Summer 2013 Handout 2

Message of the day: pp. 4, 64; login(1) pp. 2, 5

```
1$ cd /etc                               et cetera: the miscellaneous directory
2$ pwd

3$ ls -l | more                          List all the files in the current directory.
4$ ls -l motd                            List the one file named motd.
-rw-r--r--   1 root     sys           790 Aug 27  2012 motd

5$ cat motd                              relative pathname
###############################################################################
#
#  ------------------- Message last updated [2012-08-20] -------------------
#
#  Welcome to i5.nyu.edu

6$ ls -l /etc | more                     You can list a directory without cd'ing there: Handout 1, p. 10.
7$ ls -l /etc/motd                       You can list a file without cd'ing to its directory.
8$ cat /etc/motd                         full pathname: can read a file without cd'ing to its directory
```

Other interesting files to read

```
1$ cat /usr/pub/ascii                p. 42; see ascii(5). /usr/share/misc/ascii on Mac OS X
2$ more /usr/pub/ascii               press space bar, b, or q
3$ lpr -Pedlab /usr/pub/ascii

4$ more /usr/dict/words                              p. 104
5$ grep -i atlantic /usr/dict/words                  case insensitive: -y on p. 85
6$ grep -i atlantic /usr/dict/websters               ten times as many lines

7$ grep -n Toad ˜mm64/public_html/INFO1-CE9545/src/Shakespeare.complete
93399:AJAX  Toadstool, learn me the proclamation.
118425: I cannot tremble at it: were it Toad, or
151151:APEMANTUS Toad!
159867: Toad, that under cold stone

8$ grep -in Toad ˜mm64/public_html/INFO1-CE9545/src/Shakespeare.complete | more

9$ head -5 /usr/dict/words                           default is -10
10$ tail -5 /usr/dict/words                          pp. 19−20
```

Give two arguments to **awk**; enclose the first in **’**single quotes**’** (p. 75). See p. 116 for **NR**, pp. 116−117 for **<=**, and p. 120 for **&&**.

```
11$ awk '159865 <= NR && NR <= 159873' \
    ˜mm64/public_html/INFO1-CE9545/src/Shakespeare.complete
```

/etc/passwd (p. 53) and /etc/group (p. 54)

```
1$ grep abc1234 /etc/passwd
root:x:0:0:root@i5:/root:/usr/bin/bash
daemon:x:1:1::/:
mm64:x:50766:15:Mark Meretzky:/home1/m/mm64:/bin/ksh

2$ grep 15 /etc/group | more                          better yet, :15:
root::0:
other::1:root
users::15:
```

▼ **Homework 2.1: look at your line in /etc/passwd (not to be handed in)**

On each machine where you have an account, look at your line in the **`/etc/passwd`** file. Does

```
1$ finger abc1234
```

display the text in the fifth field of your line in **`/etc/passwd`**? Can you change this text with the **`passwd -f`** or **`chfn`** (“chow fun”) commands?

What is the name of your shell in the seventh field? Can you change it with the **`passwd -s`** or **`chsh`** commands? If so, change it back.

What is the number of the group to which you belong? Look at the line for your group in the **`/etc/group`** file. What is the name of the group? Do you belong to any other groups?

```
2$ grep abc1234 /etc/group
```

▲

Unix shells: p. 100

name	*exe-cutable*	*author*	*how to log out*	*name of login file*	*exit status variable*	*website*
Bourne	**`sh`**	Steve Bourne	**`exit`**	**`.profile`**	**`$?`**	
Korn	**`ksh`**	David Korn	**`exit`**	**`.profile`**	**`$?`**	**`www.kornshell.com`**
Bourne Again	**`bash`**	Brian Fox	**`exit`**	**`.bash_profile`**	**`$?`**	**`www.gnu.org/software/bash/`**
Z	**`zsh`**	Paul Falstad	**`exit`**	**`.zprofile`**	**`$?`**	**`www.zsh.org`**
C	**`csh`**	Bill Joy	**`logout`**	**`.login`**	**`$status`**	
Tenex	**`tcsh`**	Christos Zoulas	**`logout`**	**`.login`**	**`$status`**	**`www.tcsh.org`**

`bash` and **`zsh`** have the most features. **`sh5`** is a newer version of **`sh`**.

Read other sections of the online manual: pp. 308−309

To read the manual pages for the files **`/etc/passwd`** and **`/etc/group`** (and **`/etc/shadow`**),

```
1$ man -s 4 passwd        manual for the passwd file; other systems don't need the -s.
2$ man passwd             manual for the passwd program
3$ man -s 4 group
4$ man -s 4 shadow        Find out where the passwords are stored.
```

because files are in Section 4 of the manual, programs in Section 1.

The nine sections of the manual

The manual has nine main sections, stored on i5.nyu.edu in the subdirectories of the directories listed in your **`$MANPATH`** variable.

```
1$ echo $MANPATH
/usr/apache2/2.2/man:/usr/dtrace/DTT/Man:/usr/gcc/4.5/share/man:/usr/gnu/share/man:/usr
```

To locate them on other systems, say **`man man`**. To see our manual in PDF format, go to **`http://i5.nyu.edu/~mm64/man/`**.

(1) Unix programs: **`who`**, **`grep`**, **`ls`**.

(2) Unix system calls: **`fork`**, **`exec`**, **`open`**, **`close`**. See pp. 201−231.

(3c) C functions: **`printf`**, **`scanf`**, **`getchar`**, **`strlen`**, **`sqrt`**. See pp. 171−200.

(3c++)C++ classes, objects, and functions: **`vector`**, **`iterator`**, **`sort`**, **`greater`**.

(4) File formats. For example, **`passwd`**(1) is the page for the **`passwd`** command; **`passwd`**(4) is the page for the **`/etc/passwd`** file.

(5) Miscellaneous: **`environ`**(5) describes the environment variables (**`$HOME`**, **`$PATH`**, etc.); **`filesystem`**(5) describes the most important directories (**`hier`**(4) in other systems).

(6) Games in the **`/usr/games`** directory: **`backgammon`**, **`banner`**, **`fortune`**.

(7) Hardware devices in the **`/dev`** directory. For example, **`termio`**(7i) for terminals, **`mtio`**(7i) for magnetic tape I/O, **`null`**(7d) for **`/dev/null`**, etc. Also protocols: **`tcp`**(7p) for TCP, etc. See pp. 65−69.

(8) System administration utilities used mostly by the superuser.

(9) Device drivers.

Each section has an introduction:

```
2$ man -s 1 intro
3$ man -s 2 intro
4$ man -s 3 intro                    etc.; also sections 1m, 9e, 9f, 9s
```

Copy a file: pp. 16−17

```
1$ cd /third/directory               (This directory is imaginary.)
2$ pwd                               Make sure you arrived there.

3$ cp /old/directory/oldfile /new/directory/newfile
4$ ls -l /new/directory/newfile      Make sure you copied the file.
5$ chmod the newfile, if necessary   or give the -p ("preserve") option to cp
```

If the **`/new/directory`** does not already exist, you must first create it with **`mkdir`**. **`cp`** will not create it for you. You must have **`r`** permission for the **`oldfile`** and **`w`** permission for the **`/new/directory`**.

As shown above, the new file does not have to have the same name as the old file, and does not have to be in the same directory as the old file. You do not have to be in either of these directories when you perform the copy. The following abbreviations are available.

(1) In the above example, the old file was named **`oldfile`** and the new file was named **`newfile`**. If both files have the same name, you can omit the last slash and the name after it from the end of the command. For example, line 7 does the same thing as line 6. So does line 8, if you are already in the **`/new/directory`**.

Warning: if the directory **`/new`** already exists but the directory **`/new/directory`** does not, line 7 will create a copy of **`oldfile`** named **`/new/directory`**. To avoid this, write line 9.

```
6$ cp /old/directory/oldfile /new/directory/oldfile
7$ cp /old/directory/oldfile /new/directory
8$ cp /old/directory/oldfile .
9$ cp /old/directory/oldfile /new/directory/
```

In any Unix command, a single dot stands for the name of the current directory, just as two dots stand for the name of the current directory's parent. See Handout 1, p. 9.

(2) In any Unix command, you can omit the name of the current directory from the front of the name of either file. See Handout 1, p. 9. For example, line 13 does the same thing as line 12.

```
10$ cd /new/directory
11$ pwd

12$ cp /old/directory/oldfile /new/directory/newfile
13$ cp /old/directory/oldfile newfile

14$ ls -l newfile
```

In the next example, line 18 does the same thing as line 17.

```
15$ cd /old/directory
16$ pwd

17$ cp /old/directory/oldfile /new/directory/newfile
18$ cp oldfile /new/directory/newfile

19$ ls -l /new/directory/newfile
```

If both files are in the current directory, you can omit the name of the current directory from both names. See the **cp** example on p. 17.

(3) You can copy more than one file with a single **cp** command, if all of the copies are to be deposited in the same directory and if they are all to have the same names as their originals.

```
20$ cd /old/directory                                   for simplicity
21$ pwd

22$ cp oldfile1 oldfile2 /new/directory                  Copy two files.
23$ cp * /new/directory               Copy every file in your current directory.

24$ ls -l /new/directory | more
```

But to put copies into different directories, you must type a separate **cp** command for each directory. And to make the new filename different from the old filename, you must type a separate **cp** command for each individual file.

To copy entire directories, together with the files in them, see the **-r** option of **cp**.

Move and/or rename a file or directory

(4) **mv** is the same as **cp**, except that the original file does not get left behind. This means that **mv** can move a file from one directory to another, and also rename a file. We will see that **ln**, p. 59, takes the same arguments, too.

```
1$ mv /old/directory/oldname /new/directory/newname    move and rename
2$ mv /old/directory/oldname /new/directory            move without renaming
3$ mv oldname newname                                  rename without moving
4$ mv * /new/directory                                 move without renaming
```

(5) You can also move and/or rename a directory as well as a file:

```
5$ cd
6$ pwd
```

```
7$ mkdir public-html          Handout 1, p. 13. The dash should have been an underscore.
8$ ls -ld public-html         Without -d we'd see the files, not the directory.

9$ mv public-html public_html          Correct the mistake.
10$ ls -ld public_html
```

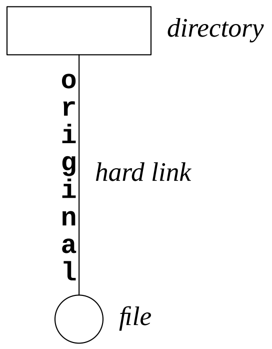
Remove a file: p. 17

```
1$ rm file             Requires w permission for file's directory; question if no r permission for file.
2$ rm file1 file2 file3
3$ rm *                Remove all files in the current directory.

4$ rm *.html           Remove all files whose names end with .html.
5$ rm junk*            Remove all files whose names begin with junk.
6$ rm *junk*           Remove all files whose names contain junk, not necessarily in the middle.

7$ rm -- -filename     Remove a file whose name begins with a dash.
8$ rm '*'              Remove the file whose name is *: pp. 26−29.
```

An additional hard link to the same directory: pp. 59−60

```
1$ cd
2$ pwd
/home1/a/abc1234

3$ date > original
4$ ls -l original
-rw-------   1 mm64     users         29 May 28 15:18 original

5$ cat original
Tue May 28 15:18:10 EDT 2013
```

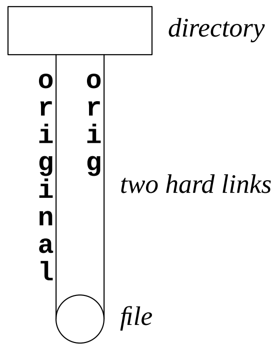
The line that connects each file or directory to the directory that contains it is called a *hard link*. The name of the file or directory is written along the link, rather than in the file or directory itself. That explains why the size of a file or directory doesn't change when you rename it. It also explains why the root directory can have no name: there's no link extending upwards from it.



Every file is born with exactly one link, leading to the directory that contains it. At all times, the file has at least one link. There is no such thing as an unattached file floating on the desktop, because there is no desktop.

To endow a file with another link, containing another name,

```
6$ ln original orig                    lowercase LN: natural logarithm
7$ ls -l original orig                 Now the file has two hard links.
-rw-------   2 mm64     users            29 May 28 15:18 orig
-rw-------   2 mm64     users            29 May 28 15:18 original

8$ ls -li original orig                See the inode ("index node") numbers (p. 58).
 273711328 -rw-------   2 mm64     users         29 May 28 15:18 orig
 273711328 -rw-------   2 mm64     users         29 May 28 15:18 original

9$ cat orig
Tue May 28 15:18:10 EDT 2013
```



```
10$ rm original
11$ ls -l orig                          orig is still there, but now has only one hard link left.
-rw-------   1 mm64     users            29 May 28 15:18 orig

12$ cat orig
Tue May 28 15:18:10 EDT 2013

13$ rm orig
14$ ls -l | more
```

Why would you want to give an additional name to a file?

```
1$ cd /usr/bin
2$ ls -li emacs-x* | more
     69273 -r-xr-xr-x   2 root     bin      25233592 Jul 19  2012 emacs-x
     69273 -r-xr-xr-x   2 root     bin      25233592 Jul 19  2012 emacs-x-23.1
```

After we installed **`emacs-x-23.1`**, we created an additional name for it like this:

```
3$ cd /usr/bin
4$ pwd

5$ ln emacs-x-23.1 emacs-x
6$ ls -li emacs-x-23.1 emacs-x
```

After we install **`emacs-x-23.2`**, we will say

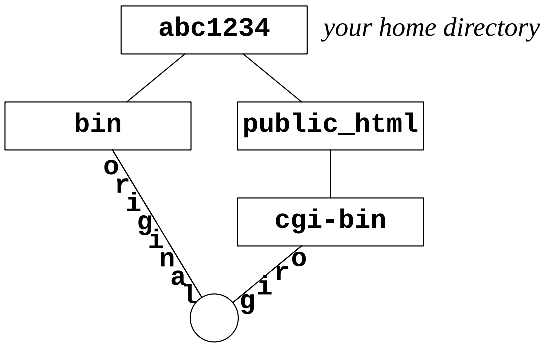
```
7$ cd /usr/bin
8$ pwd
```

```
9$ rm emacs-x
10$ ls -li emacs-x emacs-x-23.1     emacs-x-23.1 will still be there.

11$ ln emacs-x-23.2 emacs-x
12$ ls -li emacs-x-23.2 emacs-x
```

▼ Homework 2.2: how many files are there?

The directory `~mm64/public_html/INFO1-CE9545/src` contains **`ike`** and **`mike`**. Are they merely two hard links to the same file? Or are they two separate files that happen to have the same contents, permissions, owner, and date? What about **`barry`** and **`larry`**? What about **`ed`** and **`fred`**?
▲

An additional hard link to a different directory



`original` is in **`~/bin`**, **`orig`** is in **`~/public_html/cgi-bin`**.

```
1$ cd ~/bin
2$ pwd
/home1/a/abc1234/bin

3$ date > original
4$ ls -l original
-rw-------   1 mm64     users         29 May 28 15:18 original

5$ cat original
Tue May 28 15:18:10 EDT 2013

6$ cd ../public_html/cgi-bin      Go one level up and then two levels down; see below.
7$ pwd
/home1/a/abc1234/public_html/cgi-bin

8$ ln ../../bin/original orig           Go two levels up and then one level down.
9$ ls -l ../../bin/original orig
-rw-------   2 mm64     users         29 May 28 15:18 ../../bin/original
-rw-------   2 mm64     users         29 May 28 15:18 orig

10$ cat orig
Tue May 28 15:18:10 EDT 2013

11$ rm orig ../../bin/original
```

You can break the above line 6 into the following two separate commands, but why would you want to?

```
12$ cd ..
13$ cd public_html/cgi-bin
```

A hard link will stretch only so far

A hard link cannot connect a file to a directory in another disk or filesystem (‘‘file subsystem’’ on p. 67, i.e., disk partition):

```
1$ cd
2$ date > junk

3$ ln junk /tmp
ln: /tmp/junk is on a different file system
```

`df` shows which filesystem **`junk`** and **`/tmp`** are on; see Handout 1, p. 3.

```
4$ df -k junk /tmp
Filesystem           1024-blocks        Used   Available Capacity  Mounted on
i5pool/home1/m/mm64      2097152     1759364      337787    84%    /home1/m/mm64
swap                     8388608      733184     7655424     9%    /tmp
```

A symbolic link to another file in the same directory

```
1$ cd
2$ pwd
/home1/a/abc1234

3$ date > original
4$ ls -l original
-rw-------   1 mm64     users         29 May 28 15:18 original

5$ cat original
Tue May 28 15:18:11 EDT 2013
```

A *symbolic link* is a one-line file that leads you to (i.e., contains the name of) another file or directory. It’s like a shortcut in Windows or an alias on a Macintosh. The nine bits of a symbolic link are always ignored; they’re on all the time and do nothing. **`ls -l`** shows you the one-line contents after the arrow:

```
6$ ln -s original orig                Lowercase LN: create orig.
7$ ls -l original orig                Why does orig contain eight bytes?
lrwxrwxrwx   1 mm64     users          8 May 28 15:18 orig -> original
-rw-------   1 mm64     users         29 May 28 15:18 original

8$ cat orig
Tue May 28 15:18:11 EDT 2013

9$ rm original                        You should have removed orig before removing original:
10$ cat orig                          orig is now a dangling link.
cat: cannot open orig: No such file or directory

11$ rm orig
```

A symbolic link to a file in a different directory

`original` is in `~/bin`, `orig` is in `~/public_html/cgi-bin`.

A symbolic link can lead you to a file in a different directory or even in a different filesystem. A hard link, however, must lead you to a file in the same disk.

```
1$ cd ~/bin
2$ pwd
/home1/a/abc1234/bin

3$ date > original
4$ ls -l original
-rw-------   1 mm64     users         29 May 28 15:18 original

5$ cat original
Tue May 28 15:18:11 EDT 2013

6$ cd ../public_html/cgi-bin               Go one level up and then two levels down.
7$ pwd
/home1/a/abc1234/public_html/cgi-bin

8$ ln -s ../../bin/original orig           Go two levels up and then one level down.
9$ ls -l ../../bin/original orig
-rw-------   1 mm64     users         29 May 28 15:18 ../../bin/original
lrwxrwxrwx   1 mm64     users         18 May 28 15:18 orig -> ../../bin/original
```

When you follow a symbolic link containing a relative pathname, you start at the directory that contains the symbolic link. A `.` in a symbolic link therefore stands for the directory that holds the symbolic link, and a `..` in a symbolic link stands for the parent of the directory that holds the symbolic link.

```
10$ cat orig
Tue May 28 15:18:11 EDT 2013

11$ rm orig
```

We can also have a symbolic link that contains a full pathname.

```
12$ ln -s ~/bin/original orig
13$ ls -l orig
lrwxrwxrwx   1 mm64     users         26 May 28 15:18 orig -> /home1/m/mm64/bin/original

14$ rm orig ../../bin/original
```

A symbolic link to a directory

Usually `ls` and `ls -l` output the same number of items. But

```
1$ cd /
2$ pwd

3$ ls -l bin                            See the symbolic link itself.
lrwxrwxrwx   1 root     root             9 Jul  5  2012 bin -> ./usr/bin

4$ ls bin | head -3                     See the files in the ./usr/bin directory
2to3
7z
7za
```

▼ Homework 2.3: where do these symbolic links lead?

Do these symbolic links contain full pathnames (starting with `/`) or relative pathnames (not starting with `/`)? See Handout 1, p. 9 for full vs. relative.

Do these symbolic links lead to a file or to a directory?

```
1 /bin
2 /dev/msglog
3 /usr/man
4 /usr/openwin/lib/rgb.txt
```

✎✎ For space cadets only. Consider

```
1$ cd /bin
2$ pwd
/bin

3$ ls -l java
lrwxrwxrwx   1 root     root          16 Jul 11  2012 java -> ../java/bin/java
```

The `..` takes us from the directory `/bin` up one level to the root directory. But then we discover that the root directory contains nothing named `java`. Did the symbolic link therefore lead us to a nonexistent place? Ditto for `/bin/perl`.
▲

Create a Korn shell alias:

```
1$ alias g=grep          Create an alias named g.  No space around the equal sign.
2$ alias g               Verify that you created it.
3$ alias | more          See all your alias's.

4$ g atlantic /usr/dict/words
atlantic
transatlantic

5$ unalias g                          Remove the alias g.
6$ alias g                            Verify that it's gone.
```

An alias that needs single quotes

An alias needs single quotes when it contains more than one word, i.e., when it contains white space:

```
1$ alias g='grep -i'
2$ alias g
3$ g atlantic /usr/dict/words
Atlantic
atlantic
Atlantica
transatlantic
```

An alias also needs single quotes when it contains the name of more than one program, i.e., when it contains a semicolon or pipe:

```
4$ alias g='date; cal; who | grep abc1234'
5$ alias g
6$ g
```

Aliases vs. variables

An alias is easier to use than a variable: you don't have to type the dollar sign.

```
1$ alias g=grep                      Create an alias named g.
2$ g atlantic /usr/dict/words        doesn't need a dollar sign

3$ g=grep                            Create a variable named g.
4$ $g atlantic /usr/dict/words       needs the dollar sign
```

But an alias can be used only as the *first* word of a command, while a variable can be used anywhere in a command:

```
5$ cd $S45                           The variable $S45 will contain a directory name.
```

Use an alias for the name of a command; use a variable for the name of a file or directory.

Korn Shell emacs mode abbreviations to retrieve and repeat the previous commands

control-p means "press the control key *at the same time as* you press the **p**". Do not type the dash. **escape-.** means "press and release the escape key *before* you press the dot". Again, do not type the dash. See "**emacs** Editing Mode" in **ksh**(1) pp. 28−32.

Line 1 puts you into **emacs** mode; see **ksh**(1) p. 48.

```
1$ set -o emacs                Turn on the Korn Shell emacs option; minus lowercase O
2$ set -o | more               See all your options; verify that the above worked.

3$ control-p                         Go up to the previous command. Then press RETURN.
4$ control-p control-p control-p     no space between the control-p's
5$ control-n                         Back too far? Go down to next command, then press RETURN.

6$ fortune
7$ control-p                   You get a different fortune each time.

8$ lpr -Pedlab neuman          no longer need name of printer, thanks to $PRINTER variable
9$ lpq -Pedlab
10$ control-p                  Has the printer made any progress?
```

Extra material can be appended to any of the above. For example,

```
11$ ls -l                      Too much output to fit on the screen!
12$ control-p | more

13$ ls -l                      I wish I'd printed the output.
14$ control-p | lpr -Pedlab
```

To position yourself for inserting and deleting characters, type **control-f** ("forward") and **control-b** ("back") to move left and right across a command, either a command you're typing now or a previous one retrieved with **control-p**.

See the **r** alias in **ksh**(1) p. 4:

```
15$ r                    Repeat the last command.
16$ r gr                 Repeat the last command that started with gr.
17$ r 25                 Repeat the 25th command.
18$ history              See a numbered list of your most recent commands.
19$ h                    You can use h as an abbreviation for history.
```

The last word of the last command

`escape-.` is the last word of the previous command. See **`M-.`** in **`ksh`**(1) p. 31. For example,

```
1$ cat supercalifragilisticexpialidocious
2$ lpr -Pedlab supercalifragilisticexpialidocious
3$ rm supercalifragilisticexpialidocious

4$ cat supercalifragilisticexpialidocious
5$ lpr -Pedlab escape-.
6$ rm escape-.
```

Three ways to discover which shell you're using

The Bourne and Korn shells execute the commands in a file named **`.profile`** when you login in, but the C shell executes the commands in a file named **`.login`** when you login in (Handout 2, p. 2). That's why you must know which shell you're using. In either case, the file must be in your home directory.

`ps` is the Unix equivalent of right clicking on the bottom bar in Windows and selecting **`Task Manager...`** → **`Processes`** (or going to the **`Command Prompt`** and running **`tasklist`** to see the PID numbers).

```
1$ echo $SHELL
2$ grep abc1234 /etc/passwd        Look at the seventh field, p. 53.
3$ ps | more                       "Process status": list all the programs you're running, p. 34.
```

Create a Korn shell .profile file: pp. 35−38

Your **`.profile`** file should be a copy of the file **`~mm64/public_html/INFO1-CE9545/src/.profile`**:

```
1$ cd                              Go to your home directory.
2$ pwd                             Make sure you arrived there.

3$ ls -l | more                    all names except those that start with a dot
4$ ls -la | more                   all names, including those that start with a dot (Los Angeles)

5$ mv .profile old.profile         Rename your existing .profile file, if you have one.
6$ ls -la | more
```

Copy **`~mm64/public_html/INFO1-CE9545/src/.profile`** into your current directory, which is now your home directory. For the dot which is the second argument of the **`cp`**, see Handout 1, p. 9. The other dot is merely part of the filename **`.profile`**; it does not mean "the current directory".

```
7$ cp ~mm64/public_html/INFO1-CE9545/src/.profile .
8$ ls -la | more
```

`cmp` will give you dead silence if the two files are identical, p. 20:

```
9$ cmp ~mm64/public_html/INFO1-CE9545/src/.profile .profile
10$ exit            The commands in your .profile will be executed when you log back in.
```

If you change the contents of your **`.profile`** file, you must log out and log back in. The commands in your **`.profile`** are executed only when you log in, not when you put them into the file.

The contents of your Korn shell .profile file

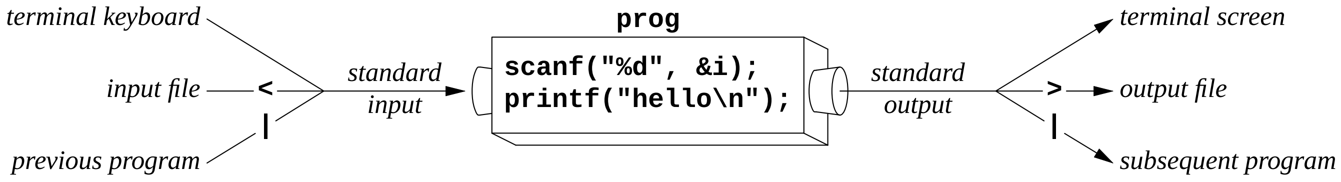
No backslash is needed to split a long command immediately after a pipe (lines 81−83). See pp. 107−108.

—On the Web at
http://i5.nyu.edu/~mm64/INFO1-CE9545/src/.profile

```
#This file is named .profile
#The Korn shell executes the commands in this file when you log in.
#Page numbers in this file refer to the ksh(1) manual page.

#"export" (p. 41) creates an environment variable (pp. 23-24).

#Directories shell searches for executables: pp. 16, 27.  gcc is C compiler.
export PATH=$PATH:/opt/gcc453/bin:.

#The -t option of man makes man run this program.
#~mm64 is the full pathname of mm64’s home directory: pp. 4-5.
export TCAT=~mm64/bin/tcat

#The editor to be invoked by mail, mailx, dbx, etc: p. 14.
export EDITOR=/bin/vi

#Make vi display the words INSERT MODE while you’re in insert mode.
export EXINIT=’set showmode’   #or export EXINIT=’set showmode number’

#The name of the computer.
export HOST=‘hostname‘

#The Korn shell will check this file for new mail every 10 minutes: p. 16.
export MAIL=/var/mail/$LOGNAME

#Default printer for the lpr, lpq, lprm programs.
export PRINTER=edlab

#Source code directory on the web at http://i5.nyu.edu/~mm64/INFO1-CE9545/src/
export S45=~mm64/public_html/INFO1-CE9545/src

#if necessary to make your screen editor (vi, emacs, pico) work properly
export TERM=vt100

#Make the command numbers in the prompt start at 1 each time you log in:
#p. 27 for .sh_history.  ~ is the full pathname of your home directory: pp. 4-5.
rm -f ~/.bash_history

#Prompt string one.  See pp. 10-12 for chopping, p. 16 for !.
#The %%.* chops off the end of $HOST: i5.nyu.edu becomes i5
#The ##*/ chops off the beginning of $PWD: /home1/a/abc1234 becomes abc1234
PS1=’! ${HOST%%.*}:${PWD##*/} $ ’

#The following command must come *after* the export EDITOR=/bin/vi.
#otherwise the export EDITOR=/bin/vi would turn emacs mode back off (p. 12).

#Retrieve and edit previous commands with emacs, pp. 28-32.
#See pp. 46-50 for set.
set -o emacs

#Make the 4 arrow keys work in emacs mode, pp. 28-32.  Only in ksh93, sorry.
#/bin/loadkeys ~mm64/public_html/INFO1-CE9545/src/loadkeys_set
```

```
53
54 #Don’t let control-d log you out (p. 48): must type "exit" (p. 41).
55 set -o ignoreeof
56
57 #Don’t let > destroy the existing contents of a file: pp. 22, 49.
58 set -o noclobber
59
60 #For alias, see pp. 3-4, 39.
61 #"history" is itself an alias for "fc -l", pp. 4, 42.
62 alias h=history
63
64 #See all the jobs you’re running, pp. 25, 43.
65 alias j=jobs
66
67 #/bin/mailx is better than /bin/mail.  Used in line 81.
68 alias mail=mailx
69
70 #Turn off last 2 r and w bits of a newborn file.  For example, date > newborn
71 #will create a newborn file whose bits are rw-------.  The octal argument of
72 #the umask command has a 1 in each bit that should be turned off.  See p. 55.
73 umask 077
74
75 #Let other people send you messages with the talk program.
76 #It turns on the two rightmost w bits of your terminal.
77 mesg y
78
79 #See new and unread mail in alphabetical order of who sent it.
80 #Can use an alias (mail) as first word of command, and also after | && || ;
81 mail -e && mail -H |
82 grep ’^.[NU]..[0-9]’ |
83 sort +0.6f -0.25 +0.6 -0.25 +0.2n
84
85 #date
86 #cal
87 #who | grep def5678
88 #˜mm64/bin/moon
```

The central mystery of Unix: pp. 29−33



	C	*C++*	*Ruby*	*Java*	*shell language*
standard output	**printf(**	**cout <<**	**puts**	**System.out.print(**	**echo**
standard input	**scanf(**	**cin >>**	**gets**	**System.in.read(**	**read**

Rows and columns of output

```
1$ who | more
esh322     pts/3        Apr  7 22:55      (172-26-196-152.dynapool.nyu.edu)
mm64       pts/1        May 28 14:37      (3a_imac_03.ndlab.its.nyu.edu)
elliott    pts/4        May 28 11:40      (njoerd.es.its.nyu.edu)
mm64       pts/8        May 28 14:39      (3a_imac_03.ndlab.its.nyu.edu)
mm64       pts/9        May 28 15:11      (3a_imac_03.ndlab.its.nyu.edu)
```

See p. 5. Other systems need **who -M** *for last column.*

The **-f** argument of **ps** gives you verbose output, like the **-l** argument of **ls**. With the **-A** argument (“all”), **ps** will output a list of everyone’s programs. Without the **-A**, it will output only your own programs. Combine **-A** and **-f** to **-Af**.

```
    2$ ps
    3$ ps -f
    4$ ps -Af | more                p. 34.
         UID   PID  PPID   C    STIME TTY          TIME CMD
        root  2360  1655   0   Mar 29 ?            0:01 /usr/sbin/sh /lib/svc/method/svc
        root  2761  1655   0   Mar 29 ?            1:14 /usr/lib/ssh/sshd
       yc801 12235 12228   0   Apr 08 pts/7        0:03 /bin/ksh
        root  2407  1655   0   Mar 29 ?            0:01 /sbin/sh /lib/svc/method/net-ipm
```

Pipe examples

```
    1$ ps -Af
    2$ ps -Af | more
    3$ ps -Af | sort | more
    4$ ps -Af | grep ksh | more

    5$ ps -Af | wc -l                              p. 30; minus lowercase L
    6$ ps -Af | grep ksh | wc -l        How many copies of the Korn Shell are running?
    7$ ps -Af | grep abc1234 | wc -l       How many programs is abc1234 running?

    8$ ps -Af | grep ksh | sort
    9$ ps -Af | grep ksh | sort | more

    10$ ps -Af | sort | grep ksh | more               Why should you never do this?
    11$ ps -Af | sort | wc -l                         Why should you never do this?

    12$ ps -Af | lpr                lpr can accept pipe input as happily as file input.
    13$ ps -Af | sort | lpr
    14$ ps -Af | grep ksh | sort | lpr

    15$ who | wc -l

    16$ cd ~mm64/public_html/INFO1-CE9545/homework
    17$ pwd
    18$ ls | wc -l                               How many people did Homework 1.5?
    19$ ls -l | lpr

    20$ cd /home1/a
    21$ pwd
    22$ ls -l | grep '^drwxr-xr-x' | wc -l     How many people did Homework 1.2?
    23$ ls -l | grep '^drwxr-xr-x' | lpr
```

Output all the “braves” in *The Tempest.*

```
24$ awk '120433 <= NR && NR <= 123831' $S45/Shakespeare.complete |
grep -i brave                                                    Bravo!
    With those that I saw suffer: a brave vessel,
PROSPERO My brave spirit!
    It carries a brave form. But 'tis a spirit.
    And his brave son being twain.
    And his more braver daughter could control thee,
GONZALO  You are gentlemen of brave metal; you would lift
    That's a brave god and bears celestial liquor.
STEPHANO O brave monster! Lead the way.
TRINCULO Where should they be set else? he were a brave
    He has brave utensils,--for so he calls them--
STEPHANO Is it so brave a lass?
    And bring thee forth brave brood.
STEPHANO This will prove a brave kingdom to me, where I shall
PROSPERO Bravely the figure of this harpy hast thou
    How beauteous mankind is! O brave new world,
    Is tight and yare and bravely rigg'd as when
PROSPERO [Aside to ARIEL]  Bravely, my diligence. Thou shalt be free.
CALIBAN  O Setebos, these be brave spirits indeed!
```

Output all the “hands” in *Titus Andronicus:* **`http://www.foxsearchlight.com/titus/`**

```
25$ awk '123842 <= NR && NR <= 127608' $S45/Shakespeare.complete |
grep -i hand
```

Output all the “deeps” in *Richard III:*
`http://www.mgm.com/title_title.php?title_star=RICHARD3`

```
26$ awk '13856 <= NR && NR <= 19631' $S45/Shakespeare.complete |
grep -i deep
```

Output all the “dogs” in *Timon of Athens* (will never be made into a movie):

```
27$ awk '147983 <= NR && NR <= 151955' $S45/Shakespeare.complete |
grep -i dog
```

Pipes: pp. 31−33

When you run two programs with a pipe

```
prog1 | prog2
```

they start at the same time, end at the same time, and run side-by-side at about the same speed. During the course of their lives, one or both of the following events may happen any number of times (or maybe not at all):

(1) **`prog1`** may produce output faster than **`prog2`** can accept it, causing the pipe to swell. When the pipe holds 16,384 bytes of data in transit, the operating system will put **`prog1`** to sleep for a little while to give **`prog2`** a chance to drain the pipe.

```
/* Excerpts from the file /usr/include/sys/fs/fifonode.h */
#define FIFOHIWAT   (16 * 1024)
#define FIFOLOWAT   (0)
```

`FIFO` is “first in, first out”. Other systems use the macro **`DEFAULT_PIPE_SIZE`** in the file **`/usr/include/sys/fifo.h`**.

(2) **`prog2`** may process its input faster than **`prog1`** can provide it. In this case, the pipe will begin to empty out. When the pipe holds 0 bytes of data in transit, the operating system will put **`prog2`** to sleep

for a little while to give **prog1** a chance to put more data into the pipe.

It is possible for neither of these two events to happen. For example, **prog1** may begin by outputting data into the pipe. Meanwhile **prog2** may have a lot of internal work to do before it begins to input data from the pipe. When **prog2** does begin to input, there may already be an ample supply of data in the pipe, so neither program is put to sleep.

To sum up: both programs begin to run simultaneously, and continue to run simultaneously except in the cases where the pipe swells to its maximum size or empties completely.

The pig in the python

sort must receive every line of its input before it can begin to produce any line of output. This restriction was imposed by the author of **sort**, not by the Unix operating system. **grep** labors under no such restriction.

Write no space around the dash in the arguments of **tr**. The character to the left of a dash in an argument of **tr** must have a lower ASCII code than the character to the right of the dash. See the ASCII code numbers in **ascii**(5).

```
1$ prog1 | prog2 | sort | prog3 | prog4
2$ prog1 | prog2 | wc -l | prog3 | prog4     p. 30 for wc -l; minus lowercase L

3$ prog1 | prog2 | grep 212 | prog3 | prog4
4$ prog1 | prog2 | tr '[A-Z]' '[a-z]' | prog3 | prog4        p. 107 for tr
5$ prog1 | prog2 | tr '[A-Z] [a-z]' | prog3 | prog4            needs 2 args
```

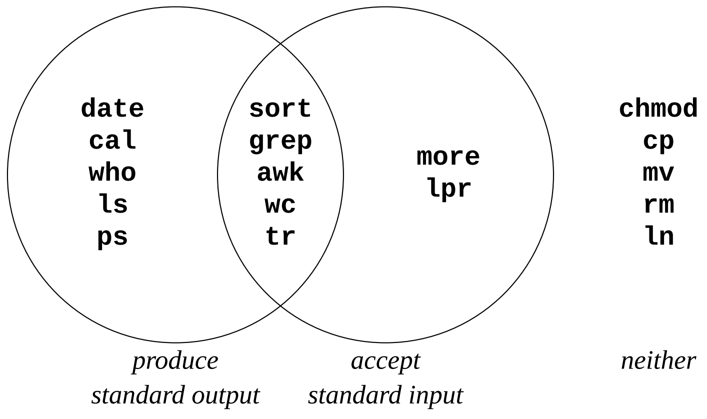
Not every program accepts standard input and/or produces standard output

Data always flows through pipes from left to right. A pipe must be preceded by a program that produces standard output and must be followed by a program that accepts standard input:

```
1$ who | sort           okay
2$ sort | who           sort receives no input from who.
```

In a pipeline, the programs in the left circle can only be the leftmost program. In a pipeline, the programs in the right circle can only be the rightmost program. For example, **lpr** produces no standard output; and although **more** does produce standard output, this output is usefully directed only to the screen.

The programs in the overlap are called *filters* (p. 101). They can be used anywhere in a pipeline.



Redirect the output to a file: pp. 29−31

To redirect the output of a program to a new file, you must have **w** permission for the directory in which the file will be created. Whenever you mention the name of a file with no leading slash, the file is assumed to be in the current directory as a consequence of Handout 1, p. 9.

```
1$ cd
2$ pwd

3$ date > junk                    You see nothing on the screen.
4$ ls -l junk

5$ cat junk                       Now you see the output of date.

6$ rm junk
7$ ls -l junk                     Make sure you removed the junk file.

8$ date > /another/directory/junk
9$ ls -l /another/directory/junk
```

Overwrite vs. append: pp. 29−31

If the output file does not already exist, **>** and **>>** will both create it. If the file already exists, **>** will overwrite it while **>>** will append extra data to the end of it without disturbing the existing contents. To append the output of a program to an existing file, you must have **w** permission for the file. Make sure there’s no space between the two **>**’s.

```
1$ cd
2$ pwd

3$ date > junk
4$ cal > junk                     Overwrite (i.e., destroy) the previous contents of junk.
5$ cat junk
6$ rm junk

7$ date > junk                    Let’s start again from the beginning.
8$ ls -l junk                     See how big junk is.
9$ cal >> junk                    Append to the end of junk. No space between the >’s.
10$ ls -l junk                    junk got bigger, didn’t it?
11$ cat junk
12$ rm junk
```

Protect a file from accidentally being overwritten

You can deny yourself permission to overwrite one of your own files. See also **noclobber** in Handout 2, p. 13, lines 57−58; **ksh**(1) pp. 22, 49.

```
1$ cd
2$ pwd

3$ date > precious
4$ ls -l precious
-rw-------   1 abc1234     users         29 May 28 15:18 precious

5$ chmod 400 precious
6$ ls -l precious
-r--------   1 abc1234     users         29 May 28 15:18 precious
```

```
7$ cal > precious                              attempt to overwrite a write-protected file
precious: Permission denied     message anyone else would get, even though it's your file
```

Hardware devices disguised as files: pp. 65−69

The earliest Unix systems had all their devices in the **/dev** directory, so a typical device name would be **/dev/ttypa**. But our system has the devices scattered in many directories.

This semester, we're not allowed to see the underlying hardware because we're in a Solaris ‘‘zone’’. But if we were the superuser, we would be able to see the following disks.

Disks are block devices.

```
1$ cd /dev/md/dsk                         filesystem(5) says this stands for "meta-disk".
2$ pwd
/dev/md/dsk
```

```
3$ ls -l d10                                 Get disk names from df (Handout 1, p. 3).
lrwxrwxrwx   1 root     root          37 Aug  9 18:04 d10 -> ../../../devices/pseudo/md@0:0,10,blk
```

```
5$ cd ../../../devices/pseudo
6$ pwd
/devices/pseudo

7$ ls -l md@0:0,10,blk
brw-r-----   1 root     sys       85, 10 Aug 10 09:49 md@0:0,10,blk
```

Terminals are character devices.

pts stands for ‘‘pseudo-terminal’’; see **pts**(7d).

```
1$ tty                                    Which terminal are you currently logged in on?
/dev/pts/10

2$ cd /dev/pts
3$ pwd
/dev/pts

4$ ls -l 0 9
crw--w----   1 root     root      36,  0 Mar 29 06:54 0
crw--w----   1 mm64     tty       36,  9 May 28 15:18 9
```

To see all the hardware devices (block and character) on a Unix machine,

```
5$ find / '(' -type b -o -type c ')' -ls 2> /dev/null | more
```

Direct a program's output to a hardware device

It looks like we're overwriting the file **/dev/pts/9**, but actually we're sending the data to the terminal's device driver.

```
1$ cd
2$ pwd
```

```
3$ who | more
mm64       pts/9        May 28 15:11      (3a_imac_03.ndlab.its.nyu.edu)

    4$ date > /dev/pts/9
    5$ echo hi there > /dev/pts/9
```

To prevent you from doing this, the victim can give the same **`who`** command to see the name of the terminal they're using.

```
6$ who | more
mm64       pts/9        May 28 15:11      (3a_imac_03.ndlab.its.nyu.edu)

    7$ cd /dev/pts
    8$ pwd
    /dev/pts

    9$ ls -l 9
    crw--w----   1 mm64     tty       36,  9 May 28 15:18 9

    10$ chmod 600 9          You can still get mail; reverts to rw--w---- when you logout
    11$ ls -l 9
    crw-------   1 mm64     tty       36,  9 May 28 15:18 9
```

/dev/null: pp. 68−69

See **`null`**(7d).

```
    1$ prog > /dev/null    Discard the output, e.g. if you're interested only in prog's exit status.

    2$ cp /dev/null empty                Could also create empty file with touch.
    3$ ls -l empty
    -rw-------   1 abc1234  users          0 May 28 15:18 empty

    4$ cd /dev                           Let's examine /dev/null itself.
    5$ pwd
    /dev

    6$ ls -l null
    crw-rw-rw-   1 root     sys      102,  2 May 28 15:18 null
```

How they programmed before they invented pipes: pp. 30−31

The pipe in line 1 is easier than the file redirections in lines 2−4. You don't have to decide what directory to put the temporary file in, making sure the directory exists and you have **`w`** permission for it. You don't have to decide what to name the file, avoiding names that already exist in that directory. You don't have to make sure that the three separate commands in lines 2−4 all mention the same filename and directory name. You don't have to worry about someone reading, removing, or tampering with your temporary file.

There's nothing to clean up afterwards if you use a pipe. And you can't run out of disk space: a pipe can transfer an indefinitely large amount of information between two programs. See the maximum size for a non-pipe file in **`intro`**(2) pp. 5, 16.

```
    1$ prog1 | prog2
```

```
2$ prog1 > file
3$ prog2 < file
4$ rm file
```

Feed a program its standard input from the keyboard

Like any Unix program, **`bc`** can take its standard input from an input file, a pipe, or the keyboard. We illustrate keyboard input below. Press **`RETURN`** at the end of each line.

```
1$ bc                  “binary calculator”
100 / 7                Space around / is optional.
14
scale = 5              Ask for five digits to the right of the decimal point; can ask for up to 99.
100 / 7
14.28571
3.14159 * (2.25 + .25) ^ 2
19.63493
control-d              Tell bc that no more input will come from the keyboard.
2$                     The shell prompt reappears.
```

control-d does not kill a program

```
1$ sort                or try this experiment with wc -l
moe
larry
curly
control-d              You type everything up to and including this line.
curly                  sort outputs everything from this line on.
larry
moe
2$                     The prompt reappears.
```

A common cause of freezing up

If you forget to specify where the input comes from, the input is assumed to come from the keyboard and the program will wait indefinitely for you to type something. When you realize what has happened, press **`control-c`** (Handout 1, p. 20, ¶ (3)) and try again with a **`control-p`** (Handout 2, p. 11).

```
1$ grep mania /usr/dict/words          okay
2$ grep mania/usr/dict/words           The computer seems to freeze up.
```

Syntax for I/O redirection

Unless you specify otherwise, the input of a program comes from the terminal keyboard. Instead of this default, you can specify that the input should come from a file or through a pipe from another program.

Unless you specify otherwise, the output of a program goes to the terminal screen. Instead of this default, you can specify that the output should go to a file or to another program through a pipe.

The word immediately to the right of a **`<`** must be the name of an input file. The word immediately to the right of a **`>`** must be the name of an output file. The word immediately to the right of a **`|`** must be the name of a program that takes its input through the pipe.

```
1$ prog1
2$ prog1 < infile                  E.g., mail abc1234@mycompany.com < infile
3$ prog1 > outfile                 For example, cal 5 2013 > outfile
4$ prog1 < infile > outfile
```

```
5$ prog1 | prog2                        For example, cal 5 2013 | lpr -Pedlab
6$ prog1 < infile | prog2               ☜ See below.
7$ prog1 | prog2 > outfile
8$ prog1 < infile | prog2 > outfile

9$ prog1 | prog2 | prog3
10$ prog1 < infile | prog2 | prog3
11$ prog1 | prog2 | prog3 > outfile
12$ prog1 < infile | prog2 | prog3 > outfile                          etc.
```

How not to redirect I/O

Mother-in-law, court-marshal, billet-doux.

```
1$ prog < file > file                   Can't use same file for both input & output: p. 152.
2$ prog < file > temp                   Create a file with a different name.
3$ mv temp file                         Rename the newly created file.

4$ prog1 | prog2 < infile               incorrect version of line 6 above
```

A Unix program is blind to I/O redirection

A Unix program doesn't have to worry about where its input comes from and where its output goes to. For example, the **grep** command in

```
1$ ps -Af | grep ksh | lpr
```

doesn't know the names of the programs that surround it. In fact, it doesn't even know that it is attached to two pipes. The only part of the command line that it can see is its own name and its own command line argument(s):

```
grep ksh
```

Similarly, the **grep** in

```
2$ grep ksh < infile > outfile
```

can see only the words

```
grep ksh
```

Who invented Unix

Unix was created in 1969−1974 by Ken Thompson and Dennis Ritchie. Brian Kernighan named it. The languages C and C++ were created by Dennis Ritchie and Bjarne Stroustrup.

```
http://cm.bell-labs.com/who/ken/
http://cm.bell-labs.com/who/dmr/
http://cm.bell-labs.com/who/bwk/
http://www.research.att.com/~bs/
```

Specify an input file name as a command line argument

Only one input file can be specified to the right of the **<** symbol:

```
1$ prog < infile                                 good
2$ prog < infile1 infile2                        bad
3$ prog < infile1 infile2 infile3                bad
```

In addition to the **<** symbol, there is another way to specify an input file for most Unix programs: simply write the names of one or more input files as command line arguments, without the **<**. Not every program allows you to do this, because not every program was written by an author who made the extra effort to provide this feature. In line 5, the shell checks that the **infile** exists and is readable, complaining if it isn't. In line 4, these checks and complaints have to be performed by the **cat**.

The **man** command will tell you if a given program will accept input file names as command line arguments. **mail**, **tr** and **tee** are the only commands in this course that take input but do not accept input file names as command line arguments. To feed a file into **mail**, **tr**, and **tee**, you will have to use a **<**.

```
4$ cat infile
5$ cat < infile
6$ cat infile1 infile2 infile3

7$ more infile
8$ more < infile
9$ more infile1 infile2 infile3

10$ lpr infile
11$ lpr < infile
12$ lpr infile1 infile2 infile3

13$ grep abc1234 /etc/passwd
14$ grep abc1234 < /etc/passwd
15$ grep abc1234 /etc/passwd /etc/group
```

cat: p. 15

The command

```
1$ cat file
```

will display a **file** on the screen. But **cat** has no specific connection with the screen—at least, it has no more connection than any Unix program does. **cat** merely copies its input to its output completely unchanged. As with any Unix program, the input comes from the terminal keyboard and the output goes to the terminal screen unless you specify otherwise.

```
2$ cat                                         Type control-d when done.
3$ cat > outfile

4$ who | more                                  Pick a victim.
mm64       pts/9       May 28 15:11      (3a_imac_03.ndlab.its.nyu.edu)

5$ cd /dev/pts
6$ pwd
/dev/pts

7$ ls -l 9
crw-rw-rw-   1 mm64     tty       36,  9 May 28 15:18 9
```

You can do the following two commands if the third **w** bit of **/dev/pts/9** is on:

```
8$ cat > /dev/pts/9                            intrusive
9$ cat /usr/dict/words > /dev/pts/9            malicious
```

You can do the following two commands if the third **r** bit of **/dev/pts/9** is on:

```
10$ cat /dev/pts/9                          Steal his or her typing.
11$ cat /dev/pts/9 > outfile                Make a permanent record of it.

12$ cat file1 file2 file3                   Concatenate 3 files onto the screen.
13$ cat file1 file2 file3 > bigfile         Concatenate 3 files into one big file.
14$ cat file1 file2 file3 | myprog          Feed 3 input files into one of our programs.

15$ lpr file1 file2 file3                   Start each file on a new page.
16$ cat file1 file2 file3 | lpr             Don’t start each file on a new page.
```

How not to use cat

Remove these three **cat**’s: they do nothing. Also remove the pipe in front of each **cat**.

```
1$ prog1 | cat | prog2
2$ prog | cat
3$ prog1 | prog2 | cat
```

★ Don’t use **cat** to input a single file into a program:

```
4$ cat file | prog          Never do this.
5$ more file | prog         Or this.
6$ tail +1 file | prog      Even worse.
7$ prog < file              You can always do this instead.
8$ prog file                Better yet, you can usually do this.
```

✉ Mail a letter: pp. 8−9

If you had a file named **letter** containing a letter, you could mail it to **abc1234** like this. **mail** is now an alias for **mailx**; see Handout 2, p. 14, line 68.

```
1$ mail abc1234@mycompany.com < letter                          mail
2$ mail abc1234 < letter            Recipient’s host defaults to sender’s host.
```

Instead of mailing a file, you could feed input to **mail** directly from the keyboard:

```
3$ mail abc1234@mycompany.com
Subject: How to send mail
Type the letter, pressing RETURN each time you near the right edge
of the screen.  Type a line consisting only of control-d to end the
letter.  (The control-d must be at the start of the line.)
Press control-c twice to abort the letter.
control-d                                Tell it that you’re done typing input.
4$

5$ mail abc1234@ibm.com def5678@un.org    Send same letter to two or more people.
i4> mail abc1234@i5.nyu.edu               Someone on another machine could type this.
```

Read your mail: pp. 8−9

The only **mailx** commands you need are **h**, **p**, **s**, **d**, and **q**. Type them in response to the **?** prompt. ★ If you choose to investigate the “reply” command, be sure to type the **r** in lowercase to avoid public humiliation. See p. 13 of **mailx**(1).

Use **alpine** instead of **mail** to read a letter with attachments. It has a help menu. See **http://www.washington.edu/alpine/**.

```
1$ cd                      Make it simpler to save a letter later.
2$ pwd

3$ mail
Mail $Revision: 4.2.4.2 $  Type ? for help.
"/usr/spool/mail/mm64": 3 messages 3 new
>N  1 abc1234  Mon May 28 13:41  8/172  "How to send mail"
>N  2 aen0000  Mon May 28 13:42  9/171  "MAD Magazine subscription"
>N  3 hsl5678  Mon May 28 13:44  7/162  "Dodsworth"
? p1                       Print letter 1 on the screen. See mailx(1) p. 8.
? p3
? p2
? s2 madmag                Save letter 2 in a file.  Type one space after the s2.
? s3 /other/directory/lewis
? h                        "headers": see the table of contents again.
? d1                       Delete letter 1.
? d2
? d3                       or d1-3
? q                        Quit.
4$                         The shell prompt reappears.
```

x is just like **q**, but it undeletes all the letters you deleted. See p. 10 of **mailx**(1).

```
5$ ls -l madmag            Verify that you created a new file named madmag.
6$ lpr madmag              or use the escape-. in Handout 2, p. 11.
7$ rm madmag
8$ ls -l madmag            Verify that you removed the file madmag.
```

▼ Homework 2.4: mail a letter to yourself and read it (not to be handed in)

Mail two short letters with subjects, from your account on i5.nyu.edu to your account on i5.nyu.edu. and read them when they arrive. How long did they take to arrive? Save them into a file, and print and remove the files.

▲

□