Programming Project 4

Due: Thursday, 10/29/20 at 11:59 pm

(text in red are updates, make sure you are following the updated instructions to pass autograder)

Fast Food Order Management

For this project, write a C program that will implement a food ordering system used in a fast food restaurant. This restaurant only serves burgers and salads and when people want to order food, they give their name and food choices to the cashier and then wait until those in front of them have been served. The program must use **a linked list** to implement the queue-like data structure.

The linked list is to maintain the following information for each group that is waiting:

- name (we assume a maximum name length of 30 characters)
- food items (number of burgers and salads ordered)
- in-restaurant status: whether the group has called ahead or is waiting in the restaurant

The system allows you to call and put your order in before you arrive at the restaurant, but it does not take orders for a specific time and date (i.e. 4 burgers and 2 salads for 7pm on Saturday). Note: these call-ahead groups will still need to check in when they arrive, so the cashier knows they are waiting in the restaurant.

Groups are added to the order list when they call-ahead or when they arrive at the restaurant. Groups are always added to the end of the order list. The system will require that each name used be unique. So, when a group is added to the order list, the system must make sure that no other group is already using that name.

The restaurant staff working in the kitchen have a steady speed in preparing food and they work on one order at a time. On average, preparing a salad takes 2 minutes while burger takes 5 minutes. The system needs to track how many food items have been ordered and provide customers with an estimated wait time.

You can assume that every time an order is being called, it will go to the first group in line. However, in order to pick up the food, the group needs to be present at the restaurant. If they are not, then the system will call the next eligible group for pick up (an eligible group is a group that the number of burgers and salads they ordered is less than or equal to prepared order).

The commands used by this system are listed below and are to come from standard input. Your program is to prompt the user for input and display error messages for unknown commands or improperly formatted commands. Note that the name of the group when given will be given as the last item on the input line. The name of the group may contain white space characters in the middle of the name but not at the beginning or end of the name. Each command given must display some information about the command being performed. Code to implement this interface is provided in the program **proj4base.c**.

Command	Description
q	Quit the program.
?	List the commands used by this program and a brief description of how to use each one.
a <#burgers> <#salads> <name>	Add the order to the order list using the given order and name specifying the group is waiting in the restaurant. The order's information is added to the end of the list. If the name already exists in the order list, give an error message and do not add the information.
c <#burgers> <#salads> <name>	Add the order to the order list using the given order and name specifying the order as a call ahead order. The order's information is added to the end of the list. If the name already exists in the order list, give an error message and do not add the information.
w <name>	Mark the call ahead order using the given name as waiting in the restaurant. If the name does not exist is the order list or is not a call ahead group, give an error message.
r <#burgers> <#salads>	Retrieve and remove the first order on the order list that is waiting in the restaurant and contains less than or equal to number of prepared burgers and salads. Note that "first" is the order that has been in the order list the longest.
l <name>	List total number of orders that are in the order list in front of the order specified by the given name. If the name does not exist, give an error message.
t <name>	Give an estimated waiting time based on the order list knowing preparing burgers will take 5 minutes and preparing salads takes 2. If the name does not exist, give an error message.
d	Display the total number of orders in the order list. Also display the names, order details and in-restaurant status of all orders in the order list in order from first to last.

Note that **<#burgers>** and **<#salads>** are to be integer values and **<name>** is a list of characters. The < and > symbols are NOT part of the input but being used to describe the input.

Error messages

The proj4base.c already performs many error checks and outputs proper statements. You still need to check for invalid names. If user enter an invalid name, print the following statement:

```
printf ("No match found for the given name. Try
again!\n");
```

if the user enters an already existing name you should give as a new order:

```
printf("An order with the given name already exists. Try
another name.\n");
```

Other printf statement you need to use to pass the autograder:

Debug mode:

```
printf( "Visiting node %s with values %d and %d\n", …);
```

Display group ahead:

```
printf("%s: %d burgers and %d salads \n", …);
```

Display the list:

```
printf("%s: %d burgers and %d salads, currently not in
the restaurant.\n ", …);
printf("%s: %d burgers and %d salads, currently in the
restaurant.\n", …);
```

if the r command cannot find a proper node to remove:

```
printf("No matching orders found.\n");
```

Use of C struct and C functions

When writing your code, you MUST create a C struct for the nodes in the linked list of the order list. These data items must include the following (and may include others if desired):

- the name of the order
- the integer variables specifying details of the order (number of burgers and salads)
- the in-restaurant status
- a pointer to the next node in the list

The pointer for the head of the linked list MUST be declared as a local variable in main() or some other function. **It may NOT be global.** If you wish to have the head of the list enclosed in a structure with some other information, that is OK (but certainly not required). The variable used to access this structure; however, may not be global. Each operation performed on the linked list MUST be done in its own function. These functions must take the head of the linked list as the FIRST parameter.

Linked List Operations/Functions

You must write C functions for the following 8 operations. These functions must be called when the specified commands are given as input.

addToList ( ) – This operation is to add a new node to the end of the linked list. This is to be used when the **a** and **c** commands are given as input.

doesNameExist ( ) – This operation is to return a Boolean value indicating whether a name already exists in the linked list. This is to be used when the **a, c, w, t** and **l** commands are given as input.

updateStatus ( ) – This operation is to change the in-restaurant status when a call-ahead order arrives at the restaurant. This operation will return a FALSE value if that order is already marked as being in the restaurant. This is to be used when the **w** command is given as input.

retrieveAndRemove ( ) – This operation is to find the first in-restaurant order that matches the order prepared for pick up at the counter. This operation is to return the name of group. This group is to be removed from the linked list. This is to be used when the **r** command is given as input.

countOrdersAhead ( ) – This operation is to return the number of orders waiting ahead of an order with a specific name. This is to be used when the **l** command is given as input.

displayWaitingTime( ) – This operation is to return the estimated waiting time for the specific name. The function will check the number of burgers and salads ordered ahead of the specified name and using known preparing time (5 minutes for burger and 2 minutes for salad) calculates the estimated wait time. This is to be used when **t** command is given as input.

displayOrdersAhead ( ) – This operation traverses down the list until a specific order name is encountered. As each node is traversed, print out that node's orders. This command is to be used when the **l** command is given.

displayListInformation ( ) – This operation to traverse down the entire list from beginning to end. As each node is traversed, print out that node's name, order details and in-restaurant status. This command is to be used when the **d** command is given as input.

Note that there may be a many-to-many relationship between the commands in the user interface and the required functions. For example, the l command ("list") relies on the following functions: doesNameExist(), countOrdersAhead() and displayOrdersAhead().

Command Line Argument: Debug Mode

Your program is to be able to take one optional command line argument, the -d flag. When this flag is given, your program is to run in "debug" mode. When in this mode, your program is to display each order's information as you traverse through the linked list of the order list. Note

that for the w, r, and t commands, the entire list may not be traversed, so you only display the part of the list that is needed to be traversed to complete the command.
List (l) and display(d)command are exceptions, no need to add debug mode as they already print the information on nodes you traverse.

When the flag is not given, this debugging information should not be displayed. One simple way to set up a "debugging" mode is to use a boolean variable which is set to true when debugging mode is turned on but false otherwise. This variable may be a global variable. Then using a simple if statement controls whether information should be output or not.

```
if ( debugMode == TRUE )
        printf (" Debugging Information \n");
```

Provided Code for the User Inteface

The code given in proj4base.c **should** properly provide for the user interface for this program including all command error checking. This program has no code for the linked list. It is your job to write the functions for the specified operations and make the appropriate calls. Most of the changes to the existing proj4base.c program need to be made in each of the **doXXXX ( )** functions. Look for the comments of:

```
// add code to perform this operation here
```

Note: the head of the linked list is required to be a local variable in main and you are required to pass the head of the linked to the operation functions. All of the **doXXXX ( )** functions currently have no parameters. It will then be expected that you will modify the function signatures of the doXXXX() functions to allow for this information to be passed as required.

MULTIPLE SOURCE CODE FILES

Your program is to be written using at least three source code files. It must also have a makefile and a header file to help with the compilation of the program. All of the storage structure code (the linked list code) is to be in one source code file. The code in proj4base.c is to be separated into two different source code files.

The following functions from proj4base.c are to be in one source code file (these are the user interface functions):

- main()
- clearToEoln()
- getNextNWSChar()
- getPosInt()
- getName()
- printCommands()

The following functions from proj4base.c are to be in another source code file (these are the functions that interact with the linked list functions):

- doAdd()
- doCallAhead()

- doWaiting()
- doRetrieve()
- doList()
- doDisplay()
- doEstimateTime()

The third source code file is to have the code that you are writing that will perform the linked list implementation: The functions in this source code file will include the following functions plus any other you write to handle the linked list:

- addToList()
- doesNameExist()
- updateStatus()
- retrieveAndRemove()
- countOrdersAhead()
- displayOrdersAhead()
- displayListInformation()
- displayWaitingTime()

If you add additional functions to your program, you can add those to whichever source code file seems appropriate for that function (or create a fourth source code file or even a fifth).
You must also create a header file. The job of the header file is to contain the information so the source code files can talk to each other. The header file (.h file) should contain the function prototypes and any struct and/or typedef statements. Please review the .h file in the example below.

The makefile MUST separately compile each source code file into a ".o" file and separately link the ".o" files together into an executable file. Review the makefile in the example below to see how this is done. The command to create the .o file is:

```
gcc –c program1.c
```

The command to link the files program1.o, program2.o and program3.o into an executable file is:

```
gcc program1.o program2.o program3.o
```

The above command will just name the executable file using the default name of a.out, most often the –o option is given to provide a specific name for the executable file.

```
gcc program1.o program2.o program3.o –o program.exe
```

Example of Multiple Source Code Files
Consider the program contained in the following files:

- max1.c
- max2.c
- max.h
- makefile

This example shows how to set up this simplest of multiple source code file program. Note that max1.c and max2.c just contain functions and a #include of max.h. The file max.h contains the prototypes (or forward declarations) for all of the functions that are called from outside its source code file and any "globally" needed information.

The makefile is a special file that helps in the compilation of the source code into the object files into the executable file. A makefile is executed by the use of the **make** command. The syntax of

a makefile can be strange. The makefile will contain multiple rules. Each rule has the following syntax:

```
target: dependencyList
        commandLine
```

The multiple rules in the make file are separated by a blank line. Also note **(this is VERY IMPORTANT)** the commandLine must use a TAB for its indentation! An example of a rule is:

```
max1.o: max1.c max.h
	gcc -c max1.c
```

The **commandLine** is **gcc -c max1.c**, which will compile the source code file of max1.c into the object code file of max1.o.

The **target** in the above example is the file **max1.o**, which is also the name of the file created when the commandLine is executed. This relationship is what causes makefiles to work.

The **dependencyList** in the above example is the two files: **max1.c** and **max.h**, which are the files needed for the commandLine to properly run. Again, this relationship is what causes makefiles to work.

The make command uses the timestamps of the files in the target and the dependencyList. If any file in the dependencyList has a more recent timestamp than the target file, the commandLine is executed. The idea is that if the user has recently changed either **max1.c** or **max.h**, then the object file **max1.o** needs to be re-compiled. Make is designed to help the programmer keep track of what needs to be compiled next.

Make and makefile tutorials can be found at:

- http://mrbook.org/tutorials/make/
- http://www.gnu.org/software/make/manual/make.html
- http://www.opussoftware.com/tutorial/TutMakefile.htm

Coding Style

Don't forget to use good coding style when writing your program. Good coding style makes your program easier to be read by other people as the compiler ignores these parts/differences in your code. Elements of good code style include (but may not be limited to):

- Meaningful variable names
- Use of functions/methods
- Proper indentation
- Use of blank lines between code sections
- In-line comments
- Function/method header comments
- File header comments

The Code Review Checklist also hints at other elements of good coding style.

Program Submission

You must zip all of your files needed for the program together and then submit the zip file for this program via the proper submission links on **Gradescope.**

In your makefile, please rename the final executable program to "Project4", e.g. ,
gcc –o Project4 program1.o program2.o program3.o

Create a directory called “Project4”, then put the 3 source code files, your header file and your makefile into this directory. Zip this directory and submit the zip file.

Note that the folder name and file name are case sensitive."