

A document on programming guidelines and how to use turnin for this course is now available and had been assigned reading for project 0. This project builds on project 0. The web page has a link to information on “Makefiles” for those who need it—At Albany, this is taught in CSI333.

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.

This project has 3 parts: Part 1, Part 2, and “Trouble”. The work for each part must be done and submitted in a separate directory.

1 Part 1

Modify your project 0 work slightly to do this...

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. For each byte read, it should write one or three bytes according to the table below. When the end-of-file condition occurs on standard input when it tries to read the next character, it should print the messages and information exactly as specified for our Project 0.

X is <code>\</code>	$F(X) = \\$
X is a printing or whitespace other than <code>\</code> , as defined by ISO C functions <code>isprint</code> and <code>isspace</code>	$F(X) = X$
Otherwise,	$F = \\dd$ where <code>dd</code> is the two digit or lower case letter numeral that represents the value of X in hexadecimal.

1.1 Warnings

BEWARE: The NULL character (value 0) is covered by the “Otherwise” case. Specifically, the output should be `\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)
aBcCD\01
          (I keyed control-d to mark end of file on input from a terminal.)
Number of characters in the file:13
Number of words in the file:3
unix%
```

1.3 Implementation Requirements

1. Input and output must be done with the system functions `read` and `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 printing characters other than `\`.

2. Here's a problem: Some input characters will be converted to one character, one will be converted to two, and the others will be converted to three.

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 *interface* of your module must be defined in a header 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 *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: (Start to think like a software architect.) You might fashion your conversion function's output interface like some of the definition of socket function `accept` since `accept` returns a value of type `struct sockaddr` whose length can vary. You can see the man page for this function with the commands `man -s 3SOCKET accept` or `man -a accept` on the `acunix` system. Another idea is used by the interface used by `write`. A third choice is to mark the end of the output by the null character; however that is not "universal" because it requires that the output never contain the null character. Note that to write the output using `write`, your program must calculate the number of characters to write anyway.

ANOTHER REQUIREMENT: Write precise and concise user documentation inside the header file that codes the interface for your module. You are encouraged to create your module as a C++ class.

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

Each Makefile must have a target named `clean` so that the command `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 `Proj1`. Create 3 subdirectories of it named `Part1`, `Part2` and `Trouble` 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 `Part1` to the other subdirectories so you can reuse it.

2 Part 2

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

2.1 Requirements

1. The program `part2` takes two command line arguments *infilename* and *outfilename*. It should try to open the files *infilename* for reading and *outfilename* for writing. If *outfilename* already exists, its contents should be overwritten. If file named *outfilename* doesn't exist, it should be created. (Hint: Appropriate argument values to `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 `cerr` or `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
```

 have the same effect, provided the second one has no errors.
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 section 9.1, 9.2 and 9.3 up to 9.3.6 of Haviland's book. You will be using the Unix `stty` command to change parameters belonging to the "terminal line discipline." (You will **not** have to write programs to change the terminal characteristics with the `termios` structure. Instead, you will run the `stty` command with various command line arguments to change the terminal characteristics.) Refer back to these readings as necessary for doing the following exercises and experiments.

However, first make sure that the shell you are using is **NOT** `tcsh`, because `tcsh` (by default) automatically resets terminal characteristics. To see what shell you are using, try the command "`echo $SHELL`" or "`ps`". If `tcsh` shows, give the command "`csh`" and continue as instructed¹

If "`tcsh`" doesn't show up, just move ahead.

Figure out and verify how to use 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.

¹This is a nuisance for this assignment. After you're done, you might delve into the problem by viewing the Unix "man" (manual) page for `tcsh` (Give the command "`man tcsh`".) and reading sections related to the `tcsh` builtin named "`setty`". But this delving is NOT important for this course and should not be done unless you finished all required work and are madly curious.

3 The Third Part “Trouble”

First take your `part2` program and convert it to an internet server. Code in your Makefile that the executable program is named `server`. Basically, you will use some socket functions instead of `open()` obtain one file descriptor and use that file descriptor for both input and output.

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 to 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 it says!

However, specifying the port this way WILL NOT WORK on PCs! See below.

To make `telnet` exit: Type `telnet`’s escape sequence (`control-]` 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. 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!
6. 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 Now for some Trouble

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

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.

You must test your **Trouble** program by making the **server** program run and then starting **Trouble** in a different window, or in the same window with **server** running in the background. Of course, you must prepare a sample input file for **Trouble** first, and decide what name **Trouble** will use for its output file.

You should find that **Trouble** works OK for small files, but gets into “trouble” for larger files. This happens even though **part2** works fine for all sizes of files. Try to determine for what size files **Trouble** fails and then try to explain why. Write your report in a file named **Trouble.txt** (or other extension depending on your word processor.) If you cannot produce a failure, see me **in advance of the due date**.

\$Id: pr1.tex,v 1.1 2004/09/08 14:16:26 sdc Exp sdc \$