The teaching assistant Xiaoyu Zheng will have office hours Tuesdays and Thursdays 1:00PM-3:30pm and by appointment in LI-96P. His phone and email are 442-4285 and xz7223@albany.edu. 

He will hold special office hours today (Fri, Sept 6) from 10:00AM-12:00PM and Monday, Sept 9 from 2:00PM to 4:00PM for the purpose of assisting people with using the ACS Unix system and any other aspect of getting started with this project.

A handout on more programming guidelines and how to use turnin for this course will be provided next week. 

Look over this assignment right away and determine whether you need any questions answered about technicalities so you can make progress in it without delay. Bring all such questions to the TA or professor. Reading: Assigned in the Sept 3 course announcement.

1 Part 1

Write a program named part1 that should read from Unix file descriptor 0, known as standard input, one (8 bit) character (i.e. byte) at a time. The program should exit if the end-of-file condition occurs when it tries to read the next character.

After each character X is read, part1 should write to Unix file descriptor 1, known as standard output, one or more characters \( F(X) \) according to the following conversion rules:

<table>
<thead>
<tr>
<th>Condition</th>
<th>( F(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>X is a lower case letter</td>
<td>( F(X) = X )</td>
</tr>
<tr>
<td>X is an upper case letter</td>
<td>( F(X) = \text{\textbackslash x} ) where x is the lower case version of X</td>
</tr>
<tr>
<td>( X = \text{\textbackslash} )</td>
<td>( F(X) = \text{\textbackslash\textbackslash} )</td>
</tr>
<tr>
<td>X is a printing or whitespace, as defined by ISO C functions isprint and isspace, and not in any preceding category</td>
<td>( F(X) = X )</td>
</tr>
<tr>
<td>Otherwise,</td>
<td>( F(X) = \text{dd} ) where dd is the two digit or lower case letter numeral that represents the value of X in hexadecimal.</td>
</tr>
</tbody>
</table>

1.1 Warnings

BEWARE: The NULL character (value 0) is covered by the “Otherwise” case. Specifically, the output should be \( \text{\textbackslash 00} \) (3 characters!). This may lead to problems with some thoughtless uses of C string functions.

Another hint: Do NOT treat control-d specially! Disk files, for example, do not relate control-d to end of file. Terminals might be put into a state where control-d is transmitted to your program.

1.2 Example run

```
unix% part1
a (I keyed in a enter)
a (Computer printed a and put cursor on the next line)
abc
abc
abCcd\'\^A (My last 2 keystrokes were control-a and enter)
a\bc\c\d\01
unix% (I keyed control-d to mark end of file on input from a terminal.)
(Sample input and output files for testing part1 with the command line below will be provided next week: part1 < infile > outfile)
```
1.3 Implementation Requirements

1. Input and output must be done with the system functions \texttt{read} and \texttt{write} respectively. You can get started by modifying the program on page 30 of Haviland to use a buffer size of 1. This will give you a program that meets all the requirements when the input is restricted to lower case letters, digits, punctuation and whitespace characters.

2. You must design and implement a module for the one character to multi-character sequence conversion function. This module must be independent of everything else in the assignment. You must design (i.e., choose wisely) the data types for the input and output of your conversion function. The \texttt{interface} of your module must be defined in a header file file that contains only type (or class) declarations and function prototypes; no function implementation code allowed. (Don’t use inline functions for this assignment.) The \texttt{implementation} of your module must be a separately compilable C/C++ file that must have an include preprocessor directive targeting your header file. Other compilable C/C++ files whose code uses your conversion function and/or datatypes must have the identical include preprocessor directive targeting your header file.

Hint: You might fashion your conversion function’s output interface like some of the definition of socket function \texttt{accept} since it returns a value of type \texttt{struct sockaddr} whose length can vary. You can see the man page for this function with the commands \texttt{man -s 3SOCKET accept} or \texttt{man -s accept} on the \texttt{acunix} system. Another idea is used by \texttt{write}.

**ANOTHER REQUIREMENT:** Write precise and concise user documentation for your conversion module in the style of a Unix man page. This document should explain to a programmer how to use your conversion module in a way the doesn’t require him or her to read the header or implementation files. It should omit explanation of your implementation.

Write a file of no more than 1 page of plain ASCII text, name it \texttt{conversion.man} and submit it under your \texttt{Part1} directory.

3. The building of the \texttt{part1} executable file must be controlled by a correct Makefile that explicitly expresses the \texttt{dependencies} upon ALL of the C/C++ code AND header files that you have written.

Each Makefile must have a target named \texttt{clean} so that the command \texttt{gmake clean} will direct the system to delete all object and executable file in the current directory.

4. Keep your project work in the following directory organization to facilitate grading: Create a directory named \texttt{Proj1}. Create 4 subdirectories of it named \texttt{Part1, Part2, Part3}, and \texttt{Part4}. Each subdirectory must have all the files (header, code and Makefile) for each part, independently of the other parts. Please do, of course, copy your conversion module files from \texttt{Part1} to the other subdirectories so you can reuse it.

2 Part 2

Write a program named \texttt{part2} that functions and is implemented in the same way as \texttt{part1}, except for what file descriptors it uses for input and output.

2.1 Requirements

1. The program \texttt{part2} takes two command line arguments \texttt{infilename} and \texttt{outfilename}. It should try to open the files \texttt{infilename} for reading and \texttt{outfilename} for writing. If \texttt{outfilename} already exists, its contents should be overwritten. If file named \texttt{outfilename} doesn’t exist, it should be created. (Hint: Appropriate argument values to \texttt{open} will do the trick.)

2. Any errors that prevent the opening of these files for their specified use should be reported with informative error messages of your own design. After the error message, the program should exit with an exit code of 1.

   Missing arguments are considered errors. Such errors should be reported by writing to \texttt{cerr} or \texttt{stderr}. Errors detected from the return value of system functions (usually system calls) should be reported
with the perror function so your message is followed by a builtin message that describes the significance of the value of errno set because of the error. (Follow the examples of Haviland’s chapter 10 instead of chapter 2.)

3. Follow the same implementation requirements as Part 1. Only the low level I/O functions open/read/write/close must be used for non-error-message data input and output.

4. Find at least one new natural modularization boundary and express it with a separately compiled implementation file and a header file for the interface, as in Part 1.

2.2 Investigations

For each investigation, write a brief account of how you did it, what were the results, and what you learned (or verified if you already knew it). Put each account in a file named INV.1, INV.2 etc. (Plain ASCII is OK. If you really need to use a word processor, the files must be in either PDF, Postscript, or HTML (with no external references). Use appropriate file name suffixes.)

1. Verify that the two shell command lines
   part1 < infile > outfile
   and
   part2 infile outfile

2. Explain why lower case letters appear duplicated when input to part1 from the keyboard but they are not duplicated in the output files produced from part2.

3. Even though the effects of the two commands are the same, the files are opened by different kinds of programs. Explain what program opens which files in each case. Compare the error messages for various error conditions in each case.

4. Read as much of Haviland’s chapter 9 as you need for using the stty command so that when you run part1:

   (a) Only one copy of lower case letters or numbers appears when you type them.

   (b) When you press a character key, the output comes out immediately instead of on the next line together with other outputs only after the enter key is pressed eventually.

   (c) Observe the cases for the min and time parameters with part1. They can be set by stty after stty is used to set “-icanon”.

   Hint: Try out “stty -a” first.

3 Part 3

Follow material from Haviland’s chapter 10 to write a server that reads characters from a client on a connection oriented link. For each character read, the server should write back to the client the one or more characters according to the conversion function specified under Part 1.

1. Your server should detect the errors, print the error reports, and exit as Haviland’s example program does.

2. Learn to get multiple xterm windows so you can simultaneously use 2 or more shells and other programs on acunix. One way is to use “X-Win32” on a Windows PC in the Lecture Center computer rooms. There are online instructions and you can get an instruction handout in LC-27. After starting X-Win32 and the telnet connection to an acunix system, give the command “xterm &” two or more times. That way, you can start your server from one xterm window and try one or more clients from one or more other xterm windows.

You can see debugging or error messages the server prints this way. You can even run the server under a debugger: Try the command “ddd part3”
3. To verify your server is in the LISTENING state (after the listen system call succeeds), use the command

`netstat -a | grep LISTEN`

The report line from `netstat` shows the port number. (You will also find out about other servers such as your classmates', so you can avoid conflicting port number choices) Find out from `man netstat` what the -a option is for.

Note the above command uses the shell’s “pipe” feature that sets up the standard output of the process running the first command to be transmitted as the standard input of the process running the second. Find out from the man page about `grep`.

4. You can test your server by using general purpose client utility program `telnet`. Just type say `telnet localhost 7000`
to connect to TCP server such as a process running the first sample program of Haviland’s chapter 10. Here, `localhost` is meant literally: “localhost” is resolved by the network system to mean the local host, what is says!

However, specifying the port this way WILL NOT WORK on PCs! See below.
To make `telnet` exit: Type `telnet`’s escape sequence (`control-c`) by default to make `telnet` display a prompt and then interpret terminal input as a `telnet` command instead of copying it to the connected server. The `quit` command will make `telnet` close the connection and exit.

5. Write 2 versions of the server program: `part3single` and `part3forked`. You can copy and modify Haviland’s chapter 10 program for `part3forked`. But `part3single` must not fork. When the accept call returns, `part3single` should do the conversion until the current connection ends. It should then close that connection’s file descriptor and then block on the accept operation.

6. You can end the server’s execution by typing `control-c` on the terminal window where you started it.
Make sure none of your servers are running when you log off the system or pause using it for this project! Otherwise, you may get messages from the system administrators, and even worse, “crackers” from around the world are more likely to discover your server with a “port scan” and try to invade your account through your server!

7. You will have to choose a port. A detail omitted (unfortunately) in Haviland’s book is that the `sin_port` value in the `struct sockaddr_in` is formed of 2 bytes in `Internet network order`, which is big-endian. That means the byte with lower address has higher numerical significance. Academic Computing’s SPARC Unix systems use big-endian multibyte integers, so that C integer say 7000 (which was used in Haviland’s book) signifies port 7000 on the internet. However, if you do the project on a PC (Pentium) system, Haviland’s program will NOT open port 7000! The fix is to write say “htons(7000)” or “htons(i)” where i is an integer value when coding the value for `sin_port` of a `struct sockaddr_in` structure. (“htns” is for “host to network short int”.)

### 3.1 Investigations

An assignment of investigations for Part 3 will be given next week (after I try it out!!).

### 4 Part 4

Write a client that functions like `part2` but uses your `part3` server for the conversion instead of doing it itself. The `part4` program should take 3 command line arguments: input file, output file, and port number in decimal.

Optional: Program `part4` to take an IP address or domain name as a 4th argument and run it and `part3` on separate hosts on the Internet.

I realize that using a server instead of a single process for such a simple computation is silly, but really useful client/server combinations work the same way.

$Id: pr1.tex,v 1.2 2002/09/06 15:03:39 sdc Exp $