A Quick Guide to Unix Commands

April 30, 2003

1 Basic UNIX commands

Below are brief descriptions of the UNIX commands you should know. Remember that you can get more information on a command via the UNIX Manual Pages. The commands in the examples below are always the first word, while the rest of the words are arguments to the commands.

man: Show the Manual Page (documentation) for a command (or function)

```
man ls
man 3 pow
```

man gives documentation for a command, such as what it does, what flags it takes, etc. Man pages are a bit technical, but you get used to using them over time.

You can also look up C(++) library functions. While looking up a library function, as in *man printf*. If you get a UNIX command by the same name instead, try looking in another section of the Manual Pages. You may add the section number before the command to lookup, as in *man 3 pow* for C library functions.

mkdir: Make a directory

```
1> mkdir foo
```

mkdir makes a directory that can hold files and other subdirectories. Above, we issued a command to make a directory called *foo* under the current directory.

cd: Change (to another) directory

```
cd foo
cd ..
cd foo/bar
```

cd allows you to go to another directory, and thus, change your current directory. Above, we first go into the directory named *foo*, which must exist under the current directory. Then, we go to the directory above us (dot dot (..) is shorthand for the directory above). Finally, we go to a directory named *bar*, under a directory named *foo* under our current directory.

pwd: Find out what is your current directory

```
pwd
```

pwd always reports the full path of your current directory.

ls: List files, directories or their information

```
ls
ls foo
ls -l prog1.cpp
```

ls is used to list files (not their contents). For example, *ls* above list the files (and subdirectories) in the current directory. Similarly, *ls foo* lists what files are in the subdirectory *foo*. In a slightly different usage, *ls -l prog1.cpp* lists information about the file *prog1.cpp*, such as how big it is etc.

cat: Look at the contents of a text file

```
cat prog1.cpp
```

Spits the contents of a text file on the screen.

less: Look at the contents of a text file page by page

```
less hw1.cpp
```

To view a longer text file, use the *less* program followed by the name of the file you want to view, as in the example. Less will allow you to page forward through the file using the ¡SPACE¿ bar and page backward using the b key.

When you are done looking at the file, type q to quit less.

cp Copy a file

```
cp prog1.cpp prog1.cpp.bak
cp testfile1 testfile2 ~/homeworks
```

cp makes a copy of a file. The last argument specifies the destination name. The arguments preceding it are the source file(s). When the destination is a directory, files with the same name as the source files are placed in that directory. In this case, any number of source files (to be copied) may precede the destination directory. In both cases, since a copy is made, the original source file is left untouched.

In our first example, a backup copy of the file *prog1.cpp* is made–the copy's name is *prog.cpp.bak* and will reside in the current directory. In the second example, copies of the files *testfile1* and *testfile2* are made in the directory *homeworks*, which resides under our home directory (i.e., referred to by the ~).

mv: Move (or rename) a file

```
mv writeup.bak writeup
mv hw3.cpp hw3.h ~/homeworks
```

mv moves or renames files. The last argument specifies the destination. When the destination the name of a directory, files with the same name as the source files are placed in that directory. In this case, any number of files may precede the destination. In both cases, the original source file won't exist any more (except with their new names and possibly new locations).

In our first example, the file *writeup* is restored from a backup copy (which we could have made with cp) named *writeup.bak*. After, the file *writeup.bak* no longer exists. The new file *writeup* will end up in the current directory. In the second example, the files *hw3.cpp* and *hw3.h* are moved into the directory *homeworks*, which resides under our home directory (i.e., referred to by the). There will no longer be files named *hw3.cpp* and *hw3.h* in the current directory.

rm: Remove a file

```
rm file1 file2
```

Removes the files. There is no easy way to undelete these files.

2 Useful shell features

Command line editing: On our system, you can use the up and down arrows to scroll through the commands that you typed during the same login session. Within a command line, you can use the left and right arrows as well as backspace to change what you typed before. Another nice feature is file completion. This works by type the beginning of a filename and then pressing the tab key.

&: running in background

```
fte prog1.cpp &
```

The ampersand (&) at the end of a UNIX command runs the command and gives you the prompt back right away. This way you can continue to type more UNIX commands. It is useful to put the ampersand at the end of commands that bring up their own windows like a web browser or emacs.

<: input redirection

```
./a.out < data1
```

The less than (<) character makes an executable take input from a file instead of waiting for something to be typed at the keyboard. This is called input redirection.

For example, if *data1* above contained the following:

```
2
4.5
3.4
```

Then, the program *a.out* would behave as if we typed 2, then hit <RETURN>, typed 4.5, then hit <RETURN>, and finally typed 3.4, then hit <RETURN>.

>: output redirection

```
./a.out > output1
g++ file.cpp >& output_and_errors
```

The greater than (>) character sends the output of an executable to a file instead of putting it on the screen. This is called output redirection. For example, if the executable *a.out* above normally prints out "Hello, world!" on the screen, then the first example would cause nothing to appear on the screen, but instead, "Hello, world!" would be found in the file *output1*.

Some output from programs is meant as error messages. Sometimes these will not be redirected by greater than (>). To redirect both regular and error output to a file, use the ampersand after the greater than symbol (i.e., >&), as in the second example.

|: piping

```
./a.out | sort
g++ file.cpp |\& less
./a.out | sort | less
```

The pipe symbol (|) sends the output of an executable as the input of the command that follows it, instead of putting the output on the screen. This is called piping. For example, if the executable *a.out* above normally prints out a bunch of names on the screen, then the first example would cause those names to go as input to the sort command, and then sort would print out the names in sorted order on the screen.

Some output from programs is meant as error messages. Sometimes these will not be piped by |. To pipe both regular and error output to another program, use the ampersand after the pipe symbol (i.e., |&), as in the second example, where the output of the compiler g++ is paged with less.

You can string along several commands with pipes. In the third example, the output of the program *a.out* is first sorted and then paged with the program

3 Compiling

g++: Compile with the GNU C++ compiler

```
g++ file1.cpp file2.cpp
  OR
g++ -Wall -o prog1 prog1.cpp
```

g++ is the compiler we are using for C++ programming courses. For more complicated programs, you may want to use the *make* utility for compiling. The easiest way to compile a program (although not always the best way) is to type *g++* followed by the names of all .cpp files that make up the program (as in the first example). By default, *g++* creates an executable named *a.out*.

With *g++*, you can use the following flags (others are listed in the Manual Pages for *g++*).

For example, -o (that's oh, not zero) followed by the name for the executable. In the second example, *-o prog1* has been used to call the executable *prog1* instead of *a.out*.

The *-Wall* flag activates diagnostic warning messages which might help you from making common programming mistakes. It is a good idea to use this flag at all times.

The compiler will give you Warnings, which you should attend to, but that may not prevent your program from running, and Errors, which will prevent the compiler from generating an executable. Messages often give you the file, line number, and a description of the error as below.

```
sum.cpp:39: unterminated string or character constant
sum.cpp:32: possible real start of unterminated constant
```

Resolve the first errors in the code first since they often cause later errors.

make: Compile with the make utility

```
make
  OR
make -f hw1.mak
```

The *make* utility makes compiling bigger programs easier, more consistent, and more efficient. You run the command *make* in the directory where your source code for a program exists. By default, the *make* utility expects a file called a make file to be present in the same directory. This make file usually specifies how to compile your program. The *make* utility looks for a make file in a UNIX file named either *Makefile* or *makefile* in the directory where *make* is run. The -f option can be used to tell make to look for a different make file as in the second example.

Note the difference between a make file, which contains instructions on how to compile a program vs. the make utility, which is the program that does all the work (using a make file).

Make will display what commands it is using to compile, etc. as it goes along.

4 Editing

emacs / xemacs / fte / joe: Edit or create files

```
fte prog1.cpp &
  OR
joe prog1.cpp
```

You can write your source code with several text editors. It is a matter of personal taste which one you choose. I would recommend using *fte*, which is easy to use and lets you compile right from the editor. To do this, press *F9* and enter the command line for compiling your program (either a call to *g++* or *make*, see above). If there are any errors, you can select the first one and press enter. The view than switches to the file and line where the error occured. Other useful keys in *fte*: *F2* saves a file, *F6* cycles through open files. Use the menu for other commands.

5 Sample Session

You want to start working on your newest assignment. You decide to create a new directory for it:

```
$ mkdir hello
$ cd hello
$ fte hello.cpp &
```

In the editor window you type:

```
#include <iostream>

int main()
{
  std::cout << "Hello World! << std::endl;;
  return 0;
}
```

You save the file and back in the shell you try to compile your program:

```
$ g++ -Wall hello.cpp -o hello
hello.cpp:5: unterminated string or character constant
hello.cpp:5: possible real start of unterminated constant
$
```

Ooops, you made a mistake. You forgot the closing " for the string constant. You correct the error in the editor, remembering to save your file and try again:

```
$ g++ -Wall hello.cpp -o hello
$ ./hello
Hello World!
$
```

It worked this time and you started your first genuine program!