

I/O Redirection on the Command Line

I/O redirection is a powerful command line feature that exemplifies the following rules from Eric Steven Raymond's *The Art of Unix Programming*:

Rule of Modularity

Write simple parts connected by clean interfaces

Rule of Composition

Design programs to be connected with other programs

- One might argue, in fact, that this capability is yet to be implemented in a graphical user interface (GUI) shell with the same elegance and universality that it has on the command line — after seeing what this is all about, you be the judge
- This function is fundamental to all Unix-based command lines, so it is not specific to *bash*; further, many other command line shells on other operating systems (e.g., *cmd* on Windows) have the same functionality
- Mastering I/O redirection, especially with certain Unix-standard tools, significantly raises your productivity on the command line

Anatomy of a Command

- You might have encountered, previously, the notion that programs have a “standard in” and a “standard out” — default channels for receiving and sending information:



- The basis of I/O redirection is the command line’s ability to *route* these data streams from one program to another — thus forming a *pipe* of data across programs:



- Further, data can be sent not only across programs, but to and from files:



- In some cases, this obviates the need to include file reading/writing code in a program!
- *Redirection* (i.e., standard in/out from/to a file, respectively) is specified through the < and > characters, respectively
- *Piping* (i.e., presenting one command’s standard out as the standard in to another command) is specified via |

Output Redirection

- Output redirection is easy with a couple of the commands that we've already seen:

```
pwd > currentDirectory.txt
```

```
ls -tF > filesByDate.txt
```

- Invoking these commands will not display the results on the command line; instead, they will create the files specified after the > character (easily verifiable with *ls*)
- Note that there's no safety net here: if files with those names were there before, they will be replaced

- How do we know that the files contain exactly what would have been shown onscreen? Enter a few new commands for displaying files:
 - ◊ *cat <filename>* dumps a file's contents to the command line (watch out for long files!)
 - ◊ Paging programs or *paggers* provide an alternative to *cat* — pagers don't dump a file's content in its entirety, but pause for each "screenful" of data
 - ◊ *more* is the original pager; *less* is a more feature-rich alternative (so named since "less is more," heh) — invoke *more/less <filename>* to page through a file
- Of course, *cat/more/less* can be used with any file, not just the results of output redirection

Input Redirection

- As you might guess, input redirection looks like this:

command < file-to-use-as-input

- You might wonder if we've seen a command yet that reads from standard in, so that we can try input redirection — and yes, we *have* seen such a command:

cat < file-to-display

- Yep, *cat file-to-display*, in some ways, is a “convenience parameter” — you get equivalent results from input redirection (at the cost of two more characters typed)

- If *cat* reads from standard in, then running *cat* with no parameters should also work — and it does: if you run *cat* by itself and start typing, *cat* will display everything you type, one line at a time
- To “end” your “file,” type *control-D* on an empty line
- Interestingly, *control-D* also serves as an equivalent to *exit* when terminating a command line shell
- This is not a coincidence, because, in fact, *the command line shell reads its commands from standard input*
- Thus, one can put a sequence of commands into a file, say *script.txt*, then run those commands in this manner:

bash < script.txt

Pipes

- Redirection to and from files on a disk is quite powerful, but in the end it is just a special case of the ability to redirect input and output from *anywhere* — in particular, from other programs
- As mentioned, one can *pipe* a program's standard out to another program's standard in; for example:

```
ls | cat
```

```
cat | bash
```

```
cat long-file.java | more
```

- The above examples are somewhat trivial (though *cat* | *bash* may not behave *exactly* as you might expect); we need some new commands/programs to make pipe a little more interesting:
 - ◊ *wc* displays the number of lines, words, and characters that were sent to standard in
 - ◊ *sort* rearranges the lines sent to standard in; ascending by default, with other options available
- Remember that pipes are just redirected input/output; you can use both *wc* and *sort* without pipes or redirection, just to see what they do “live”
- We have, however, saved a couple of the most interesting (and powerful) commands for last...

Text Filtering/Finding

- *grep*, or the *general regular expression parser*, is one of the most potent tools in the command line arsenal
- Most folks start out using *grep* as a “find-by-content” tool — it can search for regular expression matches among some set of files:

```
grep "<pattern>" <files>
```

- *<pattern>* is a regular expression (best enclosed in quotes to accommodate special characters); *grep* displays the lines in the given files that match this expression

- *grep*'s regular expression syntax resembles, but is not identical to, formal regular expression notation — it adheres to a standard though, and is worth learning; some tidbits:

- ◆ `.` matches any single character
- ◆ `[characters]` matches any character between the brackets
- ◆ `^pattern` matches lines that *start* with the given pattern
- ◆ `pattern$` matches lines that *end* with the given pattern
- ◆ `pattern*` matches zero or more instances of the given pattern
- ◆ `pattern{n}` matches exactly *n* instances of the given pattern

- There are many more of course; *man grep* provides all of the gory details
- As with many other programs (and as you might/should expect at this point), *grep* works just as well over standard in, and thus I/O redirection

Example *grep* Sequences

- While already quite useful as a file searcher, *grep* gains an additional level of utility, particularly in operating system interaction, thanks to I/O redirection
- In the following sequences, we don't give specific files or patterns, but instead focus on what they can accomplish for you when used correctly
 - ◇ *history | grep* allows you to see only the recent commands that match some pattern (say, the most recent *gcc* or *javac* invocations, or the most recent commands that mention some filename)
 - ◇ *cat | grep | more* (or *less*) is a typical sequence for finding lines that contain some text, then “paging” through the results (particularly when the search space is large)
 - ◇ *grep | wc* counts the lines, words, and characters in *grep*'s filtered lines; the pipe can begin with the *grep*, or some other command such as *history*
 - ◇ *grep | sort*, as one might expect, sorts the output; as above, other commands can precede *grep*
 - ◇ *grep | grep* filters the already filtered text...it's equivalent to finding lines that match both patterns
- One caveat with *grep* is that it's line-based — you'll need to do some visual scanning for long lines of text

Text Modification

- Have you ever stared at the search/replace dialog of your favorite GUI editor or word processor, and wished that you had more flexibility or power?
 - ◇ Replace all tabs at the beginning of a line with 4 spaces
 - ◇ Eliminate trailing whitespace from the end of a line
 - ◇ Append a semicolon to the end of every line that does *not* end with a closing brace (})
- *sed*, or *stream editor*, may just give you that power, at the small cost of using it from the command line

- As with *grep*, *sed* can take specific files off the command line; note that changes are *not* written back, but sent to standard out (at this point, you should know what to do if you do want *sed*'s changes written out to a file):

```
sed "<rule>" <files>
```

- The first argument is a *sed* "rule," i.e., an action that *sed* should do...and the most common of these rules is the replace directive:

```
sed "s/<pattern>/<replacement>/g" <files>
```

- *<pattern>* uses the same regular expressions as *grep*
- The quotes aren't absolutely necessary, but helpful for accommodating special characters

sed Replacement Notes

- At its simplest, the *<replacement>* section of the "s/*<pattern>/<replacement>/g*" rule is the text with which to replace anything that matches *<pattern>*
- The availability of a single character expands this functionality quite a bit: any ampersand (&) embedded in *<replacement>* inserts the *pattern-matched text*
- Thus, you can do replacements such as these:
 - ◆ *sed "s/[aeiou]/&&/g"* doubles up all lowercase vowels
 - ◆ *sed "s/[0-9][0-9]*/&.0/g"* adds ".0" to all integer literals

- Both *sed* and *grep* actually recognize two types of regular expression syntax: *basic* and *extended/modern*; the syntax is selected through a command line parameter, and with *man re_format* providing the details
- Of course, putting *sed* in a pipe opens up a variety of command line possibilities:

```
history | grep " scp " | sed "s/^[0-9 ]*/time /g" | bash
```

- On a completely separate note, *sed*'s search-and-replace rule also makes for good shorthand when correcting instant messaging typos

- ◆ *dondi2Imu*: Your talk was really derailed.
- ◆ *hihimanuahalua*: Huh?
- ◆ *dondi2Imu*: s/rail/tail/g
- ◆ *hihimanuahalua*: Oh, I see.

Redirection Odds & Ends

- In some respects, I/O redirection also reflects

Rule of Separation

Separate policy from mechanism;
separate interfaces from engines

...note how, instead of enforcing a *policy* on how information is sent to or received from a program, I/O redirection provides a *mechanism* for doing so in a variety of ways

- Other commands that can do some useful things as part of an I/O redirection chain include:
 - ◊ *hexdump* — Displays binary files in hexadecimal
 - ◊ *cut* — Separates data according to byte position, character position or delimiters
 - ◊ *find* — Lists files that satisfy certain criteria
- Look up the use of ```-enclosed expressions in a command: these redirect the output of the enclosed expression into the command itself (e.g., `grep "JFrame" `find . -name "*.java"``)
- *xargs* is another I/O-to-command utility: it can take input text and convert that text into arguments to be passed into another program