The Terminal as a File

- Earlier, I said that Unix thinks of almost everything as a file
    - Directories are files, as far as Unix is concerned
    - So are *printers* and *disk drives*
- Once upon a time, computers were expensive and rare
- Most computers had multiple terminals connected to them
    - This allowed more than one person to use the computer at any one time
    - Each of these terminals was a separate input and output device
    - Unix was created to work in an environment where one machine connected to many terminals

The Terminal as a File

- A terminal can be a physical device like a keyboard and monitor, or it can be an ***ssh*** session coming in from another machine
- You can have several ***ssh*** session windows going at once
  - Each window is connected to the same remote machine but is a different login session and each login session has its own terminal "file"
  - To find up what terminal you are using in your session, use the ***tty*** command

```
$ tty
/dev/pts/17
```

- In the case above, I was using terminal 17
- **This** ***tty*** is **not** the device driver ***tty***

The Keyboard and Screen as Standard Input and Standard Output

- By default...
  - standard input is taken from the keyboard,
  - standard output goes to the screen,
  - and standard error also goes to the screen
- The *cat* utility expects you to give it the name of the file you want to print to the command line
- What happens when you don't give it a file name as an argument?
  - In this case, *cat* will accept input from standard input which, by default, is the keyboard

- If you run *cat* without specifying a file it will simply echo what you type:

```
$ cat
foo
foo
bar
bar
bletch
bletch
^D
```

Redirection

- When I had you create a `.forward` file, I told you to use

```
cat > .forward [Enter]
YOUR_EMAILADDRESS [Enter]
[Control-D]
```

- This trick allows you to use *cat* as a simple text editor
  - But, it won't allow you to backspace
  - This is an example of redirection
  - By using the greater than character **>** we are telling *cat* to send output to the file `.forward` instead of printing it to the screen

Redirection

- ***Redirection*** is when you tell Unix to **take data from** or **send data to** some other "file" then it would normally use
- In the above example, we have redirected standard output
- Instead of sending the output from *cat* to the terminal, we are sending it to the file `.forward`
- Redirection is one of the features that makes Unix *flexible*
  - It allows you to take input from or send output to any file you wish
  - You can take input from something other than the keyboard like a file
  - You send output to something other than the terminal such as a file

Redirection

- Redirection is what makes pipes possible
  - When you set up a pipe you are sending the output of *one* program into the input of *another*
  - You are redirecting the output of the first command *from* the terminal *to* the input of the second command

Redirecting Standard Output

- To redirect *output*, use the greater than symbol **>** followed by a *filename*
- This tells Unix to send the output from the *command* to the *file or device* that appears after the symbol
- The format for output redirection is

```
COMMAND [ARGUMENTS] > FILENAME
```

- For example, to save a list of everyone currently logged on, you could use

```
$ who > current_logins.txt
```

Redirecting Standard Output

- That way, the output from **who** is preserved as a text file for whatever purpose you may use it:

```
$ cat current_logins.txt
bmt11989 pts/1    2011-10-02 16:43 (c-24-147-18-
                          10.hsd1.ma.comcast.net)
vtran    pts/0    2012-09-26 17:34 (c-76-119-98-
                          173.hsd1.ma.comcast.net)
abutawha pts/1    2012-09-26 17:36 (158.121.234.175)
ghoffmn  pts/2    2012-09-26 18:19 (dsl092-066-
                          161.bos1.dsl.speakeasy.net)
```

Redirecting Standard Input

- When redirecting standard output, we were sending output to something other than the terminal
- When we redirect standard input, we take input from something other than the keyboard
- To do this, we use the less than symbol **<**
- Here is the format:

```
COMMAND [ARGUMENTS] < FILENAME
```

- `repeat.sh` is a shell script that repeats the text the user enters:

Redirecting Standard Input

```
$ ./repeat.sh
Enter several lines
Type X on a line by itself
when done
asdfasd
1234132
asdfasd
1234
X

You entered
-----------
asdfasd
1234132
asdfasd
1234
X
```

- But...I can also take input *from* a file by redirecting standard *input*

```
$ ./repeat.sh < test.txt
Enter several lines
Type X on a line by itself
when done

You entered
-----------
123456789
abcdefg
987654321
hijklmnop
foo
bar
bletch
X
```

Redirecting Standard Input

- We used input from this file:

```
$ cat test.txt
123456789
abcdefg
987654321
hijklmnop
foo
bar
bletch
x
```

Redirecting Standard Output Can Destroy a File

- If you redirect standard output to a file that already exists, you will overwrite the contents of that file
- You will replace the original contents of the file with the output of the new command
- There is a "***noclobber***" option in Bash to prevent this from happening
- But, it is best to simply ***be careful*** about the file to which you redirect standard output

Adding Output to an Existing File

- If you redirect standard output to a file that already exists, you will lose the original contents of that file
- But Unix allows you to add something to the bottom of a file
- This is called **appending**
- The append symbol is two greater than symbols with no space in between **>>**
- The format is

```
COMMAND [ARGUMENTS] >> FILENAME
```

Adding Output to an Existing File

- For example:

```
$echo foo > test.txt

$ cat test.txt
foo

$ echo bar >> test.txt

$ cat test.txt
foo
bar
```

- Notice that "foo" is still in the file, and "bar" is on the following line

`/dev/null`

- Sometimes a program will do something useful but produce output you don't want
- For situations like this, Unix provides /**`dev/null`**
  - Any output you send to **`/dev/null`** will disappear
  - It will never appear on the screen
  - If you redirect input to come *from* **`/dev/null`** the command will receive an empty string
- One way to think of **`/dev/null`** is to imagine that you are redirecting output to a destination of "nothingness"

`/dev/null`

- `/dev/null` is most useful when dealing with error messages
  - Since error message *normally* go to the terminal, they will be mixed up with the regular output
  - Redirecting standard error to `/dev/null` will prevent this from happening
  - I will show you how to do this in a future class
- In fact, you already have some experience using `/dev/null` – specifically, for the purpose of testing your exercise and homework scripts!