KAIST
EE 209: Programming Structures for EE

Assignment 1: Customer Management Program
Part 1 - Argument Validation

Purpose

The purpose of this assignment is to help you learn or review (1) the fundamentals of the C programming language, (2) the details of implementing command prompts in C, (3) how to use the GNU/Unix programming tools, especially `bash`, `emacs`, and `gcc209`.

Rules

Make sure you study the course *Policy* web page before doing this assignment or any of the EE 209 assignments. In particular, note that you may consult with the course instructors, lab TAs, KLMS, etc. while doing assignments, as prescribed by that web page. However, there is one exception...

Throughout the semester, each assignment will have an "on your own" part. You must do that part of the assignment *completely on your own*, without consulting with the course instructors, lab TAs, listserv, etc., except for clarification of requirements. You might think of the "on your own" part of each assignment as a small take-home exam.

For this assignment, "checking for a missing or malformed option parameter" (as described below) is the "on your own" part. That part is worth 10% of this assignment.

Background

A *command-line interface (CLI)* is a means of interacting with a program where the user issues text commands to the program.

A *command prompt* (or just *prompt*) is a string used in a CLI to inform and literally prompt the users to type commands. A prompt usually ends with one of the characters $, %, #, :, >

- A bash shell, embedded in many Unix systems, uses a prompt of the form:

  ```
  [time] user@host: work_dir $
  ```

- DOS's COMMAND.COM and the Windows's command-line interpreter cmd.exe use the prompt of the form:

  ```
  C:\>
  ```

  where 'C' represents the default main disk label in most modern systems.

A *command-line argument* or parameter is an item of information delivered to a program when it is started. In Unix and Unix-like environments, an example of a command-line argument is:

```
mkdir ee209
```

where "ee209" is a command-line argument which tells the program mkdir to create a new folder named "ee209".

The Task

In this assignment, your task is to write a simple CLI that validates user commmand for a customer management program as described below.

Customer Management Program

Throughout Assignment 1 and 3, you will develop a customer management program, which handles customer information and the operations on them. The requirements for the customer management program are:

- A client can register a new customer and store her information.
- A client can unregister a customer and remove her information.
- A client can search for a customer and retrieve her information.

The customer information to be managed includes:

- ID: the online ID of the customer
- Name: the name of the customer
- Purchase amount: the amount of money that the customer has purchased

In this assignment, you only have to validate the command-line input line-by-line, i.e. your program will just read each line and check whether it is a valid command. If the command is invalid, the CLI prints out an error message to the standard error(`stderr`) stream and waits for the next command. If the command is valid, it does nothing and waits for the next command. There is no dependency between the commands.

Customer Management Program CLI

We provide a sample skeleton code for the customer management program CLI. Your task in this assignment is to extend this code so that it supports all the command validation features explained below. You can use your own skeleton code if you want, but it needs to behave *exactly the same* as described below.

The basic features for the CLI are as below.

- The CLI prints out a prompt of a form:

  ```
  >
  ```

- If an entered command-line includes spaces and a new line character(`'\n'`) only, the program should print the next prompt and wait for the next command-line.

  ```
  >
  >
  ```

- The `exit` command should exit the program imediately. This command should not take any argument.

  ```
  > exit
  ```

  Note that `exit` command is already implemented in the skeleton code.

- The reg(register) command should take three *arguments* - an ID, a name (NAME) and a purchase amount (PURCHASE) - of a customer to register.

  A command-line has to specify the type before each argument. In our program, '-i', '-n' and '-p' should proceed before the actual content of ID, NAME and PURCHASE, respectively. Note that there can be any number of spaces (except the new line character) between a type specifier(-i, -n, -p) and an argument. We call the combination of a type specifier and an argument an *option*. The order of the options is unimportant, but the CLI should not take duplicate options in the same line.

  ```
  reg -i ID -n NAME -p PURCHASE
  ```

  The table below shows some examples of reg.

Standard Input Stream	Standard Error Stream
reg -i ch.hwang128 -n 'Changho Hwang' -p 123	
reg -n 'Sangwook Bae' -p 2090 -i baesangwook89	
reg -n 'YoungGyoun Moon' -i ygmoon -p 50492	

- The unreg (unregister) command should take either an ID *or* a name of a customer to unregister:

  ```
  unreg -i ID
  ```

  ```
  unreg -n NAME
  ```

  The table below shows some real examples of using unreg command.

Standard Input Stream	Standard Error Stream
unreg -i ch.hwang128	
unreg -n 'Sangwook Bae'	
unreg -i ygmoon	

- The find (search) command should take either an ID *or* a name of the customer to search. The argument validation process of find is exactly the same as that of unreg while their real operations are different.

  ```
  find -i ID
  ```

  ```
  find -n NAME
  ```

  The table below shows some examples of find.

Standard Input Stream	Standard Error Stream
find -n 'Changho Hwang'	
find -i baesangwook89	
find -n 'YoungGyoun Moon'	

- There should be at least one space character (any space character except a new line character like ' ', '\t', etc.) between the commands and options. Additional space characters are allowed at the beginning, at the end and between the commands and options. You can use isspace function to match a space character including a new line character. You may have to re-check whether a character is a new line character to handle it exceptionally. The table below shows examples of this feature.

Standard Input Stream	Standard Error Stream
`   reg -i ch.hwang128 -n 'Changho Hwang' -p 431`	
`     reg    -i   baesangwook89 -n  'Sangwook Bae'   -p 2855`	
` unreg  -i ygmoon`	
`  find -n    'YoungGyoun Moon'`	
reg-i ch.hwang128 -n 'Changho Hwang' -p 6523	ERROR: Undefined Command
reg -i baesangwook89 -n 'Sangwook Bae'-p 64	ERROR: Invalid Option Argument
unreg -iygmoon	ERROR: Undefined Option

  The error messages will be explained below.

- The CLI should handle EOF properly. If the program meets EOF, it means there is no more input to read, thus the program shouldn't require any more input and it should exit (by calling exit(0);) . Before exit, if the already-typed command is invalid, the program should print the corresponding error message and then exit immediately.

  A program meets EOF when it reaches the end of a file stream. In most cases, a CLI program will not meet EOF because if there is nothing to read in `stdin`, the program just waits until something is typed in. However, you can make the CLI meet EOF using `^D`(ctrl + d). Typing `^D` forces the program to read whatever is buffered at `stdin` immediately. If there is something typed already in `stdin`, the program will read it, but if there is nothing in `stdin`, `^D` forces the program to read EOF. You can test your program whether it handles EOF properly by using this feature.

  For example, if you type in the following:

  ```
  find -i abc
  ```

  and then type `^D` without enter ('\n'), the program will read 'find -i abc' and then will wait for the rest of the command. If you continue to type in some more, for example,

  ```
  def
  ```

  and then type in `^D`, the program will think that the entered command is 'find -i abcdef', and then will wait again for the rest of the command. However, if you type in `^D` one more time (still without an enter), the program will read EOF and exit immediately, because there is nothing typed additionaly in `stdin`. If the entered command was invalid, the program should print the corresponding error message before exit. For more detail, please check how the given solution binary works with `^D`.

The program has to handle any input errors correctly. The program should scan the input line from left to right, and when it encounters an error, it should print out an error message and stop processing the line. That is, it should stop at the first error it encounters and move on to the next input line.

- The first word in a line should be a valid command. The first word refers to the first occurence of a sequence of non-space characters. If an undefined command (anything other than `exit`, `reg`, `unreg` and `find`) is given, the program should print out an error message "`ERROR: Undefined Command`" to `stderr`. The following lines are example error cases:

Standard Input Stream	Standard Error Stream
`undefcmd`	`ERROR: Undefined Command`
`undefcmd -i ygmoon`	`ERROR: Undefined Command`
`undefcmd -u UNDEFOPT`	`ERROR: Undefined Command`
`find-i ch.hwang128`	`ERROR: Undefined Command`
`fin -i baesangwook89`	`ERROR: Undefined Command`
`finda -i ygmoon`	`ERROR: Undefined Command`

- If an ambiguous option is used, the program should print an error message "`ERROR: Ambiguous Argument`". In this program, the only case corresponding to this case is when both ID and NAME are given to find or unreg:

Standard Input Stream	Standard Error Stream
`find -i ch.hwang128 -n 'Changho Hwang'`	`ERROR: Ambiguous Argument`
`unreg -n 'Sangwook Bae' -i baesangwook89`	`ERROR: Ambiguous Argument`

- If an invalid option (or valid option in a wrong format) is provided, the program should print an error message "`ERROR: Undefined Option`". Here are the examples:

Standard Input Stream	Standard Error Stream
`exit -i`	`ERROR: Undefined Option`
`exit -i ygmoon`	`ERROR: Undefined Option`

`find -p 1234`	`ERROR: Undefined Option`
`unreg -p 54512`	`ERROR: Undefined Option`
`reg -u UNDEFOPT`	`ERROR: Undefined Option`
`reg -i ch.hwang128 -n 'Changho Hwang' -p`	`ERROR: Undefined Option`
`unreg -`	`ERROR: Undefined Option`
`unreg -n`	`ERROR: Undefined Option`
`reg -i baesangwook89 -n 'Sangwook Bae' -p 654 -u UNDEFOPT`	`ERROR: Undefined Option`
`reg baesangwook89`	`ERROR: Undefined Option`
`reg -n 'Sangwook Bae' baesangwook89`	`ERROR: Undefined Option`

- If the option of the same type is provided multiple times in a command line, the program should print out an error message, "`ERROR: Multiple Same Options`" regardless of whether the content of the repeated argument is identical or not. Here are some examples:

Standard Input Stream	Standard Error Stream
`find -i ch.hwang128 -i ch.hwang128`	`ERROR: Multiple Same Options`
`unreg -n 'YoungGyoun Moon' -n 'Changho Hwang'`	`ERROR: Multiple Same Options`
`reg -i baesangwook89 -p 9 -n 'Sangwook Bae' -p 432`	`ERROR: Multiple Same Options`

- If a command ends prematurely, the program should print out an error message "`ERROR: Need More Option`". Here are some examples:

Standard Input Stream	Standard Error Stream
`find`	`ERROR: Need More Option`
`unreg`	`ERROR: Need More Option`
`reg -i baesangwook89 -n 'Sangwook Bae'`	`ERROR: Need More Option`

  Note that this error case has the lowest priority among other error cases.

- Finally, if an argument doesn't follow the 'Argument Rules' described below, the program should print out an error message "`ERROR: Invalid Option Argument`". This will be explained precisely in the next section, 'Arugment Rules'.

Note these things:
1. The program should stop processing the current line when it encounters an error.
2. All error messages should go out to stderr.
3. Nothing is printed (either to stdout or stderr) if there's no error in the line.
4. The error messages should be exactly the same as described above.

Argument Rules

Your CLI have to handle three argument types, ID, name (NAME), and purchase amount (PURCHASE). Each argument should meet the following rules.

ID Rules:

- It consists of alphabets, digits, hyphens('-'), underscores('_') and periods('.').
- The maximum length is 63, and the minimum length is 1.
- The table below shows some real examples of an ID:

Valid ID	Invalid ID
`aaaa`	`abcd!`
`ABCD1234`	`ABCD#1234`
`a-bcd_123`	`hello@gmail.com`
`-n`	`í•œê¸€ì•„ì□´ë””`
`5.......abcd....`	`*^^*`

`---------------`	`Kyoungsoo Park`
`1234567890`	`'KyoungsooPark'`
`_________________`	
`.a-b.c_d-1.2-3_4.`	`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`

PURCHASE Rules:

- It consists of one or more digits. It should be a positive integer.
- A multi-digit value that starts with '0' is not allowed.
- The number of digits cannot exceed 10.
- The table below shows some real examples of a PURCHASE:

Valid PURCHASE	Invalid PURCHASE
`0`	`01`
`402`	`042`
`4002`	`0402`
`1234`	`-1234`
`9999999999`	`55555555555`

NAME Rules:

- It consists of alphabets, hyphens('-'), periods('.'), spaces(' ') or a single quote('''). All other characters can't be used for a name (e.g., backslash ('\') and double quote ('"') are not valid characters for a name)
- A name can be enclosed with a pair of single quote (''') characters (called enclosers). An enclosed name is useful to represent spaces in a name. An encloser is not a part of a name.
- A backslash character ('\') can be used as an escape character. While an escape character is useful to represent a space or a single quote character in a name, it can be used with any other valid character. It can be used either in an enclosed name or in a barefoot name (without enclosers). An escape character is not a part of a name.
- Note again that an escape character is used only for a charcter which is a part of a name, which means that it cannot be used for an encloser because an encloser is not a part of a name. Thus when a name starts with a backslash character followed by a single quote, the single quote is just a normal valid character, not an encloser.
- The length of a name should be between 1 and 63.

Valid NAME	NAME Length	Invalid NAME
`aaaa`	4	`aaaa!`
`ABCD`	4	`ABCD1234`
`a-bcd.asdf`	10	`a_bcd.asdf`
`-n`	2	`''`
`\'`	1	`'`
`'Kyoungsoo Park'`	14	`'Kyoungsoo Park`
`Kyoungsoo\ Park`	14	`Kyoungsoo Park`
`Kyoungsoo'Park`	14	`'Kyoungsoo'Park`
`'Kyoungsoo\'Park'`	14	`'Kyoungsoo'Park'`
`Kyoungsoo\-Park`	14	`\'Kyoungsoo Park\'`
`Kyoungsoo-Park`	14	`"Kyoungsoo Park"`
`\ \ Kyoungsoo\ Park\`	17	`Kyoungsoo\!Park`
`\'Kyoungsoo\ Park\'`	16	`"Kyoungsoo\ Park"`
`Kyoungsoo\Park`	13	`'Kyoungsoo Park\'`

Note that a poorly-specified NAME arugment can be interpreted as a different error in a command line. For example, if `Kyoungsoo Park` was given as a NAME argument, the program would consider `Kyoungsoo` as a valid name argument because `Kyoungsoo` itself meets all requirements of the NAME rule. After that, the program will read `Park` considering it as another option, which is undefined. Thus the program should print `ERROR: Undefined Option`.

Design

Design your CLI program to process each kinds of commands seperately with each other. The basic structure of the program is suggested in the skeleton code. Define your own functions properly so that a single function doesn't get too long. Try to reuse the defined functions as much as possible so that you don't have to write redundant code.

We suggest you to use a *deterministic finite state automaton* (*DFA*, alias *FSA*) to validate a NAME argument. The DFA concept is described in lectures, and in Section 7.3 of the book *Introduction to CS* (Sedgewick and Wayne).

Generally, a (large) C program should consist of of multiple source code files. For this assignment, you need not split your source code into multiple files. Instead you may place all source code in a single source code file. Subsequent assignments will ask you to write programs consisting of multiple source code files.

We suggest that your program use the standard C `getchar` function to read characters from the standard input stream. Actually, `fgets` function is generally used to implement option parsing, but we are using `getchar` here as we didn't learn `fgets` and some related concepts such as arrays yet.

Logistics

You should create your program on the lab machines cluster using `bash`, `emacs`, and `gcc209`.

Step 1: Design Overall Structure

Before you jump in to programming, make an overall plan how to design your program. Read carefully and understand the skeleton code so that you can check what is already done and what you have to do additionally.

Step 2: Create Source Code

Use `emacs` to create source code in a file named `client.c` that implements your CLI.

Step 3: Preprocess, Compile, Assemble, and Link

Use the `gcc209` command to preprocess, compile, assemble, and link your program. Perform each step individually, and examine the intermediate results to the extent possible.

Step 4: Execute

Execute your program multiple times on various input commands that test all logical paths through your code. You can also use test input files provided. Download `test.tar.gz` on a lab machine and extract the file by issuing the following command:

```
tar zxvf test.tar.gz
```

Then you will find test files in `test/` directory. All commands in each test file which were named an error case, `test_invalid_arg`, `test_undef_cmd`, `test_unclear_id`, `test_need_opt` and `test_undef_opt` should print the corresponding error message for that error case, and the other test files should not print any error messages. You can use these files by redirecting these files to your program. For example, if your program is named by `client`, you can test it with the following command:

```
./client < test_file_path
```

which the `test_file_path` should be replaced by the path to the test file you want to test with. Note that you can open and read these test files using `emacs` to check what commands are there.

We also provide the solution binary created by TAs. If there are any unclear part in this assignment, you can see what the solution binary does and compare it with yours.

Step 5: Create a `readme` File

Use `emacs` to create a text file named `readme` (not `readme.txt`, or `README`, or `Readme`, etc.) that contains:

- Your name, student ID, and the assignment number.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course Policy web page.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment: Did it help you to learn? What did it help you to learn? Do you have any suggestions for improvement? Etc.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Descriptions of your code should not be in the `readme` file. Instead they should be integrated into your code as comments.

Your `readme` file should be a plain text file. Don't create your `readme` file using Microsoft Word, Hangul (HWP) or any other word processor.

Step 6: Submit

Your submission should include your `client.c` file and your `readme` file.

Create a local directory named 'YourID_assign1' and place all your files in it. Then, `tar` your submission file by issuing the following command on a lab machine (assuming your ID is 20151234):

```
mkdir 20151234_assign1
mv client.c readme 20151234_assign1
tar zcf 20151234_assign1.tar.gz 20151234_assign1
```

Upload your submission file (20151234_assign1.tar.gz) to our KLMS assignment submission page. We do not accept e-mail submission (unless our course KLMS page is down).

Grading

We will grade your work on two kinds of quality: quality from *the user's* point of view, and quality from *the programmer's* point of view. To encourage good coding practices, we will deduct points if `gcc209` generates warning messages.

From the user's point of view, a program has quality if it behaves as it should. The correct behavior of your program is defined by the previous sections of this assignment specification.

From the programmer's point of view, a program has quality if it is well styled and thereby easy to maintain. In part, style is defined by the rules given in *The Practice of Programming* (Kernighan and Pike), as summarized by the Rules of Programming Style document. For this assignment we will pay particular attention to rules 1-24. These additional rules apply:

- **Names**: You should use a clear and consistent style for variable and function names. One example of such a style is to prefix each variable name with characters that indicate its type. For example, the prefix `c` might indicate that the variable is of type `char`, `i` might indicate `int`, `pc` might mean `char*`, `ui` might mean `unsigned int`, etc. But it is fine to use another style -- a style that does not include the type of a variable in its name -- as long as the result is a clear and readable program.
- **Comments**: Each source code file should begin with a comment that includes your name, the number of the assignment, and the name of the file.
- **Comments**: Each function -- especially the `main` function -- should begin with a comment that describes *what the function does* from the point of view of the caller. (The comment should not describe *how the function works*.) It should do so by *explicitly* referring to the function's parameters and return value. The comment also should state what, if anything, the function reads from the standard input stream or any other stream, and what, if anything, the function writes to the standard output stream, the standard error stream, or any other stream. Finally, the function's comment should state which global variables the function uses or affects. In short, a function's comments should describe the flow of data into and out of the function.
- **Function modularity**: Your program should not consist of one large `main` function. Instead your program should consist of multiple small functions, each of which performs a single well-defined task. For example, you might create one function to implement each argument validation of your CLI.
- **Line lengths**: Limit line lengths in your source code to 72 characters. Doing so allows us to print your work in two columns, thus saving paper.

Special Note

As prescribed by Kernighan and Pike style rule 25, generally you should avoid using global variables. Instead all communication of data into and out of a function should occur via the function's parameters and its return value. You should use ordinary *call-by-value* parameters to communicate data from a calling function to your function. You should use your function's return value to communicate data from your function back to its calling function. You should use *call-by-reference* parameters to communicate additional data from your function back to its calling function, or as bi-directional channels of communication.

However, call-by-reference involves using pointer variables, which we have not discussed yet. So for this assignment you may use global variables instead of call-by-reference parameters. (But we encourage you to use call-by-reference parameters.)

In short, you should use ordinary call-by-value function parameters and function return values in your program as appropriate. But you need not use call-by-reference parameters; instead you may use global variables. In subsequent assignments you should use global variables only when there is a good reason to do so.