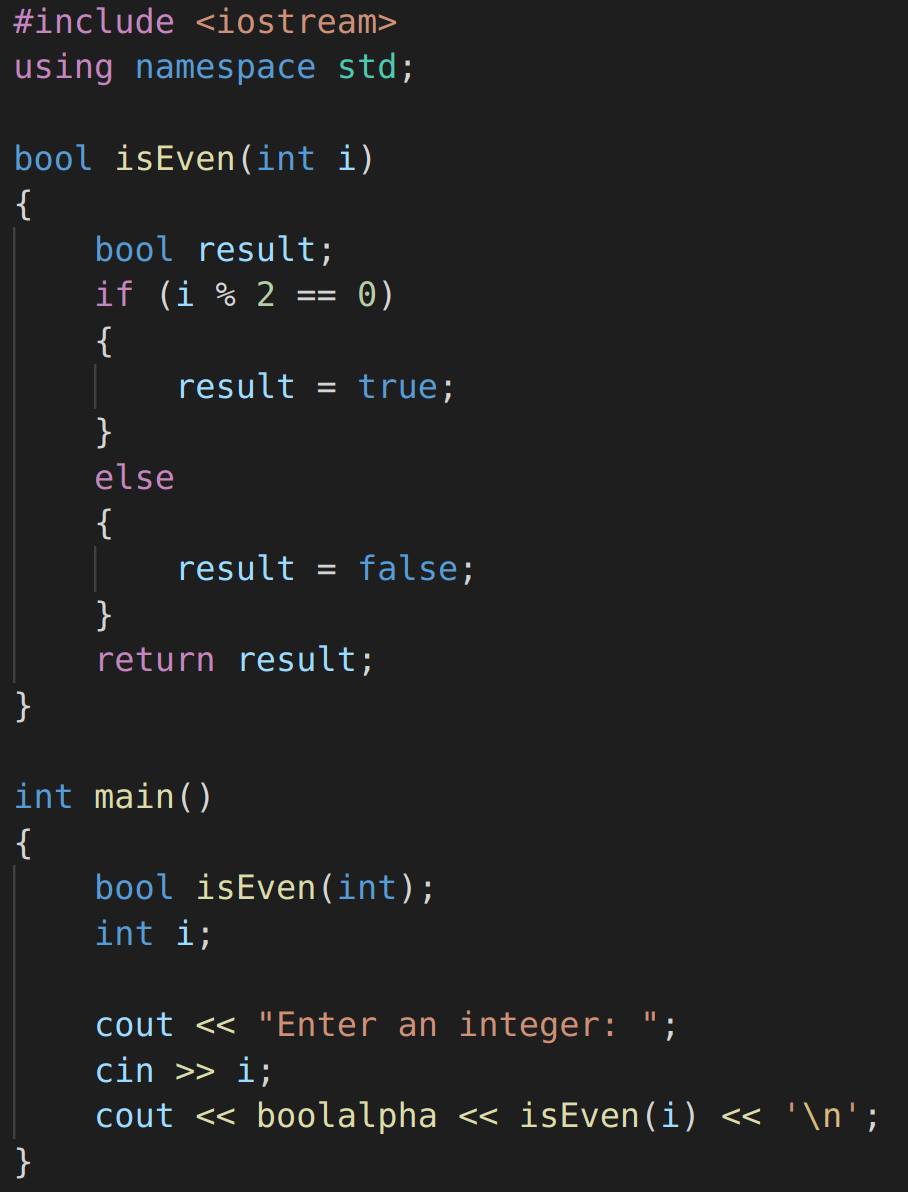
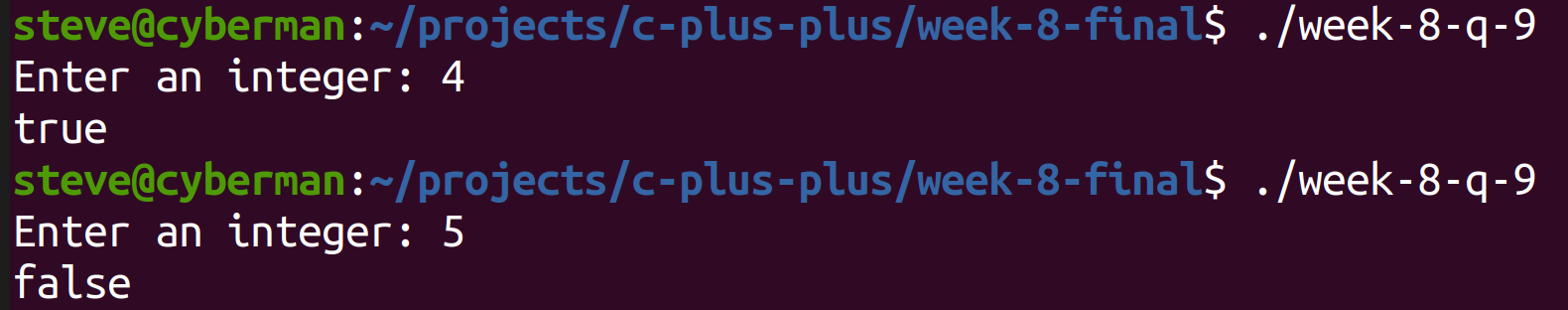
**CYBR 210 – Final Project**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. (3 Points) Write a function definition for a **bool** function called **isEven**. This function takes an integer value parameter, and returns boolean value **true** if the parameter is even and **false** if it is odd. Write a simple driver program to test your function. Show your full source code and a screenshot of your output. (Chapter 9)





1. (10 Points) In a well-written paragraph, describe the the concepts of abstraction, encapsulation, information hiding, and formal interfaces. Descibe how C++ classes and objects make use of these concepts. Include the role of access modifiers in your discussion. (Chapter 12)
2. (2 Points) Describe the role of the heap when using dynamic data. Why would a programmer prefer to use dynamic data? (Chapter 14)
3. (10 Points) The next two problems introduce command-line arguments. Command-line arguments are commonly used in console-based applications. They afford the operator the ability to execute a program by passing in arguments from the command line. We need add two arguments to our main() function. The first argument is an int that contains the number of command line arguments, the second argument is an array of char pointers, which contain the actual command line arguments. The command line argument values are stored in an array, and can be accessed to perform different tasks. The first command line argument, argv[0] is always the name of the executable file. The remaining arguments can be accessed by indexing into the array. For example, the next argument would be located in argv[1]. When writing programs with command line arguments, it is important to clarify the program’s usage. This way the operator knows how to execute the program with the correct arguments. Here is an example program where the user enters a filename as a command line argument and the program performs a word count on the file.

#include <iostream>

#include <fstream>

**using** **namespace** std**;**

//Pre: inFile is open

//Post: Returns number of words in file

int wordcount**(**ifstream**&** file**)**

**{**

int count **=** 0**;**

string word**;**

**while(**file **>>** word**)**

**{** count**++;** **}**

**return** count**;**

**}**

//Pre: None

//Post: Prints Program Usage

void printUsage**(**char exeName**[])**

**{**

cout **<<** "\n\tUsage: " **<<** exeName **<<** " [filename]\n"**;**

cout **<<** "\n\tProgram returns wordcount of filename.\n\n"**;**

**}**

int main**(**int argc**,** char**\*** argv**[])**

**{**

// Make sure user enters proper number of arguments

**if** **(**argc **!=** 2**)**

**{**

printUsage**(**argv**[**0**]);**

**return** 1**;**

**}**

// File is a command line argument

ifstream inFile**;**

inFile**.**open**(**argv**[**1**]);**

// Check to see if file exists

**if(!**inFile**)**

**{**

cout **<<** "\n\t\*\*Error opening input file.\n\n"**;**

**return** 1**;**

**}**

int wc **=** wordcount**(**inFile**);**

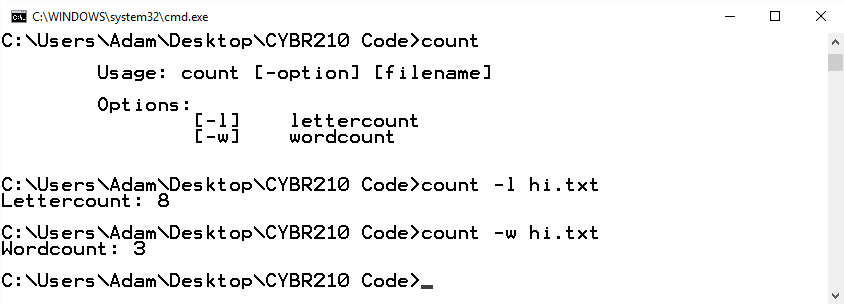
cout **<<** wc **<<** endl**;**

inFile**.**close**();**

**return** 0**;**

**}**

Your task is to write a program called “count” that lets the user retrieve a word count or a letter count of a file that is passed as a command line argument. The user enters a flag (-l or -w) to indicate if they want a letter count (-l) or word count (-w). See the figure below for an example of the program’s output. You must use a switch statement to handle the options. You should also have at least three functions: one to print the usage, one to handle the word count, and one to handle the letter count. The program must be executed from a command line environment. Show full source code and a screenshot of your working program.



#include <iostream>

#include <fstream>

using namespace std;

int wordcount(ifstream& file);

int lettercount(ifstream& file);

void printUsage(char exeName[]);

int main(int argc, char\* argv[])

{

if (argc != 3)

{

printUsage(argv[0]);

return 1;

}

ifstream inFile;

inFile.open(argv[2]);

if(!inFile)

{

cout << "\n\t\*\*Error opening input file.\n\n";

return 1;

}

char menuOption = argv[1][1];

switch (menuOption)

{

case 'l':

{

int lc = lettercount(inFile);

cout << "Lettercount: " << lc << endl;

inFile.close();

return 0;

}

break;

case 'w':

{

int wc = wordcount(inFile);

cout << "Wordcount: " << wc << endl;

inFile.close();

return 0;

}

break;

default: printUsage(argv[0]);

}

return 0;

}

//Pre: inFile is open

//Post: Returns number of words in file

int wordcount(ifstream& file)

{

int count = 0;

string word;

while(file >> word)

{ count++; }

return count;

}

int lettercount(ifstream& file)

{

int count = 0;

char character;

while(file.get(character))

{

if (isalpha(character))

count++;

}

return count;

}

//Pre: None

//Post: Prints Program Usage

void printUsage(char exeName[])

{

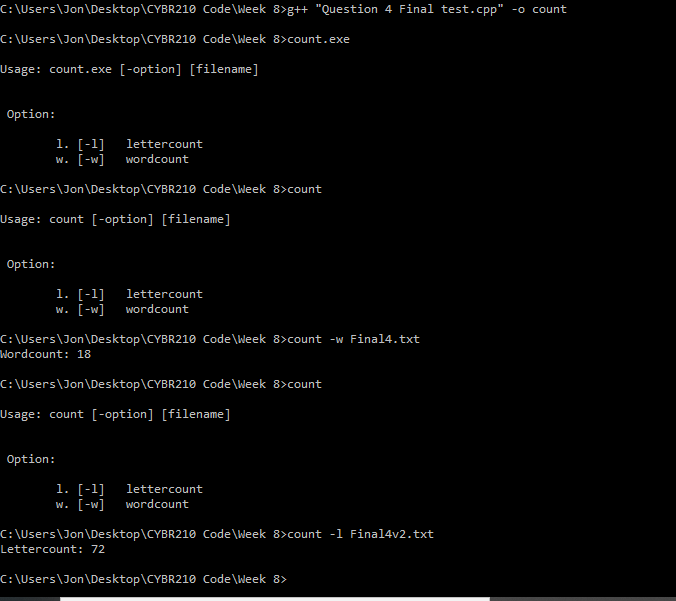
cout << "\nUsage: " << exeName << " [-option] [filename] \n\n";

cout << "\n Option: \n\n";

cout << "\tl. [-l] lettercount" << endl;

cout << "\tw. [-w] wordcount" << endl;

}



Attached will be our source code file also.

1. (10 Points) Let’s experiment1 with command line arguments some more. Suppose we need integers as command line arguments. We will need to use the **atoi()** function provided in **stdlib.h** to convert the **char\* argv[]** values to integers. **atoi()** stands for “ASCII to int.” Here is an example program that prints the sum of three integers entered as command line arguments.

#include <iostream>

#include <stdlib.h>

**using** **namespace** std**;**

void printUsage**(**char exeName**[])**

**{**

cout **<<** "\n\tUsage: " **<<** exeName **<<** " [int] [int] [int]\n"**;**

cout **<<** "\n\tReturns the sum of the 3 int command line arguments.\n\n"**;**

**}**

int main**(**int argc**,** char**\*** argv**[])**

**{**

**if** **(**argc **!=** 4**)**

**{**

printUsage**(**argv**[**0**]);**

**return** 1**;**

**}**

int a **=** atoi**(**argv**[**1**]);**

int b **=** atoi**(**argv**[**2**]);**

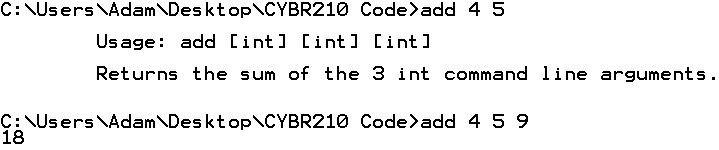
int c **=** atoi**(**argv**[**3**]);**

int sum **=** a **+** b **+** c**;**

cout **<<** sum **<<** endl**;**

**return** 0**;**

**}**



A more elegant implementation would be:

int sum **=** 0**;**

**for** **(**int i **=** 1**;** i **<** 4**;** i**++)**

**{**

sum **+=** atoi**(**argv**[**i**]);**

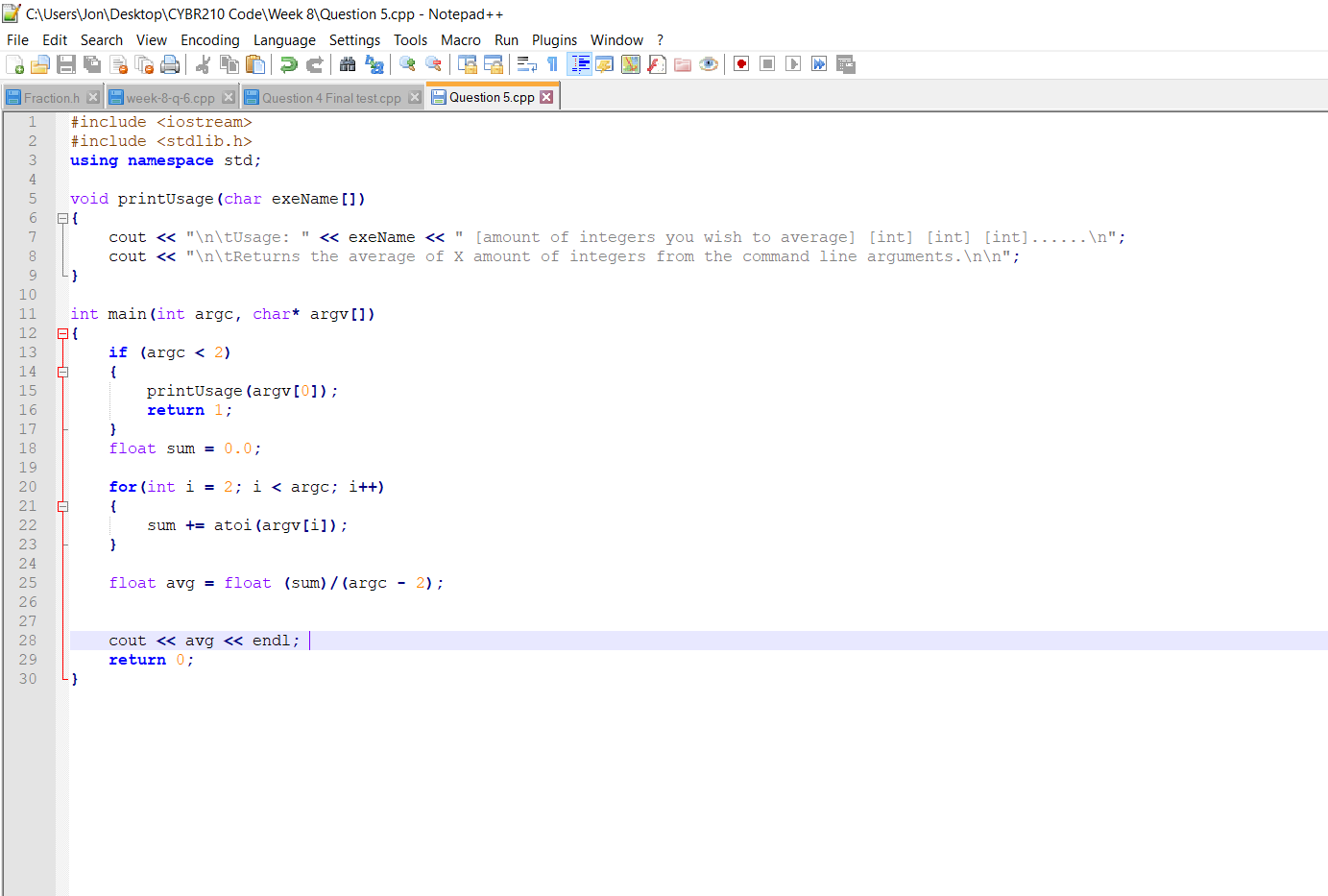
**}**

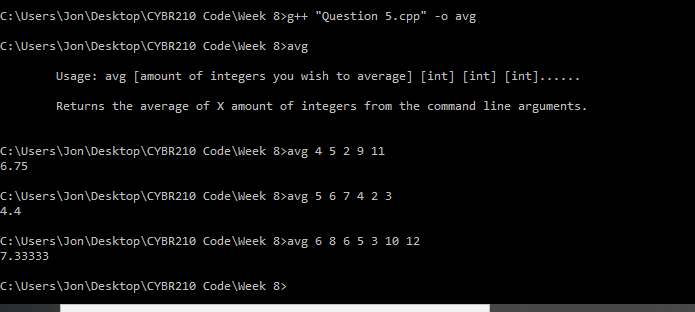
Your task is to write a C++ program that consumes integer values as command line arguments and returns the arithmetic mean of these values. To increase the flexibility of the program, there should be no set number of arguments. To overcome this, we will require the argument **argv[1]** to be the number of integers the user enters. For example, if the user wants to calculate the arithmetic mean of 4 numbers, they would pass in 4 as the first argument and then the 4 numbers as the remaining arguments. Here is an example of how the program should look in execution:



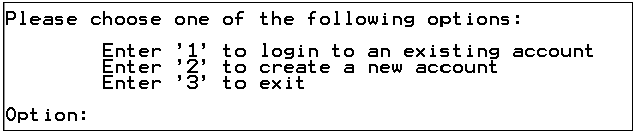
In this example, **avg** is the name of the executable and the user wants the average of 4 numbers, which are 5, 9, 2, and 11. The program then prints the arithmetic mean (average). The benefits of command line arguments should now be obvious. The user simply enters in the name of the executable program followed by the arguments. This saves time for the user by reducing interactivity. It also enables programs to be more easily scripted for automating tasks.

Include a usage statement as shown in the above addition example. Since we will need to perform division between two integers, be cautious of type casting. Your program must make use of functions. Show your full source code and a screenshot of your running program.

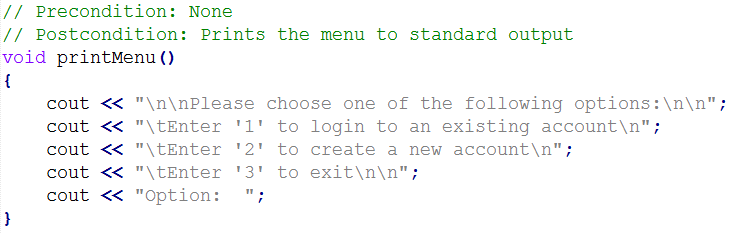




1. (65 Points) We are going to design and build an authentication program that lets users create new accounts and login to existing accounts. When launched, the program prompts a menu.



I recommend creating a void function called printMenu() that prints the menu and primes the program for the user to enter an option. Recall that ‘\t’ prints a tab and ‘\n’ prints a carriage return to help format our output.

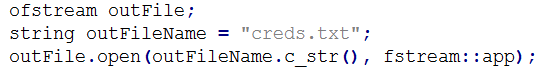


Use a switch statement to handle the menu option. If the user enters option 1, then a login() function is invoked, which authenticates a user. If option 2 is entered, then a createAccount() function is invoked, which creates a new user account. If option 3 is entered, then the program terminates. Include a default case to handle invalid entries.

You should have a do-while loop. Inside the loop, the printMenu() function is invoked and the user’s option is obtained. Following is the switch statement. The termination condition for the do-while loop should be while the user’s input is not equal to 3. This design lets the user create numerous accounts and login without having to re-execute the program.

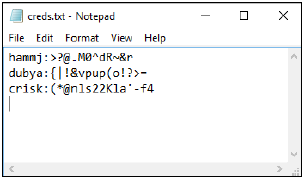
When creating a new account, the program prompts the user for a first name and a last name. It then creates a username for the user, which is the first four characters of the last name and the first character of the first name. For example, if I enter “Adam Duby” as my name, then the program will generate “dubya” as my username. The username must be in all lowercase. Therefore, the program should include a routine that converts the username to all lowercase. The program then prompts for a password. The password must be at least 10 characters long and it cannot contain any whitespace. If the user’s password does not meet the length requirement, then the program displays a message and asks the user to try again.

There needs to be a database to store user credentials. The program reads and writes a text file, called creds.txt, which stores usernames and passwords. Use the fstream::app modifer when opening the output file stream to make sure that new accounts are appended to the creds database and do not overwrite exisiting account information.



The creds.txt database should store account information in the format username:password. The username and password are stored on the same line, seperated by a colon. Each username and password combination should be on its own line in the database.

It is not secure to store passwords in plain text, so they should be stored in an encrypted form. Store the passwords in a ROT-13 encrypted format. Here is an example of a creds.txt file:

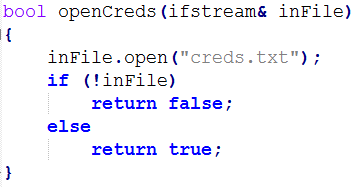


The passwords are stored in encrypted form. Upons successful creation of a new account, the program primts a message and notifies the user of their username and new email address. The email address is of the form username@norwich.edu.

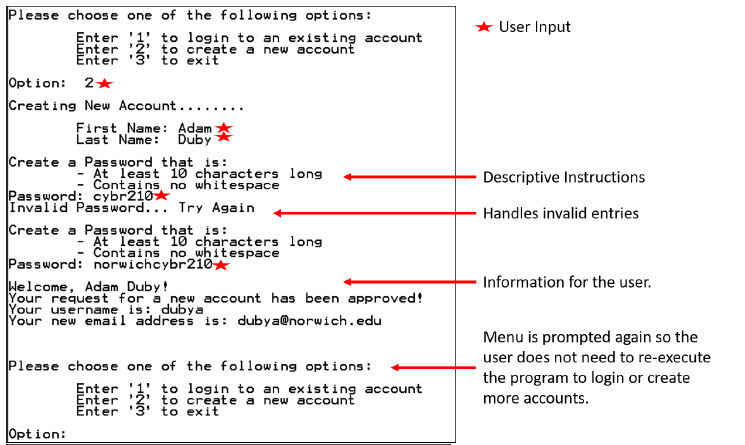
When logging in, the program prompts for the username and password then checks the creds.txt database to see if there is a match. If a match is found a welcome message is generated. If no match is found, a login failure message is generated.

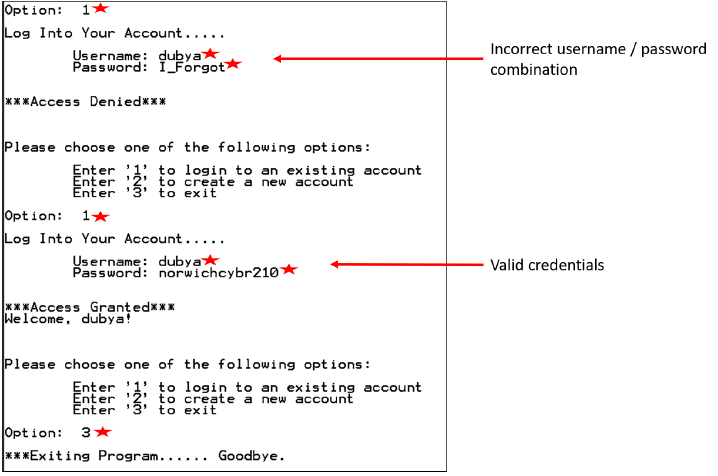
Some things to keep in mind while designing your program:

* What happens if a user has a last name less than four characters in length? Your software should be able to handle this. One way is to add letters to the end of the last name.
* Program should be able to handle multiple user accounts. Users should only have to establish their account once and can login the future, even after the program terminates.
* Be sure to practice secure coding.
  + Do not open file streams until needed and close them as soon as you are done with them.
  + Include input validation as appropriate.
* Include error handling. For example, how does the software react if the creds.txt database file gets deleted or moved? It should fail gracefully and include an error message to the console instead of crashing or freezing. if (!inFile) is a good way to do this type of checking. You can opt to do exception handling, or you can handle these types of errors as you see fit. Here is a function I use to check my input file streams to make sure that it opened successfully before I try to read from the file.

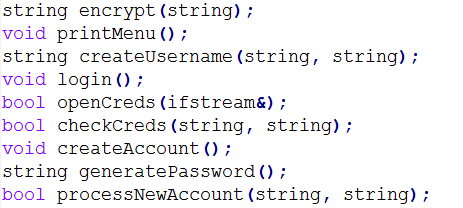


Here are some screenshots of how the program should generally look in action. Your solutions may vary:





Your overall design and architecture will be assessed during grading. Be sure to use functions as appropriate. You are free to design your functions as you deem appropriate, but keep in mind modularity, code re-use, and designing for expansion. I am including the function prototypes I used to give you a general idea of how you can approach a function decomposition for this problem.



**EXTRA CREDIT (Optional)**

(Point Value is at the Instructor’s Discretion) This challenge will present an opportunity to earn some extra credit towards your final grade. You are going to improve the 2x2 matrix inversion calculation program to support the printing of fractions. Your task is to combine the matrix program and your Fractions class to print the inverted matrix entries in fraction form. You are also tasked to add exception handling to catch division by zero, non-invertable matrices, and any other potential errors you foresee. Show all your source code and a screenshot of your working program.