Standard streams

Using streams

There are three basic streams

Streams are the flow of data in e.g. a shell. Data flows out from commands (they produce data) and data can flow into commands (they read or use data).

Normal and wanted data that flows out from a command goes to the stream called “standard out” (or stdout for short).

Unexpected data (errors etc) typically flows out from a command to the stream called “standard error” (or stderr for short).

Data that a command reads flows in to the command using the stream called standard in (or - ta-da! stdin for short).

Standard out

Normal and expected output that goes to standard out, ends up in the same place as the command was started (kind of). If you type echo “Hello” in a terminal, the standard out stream will be connected to the same terminal, so that you see the output there.

Standard out can be redirected, as we have seen, using the > character:

```
$ echo "Save me, Jebus" > save.txt
```

But, in a windows GUI program, where does standard out go?

(Probably nowhere we can look, depending on how we started the program!)

Standard error

This stream is dedicated for error messages and diagnostic messages. The reason we don’t use standard out for these kinds of messages, is that we can redirect the errors to one place, and the normal (standard out) messages to another place (or let them appear on the screen).

When you learn how to program, you will learn how to write normal and expected stuff (like Hello World) to standard out, and error messages to standard error.

If you test software, it is likely that you want to save the good stuff in one file and all the errors in another file (so that you can inspect them later).

Standard in

Standard in is a stream which can be used by a command to read data from some source. For instance, we can tell `bc` to read a file with mathematical expressions and evaluate them for us. This way, we don't need to enter them manually.

Often a program (like grep) can use either a file that we provide as an argument: `$ grep wede countries.txt` or read from input that we send to the command, e.g. using redirection $ `grep wede < countries.txt` or via a pipe (later slide!), or interactively:

```
$ grep wede
Denmark
Sweden
Swedeen
```

It’s a pipe dream

Some commands (or applications) can read from standard in, but what we don’t know yet is that the standard in stream can come from another command!

```
$ grep wede countries.txt
Swedeen
$ cat countries.txt | grep wede
Swedeen
```

The above two command lines produce the same result.

Understanding pipes

You can think of the pipe as catching the output from one command, and sending it in a “pipeline” to the next command. You could also think of it as redirecting the output not to a file, but to a command.

The command on the receiving end detects that there is data available in “standard in” and reads the data and uses it.

command1 → command2

(the arrow is the pipeline, but we write it as a vertical bar, “|”)

The real power of pipes...

...comes from combining many commands.

If we want to find out what words in a text file are most frequent, we can combine many commands using pipes!

In the next few slides, we'll see how we'd go about finding the ten most frequent words from a text file.

Breaking up the file to one word per line

```
$ tr ' ' '\n' < jabberwocky.txt
Beware
the
Jabberwock,
my
son!
The
jaws
that
bite,
the
claws
that
catch!     (etc many more lines...)
```

Removing punctuation

```
$ tr ' ' '\n' < jabberwocky.txt | tr -d "[:punct:]"
Beware
the
Jabberwock
my
son
... (and so on...)
```

(use the output of the previous command, pipe it to `tr` which deletes all punctuation)

Make all letters lower case (treat “The” as “the”)

```
$ tr ' ' '\n' < jabberwocky.txt | tr -d "[:punct:]"
  | tr A-Z a-z
beware
the
jabberwock
my
son
...(and so on...)
```

(Use the output of the previous command and send it to tr which translates all capital letters to lower case letters)

Sort all words!

```
$ tr ' ' '\n' < jabberwocky.txt | tr -d "[:punct:]"
  | tr A-Z a-z | sort
and
and
and
and
and
and
and
and
and
arms
as
as
awhile
```

(use the output from... pipe it to `sort`)

Use `uniq` to count duplicates

```
$ tr ' ' '\n' < jabberwocky.txt | tr -d "[:punct:]"
  | tr A-Z a-z | sort | uniq -c
      8 and
      1 arms
      2 as
      1 awhile
      1 back
      1 bandersnatch
(and so on...)
```

(use the output ... pipe it to uniq -c)

Sort the statistics in reverse

```
$ tr ' ' '\n' < jabberwocky.txt | tr -d "[:punct:]"
  | tr A-Z a-z | sort | uniq -c | sort -rn
     11 the
      8 and
      7 he
      4 in
      3 through
      3 my
      3 jabberwock
... and so on...
```

(use the output of the previous... pipe it to `sort -rn`)

Keep the first 10 lines of the sorted statistics

```
$ tr ' ' '\n' < jabberwocky.txt | tr -d "[:punct:]"
  | tr A-Z a-z | sort | uniq -c | sort -rn | head
     11 the
      8 and
      7 he
      4 in
      3 through
      3 my
      3 jabberwock
      2 with
      2 went
      2 vorpal
```

(And, we’re done!!!)

Join lines with same word count

```
$ tr ' ' '\n' < jabberwocky.txt | tr -d "[:punct:]"
  | tr A-Z a-z | sort | uniq -c | sort -rn | head
  | awk '{print $1,$2;}'
  |sed '$!N;/^\([^\ ]*\ \)\(.*\)\(\n\)\1/!P;s//\3\1\2\ /;D'
11 the
8 and
7 he
4 in
3 through my jabberwock
2 with went vorpal two thought that stood one it his came beware as
1 wood whiffling uffish tumtum tulgey tree took to time thou sword sought
son so snickersnack slain shun rested of o manxome long left jubjub joy
jaws its head has hand galumphing frumious frabjous foe flame eyes dead
day come claws chortled catch callooh callay by burbled boy blade bite
bird beamish bandersnatch back awhile arms
```

JUST KIDDING!!!!!!!!!! (but it works)