Lab Assignment #3: A Command Parser

1. Objectives

The objective of this assignment is to provide you with practice on the use of C++ I/O facilities and the C string library. This assignment requires you to write a command parser that takes input from the terminal or a file, parses this input and verifies that the input is correct. You will be using the functionality of this command parser to exercise several of the subsequent assignments.

2. Problem Statement

You will write a command parser that provides a textual input interface to your program. The parser should take a sequence of commands as input. Each command consists of an operation followed by its arguments. The command and the arguments are separated by white spaces. Your parser should process commands, handle errors in input and print output as described below until the end of input is encountered. All input and output must be performed using C++ I/O facilities. All string manipulation must be performed using the C string library (not the C++ string class).

3. Input, Output and Error Processing

Please read these instructions carefully.

All input should be read via standard input (`cin`). All output and error messages should be printed to standard output (`cout`). All commands should print either one line of output or one error message (but not both). If a command has multiple errors, only the first error during the parsing of the command should be printed. To distinguish output from error messages, error messages MUST begin with "`Error: `" and end with a period ("`.`"). Whenever an error message is printed, the parser should skip all input until the end of the line and then take a new command as input from the next line, i.e., it should ignore this command. When an end-of-file (i.e., end of input) character is pressed then the program must end. The end-of-file character is generated on the ECF lab machines by pressing CTRL-D on the keyboard.

Recall from Section 2 above that a command consists of an operation and a sequence of zero or more arguments. The next section describes the valid operations and their arguments. An operation is a string. If a user types an invalid operation, then the error message "`Error: unknown command.`" should be printed. Arguments are either numbers or strings. If a number argument is expected and the user does not type a number, then the number is considered invalid and the error message "`Error: argument is not a number.`" must be printed.

Below, the argument named *`emplnum`* is an integer and it MUST be printed as a 5-digit integer. For example "`303`" is printed as "`00303`". If the user types a valid number (see definition of invalid number above) but the number is larger than 5 digits, then the error message "`Error: <number> is too large.`" should be printed. If the user types a negative number (i.e., number < 0), then the error message "`Error: <number> is`

negative." should be printed. The number value in the error message is the actual number.

You must use the cin input operator (>>) in this assignment. Do not use getline or any function that converts strings to integers or vice versa. You can assume that the input will be provided one command per line. The operation and each argument will be separated by one or more spaces or tab characters, called "white space". There may be zero or more space or tab characters before the operation or after the last argument of a command (before the newline character).

4. Commands and Arguments

The commands and their arguments are shown below. All commands are in lower case. The operations are shown in bold and the arguments are shown in italics.

- **new** *emplnum*. This command should print "New: emplnum". The *emplnum* argument is described above.

- **locate** empl*num*. This command should print "Locate: emplnum". The *emplnum* argument is described above.

- **updatename** *emplnum lastname firstname*. This command should print "Updatename: empnum firstname lastname". It takes three arguments. The *emplnum* argument is described above. The last two arguments are strings. Note the reverse order of the last two arguments in the output. Assume that *lastname* and *firstname* are no more than 80 characters long.

- **updatedivision** *emplnum division*. This command should print "Updatedivision: emplnum division". This command takes two arguments all of which are numbers. The *emplnum* argument is described above. The *division* argument must be in the range 1-5 (inclusive), otherwise the error message "Error: <division> is out of the range 1-5." should be printed. The *division* argument should be printed as a single digit.

- **delete** *emplnum*. This command should print "Delete: emplnum". The *emplnum* argument is described above.

- **printall**. This command should simply print "Printall".

- **deleteall**. This command should simply print "Deleteall".

The following is an example of input commands, outputs and error messages. The output and error messages are shown in red.

```
% ./Driver
new 12345
New: 12345
locatee   12345
Error: unknown command.
```

```
locate 125
Locate: 00125
locate -20
Error: -20 is negative.
updatename  123450 Lady Gaga
Error: 123450 is too large.
updatename f12345 Lady Gaga
Error: argument is not a number.
updatename 12345 Lady Gaga
Updatename: 12345 Gaga Lady
updatedivision 12345 7
Error: 7 is out of the range 1-5.
updatedivision 125 2
Updatedivision: 00125 2
updatedivision 12345    f3
Error: argument is not a number.
updatedivision 123450    f3
Error: 123450 is too large.
delete      4949
Delete: 04949
printall
Printall
deleteall
Deleteall
(eof)
```

Note that the command "`updatedivision 123450 f9`" prints only the first error even though the command has two errors.

5. Procedure

Create a sub-directory called `lab3` in your `ece244` directory, using the mkdir command, and secure it with the chmod command. Make it your working directory. Make a `main` function in the file Driver.cpp that calls your command parser function. Write a Makefile that generates an executable called Driver from the Driver.cpp file.

6. Deliverables

Your program should be in the file Driver.cpp. Submit this file and the Makefile using the command

```
submitece244f 3 Driver.cpp Makefile
```

7. Hints

Hint 1: When reading from standard input (cin) into a C-style string (i.e., character array), then all initial white space is first skipped over and ignored before the input characters are copied into the array up until the first white space character is encountered; the string stored in the character array is properly null-terminated. For example, if the following input:

```
Lady     Gaga rocks!
```

is typed in to a program executing the following code fragment:

```
char buf[100] ;
while( !cin.eof() ) {
    cin >> buf ;
    cout << "+" << buf << "+" << endl ;
}
```

then the program produces the following output:

```
+Lady+
+Gaga+
+rocks!+
```

Hint 2: The structure of your program may be (but does not need to be) as follows:

```
int process_input()
{
   char command[COMMAND_SIZE];

   for (cin >> command; !cin.eof(); cin >> command) {
      driver(command);
   }
   return 0;
}

void
driver(char *command)
{
   if (!strcmp(command, "XXX"))
      handle_XXX();
   else if (!strcmp(command, "YYY"))
      handle_YYY();
   else if …
      …
   else {
      cout << "Error: unknown command." << endl;
      cin.ignore(1000, '\n');
   }
}
```