資料結構與程式設計

(Data Structure and Programming)

107 學年上學期複選必修課程 901 31900

Homework #3 (Due: 9:00pm, Tuesday, Oct 23, 2018)

0. Objectives

1. Get used to C++ object oriented programming styles: class, data members, member functions, header files, etc.

2. Getting familiar with more C++ advanced features, for example, operator overload, inheritance, string, iostream, etc.

3. Learning to use Standard Template Library (STL).

4. Constructing a software project: multiple makefiles, source code directories, file dependency, libraries, etc.

5. Being able to comprehend existing code and enhance/complete it.

1. Problem Description

In this homework, we are going to design a more complete user interface (on top of Homework #2) for a simple command-line database system (somewhat similar to Homework #1). The generated executable is called "**mydb**" and has the following usage:

mydb [**-File** <*dofile*>]

where the **bold words** indicate the command name or required entries, square brackets "[ ]" indicate optional arguments, and angle brackets "< >" indicate required arguments. Do not type the square or angle brackets.

This simple command-line database system should provide the following functionalities:

1. Create DB by reading in data from a .json (JSON) file. Please refer to Homework #1 for the simplified JSON format. However, we further limit the format of the "key" to a string starting with either English letters (i.e. a-z, A-Z) or underline symbol (i.e. '_'). (See command "DBAPpend" for more detailed description)

2. Add a new JSON element to the end of the JSON object.
3. Print out the JSON object or a JSON element in the JSON object by specifying a “key” string.
4. Sort the JSON elements by either “key” or “value” in ascending order.
5. Report the number of JSON elements in the JSON object.
6. Compute the summation and average of the JSON elements by their “values”.
7. Report the maximum and minimum JSON element by comparing the “values”.

2. Supported Commands

The supported commands of “**mydb**” include:

```
DBAPpend:     append a JSON element (key-value
              pair(s)) to the end of DB
DBAVerage:    compute the average of the DB
DBCount:      report the number of JSON elements in
              the DB
DBMAx:        report the maximum JSON element
DBMIn:        report the minimum JSON element
DBPrint:      print the JSON element(s) in the DB
DBRead:       read data from .csv file
DBSOrt:       sort the JSON object by key or value
DBSUm:        compute the summation of the DB
DOfile:       execute the commands in the dofile
HELp:         print this help message
HIStory:      print command history
Quit:         quit the execution
```

The lexicographic notations in this section are summarized in the following table:

CAPITAL LETTERS or leading ‘-‘	The leading ‘-‘ and capital letters in the command name or parameters are mandatory entries and will be compared “case-insensitively”. The following letters can be partially skipped. However, when entered, they should match the specification “case-insensitively”. For example, for the command “DOfile” --- ● do (ok) ● dofile (ok)

	● DoF (ok) ● d (not ok; at least "do") ● dofill (not ok; not match) ● dofile1 (not ok; extra letter)
Round bracket "()"	Meaning it should be replaced by a proper argument as suggested by the "(type variable)" description in the round brackets. For example, the parameter in "**HIStory** [(int nPrint)]" should be replaced by an integer which is the number of histories to print.
Angle brackets "<>"	Mandatory parameters; they should appear in the same relative order as specified in the command usage.
Square brackets "[]"	Optional parameters; they can appear anywhere in the command parameters.
Dot dot dot "…"	Repeatable arguments; meaning the followed argument can be repeated multiple times.
Or '|'	Or condition; either one of the argument will do.

Please note that the "[ ]" optional parameters can appear anywhere in the command line, while the "< >" mandatory parameters must follow the relative order as specified in the command usage. For example, if the command "test" has the following usage ---

```
Usage: TEST <op1> <op2> [op3] [op4]
```

The following are legal:

```
> test op1 op2            // op3 or op4 can be omitted
> test op4 op1 op3 op2    // op3 op4 order is not enforced
> test op3 op1 op2
```

But the following are illegal:

```
> test op2 op1            // op1 op2 order is enforced
> test op1 op3 op4        // op1 and op2 are mandatory
```

For the following command, use "*cerr*" (instead of "*cout*") to print out the error message.

2.1 Command "DBAPpend"

Usage: **DBAPpend** <(string key)><(int value)>

Description: Append an JSON element (i.e. key-value pair) to the end of the DB. The options <(string key)><(int value)> must present. If any of the arguments is missing, print out an error. "key" is composed of key: {char} || {'_'};

use "isValidVarName(const string&)" in util/myString.cpp;

value: valid integer;

repeated keys are not allowed, report whether they have same value.

English letters (i.e. a-z, A-Z), digits (i.e. 0-9), and underline symbol (i.e. '_'). The first character of the "key" must be an English letter or underline symbol. Use "isValidVarName(const string&)" in "util/myString.cpp" to check the validity of the "key". If any error occurs, or if there already exists a JSON element with the same key (no matter the specified value is the same or not), print out an error. "value" must be a valid integer. Otherwise, print out an error. No message is printed on the success of insertion. Note that the added element affects the JSON object in your program only. It will not change (i.e. write back to) the read-in file.

Example:

```
mydb> DBAPpend Mary 30
mydb> DBAPpend KK 4 // Assume { "KK" : 4 } already exists
Error: Element with key "KK" already exists!!
mydb> dbAPpend
Error: Missing option!!
mydb> dbAPpend ric2k1
Error: Missing option!!
mydb> DBAPpend Mary ric2k1
// Any of the error message is OK
Error: Missing option after "Mary"!!
Error: Illegal option!! (ric2k1)
```

2.2 Command "DBAVerage"

Usage: **DBAVerage**

Description: Print out the average of the values in the DB. Use "fixed" and "setprecision(2)" for *cout* to control the output precision. If the DB is empty, report NAN (a float const number, #include <cmath>) as an error.

Example:

```
mydb> DBAVe
The average of the DB is 13.38.
mydb> DBAVe
Error: The average of the DB is nan.
```

2.3 Command "DBCount"

Usage: **DBCount**

Description: Print out the number of JSON element(s) in the DB.

Example:

```
mydb> DBCount
```

```
There are 57 JSON elements in DB.
mydb> DBCount
There is 1 JSON element in DB.
mydb> DBCount
There is no JSON element in DB.
```

2.4 Command "DBMAx"

Usage: **DBMAx**

Description: Print out the maximum JSON element (i.e. with the maximum value) in the DB. If there are multiple elements with the same maximum value, print out either one (preferably, the first one encountered). If the DB is empty, report NAN (a float const number, #include <cmath>) as an error.

Example:

```
mydb> DBMAx
The max JSON element is { "Mary" : 100 }.
mydb> DBMAx
Error: The max JSON element is nan.
```

2.5 Command "DBMIn"

Usage: **DBMIn**

Description: Print out the minimum JSON element (i.e. with the minimum value) in the DB. If there are multiple elements with the same minimum value, print out either one (preferably, the first one encountered). If the DB is empty, report NAN (a float const number, #include <cmath>) as an error.

Example:

```
mydb> DBMIn
The min JSON element is { "John" : -87 }.
mydb> DBMIn
Error: The min JSON element is nan.
```

2.6 Command "DBPrint"

Usage: **DBPrint** [(string key)]

Description: Print out the JSON DB. If the option "key" is NOT specified, print out the entire JSON object. On the other hand, if the option "key" is specified, print out the JSON element with the the "key" (i.e. performing case-sensitive comparison). If not found, print out an error. The printing format is as shown in the example.

Example:

```
mydb> DBPrint
{
  "Ric" : 100,
  "John" : 50
}
Total JSON elements: 2
mydb> DBPrint Ric
{ "Ric" : 100 }
mydb> DBPrint ric
Error: No JSON element with key "ric" is found.
```

2.7 Command "DBRead"

Usage: **DBRead** <(string jsonFile)> [-Replace]

Description: Read the data from "jsonFile" to the database (DB). If file "jsonFile" doesn't exist, print out an error message. If the DB already exists and the option "-Replace" is not specified, issue an error. If the option "-Replace" is given, resplace the DB content with the data in the "jsonFile".

Example:

```
mydb> DBRead test1.json  // assume test1.json doesn't exist
Error: "test1.json" does not exist!!
mydb> DBRead test2.json  // assume DB already exists
Error: DB exists. Use "-Replace" option for replacement.
mydb> DBRead test3.json -rep
DB is replaced...
"test3.json" was read in successfully.
```

2.8 Command "DBSOrt"

Usage: **DBSOrt** <-Key | -Value>

Description: Sort the JSON elements with key (-Key) or value (-Value) in ascending order. No output message is needed for this command, even if the JSON object is empty.

Example:

```
mydb> DBSOrt
Error: Missing option!!
mydb> DBSOrt -Key
mydb> DBSOrt -Value
```

2.9 Command “DBSUm”

Usage: **DBSUm**

Description: Print out the summation of the values of the JSON elements in the DB. If the DB is empty, report NAN (a float const number, #include <cmath>) as an error.

Example:

```
mydb> DBSUm
The sum of the DB is 2880.
mydb> DBSUm
Error: The sum of the DB is nan.
```

2.10 Command “DOfile”

Usage: **DOfile** <(string filename)>

Description: Execute the commands in the dofile. After the execution, it should go back to the command prompt.

Example:

```
mydb> dofile dofile1
```

2.11 Command “HELp”

Usage: **HELp** [(string cmd)]

Description: Print out help message. If command is specified, print out its usage. Otherwise, print out the list of all commands with simple descriptions.

Examples:

```
mydb> help
mydb> help dofile
mydb> help do
```

2.12 Command “HIStory”

Usage: **HIStory** [(int nPrint)]

Description: Print command history. The argument specifies the upper bound of how many of the last command history entries it will print. If not specified, all the histories will be printed.

Example:

```
mydb> history 8
```

2.13 Command “Quit”

Usage: **Quit** [**-Force**]

Description: Quit the execution. Prompt a confirmation if the argument "-Force" is not present.

Examples:

```
mydb> quit
mydb> q -f
```

3. Implementation

3.1 File/Directory Structure

After decompressing the .tgz file, you should see the following files and directories:

```
hw3> ls
bin/  dofiles/  include/  lib/  Makefile  ref/  src/ tests/
```

library file:
stores the implementations of a header file.

"bin/" and "lib/" are the directories to store the binary (executable) and library files, respectively. The directory "include/" contains the symbolic links of the header files (.h) to be shared within different source code packages. "Makefile" is the top-level makefile. **You only need to type "make" in this root directory** and it will go to different source code directories to invoke other makefiles, check the file dependency, compile the source codes, create libraries and final executable, and return. "dofiles/" contains some dofiles for you to test, and "ref/" includes the reference executables for linux and mac platforms. Please play with them to understand the spec of the commands in this homework. "tests/" directory contains some .json files for you to test.

The "src/" contains the source codes of different packages, each defined in a sub-directory. In this homework, the packages under "src/" include:

```
hw3> ls src
cmd/  db/  main/  Makefile.in  Makefile.lib  test/  util/
```

The "main/" directory, as its name suggests, contains the main() function of the entire program. "cmd/" implements the utilities of the command interface. It also defines some common commands such as "help", "quit", "history", etc. The "db/" directory is for the simple command-line database manager. The common utilities, such as customized string functions, memory management, container classes, etc, should be placed under the "util/" directory. You should try to take advantages of these common utilities functions.

The "test/" directory is to test your "db/" implementation before completing the command interface. Please see Section 4 "What you should do?" for further guidance.

3.2 Class description

1. **Classes about command registration**: `class CmdParser`, `class CmdExec` and its derived classes

   In this program, commands in different packages (i.e. different source code directories) are "registered" through the `CmdParser` command manager. There is one global variable *cmdMgr* and commands are added through its *regCmd*() member function. For example, in file "cmdCommon.cpp":

class CmdParser:
manages the following:
1. command line interface implemented.
2. use std::map to manage registered commands for future uses.

```
bool
initCommonCmd()
{
  if (!(cmdMgr->regCmd("Quit", 1, new QuitCmd) &&
        cmdMgr->regCmd("HIStory",3,new HistoryCmd)&&
        cmdMgr->regCmd("HELp", 3, new HelpCmd) &&
        cmdMgr->regCmd("DOfile", 2, new DofileCmd)
     )) {
     cerr << "Registering \"init\" commands fails..."
          << " exiting" << endl;
     return false;
   }
   return true;
}
```

   Four commands (quit, history, help, dofile) are registered to the *cmdMgr*. The first parameter of the *CmdParser::regCmd*() function specifies the name of the command. Please note that the leading capital characters (e.g. `HIS` in `HIStory`) are mandatory matching. They are made capital for conventional reason. The second parameter specifies the number of the mandatory matching characters. The last parameter is a functional object that inherits the `class CmdExec`.

   The `class CmdExec` is the common command registration and execution interface. To create a new command, you need to declare a derived class such as `class QuitCmd` which defines at least the following three member functions: (1) *exec*(): parse the command option(s) and execute the command, (2) *usage*(): print out the command usage, and (3) *help*(): print out the command definition for the `HELp` command. For more details, please refer to functions *CmdParser::regCmd*(), *CmdParser::execOneCmd*() in file "cmdParser.cpp", and *exec*/*usage*/*help*() members functions of each derived class such as in file "cmdCommon.{h,cpp}".

   For the sake of convenience, we define a MACRO *CmdClass(T)* in the file "cmdParser.h" so that we can easily declare an inherited class of `CmdExec` as:

   CmdClass(HelpCmd);

Please refer to the file "cmdCommon.cpp" for more examples.

2. **Classes about keyboard mapping**: `class CmdParser` and `enum ParseChar`

   The `class CmdParser` defines the functions to process inputs from the standard (cin) and file inputs, and the `enum ParseChar` is to define the keyboard mapping. Please note that the grading of this homework will not include special keys such as "delete", "backspace" and arrow keys, etc. So you can actually ignore them. (i.e. Don't worry about the keyboard mapping) We will focus on testing the command registration and database's functionalities. In fact, in "src/Makefile.in" we actually define the flag "`TA_KB_SETTING`" in the macro `CFLAGS` and thus we will use our keyboard mapping by default. However, if you want to customize your keyboard mapping, please change the "`#ifndef`" part of the "`#ifndef TA_KB_SETTING`" in files "cmdParser.h", "cmdCharDef.cpp" and undefine "`TA_KB_SETTING`" in the macro `CFLAGS` of "src/Makefile.in".

3. **Classes about database manager**: The classes and member functions for JSON element and object are defined in files "dbJson.{h,cpp}". The `class DBJson` defines all the interfaces to manipulate the JSON object, and its data member "`vector<DBJsonElem> _obj`" stores the JSON elements. The `class DBJsonElem` represents the JSON element, and the `struct DBSortKey` and `DBSortValue` are used as `StrictWeakOrdering` functional objects for the STL *sort()* algorithm.

3.3 How is a command string stored in _cmdMap?

shall be the reason we need inheritance and polymorphism...

When a command is registered in the `CmdParser::cmdReg()` function, the command string is partitioned into two parts: the former mandatory part (e.g. "`HEL`" in "`HELp`" command) will be converted to all-capital and used as the key to store the command in `CmdParser::_cmdMap`. Note that the characters are made all capital in order to facilitate the case-insensitive comparison. The second template argument of "`map<const string, CmdExec*> CmdParser::_cmdMap`" is an inherited pointer object of class `CmdExec`. For example, for the command "`HELp`", a pointer object of the inherited class `HelpCmd` will be created and stored.

The latter optional part of the command string (e.g. "p" in "HELp" command) will be stored as a private data member "`string _optCmd`" of the corresponding class object. It will be checked when parsing the command line input.

i.e. in its "exec()" method.

3.4 Makefile

There are 5 types of makefiles:

1. Top-level makefile: for the entire program creation

2. “make.pkg” in each of the source code directories: calling “Makefile.in” and “Makefile.lib” to construct library for each source code package.

3. Makefile.in: common core for the makefiles in different source code directories --- (i) define the compilation rules, (ii) create file dependency, (iii) create symbolic links for the external header files from the source code directory to the “include” directory.

4. Makefile.lib: makefile to create libraries.

5. “make.main” in the “main” source code directory: to perform linking and create the final executable.

Before making the program, you are suggested to type “make linux18”, “make linux16” or “make mac” to configure the provided object file “cmdRead.o” for your environment. Type “make” for top-level Makefile to create the executable. Use “make clean” to remove all the objective files, libraries, etc.

3.5 Useful utility functions

Please pay attention that there are many prewritten utility functions that you can take advantage of for your TODOs. For example, in `class CmdExec`, *lexNoOption()*, *lexSingleOption()* and *lexOptions()* can parse the command option into tokens. In file *util/myString.cpp*, the function *myStrNCmp(const string& s1, const string& s2, unsigned n)* performs case insensitive comparison between s1 and s2 for the first n characters, and check the compatibility for the rest. The function *myStr2Int(const string& str, int& num)* can convert the string “*str*” to integer “*num*”, and the function *isValidVarName(const string& str)* can check if the parameter “*str*” is a valid key for a JSON element.

3.6 Advanced Feature: “Tab” support

When the “tab” key is pressed, all the partially matched commands will be listed. Depending on the cursor position, there can be several possible responses:

1. If nothing but space characters is before the cursor, pressing “tab” key will list all the commands.

   [Example]

   // Before pressing “tab”

   ```
   mydb> ▯
   ```

   // After pressing “tab”

   ```
   DOfile       HELp         HIStory      GNADD        GNCOMPare
   GNMULTiply   GNPrint      GNSET        GNSUBtract   GNVARiable
   mydb> ▯
   ```

Note that each command above is printed by:

```
   cout << setw(12) << left << cmd;  // cmd is a string
```

And a new line is printed for every 5 commands. After printing, you should re-print the prompt and place the cursor back to its original location (including space characters).

2. If only partial command is matched, pressing "tab" should list all the possible matched commands. (multiple matches)

   [Example]

   // Before pressing "tab"

   ```
   mydb> h
   ```

   // After pressing "tab"

   ```
   HELp        HIStory
   mydb> h
   ```

3. But if there is only one possible match, pressing tab should complete the command. A space character will also be inserted after the command to separate it from the trailing substring. The newly inserted characters should match the strings stored in `CmdParser::_cmdMap` and in "`string _optCmd`" of the corresponding inherited class object.

   [Example]

   // Before pressing "tab"

   ```
   mydb> hello world
   ```

   // After pressing "tab"

   ```
   mydb> heLp llo world
   ```

4. If no command can be matched, pressing "tab" will make a beep sound and the cursor will stay in the same location.

   [Example]

   // Before pressing "tab"

   ```
   mydb> hello world
   ```

   // After pressing "tab"

   ```
   mydb> hello world
   ```

5. If the string before the cursor has already matched a command, and if there is at least one space characters before the cursor, pressing "tab" *for the first time* will print out its command usage.

   [Example]

   // Before pressing "tab"

   ```
   mydb> hel lo world
   ```

// After pressing “tab”

```
Usage: HELp [(string cmd)]
mydb> hel lo world
```

After printing, the cursor should remain in the original location.

6. (Continued from 5) If the string before the cursor has already matched a command, and if there is at least one space characters before the cursor, pressing “tab” *for the second time and onwards* will list the file names in the current directory (Please refer to the function “*listDir()*” in “*util/util.cpp*”). Note that each command above is printed by:

   ```
      cout << setw(16) << left << fileN;  // fileN is a string
   ```

   And a new line is printed for every 5 commands.

   Several possible cases as follow:

   (6.1) If the character before the cursor is a space ‘ ‘, and…

   [Example]

   // Before pressing “tab”

   ```
   mydb> hel world
   ```

   (6.1.1) in this directory there are multiple files and they do not have a common prefix,

   // After pressing “tab” --- print out ALL the file names under current directory.

   ```
   Homework_3.docx Homework_3.pdf  Makefile        MustExist.txt   MustRemove.txt
   bin             dofiles         include         lib             mydb
   ref             src             testdb          tests
   mydb> hel world
   ```

   (6.1.2) in this directory there are multiple files and all of them have a common prefix,

   // After pressing “tab” --- auto insert the common prefix and make a beep sound ==> DO NOT print the matched files

   // (e.g.) Try this in “ref” directory

   ```
   mydb> hel mydb-world
   ```

   (6.1.3) only one file in the current directory

   // After pressing “tab” --- print out the single file name followed by a ' '

   // (e.g.) Try this in “bin” directory

   ```
   mydb> hel mydb 
   ```

   (6.2) If the character before the cursor is NOT a space ‘ ‘, treat the substring before the cursor as a “prefix”. If there are multiple files under current directory that match the prefix, print out ALL the file names that match the prefix.

[Example]

// Before pressing “tab”

```
mydb> hel McDonald
```

// After pressing “tab”

```
Makefile        MustExist.txt   MustRemove.txt
mydb> hel McDonald
```

(6.3) However, if in (6.2) the matched file names have a common prefix, automatically insert the common prefix to the command line and make a beep sound. DO NOT print out the matched files.

[Example]

// Before pressing “tab”

```
mydb> hel MucDonald
```

// After pressing “tab”

```
mydb> hel MustcDonald
```

(6.4) In (6.2), if there is only ONE matched file, insert the remaining of the matched file name followed by a space ‘ ‘.

[Example]

// Before pressing “tab”

```
mydb> hel MustEcDonald
```

// After pressing “tab”

```
mydb> hel MustExist.txt cDonald
```

(6.5) If there is NO matched file for the prefix, make a beep sound and leave the cursor in the original position.

[Example]

// Before pressing “tab”

```
mydb> hel Yellow
```

// After pressing “tab”

```
mydb> hel Yellow
```

7. If the first word is not a match of a single command, and the cursor is not on the first word, pressing “tab” should make a beep sound and the cursor will stay in the same location.

   [Example]

   // Before pressing “tab”

   ```
   mydb> he llo world
   ```

   // After pressing “tab”

```
mydb> he llo world
```

Please note that this is an advanced feature. Do this only if you have completed all the other TODO's.

3.7 Adding new source code directory (not required in this homework)

1. Under "src" directory, create a new subdirectory. Name the directory properly as the package name.

2. In the top-level makefile, add the package name (usually equal to the directory name) to the "LIBPKGS" variable.

3. In the new package directory, copy the "make.xxx" from other source code directory. Remove the assignment on the "EXTHDRS" variable if any. Add in header file name to the "EXTHDRS" later if you intend to share that header file with other packages.

4. What should you do?

You are encouraged to follow the steps below for this homework assignment:

1. Read the specification carefully and make sure you understand the requirements.

2. Think first how you are going to write the program, supposed you don't have the reference code…

3. Study the provided source code. Please be advised that the number of lines of the reference code is 2097. If you have never handled a software program in such a scale before, please read it "smartly". You may want to first figure out the layout of files and directories, major data structure (i.e. classes), and how the functions are called starting from "*main*()". Please don't dig into detailed implementation in the beginning. Try to "guess" the meaning of the functions and variables, and have a "global" view of the program first. You can also use "*ctags*" to trace the codes. For mode information about "*ctags*", please refer to the third tip in Section 5.

4. What you should do in this homework assignment are commented with "**TODO**"'s. You should be able to complete this assignment by just finishing these todo's. Roughly speaking, they contain 6 parts:

   (i) Complete the `DJson` and `DBJsonElem` classes (in DBJson.h and DBJson.cpp). Their functionalities are quite similar to Homework #1, but with some differences. Please read the descriptions carefully in Sections 1 and 2.

(ii) You can test your `DBJson` and `DBJsonElem` implementation with the "test/" directory. Please refer to the *main()* function in *test.cpp* to add more testing codes. Simply type "*make test*" in the homework root directory to generate the test program "*testdb*". Although *testdb* will not be included in the homework grading, you are encouraged to test more on your `DBJson` implementation before moving on to command interface.

(iii) Finish the command interface in "cmdParser.cpp". You need to know how to use STL "*string*", "*map*" and "*vector*".

(iv) Implement the commands for "*db*" package (in dbCmd.cpp). You need to analyze the command line to see if there is any syntax error. Please note that there are several useful "string/char*" functions in files "util/myString.cpp" and "cmd/cmdParser.cpp". Use them whenever applicable. In addition, you need to call the appropriate `DBJson` member functions for the DB manipulations.

(v) Enhance the command "DOFile". Please refer to the "TODO" in the source code "cmdCommon.cpp" for the supported features. You may need to add or modify member functions or data members of class `CmdParser`. Please refer to the fourth and fifth tips in Section 5.

(vi) Implement the "tab" function (i.e. `CmdParser::listCmd()` in "*cmdParser.cpp*").

5. Complete your coding and compile it by "*make*". Test your program frequently and thoroughly. Please note that we provide the complete code for the command line parser so that you don't need to worry about the correctness and completeness of your Homework #2. However, we only provide the object file (i.e. cmdReader.o) so that it can be used for future homework assignment. Please note that the object file is platform dependent. Different platforms may require different compilations of object files. We provide three versions of cmdReader.o: (1) **cmdReader.o.linux18** for Ubuntu 18 linux machine, (2) **cmdReader.o.linux16** for Ubuntu 16 linux machine, and (3) **cmdReader.o.mac** for MAC. The file "cmdReader.o" is actually a symbolic link to one of them. The default is "cmdReader.o.linux18". Please type "*make linux18*", "*make linux16*" or "*make mac*" to switch between different platforms.

6. Reference programs **mydb-linux18 / mydb-linux16 / mydb-mac** (for the simple command-line JSON object manipulations) are available under the "ref/" directory. Please use them to compare your result. Please also watch out the announcements in the Ceiba website and FB group.

5. Some tips you should know

1. The provided reference code can be compiled even though the TODOs are not done. However, the produced executable cannot run (i.e. will crash). Please check the TODO's and implement some of them first.

2. Sometimes you may encounter compilation error message like:

   make[1]: *** No rule to make target `../../include/util.h', needed by `cmdCommon.o'. Stop.

   This is mainly because the hidden file ".extheader.mak" in some directory is accidentally removed. You can try to "make clean" and "make" again and usually it will resolve the problem.

3. Type "make ctags" to create ctages for all the source codes. Be sure to add in the following line in your "$HOME/.vimrc" (if you don't have this file, create one):

   set tags=./tags,../tags

   Then when you use "*vim*" to edit the source code, you can jump to the function/class definition of the identifier your cursor is currently on by pressing "ctrl-]". To come back, simply press "ctrl-t".

4. The function *closeDofile()* is a TODO. However, how it is called is not included in the reference code. Here is the partial code of the function *readCmd()* in *cmdReader.cpp* . You can see how *closeDofile()* is called.

```
bool
CmdParser::readCmd(istream& istr)
{
   resetBufAndPrintPrompt();

   bool newCmd = false;
   while (!newCmd) {
      ParseChar pch = getChar(istr);
      if (pch == INPUT_END_KEY) {
         if (_dofile != 0)
            closeDofile();
         break;
      }
      switch(ch) {
         ... // Refer to the codes in homework #2
      }
   }
   return newCmd;
}
```

5. The handling of "`ifstream* _dofile`" for the "`openDofile()`" and "`closeDofile()`" may be trickier than you think. For example, if you need to open a dofile (i.e. the `DOfile` command) in a dofile, you need to store the original dofile and when the new dofile is finished, retrieve it and continue the execution from where you left. However, please note that you CANNOT "copy" fstream object. That's why we declare `_dofile` as a pointer.

6. In "cmdReader.o", there is a function "`CmdParser::reprintCmd()`" called by "`CmdParser::listCommand()`", which is for the "tab" feature.

Although you don't have the cmdReader.cpp source code, you are free to call the function `reprintCmd()`:

```
// Reprint the current command to a newline
//  cursor  should  be  restored  to  the  original
location
void
CmdParser::reprintCmd()
{
   cout << endl;
   char *tmp = _readBufPtr;
   _readBufPtr = _readBufEnd;
   printPrompt(); cout << _readBuf;
   moveBufPtr(tmp);
}
```

7. When you use output directing operator ">" to store the output of your program to a file, please note that only "standard output" is directed. The error message (i.e. "standard error" *cerr*) is not included. For "csh/tcsh", you need to use ">&" instead. For bash, you can try "&>" or something like:

   "./mydb-ref -File dofile.ref > out.mine 2>&1"   or

   "./mydb-ref -File dofile.ref 2>&1 | tee out.mine"

6. Grading

We will test your submitted program with various combinations/sequences of commands to determine your grade. The results (i.e. outputs) will be compared with our reference program. Minor difference due to printing alignment, spacing, etc can be tolerated. However, to assist TAs for easier grading work, *please try to match your output with ours.*