CS 100 Systems Programming
Homework 4

- Linux (30 pts): **Commands and Filters.**
  a. (10 pts) Write a correct Makefile for the three programs you are writing this week. Also define a target called “clean” which will remove all the files that can be rebuilt from a make. Also define a target called “strip” which will remove the symbolic debugging information from each of your executable programs. Finally, write a target called “print” which will print (output to the terminal) the source code for your String program.
  b. (20 pts) Write the following Unix commands using grep, awk, sed, at, calendar, find, rm, and mv. Each should be single command. You may not be able to do a perfect job, but try your best.
     i. print all lines containing integer literals in the files *.cpp in the current directory
     ii. print all calls to `<<` to cout in the files *.h and *.cpp
     iii. delete all blank lines from a file called foo.cpp (actually modify foo.cpp, but be careful about I/O redirection as cat < foo > foo will not work as you might like it too.)
     iv. move all files anywhere below your home directory that have the extension `.o` or `.out` into a special directory called `.trash` contained in your home directory. Be careful not to move files from .trash to .trash (hint: use find)
     v. list the names of files in all directories named `bin` on the local machine (hint: use find)
     vi. list the names of all files larger than 20 blocks in your account directory (hint: use find)
- C++ (30 pts): **Mastering classes containing pointers.** Re-write the String class from the previous homework, but use a singly-linked list as the representation. You are only allowed to have one data member, named head, which is a pointer to a ListNode. Like last week, these strings will be of varying lengths and must grow and shrink as necessary. Implement all the appropriate methods given below. Remember, as always, never copy and paste any code from anywhere - the Internet, textbooks, or from a past homework solution. You may reuse your own String declaration and main from last week’s homework, but be sure to make any changes shown below.
  Class String declaration:

```
class String
 {
 public:
   /// Both constructors should construct
   /// from the parameter s
   String( const char * s = "");
```

```
  String( const String & s );
  String operator = ( const String & s );
  char & operator [] ( const int index );
  int length() const;
  int indexOf( char c ) const;
  bool operator == ( const String & s ) const;
  /// concatenates this and s
  String operator + ( const String & s ) const;
  /// concatenates s onto end of this
  String operator += ( const String & s );
  void print( ostream & out );
  void read( istream & in );
  ~String();
private:
  bool inBounds( int i )
  {
    return i >= 0 && i < length();
  }
  struct ListNode
  {
    char info;
    ListNode * next;
    ListNode(char newInfo, ListNode * newNext)
      : info( newInfo ), next( newNext )
    {
    }
    // you may add static methods here
  };
  ListNode * head;
};
ostream & operator << ( ostream & out, String str );
istream & operator >> ( istream & in, String & str );
```

- Write a main program, called testLinkedString, which tests each function defined in your class String. You may use the same testing main from last week as a start if you like, but notice I deleted some of the methods.
- Give an estimate of the relative efficiency of each of the following two assignments (i.e., how many function calls, how many copies are made, etc):

  ```
  String s("Hello");
  String t("There");
  ```

```
    s = s + t;  // assignment 1
    s += t;     // assignment 2
```

- Systems Programming (40 pts): Implement the following two programs, but periodically check your processes from the shell with **ps aux**, and kill any run-away processes with **kill -9 pid**.
  a. (20 pts) **Signal Handling.** Write a C++ program, called handle_signals, to do the following:
     - continually print X's on the screen using the following main program:
       ```
       int main() {
               // Turn off output buffering here if you like
               init_signal_handlers();
               while (1)
                       // Print an 'X' and flush the output buffer here
       }
       ```
     - whenever the user types a ^C (interrupt) print an I, DO NOT abort
     - whenever the user types a ^\ (quit) print a Q, DO NOT quit
     - whenever the user types a ^Z (stop) print an S, then stop
     - you, the user, may resume the job with the fg command from the shell
     - on the third ^Z, print a summary of the number of signals received as follows, then exit:
       ```
       Interrupt: 5
       Stop: 2
       Quit: 3
       ```
     - You should flush the output after printing each character to prevent problems with buffered output or just turn off output buffering - your choice. Also, you may use signal() or sigaction(), but sigaction() is the preferred way to handle signals as it is the new and improved interface. Here is a link with an intro to signal handling: http://www.alexonlinux.com/signal-handling-in-linux
  b. (20 pts) **Command shell.** Write a program called my_shell, which is the next step towards a simple command shell to process commands as follows: print a prompt (e.g., "% ") then read a command on one line, then evaluate it. A command can have the following form:
     commandName [argumentList] [IORedirect] [IORedirect]
     Where an argumentList is a list of zero or more words (no patterns allowed), IORedirect is either "< inFile" or "> outFile" where inFile and outFile are valid Unix file names. You will use versions of the system calls fork(), open(), dup2(), exec(), and wait() for this assignment.

     You can test my_shell with these commands:
     ```
     % /usr/bin/tee newOutputFile  # input from and output to the terminal
     % /usr/bin/tee newOutputFile < existingInputFile
     % /usr/bin/tee newOutputFile < existingInputFile > newOutputFile
     ```

     If you want to plan ahead, next week, we will add pipes and background jobs. Only the first command in a pipeline can have its input redirected to come from a file and only the last command in a pipeline can have its output redirected to a file, so you can execute these commands:

```
% /usr/bin/tee newOutputFile > newOutputFile < existingInputFile &
% /bin/cat < existingInputFile | /usr/bin/tee newOutputFile > newOutputFile
% /bin/cat < existingInputFile | /usr/bin/tee newOutputFile > newOutputFile &
```

- For *all* files use the following header:

```
/*
* Course: CS 100 Fall 2013
*
* First Name: <your first name>
* Last Name: <your last name>
* Username: <your username>
* email address: <your UCR e-mail address>
*
*
* Assignment: <assignment type and number, e.g. "Homework #4">
*
* I hereby certify that the contents of this file represent
* my own original individual work. Nowhere herein is there
* code from any outside resources such as another individual,
* a website, or publishings unless specifically designated as
* permissible by the instructor or TA. */
```