The Unix Style

- Build special-purpose tools to do one thing well.

The Unix Style

- Build special-purpose tools to do one thing well.

The ideal; over time most tools suffer from “feature creep”. You should look at all the features but expect to use only a few.

The Unix Style

- Build special-purpose tools to do one thing well.

The ideal; over time most tools suffer from “feature creep”. You should look at all the features but expect to use only a few.

- Make as few distinctions between file types as possible.

Then a file you can read can *also* be executable: this is the basis for scripts.

The Unix Style

- Build special-purpose tools to do one thing well.

The ideal; over time most tools suffer from "feature creep". You should look at all the features but expect to use only a few.

- Make as few distinctions between file types as possible.

Then a file you can read can *also* be executable: this is the basis for scripts.

- Provide simple ways of connecting the tools together, as flexibly as possible.

The Unix Style

- Build special-purpose tools to do one thing well.

The ideal; over time most tools suffer from “feature creep”. You should look at all the features but expect to use only a few.

- Make as few distinctions between file types as possible.

Then a file you can read can *also* be executable: this is the basis for scripts.

- Provide simple ways of connecting the tools together, as flexibly as possible.

It turns out that only a couple of connections will get you a long way.

The Unix Style

- Build special-purpose tools to do one thing well.

The ideal; over time most tools suffer from "feature creep". You should look at all the features but expect to use only a few.

- Make as few distinctions between file types as possible.

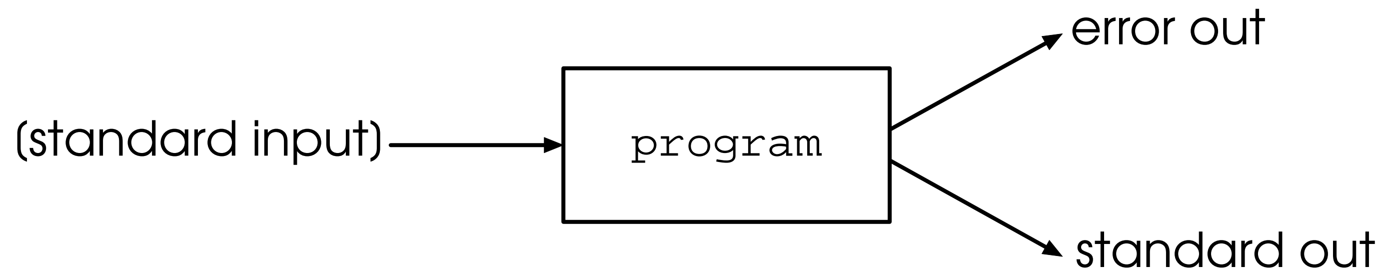
Then a file you can read can *also* be executable: this is the basis for scripts.

- Provide simple ways of connecting the tools together, as flexibly as possible.

It turns out that only a couple of connections will get you a long way.

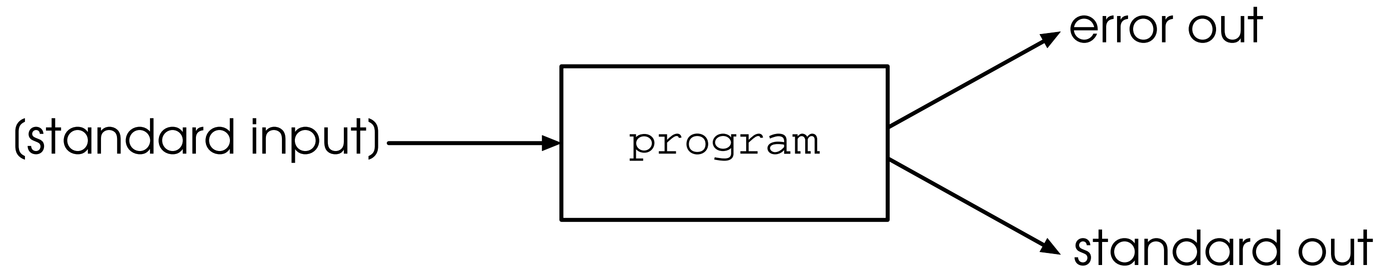
But, this means that there are only a few ways to communicate with the programs, which makes for unmemorable options: hence the complaints about Unix being hard to learn.

The Paradigmatic Unix Tool



Program reads from *stdin* (which may not be present) and writes to *stdout* and (perhaps) to *error out*.

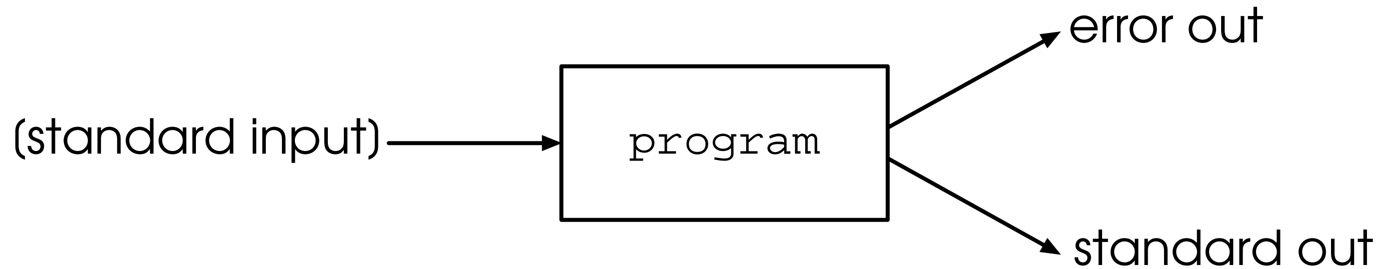
The Paradigmatic Unix Tool



Program reads from *stdin* (which may not be present) and writes to *stdout* and (perhaps) to *error out*.

The default for *stdin* is what you type.

The Paradigmatic Unix Tool



Program reads from *stdin* (which may not be present) and writes to *stdout* and (perhaps) to *error out*.

The default for *stdin* is what you type.

The default for *stdout* is what you see; so is the default for *error out*.

Connection Methods I

- | ("**pipe**") connects the *stdout* of one program to the *stdin* of the next.

Connection Methods I

- | ("**pipe**") connects the *stdout* of one program to the *stdin* of the next.

For example

```
% pwd
/Users/agnew/courses/sio233/notes
% pwd | wc
       1       1      34
%
```

The `pwd` command tells me where I am; if I "pipe this to `wc`", this string is sent to wc, which reads it and sends (to *stdout*) the number of lines, words, and characters (bytes).

A Piping Example: Sorting Words

1. `% cat tmp`
2. `% cat tmp | sort`
3. `% cat tmp | sort | uniq`
4. `% cat tmp | sort | uniq -c`

1	2	3	4
O	Absalom!	Absalom!	1 Absalom!
my	Absalom,	Absalom,	2 Absalom,
son	Absalom,	God	1 God
Absalom,	God	I	1 I
my	I	O	2 O
son,	O	died	1 died
my	O	for	1 for
son	died	had	1 had
Absalom!	for	my	5 my
would	had	son	2 son
God	my	son!	1 son!
I	my	son,	2 son,
had	my	thee,	1 thee,
died	my	would	1 would
for	my		
thee,	son		
O	son		
Absalom,	son!		
my	son,		
son,	son,		
my	thee,		
son!	would		

Connection Methods II

- > ("**redirect**") connects the *stdout* of one program to a file.

Connection Methods II

- > ("**redirect**") connects the *stdout* of one program to a file.

For example

```
% cat tmp | sort | uniq -c > tmp1
```

would put the counted results of the phrase (option 4) into the file *tmp1*.

Obviously, this is the end of the line (in all senses).

Connection Methods II

- > ("**redirect**") connects the *stdout* of one program to a file.

For example

```
% cat tmp | sort | uniq -c > tmp1
```

would put the counted results of the phrase (option 4) into the file *tmp1*.

Obviously, this is the end of the line (in all senses).

If you try to do this to an existing file, you will (or should) get an error message – one of the few things done to protect you. (If not, you should set up your "environment" so this will be the case, by running `set noclobber`).

Connection Methods II

- `>` ("**redirect**") connects the *stdout* of one program to a file.

For example

```
% cat tmp | sort | uniq -c > tmp1
```

would put the counted results of the phrase (option 4) into the file *tmp1*.

Obviously, this is the end of the line (in all senses).

If you try to do this to an existing file, you will (or should) get an error message – one of the few things done to protect you. (If not, you should set up your "environment" so this will be the case, by running `set noclobber`).

- `>!` connects the *stdout* of one program to a file, **and overwrites what is there**.

Connection Methods II

- `>` ("**redirect**") connects the *stdout* of one program to a file.

For example

```
% cat tmp | sort | uniq -c > tmp1
```

would put the counted results of the phrase (option 4) into the file *tmp1*.

Obviously, this is the end of the line (in all senses).

If you try to do this to an existing file, you will (or should) get an error message – one of the few things done to protect you. (If not, you should set up your "environment" so this will be the case, by running `set noclobber`).

- `>!` connects the *stdout* of one program to a file, **and overwrites what is there**.

- `>>` connects the *stdout* of one program to a file, and **appends** to what is there.

Error Output

If you send the output to a file (`>`, `>>`, or `>!`) or pipe it to another program (`|`) the *error out* will still go to the screen. If you want it, also, to go to a file, you need to use `>&`.

Options Using Command-Line Flags

The usual way of setting various options is to add flags on the command line; sometimes these have a - before them, sometimes a --; usually the option setting can be combined.

Options Using Command-Line Flags

The usual way of setting various options is to add flags on the command line; sometimes these have a - before them, sometimes a `--`; usually the option setting can be combined.

For example, `ls` has the options (of which I know a few):

Options Using Command-Line Flags

The usual way of setting various options is to add flags on the command line; sometimes these have a - before them, sometimes a --; usually the option setting can be combined.

For example, `ls` has the options (of which I know a few):

```
-a   --all
-A   --almost-all                -m
-b   --escape                    -n   --numeric-uid-gid
     --block-size=SIZE           -N   --literal
-B   --ignore-backups            -o
-c                               -p   --file-type
-C                               -q   --hide-control-chars
     --color[=WHEN]                   --show-control-chars
-d   --directory                 -Q   --quote-name
-D   --dired                          --quoting-style=WORD
-f                               -r   --reverse
-F   --classify                  -R   --recursive
     --format=WORD               -s   --size
     --full-time                 -S
-g                                    --sort=WORD
-G   --no-group                       --time=WORD
-h   --human-readable            -t
     --si                        -T   --tabsize=COLS
-H                               -u
     --indicator-style=WORD      -U
-i   --inode                     -v
-I   --ignore=PATTERN            -w   --width=COLS
-k   --kilobytes                 -x
-l                               -X
-L   --dereference               -1
                                      --help
                                      --version
```

`ls -altr` gives a full listing of all files in reverse chronological order.

Getting Help I: `man`

The command

`% man` *name*

will produce the manual page for the program called *name*, if there is one.

Getting Help I: `man`

The command

`% man` *name*

will produce the manual page for the program called *name*, if there is one.

For example,

```
% man ls
```

produces

LS(1) FSF LS(1)

NAME

ls - list directory contents

SYNOPSIS

ls (OPTION)... (FILE)...

DESCRIPTION

List information about the FILE's (the current directory by default).
Sort entries alphabetically if none of -cftuSUX nor --sort.

-a, --all

do not hide entries starting with .

-A, --almost-all

do not list implied . and ..

etc.

Getting Help II: `apropos`

The command

`% apropos` *string*

will produce all *NAME* lines from manual page that contain *string*: usually too many.

Getting Help II: `apropos`

The command

`% apropos` *string*

will produce all *NAME* lines from manual page that contain *string*: usually too many.

For example

```
% apropos list
```

produces, among other things:

ciphers(1ssl)	- SSL cipher display and cipher list tool
column(1)	- columnate lists
history(n)	- Manipulate the history list
join(n)	- Create a string by joining together list elements
list(n)	- Create a list
ls(1)	- list directory contents
lsort(n)	- Sort the elements of a list
mkdep(1)	- construct Makefile dependency list
users(1)	- list current users
vgrind(1)	- grind nice listings of programs