CS120 Programming Instructions

Terence Soule

August 20, 2012

Contents

Introduction

This manual was written to accompany the Computer Science I course (CS120) for the University of Idaho Computer Science program. It covers expectations for programs (for both labs and assignments), basic Unix, and nano.

1 Expectations

All students must show up at the beginning of the lab period. All work must be complete and turned in by the end of the lab period. Late work will not be accepted. If the exercise(s) are assigned before the lab meets students may start, and possibly complete, the exercise(s) before the lab meets. In these cases students may turn in their work at the beginning of the lab period, but they still must show up for lab.

All work should be typed. Any programs written as part of a lab exercise must be turned in along with a copy of the output showing the results.

All work must include the student's name, the date and the lab number. Individual exercises should be clearly marked.

It is up to the student to show that their programs function properly. Every program should be run multiple times with different inputs or parameters to show the it functions properly under a range of conditions.

It is acceptable to request help from or to discuss problems with other students. However, any work you turn in must be your own work. You must understand it and be able to explain it. Copying another students code is strictly forbidden.

There are a number of expectations that apply to any programs turned in:

- All programs should be adequately commented.
- All programs should begin with a comment containing the student's name, date and assignment number.
- Variable names should be descriptive.
- Programs should be broken into functions where appropriate.
- Output should be clear.
- Requests for input from the user should be clear.
- Programs should be adequately tested to show that they function properly under 'reasonable' conditions.

2 Introduction to Unix

Unix is an operating system originally designed in the 60's and 70's to be multi-user, multi-taking operating system. It is one of the most, if not the most, influential operating system ever developed.

Unix must have a way of organizing the users' files and of allowing multiple programs to run simultaneously without interfering with each other. Files are organized in a hierarchical tree structure. The root of the tree is called '/'. Each 'branch' of the tree is a subdirectory. Each subdirectory has its own name. (Subdirectories are equivalent to folders in a Windows or Macintosh environment, but you don't get the pretty graphical interface.)

Every user is assigned their own subdirectory whose name is the same as the user's login. Thus, the user tsoule, keeps all of his files in a subdirectory called tsoule or in subdirectories within the tsoule subdirectory. Each subdirectory is protected so that only certain users are allowed access to it. In addition, there are several levels of access, for example you might be allowed to read the files in a certain subdirectory, but not to modify those files. (This is common in subdirectories containing programs that every user wants to use, such as the c++ program, but that users shouldn't be allowed to change.)

There are a number of basic commands that are used to negotiate Unix's file structure:

- **ls** Short for LiSt, this command lists the files and subdirectories in the current subdirectory. It does not list 'hidden' files; ones whose names starts with a '.'.
- **ls -a** This command lists all of the files and subdirectories in the current subdirectory including the hidden files.
- **ls -l** This command lists all of the files and subdirectories in the current directory in the 'long' format - it lists information other than just the names.
- **pwd** Short for Print Working Directory, this command tells the used where they are in the files structure. A typical output would be */users/faculty/tsoule/* showing that the user is currently in the subdirectory tsoule, which is in the subdirectory faculty, which is in the subdirectory users, which is in the root directory /.
- **cd** *directoryname* Short for Change Directory the command changes which directory the user is currently in.
- **cp** *filename1 filename2* Short for CoPy, this command copies filename1 into filename2. If filename2 already exists, it is replaced with the new file. The file names can include directories.
- **mv** *filename1 filename2* Short for MoVe, this command copies filename1 into filename2 and removes filenmae1. If filename2 already exists, it is replaced with the new file. The file names can include directories.
- **mkdir** *directoryname* Short for MaKe DIRectory, this command creates a new subdirectory.
- **rm** *filename* Short for ReMove, this command removes (deletes) a file.
- **rmdir** *directoryname* Short for ReMove DIRectory, this command removes (deletes) a directory. A directory must be empty before you can remove it.
- **script** *filename* The script command makes a record of everything printed on the screen by writing it to the file *filename*. (If no filename is given it is written to a file called *typescript*.) To use the script

command type script *filename*, run the program, **then type exit**. Typing *exit* stops the script. The script file can be printed and turned in as the sample output of a program.

There are two errors to avoid when using the script command. First, if you fail to type exit everything you do will continue to be dumped into the script file. In particular, if you attempt to open the script file before exiting the script command you will set up an infinite loop. Second, make sure that the *filename* you use is not the same as the name of a program you wrote; otherwise the script file will overwrite (erase) the program and you will have to rewrite the program.

- **man** *commandname* Short for MANual, this command presents help information on the given command.

All Unix directories include two special subdirectories called '.' and '..' The directory called '.' is the current directory. Thus, for example, the command *cd .* moves the user to the current directory, which has no effect. The directory called '..' refers to the directory above the current directory. So, *cs ..* moves you up one level in the directory hierarchy.

3 Nano

Nano is a simple text editor that we will use for writing programs and occasionally other documents. Nano can be found on almost all UNIX machines. We begin with Nano only because it is one of the simplest text editors to learn and use. Most computer science majors eventually chose to use a more advanced and more powerful editor such as vi or emacs. Tutorials on the use of vi and emacs are available on the web and if you do chose to learn one of these text editors (or some other editor) please feel free to use it for class assignments.

To begin using nano simply type:
nano *filename*
at the Unix prompt. This will begin nano and either open an existing file or create a new file depending on whether the given file already exists. Once the file is open you can begin typing.

Note that nano does not use the mouse. You use the arrow keys to move the cursor.

Once nano is started you will see the file's current contents (nothing if it is a new file) plus two rows of commands listed across the bottom of the screen. For example the first command listed is ^*G Get Help*. The ^ symbol stands for the control key in nano. So, to 'get help' hold down the control key (labeled Ctrl on the keyboard) and simultaneously press the g key (capitals not required). This opens a help page. You can exit the help page by pressing ctrl and x simultaneous as noted at the bottom of the help page or you can skip to the next page of help by pressing ctrl and v simultaneously.

Other common and useful commands are:

^o (press ctrl and o simultaneously) this saves your current work. Nano will ask for a filename (all of nano's prompts are at the bottom of the screen) and prompt you with the current filename. So, if you just hit enter the file is saved under the current name.

^x exits nano. If you haven't saved the latest changes to the file nano will ask you whether you want to save, and if so what file name to use, just as with ^o.

^k cuts the current line of text deleting from the file.

^u uncuts the last set of cut lines restoring them to the file at the cursors current location. ^k and ^u can be used as a primitive cut and paste to move one or more lines of text.

^w allows you to search for a string. After typing ^w (remember that's ctrl and w simultaneously) nano will ask for the string you want to search for. Again the prompt for the search string appears at the bottom of the screen.

^c tells you the current line number. Useful for finding errors in your code.

^v jumps down a whole page. Useful for navigating large documents.

^y jumps up a whole page.

4 The Curses Library

The curses library defines a number of functions that give you much more precise control over displaying characters on the screen. The name curses was apparently derived from the term cursor by someone with a crude sense of humor. (There is also a newer version of the library called ncurses, short for new curses, that is available, but it is not currently installed.)

To use the curses library you need to include it:

#include<curses.h>

and explicitly link it during the compiling process. Common libraries are linked automatically, but because the curses library is not always used it has to be explicitly linked. So, to compile a program that includes the curses library type:

g++ -lcurses *programname*

There are a few basic commands that are a part of any program using the curses library:

- WINDOW **variable name*; - This declares a variable of type 'window'. For now ignore the *.
- *variable name* = initscr(); - This connects the WINDOW variable to the screen
- clear(); - This function clears the screen.
- refresh(); - This function redraws the screen. It must be used every time you want something new to be displayed.
- endwin(); - This function ends curses control of the screen. It should go at the end of the program. If you forget to include it at the end of the program, when the program exits the screen will no longer respond properly and you will have to exit and log back on to the system.

Note how each of these commands is used in the sample program given below.

There are three simple commands that help you place and remove characters from the screen:

- move(*r,c*); - This function moves the cursor to row r and column c. r and c can be integers, integer variables or integer expressions.
- insch(*ch*); - This function places the character ch at the cursor's current position. ch can be a chracter such as 'R' or a character variable.
- delch(); - This function deletes the character at the cursor's current location.

For example, the code:

```
int row = 5, column = 10;
move(row,column);
delch();
insch('X');
refresh();
```

removes whatever character is currently at row 5 and column 10 and replaces it with the character X. Note that without the refresh command the changes wouldn't show up on the screen. Also note that the preliminary commands (defining a variable of type WINDOW, initializing the screen, etc.) would need to be done first.