資料結構與程式設計

(Data Structure and Programming)

101 學年上學期複選必修課程 901 31900

Homework #3 (Due: 11:00pm, Wed, Nov 7, 2012)

Department:___________________________ Grade:__________

Id:_______________________ Name:________________________

0. Objectives

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

2. Learning to use Standard Template Library (STL)

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

4. 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 calculator under the modular number system. The generated executable is called "**modCalc**" and has the following usage:

modCalc [**-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 command-line calculator should provide the following functionalities:

1. Define modulus *m*: specify that all the variables are integers under the modulo-*m* number system. In other words, all the numbers are within [0, *m* – 1] and when overfloat occurs, they should “wrap” around the modulus *m* (by the modulo ‘%’ operator).
2. Set variables: define variables with names and initial values.
3. Perform arithmetic operations: support ‘+’, ‘-‘, ‘*’, and compare ‘==’.
4. Examine variables: print out the variable names and values.

A modular number system is a number system represented best by numbers on a circle because the numbers repeat. The numbers on a clock face are an example. None of the arithmetic operations on the numbers of this system can produce results that are not in the system. For example, in a modulo-7 number system, 3 + 4 = 0, 5 + 5 = 3, 3 - 5 = 5, 3 * 4 = 5.

In this homework, you don’t need to handle “division /”, “mod %”, “shift <<” operators, etc.

2. Supported Commands

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

`DOfile:`	`execute the commands in the dofile`
`HELp:`	`print this help message`
`HIStory:`	`print command history`
`MADD:`	`perform modular number addition`
`MCOMPare:`	`compare if two variables or values are equal`
`MMULTiply:`	`perform modular number multiplication`
`MPrint:`	`print the variables of the modular number calculator`
`MSET:`	`set the modulus of the modular number calculator`
`MSUBtract:`	`perform modular number subtraction`
`MVARiable:`	`set the variable value of the modular number calculator`
`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 forced
> test op3 op1 op2
```

But the following are illegal:

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

2.1 Command “DOfile”

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

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

Example:

mcalc> dofile dofile1

2.2 Command “HELp”

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

Description: print this help message. If no command is specified, print out the list of all commands with simple descriptions.

Examples:

mcalc> help

mcalc> help dofile

mcalc> help do

2.3 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:

mcalc> history 8

2.4 Command “MADD”

Usage: **MADD** <(string y)> <(string a) | (int va)> <(string b) | (int vb)>

Description: perform modular number addition. The summation of the second and third arguments will be stored in the corresponding variable of the first argument. Therefore, the first argument must be a variable name. It can be preexistent or not in the variable map. If preexistent, the previous value will be overwritten. Otherwise, a new variable will be created. The second and third arguments can be variable names or integral values. If they are variable names, they must preexist in the variable map.

Examples:

```
mcalc> madd a b c        // b, c must preexist
mcalc> madd a 8 d
mcalc> madd a 7 -8
```

2.5 Command “MCOMPare”

Usage: **MCOMPare** <(string var1) | (int val1)> <(string var2) | (int val2)>

Description: compare if two variables or values are equal under modular number system. If variable names are specified, they must preexist in the variable map.

Examples:

```
mcalc> mcompare a b
mcalc> mcomp a 7
mcalc> mcomp 7 8
```

2.6 Command “MMULTiply”

Usage: **MMULTiply** <(string y)> <(string a) | (int va)> <(string b) | (int vb)>

Description: perform modular number multiplication. See command “MADD” for description on the arguments.

Example:

```
mcalc> mmult y a b
```

2.7 Command “MPrint”

Usage: **MPrint** [(string var)...]

Description: print the variables of the modular number calculator.

Examples:

```
mcalc> mprint
mcalc> mprint a
mcalc> mprint a b c
```

2.8 Command “MSET”

Usage: **MSET** <(int modulus)>

Description: set the modulus of the modular number calculator. The default modulus is 100,000,000. Note that when a new modulus is set, if it is different from the previous setting, all the variables in the variable map will be invalidated (i.e. the map will be clear()).

Example:

```
mcalc> mset 16
```

2.9 Command "MSUBtract"

Usage: **MSUBtract** <(string y)> <(string a) | (int va)> <(string b) | (int vb)>

Description: perform modular number subtraction. See command "MADD" for description on the arguments.

Example:

```
mcalc> msub y 10 b
```

2.10 Command "MVARiable"

Usage: **MVARiable** <(string var)> <(string var) | (int val)>

Description: set the variable value of the modular number calculator. The second argument can be variable name or integral value. However, if variable name is entered for the second argument, the corresponding variable must preexist.

Examples:

```
mcalc> mvar a 10
mcalc> mvar a b
```

2.11 Command "Quit"

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

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

Examples:

```
mcalc> quit
mcalc> 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/
```

“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 64 and 32-bit platforms. Please play with them to understand the spec of the commands in this homework.

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
calc/  cmd/  main/  Makefile.in  Makefile.lib  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 “calc/” directory is for the command-line calculator. 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.

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”:

   ```
   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 actually can ignore them. (i.e. Don't worry about the keyboard mapping) We will focus on testing the command registration and modular calculator's functionality. 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 modular calculator**: `class ModNum`, `class MsetCmd`, etc.

   The `class ModNum` serves as a wrapper class for modular numbers. It has only one non-static data member "`int _num`" and overloads several arithmetic operators (e.g. +, +=, *...) so that its instantiated objects can be operated as regular data type (e.g. `y = a + b`). The other two data members in `class ModNum`, "`int _modulus`" and "`CalcMap _varMap`", are static. They are used to define the modulus and store the variables for modular calculation, respectively.

   The classes in file "calcCmd.h" are derived classes of `CmdExec`. They are used to define the modular calculator commands.

3.3 How is a command string stored in _cmdMap?

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 `CmdParser::_cmdMap` is an inherited pointer object of class `CmdExec`. For example, for the command "`HELp`", a pointer 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 in the private data member "`string _optCmd`" of the corresponding class object.

3.4 Makefile

There are 5 types of makefiles:

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

2. 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.

3. Makefile.lib: makefile to create libraries.

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

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

Before making the program, you are suggested to type “make 32” or “make 64” 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. A test program “testMC” can be created by typing “make test”. You can modify the file “test.cpp” in “src/test” to test your “`class ModNum`” before implementing the modular calculator commands.

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`, *lexSingleOption()* and *lexOptions()* can parse the command option into tokens. In file *util/myString.cpp*, the function *isValidVarName(const string& str)* can check if the parameter “*str*” is a valid variable name.

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”

   ```
   mcalc>   ▯
   ```

   // After pressing “tab”

   ```
   DOfile      HELp        HIStory     MADD        MCOMPare
   MMULTiply   MPrint      MSET        MSUBtract   MVARiable
   ```

```
mcalc>    ▯
```

Note that each command above is printed by:

```
   cout << setw(12) << left << cmd;
```

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”

   ```
   mcalc> h▯
   ```

   // After pressing “tab”

   ```
   HELp        HIStory
   mcalc> h▯
   ```

   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”

   ```
   mcalc> hello world
   ```

   // After pressing “tab”

   ```
   mcalc> heLp llo world
   ```

3. 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”

   ```
   mcalc> hello world
   ```

   // After pressing “tab”

   ```
   mcalc> hello world
   ```

4. 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” will print out its command usage.

   [Example]

   // Before pressing “tab”

   ```
   mcalc> hel lo world
   ```

   // After pressing “tab”

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

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

5. 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”

   ```
   mcalc> he llo world
   ```

   // After pressing “tab”

   ```
   mcalc> 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 “Makefile” from other source code directory. Remove the assignment on the “EXTHDRS” variable if any. Add in header file name to the “EXTHDRS” later if that header file is intended to be shared 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 1804. 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 4 parts: (i) Finish the command interface in "cmdParser.cpp". You need to know how to use STL "*string*", "*map*" and "*vector*". (ii) Complete the "`ModNum`" class (in calcModNum.h and calcModNum.cpp). You need to implement operator overloads for various arithmetic operations. (iii) Implement the commands for the "*calc*" package (in calcCmd.cpp). You need to understand how to use STL "*map*" to record the variable names of "`ModNum`" type and their corresponding values. You also 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 `ModNum` functions for the calculated result. (iv) 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.

5. Before you finish the command interface in (4) above, you can actually first test your overload operators in class `ModNum` by the test program in "src/test". Simply type "*make test*" and a test program "*testMC*" will be generated. Please note that "*testMC*" will be part of the grading of this homework.

6. 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 two versions of cmdReader.o: (1) **cmdReader-32.o** for 32-bit machine, and (2) **cmdReader-64.o** for 64-bit. The file "cmdReader.o" is actually a symbolic link to one of them. The default is "cmdReader-64.o". Please type "*make 32*" or "*make 64*" to switch between 32 and 64-bit platforms. You can use "uname -a" to figure out the type of your platform.

7. Reference programs **modCalc-64 / modCalc-32** (for the command-line modular calculator) and **testMC-64 / testMC-32** (to test the overloaded operators in class `ModNum`) are available under the "ref/" directory. Please use them to compare your result. Please also watch out the announcements in the ceiba website and BBS.

5. Some tips you should know

1. The reference code is NOT complete and thus cannot be compiled. To make it "compile-able", you should at least initialize the static data members of class `ModNum` (i.e. the first TODO in "calcModNum.cpp"). However, even after the code become compile-able, it 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 currnet 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") is not included. For "csh/tcsh", you need to use ">&" instead. For bash, you can try "&>" or something like:

```
"modCalc.ref -File dofile.ref > out.mine 2>&1".
```

6. Grading

We will test your submitted program with various combinations/sequences of commands to determine your grade. The results (i.e. outputs) from both **modCalc** and **testMC** 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.*