There are a few commands, terms and concepts that are useful to know at the start of this discussion, so I will start with those.

standard input device – refers to your keyboard. You type something while using an program, and the program gets what you type from the system – provided it accepts input from the standard input device.

standard output device – refers to the screen of a computer terminal, or a window that acts as a terminal screen. Putty opens a window you use to interact with a computer (in this course the csx server), and that window acts like a terminal or display device for the computer you are interacting with. Text displayed by the program (putty, in this case) to the standard output device is displayed as text in the program window.

I/O redirection – refers to the ability of the operating system to route inputs and outputs to different devices of files. This is usually limited to redirection of the standard input and output device traffic. There are three symbols used in command lines to invoke various kinds of I/O redirection. The "<" symbol tells the operating system to take characters from the file or device named to the right of the symbol and use it as the source of standard input for the program or device to the left of the symbol, rather than letting that program read from your keyboard. The ">" symbol tells the operating system to route the standard output stream from the program to the left of the symbol to the file or device to the right of the symbol. The "|" symbol tells the operating system to route the standard output from the program to the left of the symbol to the standard input of the program or device to the right of the symbol.

current working directory – csx and almost all computers today have file systems based upon the concept of directories or folders. Graphical interfaces and applications for most modern operating systems (including Windows and OS X) hide some of this from users. The Linux command-line environment on csx does not conceal this as much. You need to be aware of where you are in the directory system more than you do in those other more user-friendly, graphically-oriented environments. A directory or folder is simply a logical location in the file system, rather than a physical location. Programs need to know where to look for files, and to simplify the problem, there is the concept of a current working directory. Programs can access files in the current working directory without having to know the full path to the files because the operating system will simply look for a file that does not have a path associated with the name in that current working directory.

To find out what your current working directory is, use the **pwd** (print working directory) command, for which the output is pretty self-explanatory. To change the current directory, use the **cd** command. You can change the current directory by giving the command the full path to the directory you are switching to, or a relative path. A full or absolute path includes the root of the directory system, and shows all successive child directories in sequence until reaching the target directory. An example of a full path is **/home/rlchurc/test**. You will rarely need to use full paths. Relative paths start from the current directory. To move up one directly level, use "**..**" in the relative path. To move to a subdirectory, simply give the name of the subdirectory. In both full and relative paths, the "/" character separates components of the path. The command

cd source/datafiles moves the working directory from the starting directory through the **source** subdirectory to the **datafiles** directory – assuming these directories exist. The command **cd ../../c-code/project_1/includes** would change your current directory up two levels, then down from there to the **c-code/project_1/includes** directory.

You can create a new subdirectory using the **mkdir** command. The command **mkdir cs2433** creates the directory **cs2433** in your current working directory. You can create other directories elsewhere buy using the appropriate full or relative paths to the directory you want to create a subdirectory in, then the name of the subdirectory to be created. The command **mkdir cs2433/p0** creates the subdirectory **p0** in the subdirectory **cs2433**, if the **cs2433** subdirectory exists.

To see what files or subdirectories are in your current directory, the **ls** command will do the job. You can list the contents of a specific directory by providing a path to the directory in questions. You can narrow the listing to just C source files by using the **ls *.c** command. Using an **ls *.c*** command will list all files whose file name extensions start with "c". (Note that Linux is case-sensitive, so files whose file name extensions start with "C" will be skipped.) The "*****" is called a "wildcard" character, and says to match any number of characters, while the substring **.c** says to require the items listed to end with those characters. The "**?**" character is another wildcard, but matches precisely one arbitrary character. Both symbols are used in creating regular expression, which can be very useful, but are beyond the present scope. You should explore the wonders of regular expressions in Linux and Unix command-lines for yourself. Also, there are many useful modifiers you may use with the **ls** command. You should read about these by using the command **ls --help**.

Now that we have some concepts and useful system commands down, we can look at the problem of compiling C and C++ files, while introducing a few more tools.

Compiling a simple program in the C language on the csx server can be done using any of the C/C++ compilers on the csx server. As a starting point, we will be using the **gcc** compiler. This and the other C/C++ compilers are very powerful tools with lots of options available, some of which you may need in more advanced CS courses. Therefore, we start by showing you how to learn about these options.

If you want to know more frequently used compiler options available to you in using the **gcc** compiler, type the following command at the command prompt on the csx server.

gcc --help<enter>

It is unlikely that you will actually need to use any of the more complex options for the **gcc** compiler, but it is a convenient way to get more information about the compiler options, should you need them. You can get even more information using the **man** command.

man gcc<enter>

This command prints the **gcc** users manual to the standard output device using the **more** filter. The **more** filter displays the first page of the text to be displayed, then stops, if there is more than

one screen-page to be displayed. If you press <enter>, the next line will be displayed. If you press the space-bar, a full, new screen-page will be displayed. You cannot move backwards, though, so if you want to read this material and be able to refer back to parts that have been pushed off-screen, you have two options. First, some material will remain accessible by using the scroll-bar at the right of the **putty** window – though very long man files like that for **gcc** will exceed the buffering available. The other is to make a copy of the **man** documentation available as a file in your account. You can do this using the following command.

man gcc > man_gcc.txt<enter>

This will create a copy of the **gcc** manual in the file named "man_gcc.txt" in whatever directory is your current working directory.

For most or all of the work in this class, you can compile a C source file by typing the following command and the command prompt.

gcc mysource.c<enter>

where "**mysource.c**" is simply an arbitrary C source file. Assuming the file compiles successfully, a file name "**a.out**" will be created. This file is executable, and you can run it by typing **a.out** at the command prompt and hitting <enter>. (Note that you may need valid command-line parameters for the program to run successfully.)

Having all executable files be named a.out would not be very helpful, so **gcc** and other compilers allow you to specify the name of the target file.

gcc –o myprog mysource.c

The above command causes the **gcc** compiler to compile the C source file **mysource.c** and generate an output file named **myprog**.

NOTE: You must take care to ensure that the name of the target file, including file name extensions, is not identical that of the source file.

Finally, for those of you who are familiar with batch files, the **chmod** command is important for you to know. Any time you write a batch file, you have to tell the operating system that it is an executable file. Rather than get into a long discussion of how to set and clear access privileges, remember these two commands.

chmod 700 afile

chmod 600 afile

The first octal (base 8) digit in the first command indicates that the file named "afile" is to be made executable, while the first octal digit in the second command makes the file non-executable. Both commands set the bits that allow you to read and write the file (so you can edit it), while clearing all the group and global privilege bits.