Chapter 4a. I/O redirection

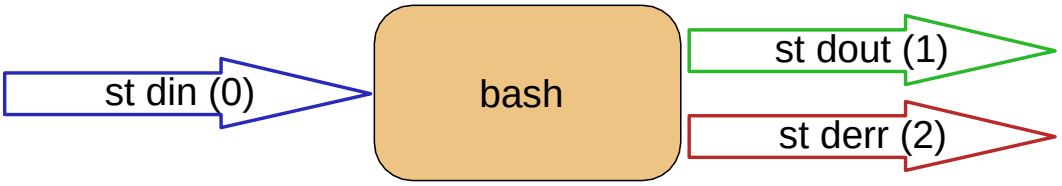
One of the powers of the Unix command line is the use of **input/output redirection** and **pipes**.

This chapter explains **redirection** of input, output and error streams.

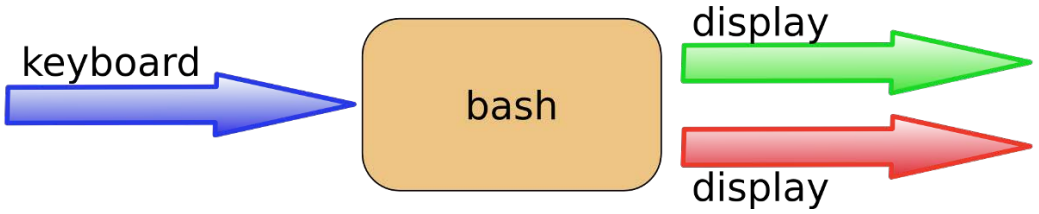
18.1. stdin, stdout, and stderr

The bash shell has three basic streams; it takes input from **stdin** (stream **0**), it sends output to **stdout** (stream **1**) and it sends error messages to **stderr** (stream **2**) .

The drawing below has a graphical interpretation of these three streams.



The keyboard often serves as **stdin**, whereas **stdout** and **stderr** both go to the display. This can be confusing to new Linux users because there is no obvious way to recognize **stdout** from **stderr**. Experienced users know that separating output from errors can be very useful.

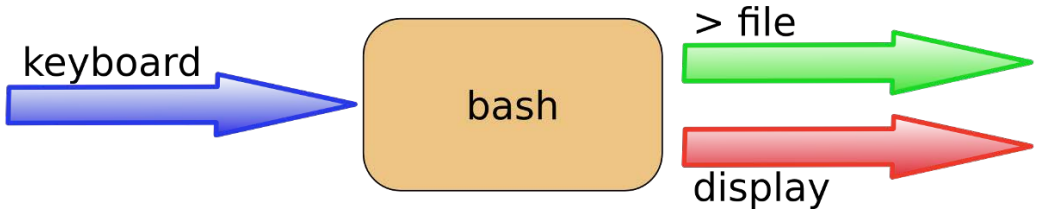


The next sections will explain how to redirect these streams.

18.2. output redirection

18.2.1. > stdout

stdout can be redirected with a **greater than** sign. While scanning the line, the shell will see the > sign and will clear the file.



The > notation is in fact the abbreviation of **1>** (**stdout** being referred to as stream **1**).

```
[clim@sop ~]$ echo It is cold today!

It is cold today!

[clim@sop ~]$ echo It is cold today! > winter.txt
[clim@sop ~]$ cat winter.txt

It is cold today!
[clim@sop ~]$
```

Note that the bash shell effectively **removes** the redirection from the command line before argument 0 is executed. This means that in the case of this command:

```
echo hello > greetings.txt
```

the shell only counts two arguments (echo = argument 0, hello = argument 1). The redirection is removed before the argument counting takes place.

18.2.2. output file is erased

While scanning the line, the shell will see the > sign and **will clear the file**! Since this happens before resolving **argument 0**, this means that even when the command fails, the file will have been cleared!

```
[clim@sop ~]$ cat winter.txt

It is cold today!

[clim@sop ~]$ zcho It is cold today! > winter.txt

-bash: zcho: command not found
[clim@sop ~]$ cat winter.txt
[clim@sop ~]$
```

18.2.3. noclobber

Erasing a file while using > can be prevented by setting the **noclobber** option.

```
[clim@sop ~]$ cat winter.txt

It is cold today!

[clim@sop ~]$ set -o noclobber

[clim@sop ~]$ echo It is cold today! > winter.txt

-bash: winter.txt: cannot overwrite existing file
[clim@sop ~]$ set +o noclobber

[clim@sop ~]$
```

18.2.4. overruling noclobber

The **noclobber** can be overruled with >|.

```
[clim@sop ~]$ set -o noclobber

[clim@sop ~]$ echo It is cold today! > winter.txt

-bash: winter.txt: cannot overwrite existing file

[clim@sop ~]$ echo It is very cold today! >| winter.txt
[clim@sop ~]$ cat winter.txt

It is very cold today!
[clim@sop ~]$
```

18.2.5. >> append

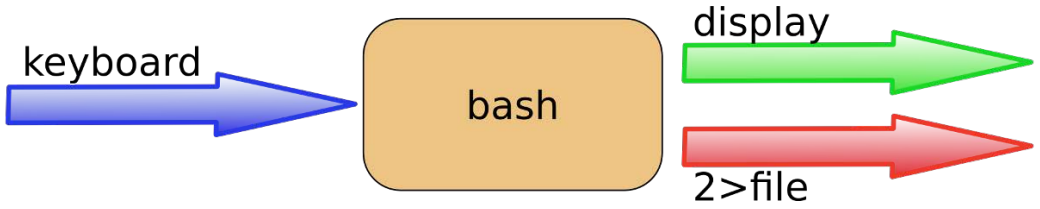
Use >> to **append** output to a file.

```
[clim@sop ~]$ echo It is cold today! > winter.txt
[clim@sop ~]$ cat winter.txt

It is cold today!

[clim@sop ~]$ echo Where is the summer ? >> winter.txt
[clim@sop ~]$ cat winter.txt

It is cold today!
Where is the summer ?
[clim@sop ~]$
```

18.3. error redirection

18.3.1. 2> stderr

Redirecting **stderr** is done with **2>**. This can be very useful to prevent error messages from cluttering your screen.



The screenshot below shows redirection of **stdout** to a file, and **stderr** to **/dev/null**. Writing **1>** is the same as >.

```
[clim@sop ~]$ find / > allfiles.txt 2> /dev/null
[clim@sop ~]$
```

18.3.2. 2>&1

To redirect both **stdout** and **stderr** to the same file, use **2>&1**.

```
[clim@sop ~]$ find / > allfiles_and_errors.txt 2>&1
[clim@sop ~]$
```

Note that the order of redirections is significant. For example, the command

```
ls > dirlist 2>&1
```

directs both standard output (file descriptor 1) and standard error (file descriptor 2) to the file dirlist, while the command

```
ls 2>&1 > dirlist
```

directs only the standard output to file dirlist, because the standard error made a copy of the standard output before the standard output was redirected to dirlist.

18.4. output redirection and pipes

By default you cannot grep inside **stderr** when using pipes on the command line, because only **stdout** is passed.

```
clim@sop:~$ rm file42 file33 file1201 | grep file42

rm: cannot remove ‘file42’: No such file or directory
rm: cannot remove ‘file33’: No such file or directory
rm: cannot remove ‘file1201’: No such file or directory
```

With **2>&1** you can force **stderr** to go to **stdout**. This enables the next command in the pipe to act on both streams.

```
clim@sop:~$ rm file42 file33 file1201 2>&1 | grep file42

rm: cannot remove ‘file42’: No such file or directory
```

You cannot use both **1>&2** and **2>&1** to switch **stdout** and **stderr**.

```
clim@sop:~$ rm file42 file33 file1201 2>&1 1>&2 | grep file42

rm: cannot remove ‘file42’: No such file or directory clim@sop:~$
echo file42 2>&1 1>&2 | sed 's/file42/FILE42/' FILE42
```

You need a third stream to switch stdout and stderr after a pipe symbol.

```
clim@sop:~$ echo file42 3>&1 1>&2 2>&3 | sed 's/file42/FILE42/'
file42

clim@sop:~$ rm file42 3>&1 1>&2 2>&3 | sed 's/file42/FILE42/' rm:
cannot remove ‘FILE42’: No such file or directory
```

18.5. joining stdout and stderr

The **&>** construction will put both **stdout** and **stderr** in one stream (to a file).

```
clim@sop:~$ rm file42 &> out_and_err
clim@sop:~$ cat out_and_err

rm: cannot remove ‘file42’: No such file or directory
clim@sop:~$ echo file42 &> out_and_err

clim@sop:~$ cat out_and_err

file42
clim@sop:~$
```

18.6. input redirection

18.6.1. < stdin

Redirecting **stdin** is done with < (short for 0<).

```
[clim@sop ~]$ cat < text.txt
one

two

[clim@sop ~]$ tr 'onetw' 'ONEZZ' < text.txt
ONE

ZZO

[clim@sop ~]$
```

18.6.2. << here document

The **here document** (sometimes called here-is-document) is a way to append input until a certain sequence (usually EOF) is encountered. The **EOF** marker can be typed literally or can be called with Ctrl-D.

```
[clim@sop ~]$ cat <<EOF > text.txt

> one

> two

> EOF

[clim@sop ~]$ cat text.txt
one

two

[clim@sop ~]$ cat <<brol > text.txt

> brel

> brol

[clim@sop ~]$ cat text.txt
brel

[clim@sop ~]$
```

18.6.3. <<< here string

The **here string** can be used to directly pass strings to a command. The result is the same as using **echo string | command** (but you have one less process running).

```
clim@sop~$ base64 <<< linux-training.ro
bGludXgtdHJhaW5pbmcuYmUK

clim@sop~$ base64 -d <<< bGludXgtdHJhaW5pbmcuYmUK
linux-training.ro
```

See rfc 3548 for more information about **base64**.

18.7. confusing redirection

The shell will scan the whole line before applying redirection. The following command line is very readable and is correct.

```
cat winter.txt > snow.txt 2> errors.txt
```

But this one is also correct, but less readable.

```
2> errors.txt cat winter.txt > snow.txt
```

Even this will be understood perfectly by the shell.

```
< winter.txt > snow.txt 2> errors.txt cat
```

18.8. quick file clear

So what is the quickest way to clear a file ?

```
>foo
```

And what is the quickest way to clear a file when the **noclobber** option is set ?

```
>|bar
```

18.9. practice: input/output redirection

1. Activate the **noclobber** shell option.

2. Verify that **noclobber** is active by repeating an **ls** on **/etc/** with redirected output to a file.

3. When listing all shell options, which character represents the **noclobber** option ?

4. Deactivate the **noclobber** option.

5. Make sure you have two shells open on the same computer. Create an empty **tailing.txt** file. Then type **tail -f tailing.txt**. Use the second shell to **append** a line of text to that file. Verify that the first shell displays this line.

6. Create a file that contains the names of five people. Use **cat** and output redirection to create the file and use a **here document** to end the input.