TERM ASSIGNMENT

WINTER 2022

Contents

Submission Policy

- **Each Milestone is due on Sunday by the end of day 23:59 EST (UTC – 5)**
- **Source (.c) and text (.txt) files that are provided with each milestone MUST be used or your work will not be accepted. Resubmission will be required attracting the respective late deduction based on the date/time submitted relative to the deadline date/time.**
- **Late submissions are accepted, but there is a 25% deduction per day late**
- **All work must be submitted by the matrix submitter – no exceptions**
- **All files you create or modify MUST contain the following declaration at the top of all documents or it will require a resubmission attracting the respsective late deduction based on the date/time resubmitted to the deadline date/time**
- **Coding violations (principles, style guidelines, etc.) deemed unacceptable as set by your instructor will not be accepted and will require a resubmission attracting the respsective late deduction based on the date/time resubmitted to the deadline date/time**

```
*****************************************************************************
              <assessment name example: Assignment 1 MS-1 >
Full Name  :
Student ID#:
Email      :
Section    :

Authenticity Declaration:
I declare this submission is the result of my own work and has not been
shared with any other student or 3rd party content provider. This
submitted
piece of work is entirely of my own creation.
*****************************************************************************
```

Notes

- Due dates are in effect **even during a holiday**
- You are responsible for **backing up your work regularly**
- It is expected and assumed that for each milestone, you will plan your coding solution by using the computational thinking approach to problem solving and that **you will code your solution based on your defined pseudo code algorithm**.

Assignment #1

Worth: 15% of final grade

Veterinarian Clinic System

Milestone	Worth	Due Date
1	**5%**	**March 27 (23:59 EST)**
2	**5%**	**April 3 (23:59 EST)**
3	**5%**	**April 10 (23:59 EST)**

Introduction

This assignment has been 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 milestone #3, you will have created a basic patient (pet) appointment system for a veterinary clinic. Patient contact information will be managed as well as the scheduling and management of appointments.

Each milestone will build upon the previous, adding more functionality and components. Milestone #1 is focused on providing helper functions that will aid you in 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 (so use them whenever possible and don't duplicate logic already done).

Each milestone will be released weekly and can be downloaded or cloned from GitHub:
https://github.com/Seneca-144100/IPC-Project

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

Preparation

Download or clone the Assignment 1 from GitHub.

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 (Code: weight 2.5%)

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.

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 yet). 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 the three milestones. 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).

Core Module

1. Create a module called "core". To do this, you will need to create two files: "***core.h***" and "***core.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).
   - Copy and paste the **commented section** provided for you in the ***a1ms1.c*** file (top portion) to all files you create
   - Fill in the information accordingly
3. The "***core.c***" file will require the usual **standard input output** system library as well as the new user library "***core.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 "***core.h***" header file:

```
// Clear the standard input buffer
void clearInputBuffer(void);

// Wait for user to input the "enter" key to continue
void suspend(void);
```

Functions

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

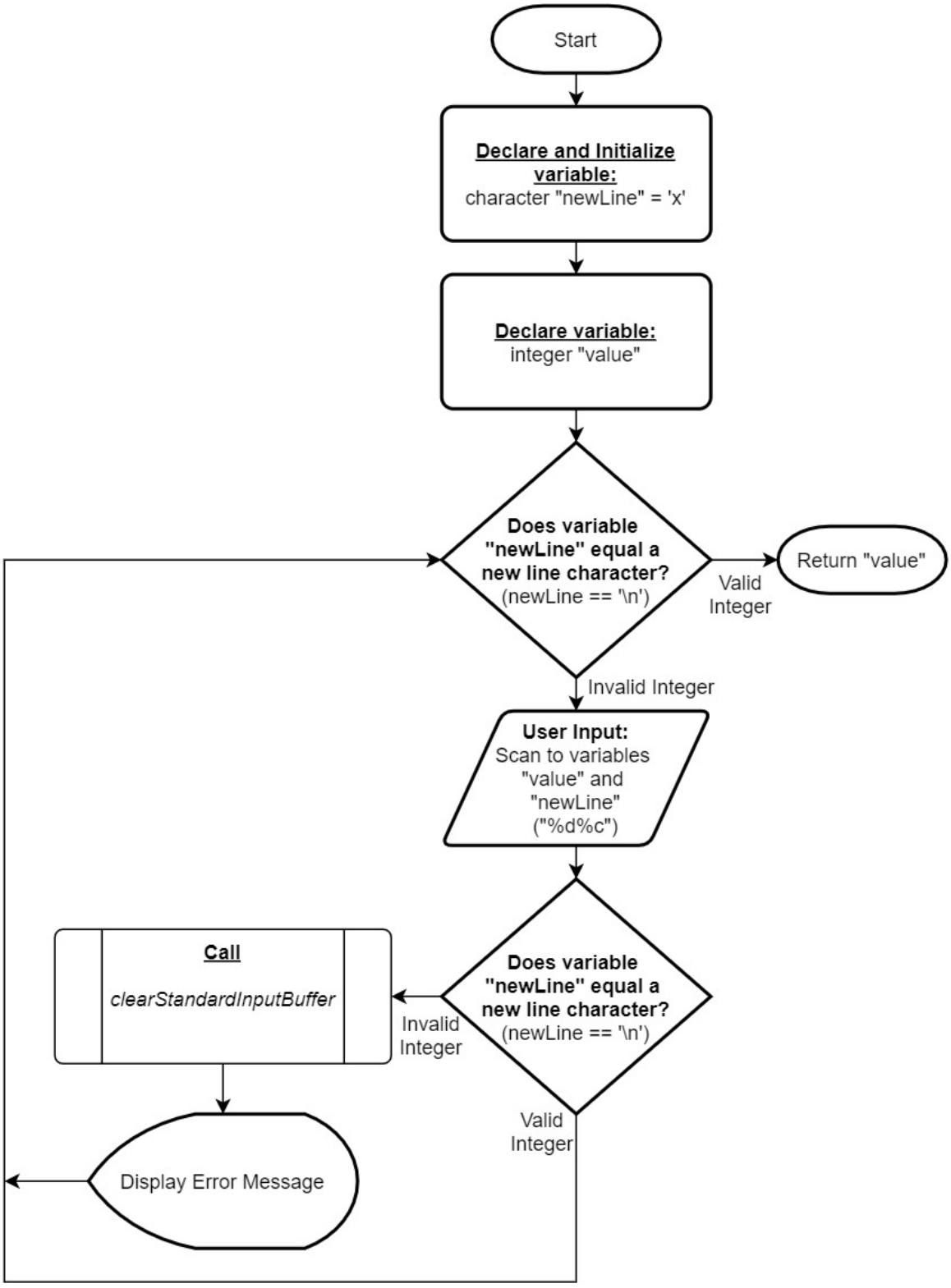
```
// As demonstrated in the course notes:
// https://intro2c.sdds.ca/D-Modularity/input-functions#clearing-the-buffer
// Clear the standard input buffer
void clearInputBuffer(void)
{
    // Discard all remaining char's from the standard input buffer:
    while (getchar() != '\n')
    {
        ; // do nothing!
    }
}
```

```
// Wait for user to input the "enter" key to continue
void suspend(void)
{
    printf("<ENTER> to continue...");
    clearInputBuffer();
    putchar('\n');
}
```

6. Each function briefly described below will require a function prototype to be placed in the "***core.h***" file, and their respective function definitions in the "***core.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 (6) functions** in total):

   - Function: **inputInt**
     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.

- 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 ***clearInputBuffer*** function, followed by displaying an error message and continue to prompt for a valid integer. Review the flowchart below that describes this process.



- Function: **inputIntPositive**
  This function must:
  - return an integer value and receives no arguments.
  - perform the same operations as **inputInt** 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: **inputIntRange**
  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:
    - A range is a set of numbers that includes the upper and lower limits (bounds)
    - You must provide **meaningful** parameter identifiers (names)
  - performs the same operations as **inputInt** 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.

- Function: **inputCharOption**
  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: **inputCString**
  The purpose of this function is to obtain user input for a C string value with a length (number of characters) in the character range specified by 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.*

- Function: **displayFormattedPhone**
  The purpose of this function is to display an array of 10-character digits as a formatted phone number.
  This function:
  - must receive one (1) argument and therefore requires one (1) parameter:
    - 1st parameter is an unmodifiable character pointer representing a C string..
  - does not **return** a value..
  - should not assume a valid C string array, and therefore, should carefully validate the argument char array to determine:
    - it is exactly 10 characters long
    - only contains digits (0-9)
  - should display "`(___)___-____`" when the argument C string char array is not a 10-character all digit value.
  - should display the phone number in the following format when it is a valid C string phone number: "`(###)###-####`" (where each # is the character digit from the C string argument char array).
  - NOTE: Do not add a newline character at the beginning or end of the displayed value.

A1-MS1: Sample Output

```
Assignment 1 Milestone 1: Tester
================================

TEST #1 - Instructions:
1) Enter the word 'error'  [ENTER]
2) Enter the number '-100' [ENTER]
:>error
Error! Input a whole number: -100
//////////////////////////////////////
TEST #1 RESULT: *** PASS ***
//////////////////////////////////////

TEST #2 - Instructions:
1) Enter the number '-100' [ENTER]
2) Enter the number '200'  [ENTER]
:>-100
ERROR! Value must be > 0: 200
//////////////////////////////////////
TEST #2 RESULT: *** PASS ***
//////////////////////////////////////

TEST #3 - Instructions:
1) Enter the word 'error' [ENTER]
2) Enter the number '-4'  [ENTER]
3) Enter the number '12'  [ENTER]
4) Enter the number '-3'  [ENTER]
:>error
Error! Input a whole number: -4
ERROR! Value must be between -3 and 11 inclusive: 12
ERROR! Value must be between -3 and 11 inclusive: -3
//////////////////////////////////////
TEST #3 RESULT: *** PASS ***
//////////////////////////////////////

TEST #4 - Instructions:
1) Enter the number '14' [ENTER]
:>14
//////////////////////////////////////
TEST #4 RESULT: *** PASS ***
//////////////////////////////////////

TEST #5 - Instructions:
1) Enter the character 'R' [ENTER]
2) Enter the character 'e' [ENTER]
3) Enter the character 'p' [ENTER]
4) Enter the character 'r' [ENTER]
:>R
ERROR: Character must be one of [qwErty]: e
ERROR: Character must be one of [qwErty]: p
ERROR: Character must be one of [qwErty]: r
//////////////////////////////////////
TEST #5 RESULT: *** PASS ***
//////////////////////////////////////
```

```
TEST #6: - Instructions:
1) Enter the word 'horse'   [ENTER]
2) Enter the word 'chicken' [ENTER]
3) Enter the word 'SENECA'  [ENTER]
:>horse
ERROR: String length must be exactly 6 chars: chicken
ERROR: String length must be exactly 6 chars: SENECA
//////////////////////////////////////
TEST #6 RESULT: SENECA (expected result: SENECA)
//////////////////////////////////////

TEST #7: - Instructions:
1) Enter the words 'Seneca College' [ENTER]
2) Enter the word 'CATS'            [ENTER]
:>Seneca College
ERROR: String length must be no more than 6 chars: CATS
//////////////////////////////////////
TEST #7 RESULT: CATS (expected result: CATS)
//////////////////////////////////////

TEST #8: - Instructions:
1) Enter the word 'dogs'  [ENTER]
2) Enter the word 'HORSES' [ENTER]
:>dogs
ERROR: String length must be between 5 and 6 chars: HORSES
//////////////////////////////////////
TEST #8 RESULT: HORSES (expected result: HORSES)
//////////////////////////////////////

//////////////////////////////////////
TEST #9 RESULT:
Expecting (___)___-____ => Your result: (___)___-____
Expecting (___)___-____ => Your result: (___)___-____
Expecting (___)___-____ => Your result: (___)___-____
Expecting (___)___-____ => Your result: (___)___-____
Expecting (___)___-____ => Your result: (___)___-____
Expecting (___)___-____ => Your result: (___)___-____
Expecting (416)111-4444 => Your result: (416)111-4444
//////////////////////////////////////

Assignment #1 Milestone #1 testing completed!
```

Reflection (Weight: 2.5%)

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 core 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. It is good practice to initialize variables to a "safe empty state". With respect to variable initialization, what is the difference between assigning **0** and **NULL**? When do you use one over the other and why?

3. Your friend (also a beginner programmer) is having difficulty understanding how to manage the "standard input buffer" (particularly when there is residual data). Your friend has read all the course notes, Googled the topic, followed along with course lectures about this topic, but is still struggling with this concept. Describe exactly how you would attempt to help your friend understand this topic?

Milestone – 1 Submission

1. Upload (file transfer) your all header and source files including your reflection:
   `core.h core.c a1ms1.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 a1ms1.c core.c -o ms1 <ENTER>
   ```
   If there are no error/warnings are generated, execute it: ***`ms1`*** *`<ENTER>`*
4. Run the submission command below (replace **profname.proflastname** with **your professors** Seneca userid and replace **NAA** with your section):
   ```
   ~profName.proflastname/submit 144a1ms1/NAA_ms1 <ENTER>
   ```
5. Follow the on-screen submission instructions.

Milestone – 2 (Code: weight 2.5%)

Milestone-2 comes with a starting framework for the veterinary clinic system (the menu system and basic display functions). Although a lot of code is already provided for you, there are still many functions you will need to define.

Milestone-2 focuses on the management of the clinic's "patients" which are the records representing the family pets. You will need to create some data structures along with several functions that will be responsible for carrying out menu selections which perform specific tasks in the managing of patient data.

A new module "clinic" is now required which will be used to organize all the clinic-centric components and where you will be placing most of your remaining work.

Specifications

It is important to note that this code will not compile until you copy your work from Milestone-1 into the files for Milestone-2, create some new data types, and define the remaining uncoded function definitions.

Core Module

The first thing you will need to do is copy and paste the contents of your Milestone-1 core module code (***core.h*** and ***core.c*** files) into the Milestone-2 ***core.h*** and ***core.c*** files. **Carefully review the source code comments** provided in this milestone's ***core.h*** and ***core.c*** files and insert your code from Milestone-1 where directed.

The Core module you developed from Milestone-1 will need to be reviewed and upgraded to include the application of ***string library functions*** wherever it makes sense to apply them. The string library contains many functions, so to help you narrow your efforts, a list of functions to evaluate and consider have been provided for you below.

String Library

- Review your code in the ***core.c*** file and upgrade where necessary to use functions available from the string library. Functions you should be considering can be any of the following (but no others):

  strlen, strcpy, strcat, strcmp, strncat, strncmp, strncpy, strchr, strrchr

Reminder: You will need to include the string library to implement any of the string functions!

Clinic Module

You will need to create a couple of new data structures to represent the patient information. This includes a Patient and Phone type. These types need to be defined in the ***clinic.h*** file (**review the comments** in the ***clinic.h*** file for placement).

Data Structure Types

Review the function "**displayPatientData**" in the ***clinic.c*** file and the pets variable initialization in the **main** function (***a1m2.c*** file) to determine the member names and order/sequence.

Phone

The Phone type will have **two** members:

1. A C string member for storing the phone description that can be any one of the following values:
   - "**CELL**", "**HOME**", "**WORK**", "**TBD**"
   - Use the appropriate provided macro's for sizing

2. A C string member for storing the 10-digit phone number (example: "**1112224444**")

Patient

The Patient type will have **three** members:

1. An integer member responsible for storing a unique patient number

2. A C string member for storing the patient's (pet's) name
   - Use the appropriate provided macro for sizing

3. A Phone type for storing the patient's contact information

Now that this is done, the only remaining thing to do is to define the functions!

Functions

Familiarize yourself with the functions already coded for you in the ***clinic.c*** file that implement the function prototypes defined in the ***clinic.h*** file. These functions implement the main menu system for the application. Notice how they call other functions to perform specific tasks? Many of these functions need to be defined.

The functions needing to be defined are identified in the ***clinic.h*** file under the commented section "**All the below functions need defining**".

Apply the same development strategy as mentioned in Milestone-1 (Strategy-2) whereby 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. Applying this strategy will allow you to compile the project and begin coding the function definitions gradually (testing along the way as you complete one by one).

clinic.c

1. Locate the section of code where you need to begin defining the remaining function definitions ("!!! Put all the remaining function definitions below !!!"

2. Copy the function prototypes in-place of the "**ToDo:**" comment for each respective function, and remove the trailing semi-colon ';' and replace with a code block using curly braces { }

3. If the function requires a return value (not a void) then return a hardcoded value as a temporary measure (obviously you will have change this later).

4. At this point you should be able to compile your project without errors. If you are not able to, review the preceding steps carefully and don't proceed until you can compile the code.

The following is a short description of what each function needs to accomplish (these are described in the order they are listed in the ***core.h*** file).

Note: You may code the definitions to these functions in any order you like

Menu & Item Selection Functions

// Display's all patient data in the FMT_FORM | FMT_TABLE format
```
void displayAllPatients(const struct Patient patient[], int max, int fmt);
```

- This function should iterate the patient array for up to the max number of elements and display each patient record in the desired format as specified by the last argument 'fmt'.
- Patient records that have a zero value for the patient number should NOT be displayed.
- This function should display: "**`*** No records found ***`**" if there were no eligible records to display.
- Reminder: There are functions already provided to you that will display the table header (if required based on '***fmt***') and display a single record!

// Search for a patient record based on patient number or phone number
```
void searchPatientData(const struct Patient patient[], int max);
```

- This function should present a menu with two search options:
  1. "By patient number"
     - When this option is chosen, you should call the function ***searchPatientByPatientNumber***
  2. "By phone number"
     - When this option is chosen, you should call the function ***searchPatientByPhoneNumber***
- Review the sample output to see how this is used by the user

// Add a new patient record to the patient array
```
void addPatient(struct Patient patient[], int max);
```

- This function should test to see if the patient array has a free element for a new record (this is identified when the patient number value is zero). Be sure to store the array index!
- If the array does not have room for a new record, it should display "**`ERROR: Patient listing is FULL!`**"
- If an index was found where a new record can be stored, you must determine the next unique patient number (call function: ***nextPatientNumber***) and assign that number to the element at the index where the new record will be stored
- Finally, get user input for the new record by calling the function: ***inputPatient***
- Display a confirmation message after the data is input: "**`*** New patient record added ***`**"

// Edit a patient record from the patient array
v**oid editPatient(struct Patient patient[], int max);**

- The user must be prompted to enter the unique patient number for the record to be edited: "**Enter the patient number:** "
- You must then locate that record to see if it exists by calling the function: ***findPatientIndexByPatientNum***
- If the patient record exists, then call the function: ***menuPatientEdit*** for editing options
- If the patient record does not exist, display an error message: "**ERROR: Patient record not found!**"

// Remove a patient record from the patient array
void removePatient(struct Patient patient[], int max);

- Like the ***editPatient*** function (above), you must prompt the user to enter the unique patient number for the record to remove
- You must then locate that record to see if it exists by calling the function: ***findPatientIndexByPatientNum***
- If the patient record exists, then you must display the record to the user in "form" format and prompt for confirmation to remove the record: "**Are you sure you want to remove this patient record? (y/n):** "
    - If the user confirms to remove the record, the patient information should be set to a safe empty state to make it available again for a new record and display the message: "**Patient record has been removed!**"
    - If the user does not confirm the removal, display the message: "**Operation aborted.**"
- If the patient record does not exist, display an error message: "**ERROR: Patient record not found!**"

Utility Functions

// Search and display patient record by patient number (form)
void searchPatientByPatientNumber(const struct Patient patient[], int max);

- The user must be prompted for the unique patient number: "**Search by patient number:** "
- You must then locate that record to see if it exists by calling the function: ***findPatientIndexByPatientNum***
- If the patient record exists, then you must display the record to the user in "form" format
- If the patient record can't be located, display the message: "***** No records found *****"

// Search and display patient records by phone number (tabular)
void searchPatientByPhoneNumber(const struct Patient patient[], int max);

- The user must be prompted for the patient 10-digit phone number: "**Search by phone number:** "
- The patient array must be searched for all matchs on the entered phone number
    - Note: There can be more than one match since one family can have several pets
- For each patient record found, display the patient record in "tabular" format
- If no record matches could be found, display the message: "***** No records found *****"

// Get the next highest patient number
```
int nextPatientNumber(const struct Patient patient[], int max);
```
- The next patient number is determined by adding one to the largest patient number in the patient array
- The calculated next number should be returned

// Find the patient array index by patient number (returns -1 if not found)
```
int findPatientIndexByPatientNum(int patientNumber,
                                 const struct Patient patient[], int max);
```
- This function should search the patient array for the element that matches the "patientNumber" argument value
- If the record is found, the index position of the matched element should be returned.
- If the record can't be located (matched) then -1 should be returned.

User Input Functions

// Get user input for a new patient record
```
void inputPatient(struct Patient* patient);
```
- This function will receive a pointer to a Patient type that will already have the next patient number assigned (the user entered data should be stored to the patient argument pointer)
- A title should be displayed: "**`Patient Data Input`**" (with dashed characters as an underline, see sample output)
- The patient number should be displayed to 5 digits with zero's padding on the left side
- The user should be prompted for the patient (pet) name: "**`Name  :`**" (Note: there are 2 spaces after the 'e' and before the colon)
- The user should then be prompted to enter the phone information (call function: ***inputPhoneData***)

// Get user input for phone contact information
```
void inputPhoneData(struct Phone* phone);
```
- The user should be presented with a menu to choose how the patient would like to be contacted. There are four options:
  1. Cell
  2. Home
  3. Work
  4. TBD *Note: "TBD" is short for "To be determined".*
- The phone description should be assigned the UPPERCASE equivalent of the user selection (example, if the user enters option 3, then "WORK" will be assigned to the phone description.)
- IF option 4 "TBD" is selected, the phone number should be set to a safe empty state
- Options 1 to 3 however, should prompt the user for a 10-digit phone number and assign the entered value to the phone number member.

A1-MS2: Sample Output

Please review the provided text file for this milestone (on GitHub) "***a1ms2_output.txt***"

Reflection (Weight: 2.5%)

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. What factors must you consider when naming a module or library? Why do you think it is a suggested best practice to identify a library's header and source code files using the same name? Give an example from this assignment to support your argument.

2. So far, you have created and are maintaining two modules in the development of this application. Why do we have these two modules and explain why these aren't combined into a single module?

3. Modules are typically split into a header file and an implementation file. What purpose and benefit is there to this division? Refer to the module(s) in this application in your explanation.

Milestone – 2 Submission

1. Upload (file transfer) your all header and source files including your reflection:
   `core.h core.c clinic.h clinic.c 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 core.c clinic.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 144a1ms2/NAA_ms2 <ENTER>
   ```
5. Follow the on-screen submission instructions.

Milestone – 3 (Code: weight 2.5%)

Milestone-3 completes the veterinary clinic system with the addition of appointment management functionality and the importing of patient and appointment information from data text files. You will need to create additional data structures along with several functions that will be responsible for carrying out the new appointment management menu options that perform specific tasks in the management of appointment data.

Specifications

Like Milestone-2, it is important to note that this code will not compile until you copy your work from Milestone-2 into the provided files for Milestone-3. You will also need to create some new data types, as well as define the mandatory uncoded function prototypes and definitions that are new to Milestone-3.

Three ***clinic*** module/library functions have been completely supplied for you: "***displayScheduleHeader***", "***displayScheduleData***", and "***menuAppointment***", along with two function prototypes: "***importPatients***" and "***importAppointments***" (responsible for data import). These functions provide you with the necessary framework to get started in this final milestone.

In this milestone, **it is expected you will create additional functions** as you see fit to help you get the job done. It is also expected you will **follow and adhere to the structured design principles** (Functions | Introduction to C (sdds.ca)).

When you create your own functions, be sure to organize them into the established commented sections for each module/library so you can easily find and maintain them:

Core
- User Interface
- User Input
- Utility

Clinic
- Display Functions
- Menu & Item Selection
- Utility
- User Input
- File

Recommended Approach

You need to get the project ready for development so it can compile and test your new functions with actual data. It is recommended you develop the new components in the following sequence:

1. Create/define the new data types: ***Time***, ***Date***, and ***Appointment***

2. Code the ***importPatients*** and ***importAppointments*** function definitions. The ***main*** function requires these functions to be working so the application can start with data preloaded.

3. Create the function prototypes and empty function definitions (stubs) for those functions called within the ***menuAppointment*** function.

Clinic Module

In this milestone, you will be working almost exclusively with the ***clinic*** module unless you find reason to add more generalized functions to the ***core*** module.

Data Structures

A few new data structures will be needed to represent the appointment information. This includes a ***Time***, a ***Date***, and an ***Appointment*** type. These types need to be defined in the ***clinic.h*** file (**review the comments** in the ***clinic.h*** file for placement).

Hint

Review the following resources to help you determine the members for these new data types:

- Function: "***displayScheduleData***" (clinic.c file)
- Data file: "***appointmentData.txt***" (see description in next section)

Data Files

There are two data files you will need to import into the application. One contains patient data, while the other contains appointment information. These files are provided along with the project files on GitHub.

patientData.txt

The data fields for the patient data are **separated by a pipe (|)** character in the following order:

- Patient number
- Patient name
- Contact phone description
- Contact phone number

appointmentData.txt

The data fields for the appointment data are **separated by a comma (,)** character in the following order:

- Patient number
- Appointment year
- Appointment month
- Appointment day
- Appointment hour
- Appointment minute

Functions

Data Import (text files)

You must create the "***importPatients***" and "***importAppointments***" function definitions (in the ***clinic.c*** file) that will read-in the text data and store the data to their respective ***struct Patient*** and ***struct Appointment*** arrays. Review the ***main*** function to see how they are called. The function prototypes for these functions have been provided for you located in the ***clinic.h*** header file.

Here is a brief overview of how these functions should work. Both import functions should do the following:

- Read the data file (file name is provided in the first argument)
- Store the data to the second argument, an array of the respective struct data type
- Respect the array size specified by the 3rd argument even when the data file has more records
- Must return the total number of records read from the file and stored to the array, which may be less than the total number of records in the file

Appointment Management

To get you started, the appointment management menu and display functions have been provided for you:

- ***menuAppointment***
- ***displayScheduleHeader***
- ***displayScheduleData***

Hints

- Create temporary empty function shells for the following functions:
    - ***viewAllAppointments***
    - ***viewAppointmentSchedule***
    - ***addAppointment***
    - ***removeAppointment***

  All these functions are called from the "***menuAppointment***" function and will need to exist for you to be able to compile your code.

- Review the sample output text file (***a1ms3_output.txt***) that came with the project files to see how these menu options should work

viewAllAppointments

- Carefully review the sample output text file (***a1ms3_output.txt***) that came with the project files and notice the order of the data is not presented in the order it is positioned in the appointments array.
- At some point in the logic for this menu option, you will need to call the "***displayScheduleHeader***" and "***displayScheduleData***" functions

viewAppointmentSchedule

- This process should display only the appointments scheduled for a specific date. Therefore, the user must be prompted for a specific date (year, month, and day) prior to displaying the results.
- Date input prompting and validations must accurately determine the number of days in a given month and accommodate leap years
- Carefully review the sample output text file (***a1ms3_output.txt***) that came with the project files and notice the order of the data is not presented in the order it is positioned in the appointments array.
- At some point in the logic for this menu option, you will need to call the "***displayScheduleHeader***" and "***displayScheduleData***" functions

addAppointment

- This process should test if there is an available element in the appointments array for a new appointment to be added. Available appointments can be determined by testing the patient number which must be less than 1 to indicate an empty/available element.
- Validation of the entered patient number must be performed
- Appointment times must adhere to the following rules:
  - **Operation hours** are determined based on macro's that define the start and end hours so they can be modified in one place without affecting the code logic
  - The effective appointment start times must fall within the inclusive hour range defined by the macros mentioned above (example: 9:00 – 16:00, so the last valid appointment time can be 16:00)
  - The entered minute value must align with the set appointment minute interval which should be represented as a macro so it can be modified in a single place and not affect the code logic (example: if appointments are in 15-minute intervals it is only possible to have appointments starting at 0, 15, 30, or 45 minutes on the hour)
  - Only ONE appointment can occupy a time slot for the given year, month, and day (**no double booking or overlap is allowed**)
- Again, carefully review the sample output text file (***a1ms3_output.txt***) that came with the project files to see how this process should work.

removeAppointment

- Appointments are removed by patient number and a specific date (year, month, and day).
- The entered patient number must be validated as a patient who has not been removed from the ***patients*** array or the removal should be denied (this will ensure that past appointment data will remain intact).
- When an appointment match occurs (based on the entered patient number and date), the patient information details should be displayed, and user confirmation must be received to remove the appointment
- To mark an appointment as removed, the appropriate appointments array element must be set to a safe empty state.
- Again, carefully review the sample output text file (***a1ms3_output.txt***) that came with the project files to see how this process should work.

A1-MS3: Sample Output

Please review the provided text file for this milestone (on GitHub) "***a1ms3_output.txt***"

Reflection (Weight: 2.5%)

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**" be sure to **include your confirmation of authenticity and personal information at the beginning of the file** and record your answers to the below questions in this file.
- **Refer to the reflection rubric link noted at the beginning of this document on page 3**
- Answer each question in sentence/paragraph form unless otherwise instructed.
- Do not include the question in your response.
- A minimum **600** overall word count is required.
- Whenever possible, be sure to substantiate your answers with a brief example to demonstrate your view(s).

1. This milestone required you to create additional functions that were not specified or provided for you. Briefly describe **three functions** you created and include the **purpose** and **value** those functions contributed towards the application. Organize your answer for each function by first including the function prototype (not the definition) followed by your explanation. (Minimum: 200 words)

2. The "***addAppointment***" function must perform many operations. How many lines of code did you use for this function? How many lines did you save by applying pattern recognition and the use of functions? Identify all the sections of logic you were able to consolidate into useful functions to help with readability and maintainability of the code. (Minimum: 200 words)

3. This milestone demanded a concerted effort in time management to insure you could complete the work on-time. Breakdown how you spent your week of development and include each function/section of logic you worked on describing the overall success/challenges you had along the way. (Minimum: 200 words)

Milestone – 3 Submission

1. Upload (file transfer) your all header and source files including your reflection:
   `core.h core.c clinic.h clinic.c a1ms3.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 a1ms3.c core.c clinic.c -o ms3 <ENTER>
   ```
   If there are no error/warnings are generated, execute it: ***`ms3`*** *`<ENTER>`*
4. Run the submission command below (replace **profname.proflastname** with **your professors** Seneca userid and replace **NAA** with your section):
   ```
   ~profName.proflastname/submit 144a1ms3/NAA_ms3 <ENTER>
   ```
5. Follow the on-screen submission instructions.