Assignment #1

Worth: 10% of final grade

Account Ticketing System

Milestone	Worth	Due Date	Submission Required
1	*10%*	*(Suggested Target: March 5th)*	*NO*
2	**40%**	**March 12th (23:59 EST)**	**YES**
3	*10%*	*(Suggested Target: March 15th / 16th)*	*NO*
4	**40%**	**March 19th (23:59 EST)**	**YES**

Introduction

This is the first of two assignments. Assignment #2 builds and extends upon Assignment #1. Each assignment is broken down into critical deadlines called milestones. Implementing projects using milestones will help you stay on target with respect to timelines and balancing out the workload.

By the end of assignment #2 (milestone #4), you will have created a basic ticketing system. Think of this as a tracking system for customer reported problems. When a customer has a problem they will phone or email for support. The person handling the support request will create a ticket for the request that contains the details of the problem and the customer contact information so that the customer can be notified when there is a solution. The application will be incrementally built (adding more functionality and components) with each assignment milestone.

Assignment #1 milestones 1-2 are focused on providing helper functions that will aid you the development of the overall solution in future milestones. These functions will streamline your logic and simplify the overall readability and maintainability of your program by providing you with established routines that have been thoroughly tested for reliability and eliminate unnecessary code redundancy.

Preparation

Download or clone the Assignment 1 (**A1**) from https://github.com/Seneca-144100/BTP-Project

In the directory: A1/MS1 you will find the Visual Studio project files ready to load. Open the project (**a1ms1.vcxproj**) in Visual Studio.

Note: the project will contain only one source code file which is the main tester "**a1ms1.c**".

Milestone – 1 *(Worth 10%, Target Due Date: March 5th)*

Milestone – 1 does not require a submission and does not have a specific deadline, however, you should target to have this part completed no later than **March 5th** to ensure you leave enough time to complete Milestone – 2 which must be submitted and is due **March 12th**.

Milestone-1 includes a unit tester (***a1ms1.c***). A unit tester is a program which invokes your functions, passing them known parameter values. It then compares the results returned by your functions with the correct results to determine if your functions are working correctly. The tester should be used to

confirm your solution meets the specifications for each "helper" function. The helper functions should be thoroughly tested and fail-proof (100% reliable) as they will be used throughout your assignment milestones. An optional matrix submitter tester is also at your disposal so you can receive additional confirmation that your solution meets the minimum milestone requirements.

Note: Inevitably, all these functions will be tested as part of the Milestone #2 submission

Development Suggestions

You will be developing several functions for this milestone. The unit tester in the file "***a1ms1.c***" assumes these functions have been created and, until they exist, the program will not compile.

Strategy – 1

You can comment out the lines of code in the "***a1ms1.c***" file where you have not yet created and defined the referenced function. You can locate these lines in the function definitions (after the main function) and for every test function, locate the line that calls the function you have not yet developed and simply comment the line out until you are ready to test it.

Strategy – 2

You can create "empty function shells" to satisfy the existence of the functions but give them no logic until you are ready to program them. These empty functions are often called *stubs*.

Review the specifications below and identify every function you need to develop. Create the necessary function prototypes (placed in the .h header file) and create the matching function definitions (placed in the .c source file), only with empty code blocks (don't code anything). In cases where the function MUST return a value, hardcode (temporarily until you code the function later) a return value so your application can compile.

Specifications

Milestone-1 will establish the function "helpers" we will draw from as needed throughout these two assignments. These functions will handle routines that are commonly performed (greatly reduces code redundancy) and provide assurance they accomplish what is expected without fail (must be reliable).

1. Create a module called "commonHelpers". To do this, you will need to create two files: "***commonHelpers.h***" and "***commonHelpers.c***" and add them to the Visual Studio project.
2. The **header file (.h)** will contain the function prototypes, while the **source file (.c)** will contain the function definitions (the logic and how each function works).
   - For each of these files, create a **commented section at the top** containing the following information (you may want to use what was similarly provided to you in the workshops):
     - Assignment #1 Milestone #1
     - Your full name
     - Your student ID number
     - Your Seneca email address
     - Your course section code
3. The "***commonHelpers.c***" file will require the usual **standard input output** system library as well as the new user library "***commonHelper.h***", so be sure to include these.

4. Review the "***a1ms1.c***" tester file and examine each defined tester function (after the main function). Each tester function is designed to test a specific helper function.
5. Two (2) functions are provided for you. Here are the function prototypes you must copy and place into the "***commonHelper.h***" header file:

```
int currentYear(void);
void clearStanadardInputBuffer(void);
```

The source code file "***commonHelper.c***" must contain the function definitions (copy and place the function definitions below in the "***commonHelper.c***" file):

```
// Uses the time.h library to obtain current year information
// Get the current 4-digit year from the system
int currentYear(void)
{
    time_t currentTime = time(NULL);
    return localtime(&currentTime)->tm_year + 1900;
}
```

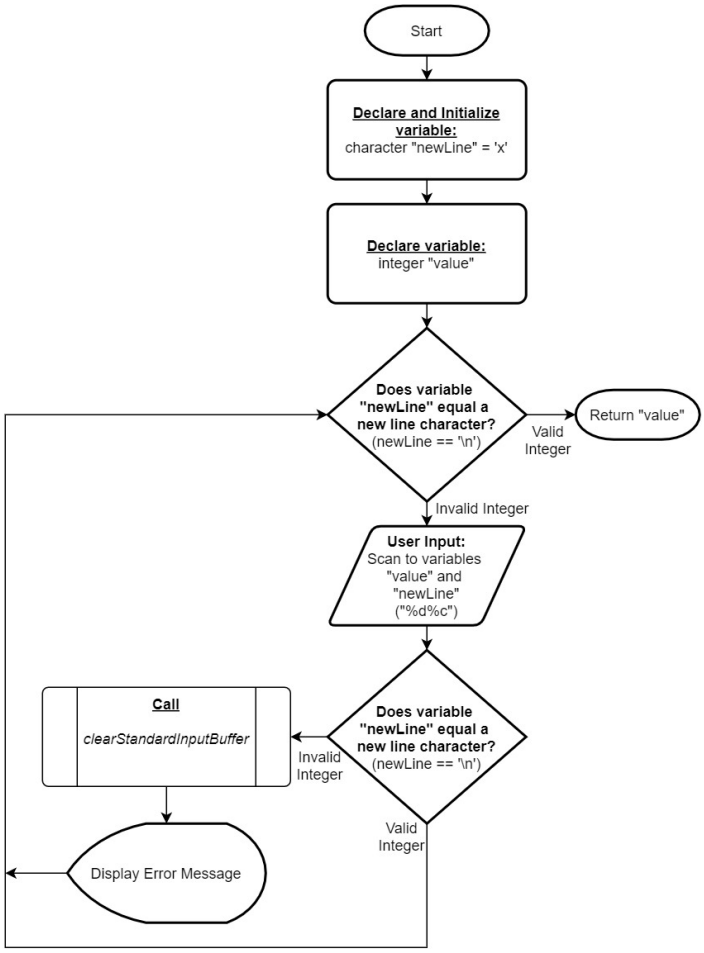
***Note**: You will need to **#include <time.h>** system library for the above function to compile.

```
// As demonstrated in the course notes:
https://ict.senecacollege.ca/~btp100/pages/content/formi.html#buf
// Empty the standard input buffer
void clearStandardInputBuffer(void)
{
    while (getchar() != '\n')
    {
        ; // On purpose: do nothing
    }
}
```

6. Each function briefly described below will require a function prototype to be placed in the "***commonHelpers.h***" file, and their respective function definitions in the "***commonHelpers.c***" file. The function identifiers (names) are provided for you however **you are responsible for constructing the full function prototype and definitions** based on the descriptions below (there are **seven (7) functions** in total):
   - Function: **getInteger**
     This function must:
     - return an integer value and receives no arguments.
     - get a valid integer from the keyboard.
     - display an error message if an invalid value is entered (review the sample output for the appropriate error message)
     - guarantee an integer value is entered and returned.

Hint: You can use scanf to read an integer and a character ("%d%c") in one call and then assess if the second value is a newline character. If the second character is a newline (the result of an <ENTER> key press), scanf read the first value successfully as an integer. **This technique can be used in other "get" functions you need to create for different data types!**

If the second value (character) is not a newline, the value entered was not an integer or included additional non-integer characters. If any invalid entry occurs, your function should call the ***clearStandardInputBuffer*** function, followed by displaying an error message and continue to prompt for a valid integer. Review the following flowchart that describes this process:



- Function: **getPositiveInteger**
  This function must:
  - return an integer value and receives no arguments.
  - perform the same operations as **getInteger** but validates the value entered is greater than 0.
  - display an error message if the value is a zero or less (review the sample output for the appropriate error message).
  - continue to prompt for a value until a value is greater than 0.
  - guarantee a positive integer value is entered and returned.

- Function: **getDouble**
  This function must:
  - return a double value and receives no arguments.
  - get a valid double value from the keyboard.
  - display an error message if an invalid value is entered (review the sample output for the appropriate error message)
  - guarantee a double value is entered and returned.
  - **Hint**: Process is the same as described in the flowchart for **getInteger** only this is for a double type

- Function: **getPositiveDouble**
  This function must:
  - return a double value and receives no arguments.
  - perform the same operations as **getDouble** but validates the value entered is greater than 0.
  - display an error message if the value is a zero or less (review the sample output for the appropriate error message).
  - continue to prompt for a value until a value is greater than 0.
  - guarantee a positive double value is entered and returned.

- Function: **getIntFromRange**
  This function must:
  - return an integer value and receives two arguments:
    - First argument represents the **lower-bound** of the permitted range.
    - Second argument represents the **upper-bound** of the permitted range.

    Note: You must provide **meaningful** parameter identifiers (names)
  - performs the same operations as **getInteger** but validates the value entered is between the two arguments received by the function (**inclusive**).
  - display an error message if the value is outside the permitted range (review the sample output for the appropriate error message).
  - continue to prompt for a value until a value is between the permitted range (inclusive)
  - guarantee an integer value is entered within the range (inclusive) and returned.

Note

You will need to review the supplemental document "**Introduction to C Strings**" (https://github.com/Seneca-144100/BTP-Project/tree/master/A1/Introduction%20to%20C%20Strings.pdf) before attempting to do the next two functions

- Function: **getCharOption**
  This function must:
  - return a single character value and receives one argument:
    - an unmodifiable C string array representing a list of valid characters.

    Note: You must provide a **meaningful** parameter identifier (name)
  - get a single character value from the keyboard.
  - validate the entered character matches any of the characters in the received C string argument.
    Reminder: A C string will have a ***null terminator*** character marking the end of the array
  - display an error message if the entered character value is not in the list of valid characters (review the sample output for the appropriate error message)
    Note: Include in the error message the C string permitted characters
  - Continue to prompt for a single character value until a valid character is entered.
  - Guarantee a single character value is entered within the list of valid characters (as defined by the C string argument received) and returned.

- Function: **getCString**
  The purpose of this function is to obtain user input for a C string value with a length (number of characters) between the character range specified in the 2nd and 3rd arguments received (inclusive).

  This function:
  - must receive three (3) arguments and therefore needs three (3) parameters:
    - 1st parameter is a character pointer representing a C string
      ***Note**: Assumes the argument has been sized to accommodate at least the upper-bound limit specified in the 3rd argument received*
    - 2nd parameter represents an integral value of the **minimum** number of characters the user-entered value must be.
    - 3rd parameter represents an integral value of the **maximum** number of characters the user-entered value can be.
  - does not ***return*** a value, but does return a C string via the 1st argument parameter pointer.
  - must validate the entered number of characters is within the specified range. If not, display an error message (review the sample output for the appropriate error message). Note: If the 2nd and 3rd arguments are the same value, this means the C string entered must be a specific length.
  - must continue to prompt for a C string value until a valid length is entered.
  - guarantee's a C string value is entered containing the number of characters within the range specified by the 2nd and 3rd arguments (and return via the 1st argument pointer).

  [IMPORTANT]
  *You are NOT to use any of the **string library functions**; you must manually determine the entered C string length using a conventional iteration construct.*

A1-MS1: Sample Output

```
Assignment 1 Milestone 1
========================================

TEST #1: Enter the word 'error' [ENTER], then the number -100: error
ERROR: Value must be an integer: -100
*** PASS ***

TEST #2: Enter the number -100 [ENTER], then the number 200: -100
ERROR: Value must be a positive integer greater than zero: 200
*** PASS ***

TEST #3: Enter the word 'error' [ENTER], then the number -4 [ENTER], then 12 [ENTER], then
-3: error
ERROR: Value must be an integer: -4
ERROR: Value must be between -3 and 11 inclusive: 12
ERROR: Value must be between -3 and 11 inclusive: -3
*** PASS ***

TEST #4: Enter the number 14: 14
```

```
*** PASS ***

TEST #5: Enter the word 'error' then, the number -150.75: error
ERROR: Value must be a double floating-point number: -150.75
*** PASS ***

TEST #6: Enter the number -22.11 [ENTER], the number 225.55: -22.11
ERROR: Value must be a positive double floating-point number: 225.55
*** PASS ***

TEST #7: Enter the character 'R' [ENTER], then 'p' [ENTER], then 'r': R
ERROR: Character must be one of [qwerty]: p
ERROR: Character must be one of [qwerty]: r
*** PASS ***

TEST #8: Enter the word 'horse' [ENTER], then 'SENECA': horse
ERROR: String length must be exactly 6 chars: SENECA
Your Result: SENECA (Answer: SENECA)

TEST #9: Enter the words 'Seneca College' [ENTER], then 'IPC': Seneca College
ERROR: String length must be no more than 6 chars: IPC
Your Result: IPC (Answer: IPC)

TEST #10: Enter the words 'ipc' [ENTER], then 'SCHOOL': ipc
ERROR: String length must be between 4 and 6 chars: SCHOOL
Your Result: SCHOOL (Answer: SCHOOL)

Assignment #1 Milestone #1 completed!
```

Milestone – 1 Submission

1. ***This is a <u>test submission</u> for verifying your work only*** – no files will be submitted to your instructor – this will test your functions and confirm the outputs match to the expected output.
2. Upload (file transfer) your all header and source files:
   - **commonHelpers.h**
   - **commonHelpers.c**
   - **a1ms1.c**
3. Login to matrix in an SSH terminal and change directory to where you placed your source code.
4. Manually compile and run your program to make sure everything works properly:

   ```
   gcc -Wall a1ms1.c commonHelpers.c -o ms1 <ENTER>
   ```

   If there are no error/warnings are generated, execute it: ***ms1*** *<ENTER>*
5. Run the submission command below (replace **profname.proflastname** with **<u>your professors</u>** Seneca userid and replace **NAA** with your section):

   ```
   ~profName.proflastname/submit 100a1ms1/NAA_ms1 <ENTER>
   ```
6. Follow the on-screen submission instructions.

Milestone – 2 *(Worth 20%, Due Date: March 12th)*

In Milestone – 2, will expand on what was done from Milestone – 1. You will need to copy the header and source code files for the "**commonHelpers**" module to the Milestone – 2 directory and include them in the Milestone – 2 Visual Studio project.

You will begin this milestone by creating some new data types in an "***account.h***" header file, and the main function creates variable instances of those new types. You will prompt for user input and store the entered values to the appropriate variable members. It is expected you will call functions from the common helper library where appropriate. After data has been entered and stored, you will display the information back to the user in a tabular format (review the sample output section).

Specifications

1. In the "***account.h***" header file, you will create three (3) new data types that will be used to represent an account and related customer demographic and login information. These new data types will be defined in the module "account" and will require you to create another header file called "***account.h***" (add this new file to the Visual Studio project). We will NOT be creating a source code file for this module yet (this will be done in a later milestone). Create the following new structures:

   "Demographic"
   - This structure has **three (3) members**. You must provide the appropriate data type and meaningful identifiers/names for each described member:
     - An integer type that represents the birth year of a customer.
     - A double floating-point type that represents the household income.
     - A C string that represents the country the customer resides and should be able to store up to thirty (30) displayable characters.

   "UserLogin"
   - This structure has **three (3) members**. You must provide the appropriate data type and meaningful identifiers/names for each described member:
     - A C string that represents the name of the customer account holder and is what will be displayed when they are logged into the system and should be able to store up to thirty (30) displayable characters.
     - A C string that represents the user login name and should be able to store up to ten (10) displayable characters.
     - A C string that represents the password for the user and should be able to store up to eight (8) displayable characters.

   "Account"
   - This structure has **two (2) members**. You must provide the appropriate data type and meaningful identifiers/names for each described member:
     - An integer type that represents the account number associated to a customer.
     - A single character type that represents the account type (for example, an 'A' would represent a customer service agent, and a 'C' would represent a customer).

2. In the "***a1ms2.c***" source code file, you will find three variables declared ("**account**", "**login**", and "**demographic**") which are instances of the new types you created in the "***account.h***" header file. You

need to provide the necessary code that will assign user input to each of the variable members. Use the example output and the source code comments to help guide you in accomplishing this task.

Reminder

You should be utilizing the common helper functions you created in Milestone – 1 as much as possible where appropriate to help you do this task!

3. The last task you need to do, is complete the "***displayAccount***" function definition located after the main function. The formatted table header is provided for you. To help you format the data values to properly align with the header, you can use the following format specifiers in your printf statement for the respective fields:

Column Name	Format Specifier
Acct#	%05d
Acct.Type	%-9s
Birth	%5d
Household-Income	$%15.2lf
Country	%-15s
Disp.Name	%-15s
Login	%-10s
Password	%8s

You will need to reference the appropriate arguments received by this function and their respective members to provide your printf function with the required data.

A1-MS2: Sample Output

```
Assignment 1 Milestone 2
=========================================

TEST #1: Enter the word 'error' [ENTER], then the number -100: error
ERROR: Value must be an integer: -100
*** PASS ***

TEST #2: Enter the number -100 [ENTER], then the number 200: -100
ERROR: Value must be a positive integer greater than zero: 200
*** PASS ***

TEST #3: Enter the number -4 [ENTER], then 12 [ENTER], then -3: -4
ERROR: Value must be between -3 and 11 inclusive: 12
ERROR: Value must be between -3 and 11 inclusive: -3
*** PASS ***

TEST #4: Enter the word 'error' then, the number -150.75: error
ERROR: Value must be a double floating-point number: -150.75
*** PASS ***

TEST #5: Enter the number -22.11 [ENTER], the number 225.55: -22.11
ERROR: Value must be a positive double floating-point number: 225.55
*** PASS ***
```

```
TEST #6: Enter the word 'error' then, the number 11: error
ERROR: Value must be an integer: 11
*** PASS ***

TEST #7: Enter the character 'R' [ENTER], then 'p' [ENTER], then 'r': R
ERROR: Character must be one of [qwerty]: p
ERROR: Character must be one of [qwerty]: r
*** PASS ***

TEST #8: Enter the word 'horse' [ENTER], then 'SENECA': horse
ERROR: String length must be exactly 6 chars: SENECA
Your Result: SENECA (Answer: SENECA)

TEST #9: Enter the words 'Seneca College' [ENTER], then 'IPC': Seneca College
ERROR: String length must be no more than 6 chars: IPC
Your Result: IPC (Answer: IPC)

TEST #10: Enter the words 'ipc' [ENTER], then 'SCHOOL': ipc
ERROR: String length must be between 4 and 6 chars: SCHOOL
Your Result: SCHOOL (Answer: SCHOOL)

Account Data Input
----------------------------------------
Enter the account number: 50001 Account
ERROR: Value must be an integer: 50001
Enter the account type (A=Agent | C=Customer): Agent
ERROR: Character must be one of [AC]: a
ERROR: Character must be one of [AC]: c
ERROR: Character must be one of [AC]: C

User Login Data Input
----------------------------------------
Enter user login (10 chars max): Williamson Willie
ERROR: String length must be no more than 10 chars: Williamson
Enter the display name (30 chars max): Seneca Student
Enter the password (must be 8 chars in length): jump
ERROR: String length must be exactly 8 chars: jumping
ERROR: String length must be exactly 8 chars: seventeen
ERROR: String length must be exactly 8 chars: seneca21

Demographic Data Input
----------------------------------------
Enter birth year (current age must be between 18 and 110): 2004
ERROR: Value must be between 1911 and 2003 inclusive: 1910
ERROR: Value must be between 1911 and 2003 inclusive: 1988
Enter the household Income: $1 million 5 hundred
ERROR: Value must be a double floating-point number: -500.25
ERROR: Value must be a positive double floating-point number: 0.0
ERROR: Value must be a positive double floating-point number: 188222.75
Enter the country (30 chars max.): CANADA

Acct# Acct.Type Birth Household-Income Country         Disp.Name       Login      Password
----- --------- ----- ---------------- --------------- --------------- ---------- --------
50001 CUSTOMER   1988 $      188222.75 CANADA          Seneca Student  Williamson seneca21

Assignment #1 Milestone #2 completed!
```

Reflection (Worth 20%, Due Date: March 12th)

Academic Integrity

It is a violation of academic policy to copy content from the course notes or any other published source (including websites, work from another student, or sharing your work with others).

Failure to adhere to this policy will result in the filing of a violation report to the Academic Integrity Committee.

Instructions

- Create a text file named "**reflect.txt**" and record your answers to the below questions in this file.
- Answer each question in sentence/paragraph form unless otherwise instructed.
- A minimum **300** overall word count is required (does NOT include the question).
- Whenever possible, be sure to substantiate your answers with a brief example to demonstrate your view(s).

1. From the helper functions library, what function was the most challenging to define and clearly describe the challenge(s) including how you managed to overcome them in the context of the methods used in preparing your logic, debugging, and testing.

2. Describe how the "helper functions" library contributes toward making the code easier to read and include in your analysis why the library will make your code easier to maintain.

3. Comment on why the C programming language provides a programmer the ability to create new data types (struct) and what advantages does this have? Are there limitations in the construction of a new data type – if so, what specifically?

Reflections will be graded based on the published rubric:
https://github.com/Seneca-144100/BTP-Project/tree/master/Reflection%20Rubric.pdf

Milestone – 2 Submission

1. Upload (file transfer) your all header and source files including your reflection:
   `commonHelpers.h commonHelpers.c account.h a1ms2.c reflect.txt`
2. Login to matrix in an SSH terminal and change directory to where you placed your source code.
3. Manually compile and run your program to make sure everything works properly:
   ```
   gcc -Wall a1ms2.c commonHelpers.c -o ms2 <ENTER>
   ```
   If there are no error/warnings are generated, execute it: ***`ms2`*** *<ENTER>*
4. Run the submission command below (replace **profname.proflastname** with **your professors** Seneca userid and replace **NAA** with your section):
   ```
   ~profName.proflastname/submit 100a1ms2/NAA_ms2 <ENTER>
   ```
5. Follow the on-screen submission instructions.