Before we start:

- Open a terminal and cd to data-temp directory
- List the content of the data directory. You will find data files **temp.dat, temp-clean.dat,** and **temp-clean1.dat,** which are used during this lecture

If you do not have the data files, go on Canvas -> Files -> zip files and download **data-temp.zip**

- Put the data-temp.zip into your home directory
- Unzip the file data-temp.zip and a directory called data-temp will be created
  You can also unzip by typing at the terminal

```
unzip data-temp.zip
```

Slides 1-3 refer to video Lecture5-stdoutput-redirection

Standard output

Standard output, sometimes abbreviated *stdout*, refers to the *streams* of data (plain text) that are produced by command line programs.

By default, the standard output (stdout) is sent to the terminal.

For example:

```
echo Hello Unix

ls ~/

cat program_1.bash

grep UT temp.dat
```

Some commands such as **mv** do not generate standard output.
Think of another command that generates a standard output.

Redirecting standard output with >

The output of a command can be redirected into a file. For this use **>**

```
command  (options) argument(s) > file
```

If file does not exist it will be created, if file already exists it will be overwritten

```
echo Hello Unix           # output to screen
echo Hello Unix > file1   # output redirected to a file
cat file1  # convince yourself that the output is correct
```

```
ls ~/               # output to screen
ls ~/ > my_homedir  # output redirected to a file
```

```
grep UT temp.dat           # output to screen
grep UT temp.dat > UT.dat  # output redirected to a file
```

Redirecting standard output with >

```
command  (options) argument(s) > file
```

If a file already exists, it will overwrite that file.

```
echo Hola Python > file1
# Now you have overwritten your file file1
```

Since > will overwrite an existing file, you can use >> to **append** the output of a line of code into a file

Redirecting and appending standard output with >>

```
command  (options) argument(s) >> file
```

```
echo Hello Unix >> file1
#output is redirected and appended to the existing file1
```

Practice

Practice redirecting the output of these commands into a file:

```
head temp-clean.dat
tail temp-clean.dat
cal
pwd
```

Before you redirect, use the command the usual way (without redirection) and make sure the command prints the correct output to screen.

Only in the next step redirect the output

Also, try overwriting some files, just for fun.

Try appending the output of command **`date`** into a file date.txt

Try it multiple times ...

Slides 4-5 refer to video Lecture5-stderror-redirection

Standard error

Standard error (stderr) is another output stream (plain text) typically used by programs to report **status messages** such as error messages and warning messages.

Status messages (stderr) are displayed in the terminal by default.

For example, if you type **`data`** instead of **`date`** you will get an error message:

```
data
```

Or if you try listing a file that does not exist:

```
ls date2.txt
```

Also, for example if you try using find commands to search files in the root directory:

```
find / -type f -name pwd
```

You will get messages that permission was denied to search certain folders.

Redirecting standard error with 2>

Use **2>** to redirect standard error (*stderr*) into a file:

Example :

```
data 2> my_error
```

now look at the content of my_error

```
ls date2.txt 2> my_error2
```

now look at the content of my_error2

```
find / -type f -name pwd 2> /dev/null
```

You can get rid of the standard error message completely by redirecting it to the so-called null device (/dev/null)

Pipelines

A *pipe* | is a form of *redirection* that is used to send the output of a command to another command for further processing.

Pipes are used to create what can be visualized as *a pipeline of commands*

By using the pipe operator | the output text (*stdout)* of one command can be piped into the input (*stdin)* of another command

```
command (options) (arguments) | command (options)
```

Commands after the first pipe do not have an argument.

```
grep AZ temp.dat | head -3
```

Come up with a pipeline of commands that use grep and tail

Pipelines

It is possible to put several commands into pipeline.

```
grep AZ temp.dat | head -3
grep AZ temp.dat | head -3 | tail -1
```

Note that commands after the first pipe do not have an argument.

Always add commands one by one. First try out the 1st command, then the 2nd, and then 3rd. This will minimize the chance of errors.

Common error: Repeat filename in 2nd , 3rd , 4th commands and so on…of a pipeline

Example:

Q: Extract all lines containing keyword AZ from the first 8 lines of temp.dat

Which pipeline is incorrect and why?

```
head -8 temp.dat | grep AZ
head -8 temp.dat | grep AZ temp.dat
```

More on command grep

Search and print all the lines in a file that match multiple patterns.

```
grep 'pattern1' file.txt | grep 'pattern2' # in any order
```

```
……pattern1 …. . pattern2
```

```
 grep AZ temp.dat  | grep 203
```

Slides 9 refers to video Lecture5-command substitution

Command substitution

Output of a line of command can be stored in a variable.
To do this use the normal variable assignment technique,
and the command stored inside ` ` (back quotes)

```
output_var=`line of code` #need to use back quotes
```

Example:

```
grep –c UT temp.dat
x=`grep –c UT temp.dat`
echo $x
```

Example

```
mycurrentdir=`pwd`
echo $mycurrentdir
```

If you change directories, the variable will stay the same.

Slides 10-13 refer to video Lecture5-input redirection and tr command

Redirecting standard input with “<“ and the tr command

The tr utility takes input from standard input (stdin), i.e. your keyboard, performs substitution of selected characters, and prints output to standard output (i.e. your terminal)

tr *translates* specified characters into other characters.

```
tr (options) charset1 charset2
```

```
tr a-z A-Z   # takes input from default standard input (keyboard)
             # write something and press return
             # to exit Control-D
```

```
tr A-Z a-z < temp-clean.dat  #redirect standard input, i.e. tr takes
input from temp-clean.dat and not from keyboard
```

Try to save the output of the command above in a new file temp_lower.dat

Try this

~~tr A-Z a-z temp-clean.dat~~ # you get an error

tr does not accept file names as arguments

tr command - translate characters

The *tr command* is used to *translate* specified characters into other characters

```
tr [options] charset1 charset2      #translate characters
```

```
echo Thos os onsode | tr 'o' 'i'    #in pipeline

echo Thos os onsode | tr a-z A-Z

echo I really like tr! | tr ' ' '\n’
echo I realllly like tr! | tr -s 'l'
```

Can you guess what the –s option does?

Remember the sed command

sed (*stream editor*) is a Unix utility that parses and transforms text

```
sed 's/word1/word2/'  filename
sed 's/word1/word2/g'  filename #If you add g in the end it will replace all occurrences
```

Make a file called sed-example and write in it:

```
I love bla. I said I love bla
```

Then run sed:

```
sed 's/bla/tea/' sed-example
sed 's/bla/tea/g' sed-example
```

If you add g in the end, it will replace all occurrences

sed in pipeline

```
echo bla | sed 's/bla/tea/'
```

Difference between tr and sed

If you do this example you would think that tr and sed are quite similar:

```
    echo Test+for+tr+and+sed | tr '+' ' '
    echo Test+for+tr+and+sed | sed 's/+/ /g'
```

However, if you do this example, you will realize that they are different:

```
    echo good | tr 'good' 'best'
```

tr has done character-based transformation and it is replacing good to best as **g=b, o=e, o=s, d=t**

```
    echo  good | sed 's/good/best/g'
```

```
sed  s/word1/word2/g filename

tr (options) charset1 charset2 < filename
```