COMP 2718: Redirection

By: Dr. Andrew Vardy

Adapted from the notes of Dr. Rod Byrne

Outline

- Redirection — Chapter 6 of TLCL
- Standard Input, Standard Output, and Standard Error
- Redirecting Standard Output
- File Desriptors
- Redirecting Standard Error
- Redirecting Both **stdout** and **stderr**
- Send your complaints to `/dev/null`
- Redirecting Standard Input
- Pipelines
- The Difference Between > and |
- Concept: Filters

Redirection — Chapter 6 of TLCL

We're going to cover **redirection** which is described in chapter 6 of the textbook. Along the way, we'll introduce the following commands:

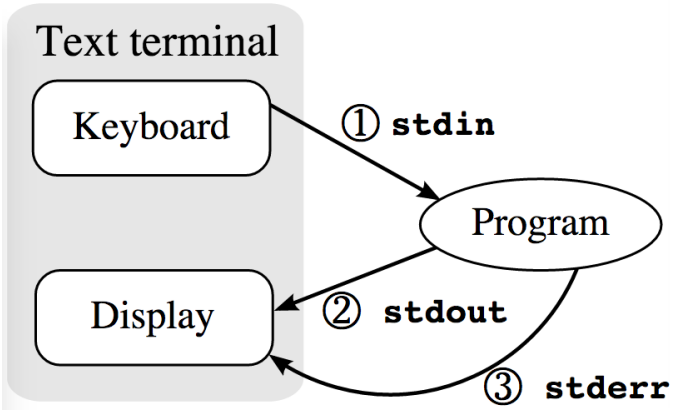
- `cat`: Concatenate files
- `sort`: Sort lines of text
- `uniq`: Report or omit repeated lines
- `grep`: Print lines matching a pattern
- `wc`: Print newline, word, and byte counts for each file
- `head`: Output the first part of a file
- `tail`: Output the last part of a file
- `tee`: Read from standard input and write to standard output and file

In this case our focus will be on the concept of redirection and the above commands will be introduced as we go.

Standard Input, Standard Output, and Standard Error

Recall the definitions of the standard streams **stdin**, **stdout**, and **stderr** (from "The OS, Shell, Terminal, and Text"):

- **standard input** (stdin) is the input that comes from the keyboard
- **standard output** (stdout) is the program's output displayed by the terminal
- **standard error** (stderr) are the program's error/debug messages displayed by the terminal



We are now going to look into **redirecting** these streams.

Redirecting Standard Output

To redirect standard output to another file instead of the screen use the > operator followed by a filename:

```
command > filename
```

For example, to list the contents of `/usr/bin` and store the results to `ls-output.txt` instead of displaying those results, do the following:

```
$ ls -l /usr/bin > out.txt
```

What happens if the directory specified is non-existent (i.e. there is an error):

```
$ ls -l /nonexistent > out.txt
```

```
ls: /nonexistent: No such file or directory
```

That makes sense, but what about `out.txt`?

```
$ ls -l out.txt
```

```
-rw-r--r--  1 av  wheel  0 26 Jan 15:11 out.txt
```

The file is empty. Meanwhile, why didn't the error message ("No such file or directory") go into `out.txt`?

Because the error message goes to **standard error**.

We often want to append to an existing file, as opposed to overwriting it. For this we use the >> operator.

```
command >> filename
```

For example, the following can be used to add items to a "To Do" list:

```
$ echo "My To Do List" > todo.txt

$ echo "--------------" >> todo.txt

$ echo "- Buy milk" >> todo.txt

$ echo "- Walk dog" >> todo.txt

$ echo "- Learn redirection" >> todo.txt
```

File Desriptors

A Unix **file descriptor** is an identifier for a file or other input/output stream. It could represent input from the keyboard (standard input), output to the display (standard output), or even more exotic things like a network connection.

File descriptors are really just integers and 0, 1, and 2 are assigned to the standard streams:

File Descriptor	Stream
0	Standard Input (stdin)
1	Standard Output (stdout)
2	Standard Error (stderr)

Redirecting Standard Error

To redirect standard error to another file instead of the screen use the 2> operator followed by a filename:

```
command 2> filename
```

Why 2>? Because file descriptor 2 is standard error. So the following should be equivalent to our previous example of redirecting to standard output:

```
$ ls -l /usr/bin 1> out.txt
```

This works!

Redirecting Both **stdout** and **stderr**

To capture all of a command's output we may wish to redirect both standard output and error to one file. Here is one method:

```
command > filename 2>&1
```

Yes that does look weird! First is the redirection of standard output with >. Second, standard error (2) is redirected to standard output (1), hence the notation `2>&1`.

Note that reversing the order of > and 2>&1 does not generate the right results.

```
$ ls -l /usr/bin 2>&1 out.txt >
```

This doesn't make sense because the final > should be directed to a file. So how about this?

```
$ ls -l /usr/bin 2>&1 > out.txt
```

The 2>&1 redirects stderr to stdout (currently the screen). The > redirects stdout to the file as before. Executing this is legal but stderr will go to the screen, not the desired file.

There is a newer, more streamlined syntax for redirecting both stdout and stderr to the same file:

```
command &> filename
```

e.g.

```
ls /bin &> out.txt
```

Redirecting to separate files.

Meanwhile, it remains quite possible to keep stdout and stderr separate: e.g.

```
ls /bin > out.txt 2> err.txt
```

Send your complaints to `/dev/null`

In Unix and Unix-like operating systems, there is a saying:

> *"Everything is a file."*

This means that all sorts of input/output resources such as modems, keyboards, printers, and even network connections can be treated as files.

Even **nothing** is a file. Anything sent to the "file" `/dev/null` is simply discarded.

This is sometimes useful for discarding unwanted output. For example, to discard all error/status messages from an `ls` command, redirect stderr to `/dev/null`:

```
ls /bin > out.txt 2> /dev/null
```

`cat` - Concatenate Files

The next natural topic is redirecting standard error. But we will introduce the very useful command `cat` first. `cat` simply displays (to stdout) the file(s) passed in as arguments:

```
cat [file...]
```

This implies that there may be multiple files as arguments, possibly even none. Can be used to display text, such as `todo.txt` created earlier:

```
$ cat todo.txt
My To Do List
-------------
- Buy milk
- Walk dog
- Learn redirection
```

`cat` really stands for “concatenate” which means to link together. The following example shows how `cat` is used to join separate files together:

```
$ echo "My To Do List" > l1.txt

$ echo "--------------" > l2.txt
$ echo "- Buy milk" > l3.txt
$ echo "- Walk dog" > l4.txt
$ echo "- Learn redirection" > l5.txt

$ cat l1.txt l2.txt l3.txt l4.txt l5.txt > todo2.txt
```

Or use globbing:

```
$ cat l* > todo2.txt
```

Remember that `cat` accepts 0, 1, or many arguments. What about 0? In this case, `cat` expects input from the keyboard.

```
$ cat
Type some text
Type some text
[Hit control-D to generate an end of file (EOF)
 character when finished]
```

With 0 arguments, `cat` simply copies stdin to stdout. The first copy of "Type some text" was what I typed—this went into stdin. The second copy was stdout coming out.

Redirecting Standard Input

Redirect standard input to come **from** a file instead of the keyboard:

```
command < filename
```

For example:

```
cat < todo.txt
```

The command `uniq` removes repeated lines from its input. Lets start our example with a redundant listing of fruit:

```
echo "apples" > fruit.txt
echo "apples" >> fruit.txt
echo "oranges" >> fruit.txt
```

The following feeds `fruit.txt` to `uniq`, redirects the output to `fruit2.txt`.

```
uniq < fruit.txt > fruit2.txt
```

It would be simpler to execute the following:

```
uniq fruit.txt > fruit2.txt
```

Read man page for `uniq` to understand why these are equivalent.

Pipelines

Suppose you want to sort the output from some command. There is a `sort` function that can do this for you (see `man sort`). The following is one way of achieving this sort on a directory listing form `ls`:

```
$ ls /bin > /tmp/m
$ sort /tmp/m
$ rm /tmp/m
```

(Actually the output from `ls` is already sorted in alphabetical order, but lets assume that `sort` provides some alternative sorting that is preferred.)

Note the creation of a temporary file in `/tmp` to store the intermediate result. This is an extremely common workflow. We can achieve the same result with a **pipeline**:

```
$ ls /bin | sort
```

A pipeline is formed with the *pipe* operator | which connects the standard output of one command to the standard input of another:

```
command1 | command2
```

In fact, we can pipeline many commands together. For example, to allow the result of the previous example to be viewed with `less` do the following:

```
$ ls /bin | sort | less
```

The Difference Between > and |

The essential difference is that the redirection operator > connects a command with a file, while the pipe operator | connects the output of one command with the input of another:

```
command > file
```

```
command1 | command2
```

If you were to do the following (bad idea):

```
command1 > command2
```

. . . this would create a file called `command2`. If you were in the same directory as the actual `command2`, it would be overwritten!

`wc` - Print Line, Word, and Byte Counts

Prior to showing some more interesting examples of pipelines, we'll introduce some more useful commands. The first is `wc` which displays the number of lines, words, and bytes of a file. For example:

```
$ wc todo.txt
       5      14      70 todo.txt
```

The option `-l` just prints the number of lines:

```
$ wc -l todo.txt
       5 todo.txt
```

grep - Print Lines Matching a Pattern

grep looks for patterns of text in files, or stdin:

```
grep pattern [file...]
```

pattern is expressed as a **regular expression**, a sophisticated language for pattern matching. We will cover regular expressions later, but will focus for the moment on simple strings.

grep can look for patterns inside files. e.g.

```
$ grep milk todo.txt
```

```
- Buy milk
```

Without arguments grep looks for the pattern in stdin, which allows pattern matching on the names of files or anything else!

```
$ ls /usr/bin | grep latex
pod2latex
pod2latex5.16
```

uniq - Report or Omit Repeated Lines

The uniq command accepts sorted data from either an input file or stdin and simply removes any duplicate lines when displaying to stdout.

For an example, note that some of the same files exist in both /bin and /usr/bin. The following pipeline allows you to view (using less) the complete contents of both directories:

```
$ ls /bin /usr/bin | sort | less
```

We can add uniq to eliminate duplicate entries.

```
$ ls /bin /usr/bin | sort | uniq | less
```

uniq has a -d option which means to display the duplicates and suppress everything else. So the following shows only the duplicated commands:

```
$ ls /bin /usr/bin | sort | uniq -d | less
```

`head` / `tail` - Print First / Last Part of Files

To view only the first few lines of a file, use `head`. To view only the last few lines, use `tail`. Both commands take "few" as meaning 10 by default, but these can be altered with the `-n` option:

```
$ head -n 2 todo.txt
My To Do List
-------------
```

Actually, you can abbreviate even further: `head -2`. This also works with `tail`:

```
$ tail -3 todo.txt
- Buy milk
- Walk dog
- Learn redirection
```

We can combine `head` and `tail` with a pipeline to extract ranges of entries:

e.g. Extract the first "To Do" item on line 3:

```
$ tail -3 todo.txt | head -1
```

e.g. Extract lines 3 and 4:

```
$ head -4 todo.txt | tail -2
```

or

```
$ tail -3 todo.txt | head -2
```

`tail` also has the very useful feature of allowing you to view changes made to a file in real-time with the `-f` option.
Try the following:

```
$ tail -f /var/log/syslog
```

while this is running, try disconnecting and reconnecting your internet connection, or make some other change to your system.

`tee` - Split Input to Both **stdout** and File(s)

Just like a 'T' pipe which connects a single source of water to two outputs, the `tee` command splits stdin so that it goes to both stdout and to some file(s). This is often useful if we want to capture some intermediate result.

For example, the following lists all of the files in `/usr/bin/` which include the "zip" in their names:

```
ls /usr/bin | grep zip
```

Maybe we also want to capture the listing of all files, prior to `grep`. We could do the following:

```
ls /usr/bin | tee ls.txt | grep zip
```

Concept: Filters

We can think of pipelines of multiple commands as filters. Each command changes its input somehow and then passes on the result.