Practical Computing for Scientists

Armin Sobhani

CSCI 2000U

UOIT – Fall 2015

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

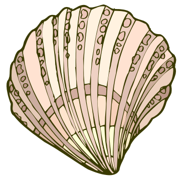
Assignment 1

- Please complete and submit Assignment 1:
  - Blackboard > Course Content > Week 3 (Sept. 28-Oct. 2) > Monday Sept. 28 > Assignment 1
  - Due Oct. 3, 2015, 17:00 EDT

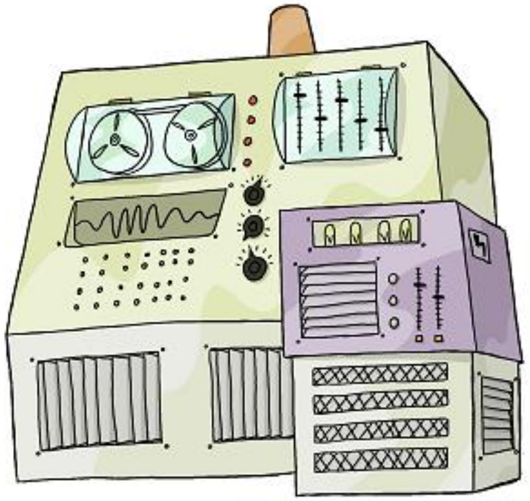
The Unix Shell

Variables

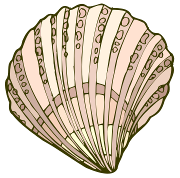
Created by Greg Wilson



shell

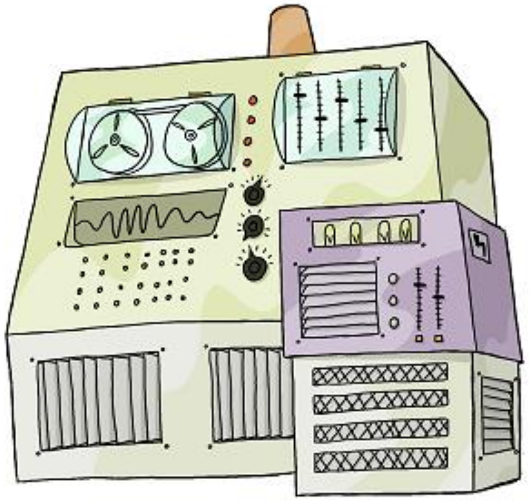


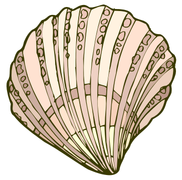
UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY



shell

The shell is a program

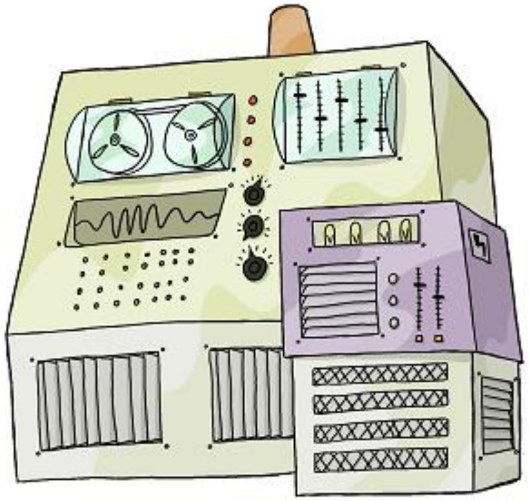


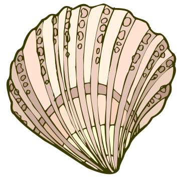


shell

The shell is a program

It has variables





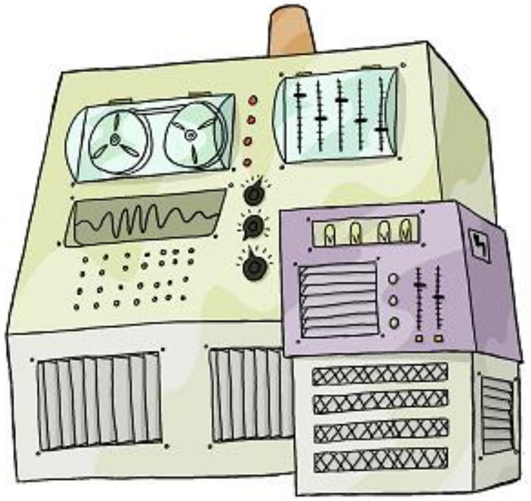
shell

The shell is a program

It has variables

Changing their values

changes its behavior



UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

```
$ set
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
MANPATH=/usr/local/man:/usr/share/man:/usr/man
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/usr/local/bin:/usr/bin:/bin:/cygdrive/c/Windows/system32:
/cygdrive/c/Windows:/cygdrive/c/bin:/cygdrive/c/Python27
PWD=/home/vlad
UID=1000
USERNAME=vlad
```

```
$ set
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
MANPATH=/usr/local/man:/usr/share/man:/usr/man
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/usr/local/bin:/usr/bin:/bin:/cygdrive/c/Windows/system32:
/cygdrive/c/Windows:/cygdrive/c/bin:/cygdrive/c/Python27
PWD=/home/vlad
UID=1000
USERNAME=vlad
```

With no arguments, shows all variables and their values

Standard to use upper-case names

```
$ set
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
MANPATH=/usr/local/man:/usr/share/man:/usr/man
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/usr/local/bin:/usr/bin:/bin:/cygdrive/c/Windows/system32:
/cygdrive/c/Windows:/cygdrive/c/bin:/cygdrive/c/Python27
PWD=/home/vlad
UID=1000
USERNAME=vlad
```

```
$ set
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
MANPATH=/usr/local/man:/usr/share/man:/usr/man
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/usr/local/bin:/usr/bin:/bin:/cygdrive/c/Windows/system32:
/cygdrive/c/Windows:/cygdrive/c/bin:/cygdrive/c/Python27
PWD=/home/vlad
UID=1000
USERNAME=vlad
```

All values are strings

```
$ set
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
MANPATH=/usr/local/man:/usr/share/man:/usr/man
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/usr/local/bin:/usr/bin:/bin:/cygdrive/c/Windows/system32:
/cygdrive/c/Windows:/cygdrive/c/bin:/cygdrive/c/Python27
PWD=/home/vlad
UID=1000
USERNAME=vlad
```

All values are strings

Programs must convert to other types when/as necessary

```
$ set
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
MANPATH=/usr/local/man:/usr/share/man:/usr/man
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/usr/local/bin:/usr/bin:/bin:/cygdrive/c/Windows/system32:
/cygdrive/c/Windows:/cygdrive/c/bin:/cygdrive/c/Python27
PWD=/home/vlad
UID=1000
USERNAME=vlad
```

`int(string)` for numbers

```
$ set
COMPUTERNAME=TURING
HOME=/home/vlad
HOMEDRIVE=C:
HOSTNAME=TURING
HOSTTYPE=i686
MANPATH=/usr/local/man:/usr/share/man:/usr/man
NUMBER_OF_PROCESSORS=4
OS=Windows_NT
PATH=/usr/local/bin:/usr/bin:/bin:/cygdrive/c/Windows/system32:
/cygdrive/c/Windows:/cygdrive/c/bin:/cygdrive/c/Python27
PWD=/home/vlad
UID=1000
USERNAME=vlad
```

`split(':')` for lists

`PATH` controls where the shell looks for programs

PATH controls where the shell looks for programs

```
$ ./analyze
```

Run the `analyze` program in the current directory

PATH controls where the shell looks for programs

```
$ ./analyze
$ /bin/analyze
```

Run the `analyze` program in the /bin directory

PATH controls where the shell looks for programs

```
$ ./analyze
$ /bin/analyze
$ analyze
```

PATH controls where the shell looks for programs

```
$ ./analyze
$ /bin/analyze
$ analyze
```

```
directories = split(PATH, ':')
for each directory:
  if directory/analyze exists, run it
```

`PATH` controls where the shell looks for programs

```
$ ./analyze
$ /bin/analyze
$ analyze
```

```
/usr/local/bin
/usr/bin
/bin
/cygdrive/c/Windows/system32
/cygdrive/c/Windows
/cygdrive/c/bin
/cygdrive/c/Python27
```

PATH controls where the shell looks for programs

```
$ ./analyze
$ /bin/analyze
$ analyze
```

```
/usr/local/bin
/usr/bin
/bin                                /bin/analyze
/cygdrive/c/Windows/system32
/cygdrive/c/Windows
/cygdrive/c/bin                     /cygdrive/c/bin/analyze
/cygdrive/c/Python27
                                    /users/vlad/analyze
```

PATH controls where the shell looks for programs

```
$ ./analyze
$ /bin/analyze
$ analyze
```

```
/usr/local/bin
/usr/bin
/bin                                  /bin/analyze
/cygdrive/c/Windows/system32
/cygdrive/c/Windows
/cygdrive/c/bin                       /cygdrive/c/bin/analyze
/cygdrive/c/Python27
                                      /users/vlad/analyze
```

PATH controls where the shell looks for programs

```
$ ./analyze
$ /bin/analyze
$ analyze
```

```
/usr/local/bin
/usr/bin
/bin                                /bin/analyze
/cygdrive/c/Windows/system32
/cygdrive/c/Windows
/cygdrive/c/bin                     /cygdrive/c/bin/analyze
/cygdrive/c/Python27
                                    /users/vlad/analyze
```

`echo` prints its arguments

`echo` prints its arguments

Use it to show variables' values

`echo` prints its arguments

Use it to show variables' values

```
$ echo hello transylvania!
hello transylvania!
$
```

`echo` prints its arguments

Use it to show variables' values

```
$ echo hello transylvania!
hello transylvania!
$ echo HOME
```

echo prints its arguments

Use it to show variables' values

```
$ echo hello transylvania!
hello transylvania!
$ echo HOME
HOME
$
```

echo prints its arguments

Use it to show variables' values

```
$ echo hello transylvania!
hello transylvania!
$ echo HOME
HOME
$ echo $HOME
/home/vlad
$
```

echo prints its arguments

Use it to show variables' values

```
$ echo hello transylvania!
hello transylvania!
$ echo HOME
HOME
$ echo $HOME
/home/vlad
$
```

Ask shell to replace variable name with value before program runs

echo prints its arguments

Use it to show variables' values

```
$ echo hello transylvania!
hello transylvania!
$ echo HOME
HOME
$ echo $HOME
/home/vlad
$
```

Ask shell to replace variable name with value before program runs

Just like * and ? are expanded before the program runs

`echo` prints its arguments

Use it to show variables' values

```
$ echo hello transylvania!
hello transylvania!
$ echo HOME
HOME
$ echo $HOME ──────→ echo /home/vlad
/home/vlad
$
```

Create variable by assigning to it

Create variable by assigning to it

Change values by reassigning to existing variables

Create variable by assigning to it

Change values by reassigning to existing variables

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
$ SECRET_IDENTITY=Camilla
$ echo $SECRET_IDENTITY
Camilla
$
```

Assignment only changes variable's value

in *this* shell

Assignment only changes variable's value in *this* shell

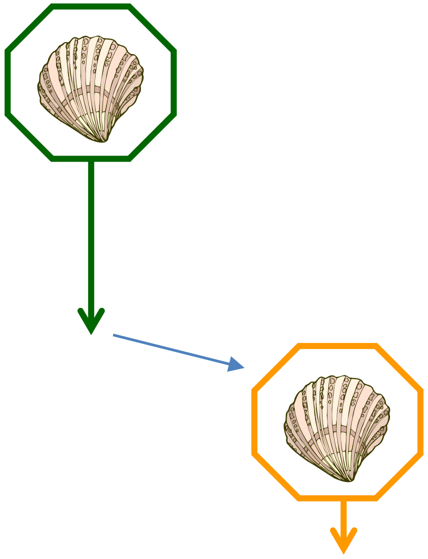
```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
$
```

Assignment only changes variable's value in *this* shell

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
$ bash
$
```

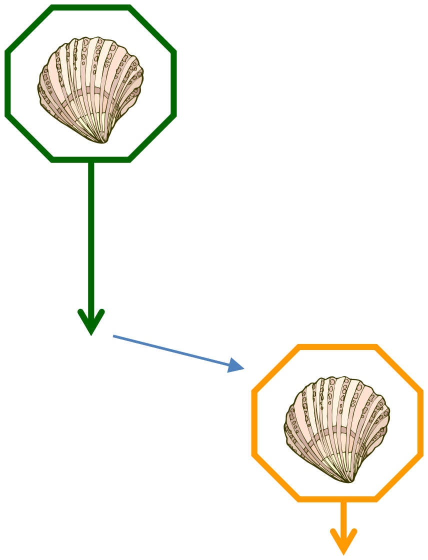
Assignment only changes variable's value in *this* shell

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
$ bash
$
```



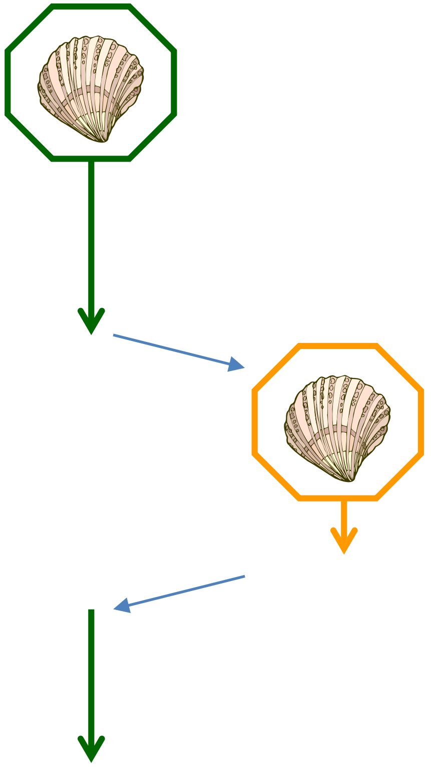
Assignment only changes variable's value in *this* shell

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
$ bash
$ echo $SECRET_IDENTITY
$
```



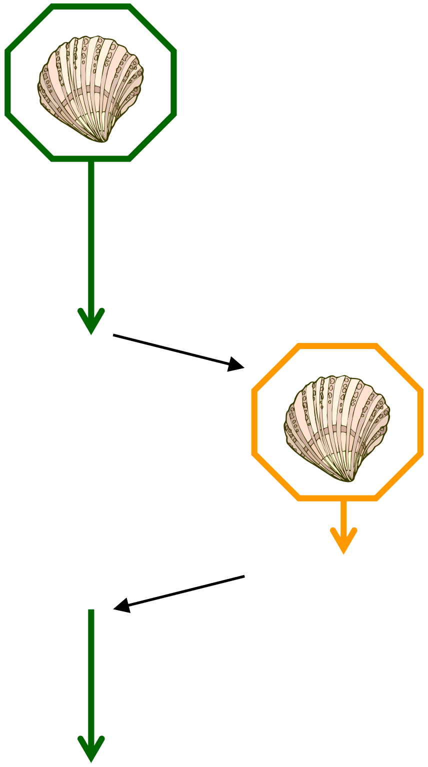
Assignment only changes variable's value in *this* shell

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
$ bash
$ echo $SECRET_IDENTITY
$ exit
$
```



Assignment only changes variable's value in *this* shell

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
$ bash
$ echo $SECRET_IDENTITY
$ exit
$ echo $SECRET_IDENTITY
Dracula
$
```

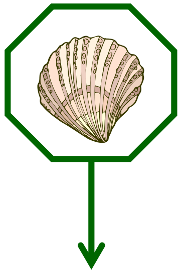


UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Use `export` to signal that the variable should be visible to subprocesses

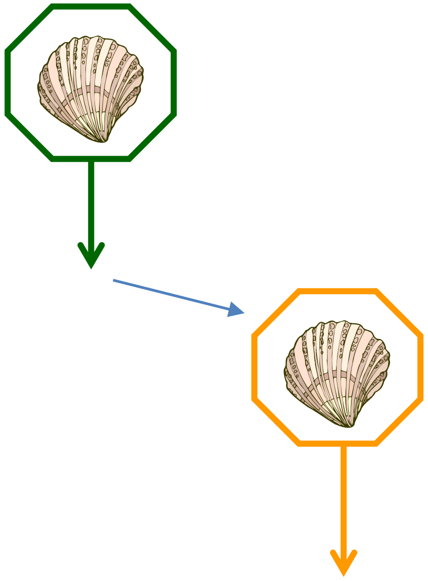
Use `export` to signal that the variable should be visible to subprocesses

```
$ SECRET_IDENTITY=Dracula
$ export SECRET_IDENTITY
$
```



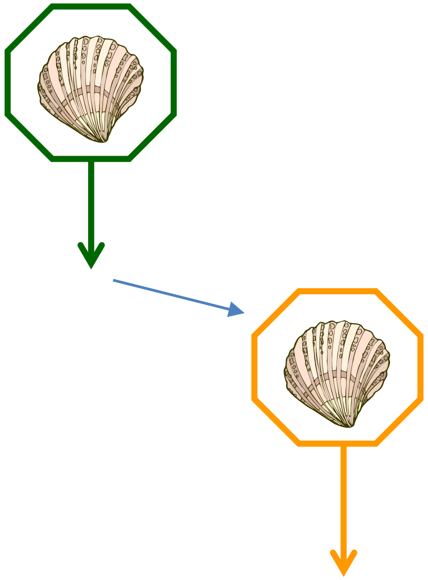
Use `export` to signal that the variable should be visible to subprocesses

```
$ SECRET_IDENTITY=Dracula
$ export SECRET_IDENTITY
$ bash
$
```



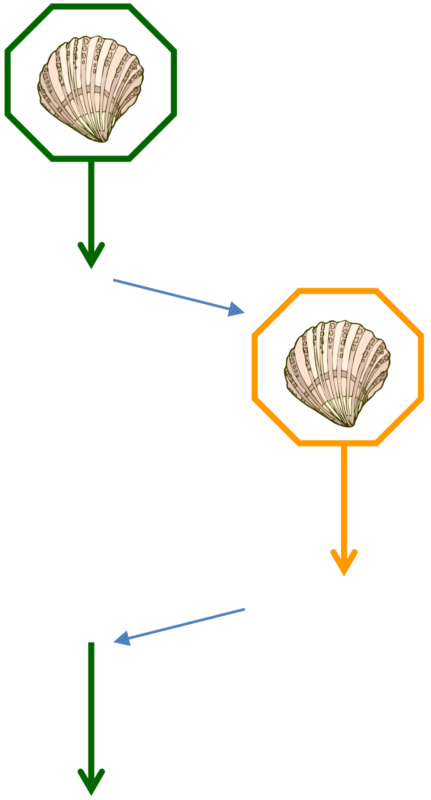
Use `export` to signal that the variable should be visible to subprocesses

```
$ SECRET_IDENTITY=Dracula
$ export SECRET_IDENTITY
$ bash
$ echo $SECRET_IDENTITY
Dracula
$
```



Use `export` to signal that the variable should be visible to subprocesses

```
$ SECRET_IDENTITY=Dracula
$ export SECRET_IDENTITY
$ bash
$ echo $SECRET_IDENTITY
Dracula
$ exit
$
```



Commands in `$HOME/.bashrc` are executed when shell starts

Commands in `$HOME/.bashrc` are executed when shell starts

```
export SECRET_IDENTITY=Dracula
export BACKUP_DIR=$HOME/backup
```

`/home/vlad/.bashrc`

Commands in `$HOME/.bashrc` are executed when shell starts

```
export SECRET_IDENTITY=Dracula
export BACKUP_DIR=$HOME/backup
```

Also common to use `alias` to create shortcuts

Commands in `$HOME/.bashrc` are executed when shell starts

```
export SECRET_IDENTITY=Dracula
export BACKUP_DIR=$HOME/backup
```

Also common to use `alias` to create shortcuts

```
alias backup=/bin/zarble -v --nostir -R 20000 $HOME $BACKUP_DIR
```

Commands in `$HOME/.bashrc` are executed when shell starts

```
export SECRET_IDENTITY=Dracula
export BACKUP_DIR=$HOME/backup
```

Also common to use `alias` to create shortcuts

```
alias backup=/bin/zarble -v --nostir -R 20000 $HOME $BACKUP_DIR
```

Not something you want to type over and over

The Unix Shell

Advanced Shell Tricks

Created by Steve Crouch

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

“How should I do this?”

With smartphones, you'll often hear people say something like

"How should I do this?"

"There's an app for that… check this out!"

"How should I do this?"

With smartphones, you'll often hear people say something like

"There's an app for that... check this out!"

Whereas Unix shell programmers will say

"There's a shell trick for that...
check this out!"

In previous lessons, we've seen how to:

– Combine existing programs using pipes & filters

```
$ wc -l *.pdb | sort | head -1
```

In previous lessons, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files

```
$ wc -l *.pdb > lengths
```

In previous lessons, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
```

In previous lessons, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

Very powerful when used together

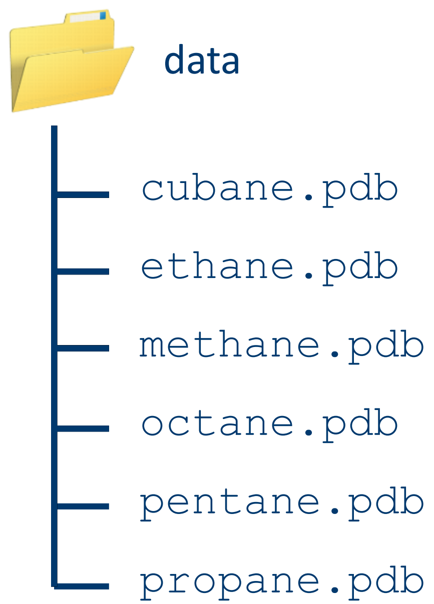
In previous lessons, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

Very powerful when used together

But there are other useful things we can do with these – let's take a look…

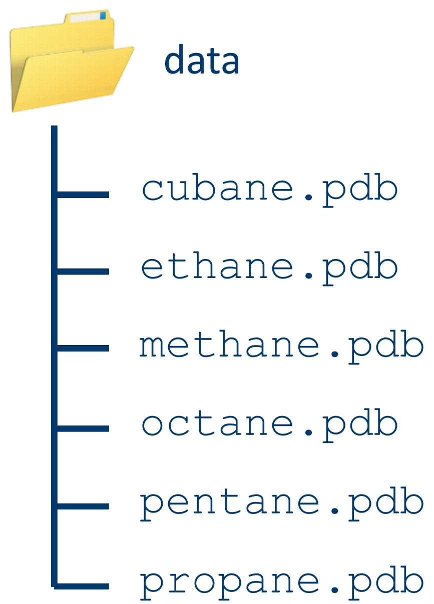
First, let's revisit redirection…



First, let’s revisit redirection…

```
$ ls *.pdb > files
```

list all pdb files
redirect to a file

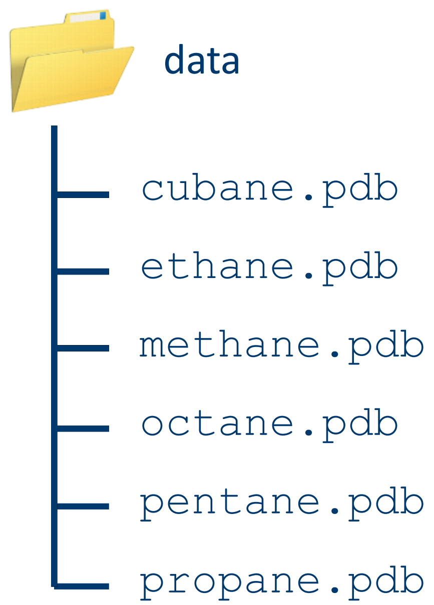


First, let's revisit redirection…

```
$ ls *.pdb > files
```

list all pdb files
redirect to a file

The 'redirection' operator

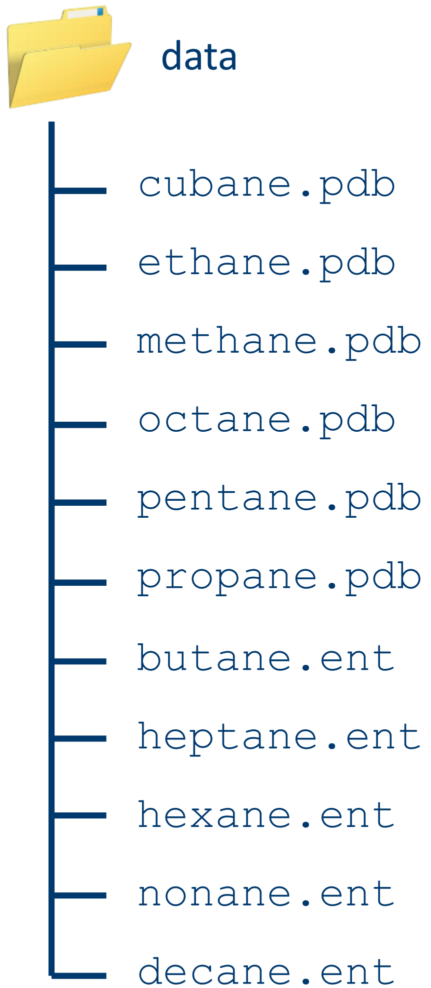


First, let's revisit redirection...

```
$ ls *.pdb > files
```

list all pdb files
redirect to a file

But what about adding this together with other results generated later?



UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

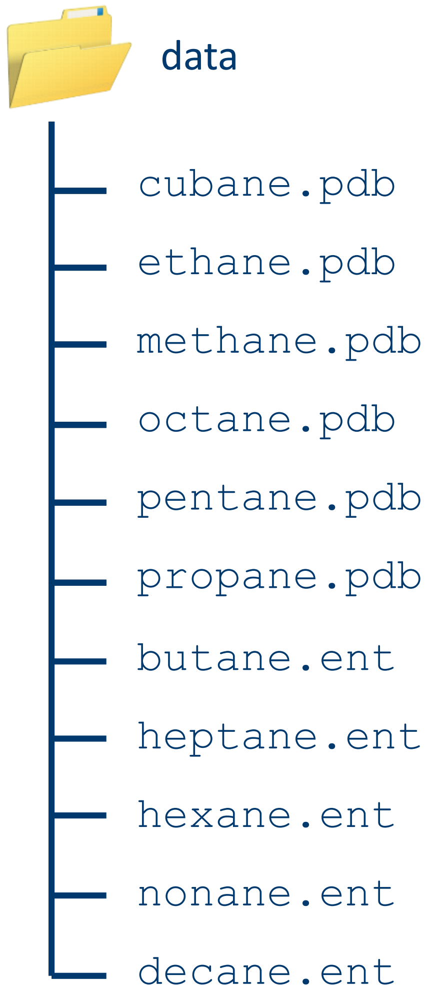
First, let's revisit redirection…

```
$ ls *.pdb > files
```

list all pdb files redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
```



First, let's revisit redirection...

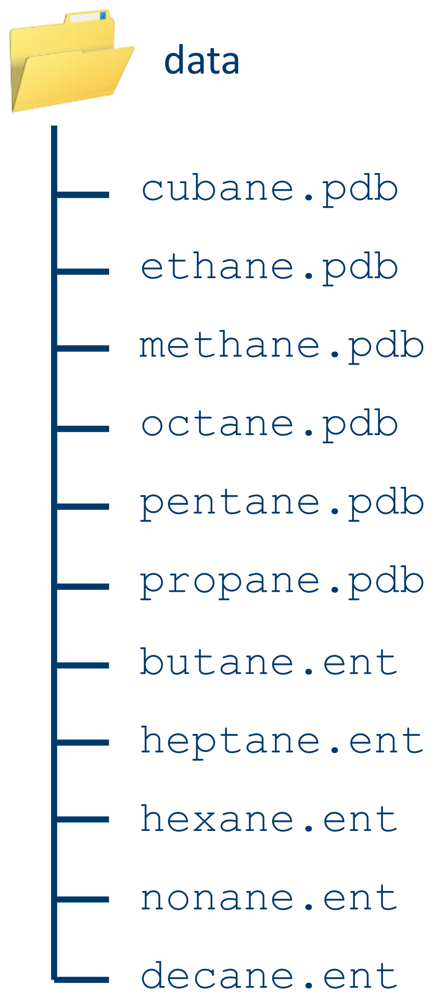
```
$ ls *.pdb > files
```

list all pdb files redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
```

We just want the ent files



First, let's revisit redirection...

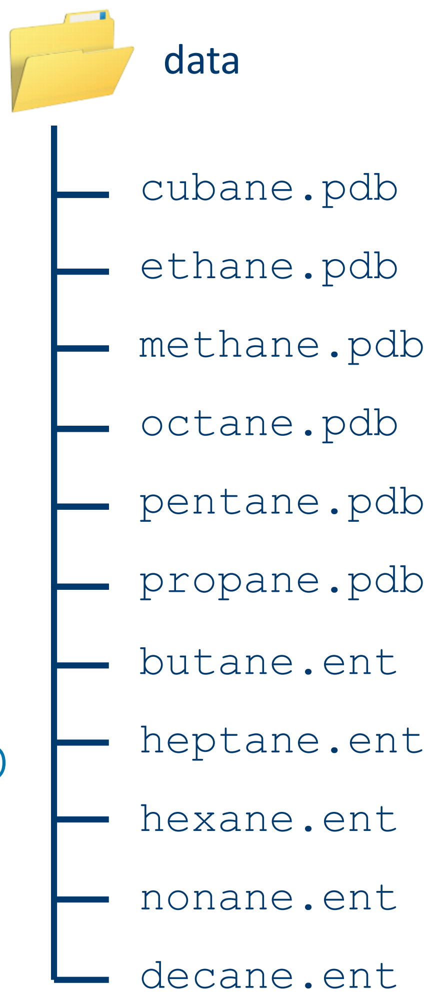
```
$ ls *.pdb > files
```

list all pdf files redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

append files into a single new file



First, let's revisit redirection...

```
$ ls *.pdb > files
```

list all pdb files redirect to a file

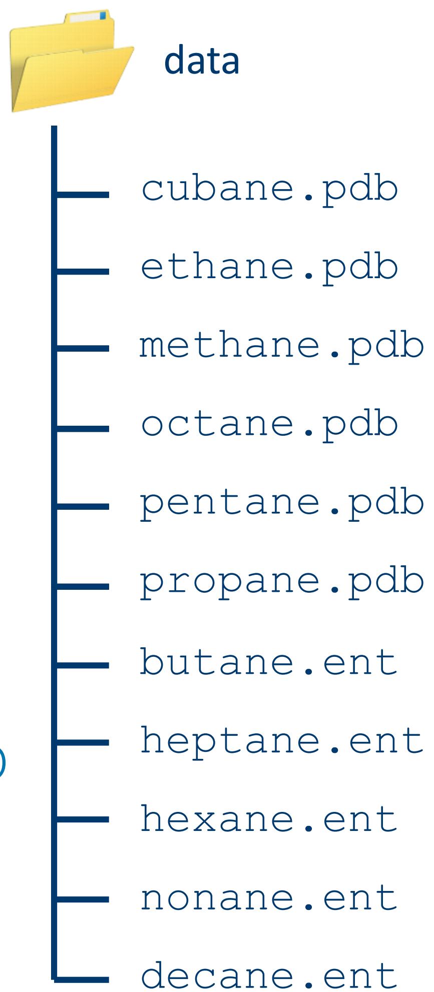
But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

append files into a single new file

Instead, we can do...

```
$ ls *.ent >> files
```



UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

First, let's revisit redirection…

```
$ ls *.pdb > files
```

list all pdb files redirect to a file

But what about adding this together with other results generated later?

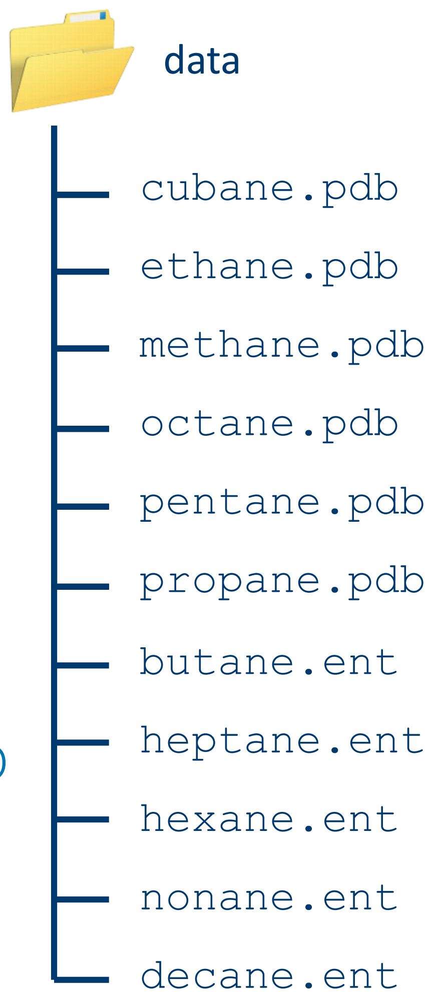
```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

append files into a single new file

Instead, we can do…

```
$ ls *.ent >> files
```

Note the double >'s – the append' operator

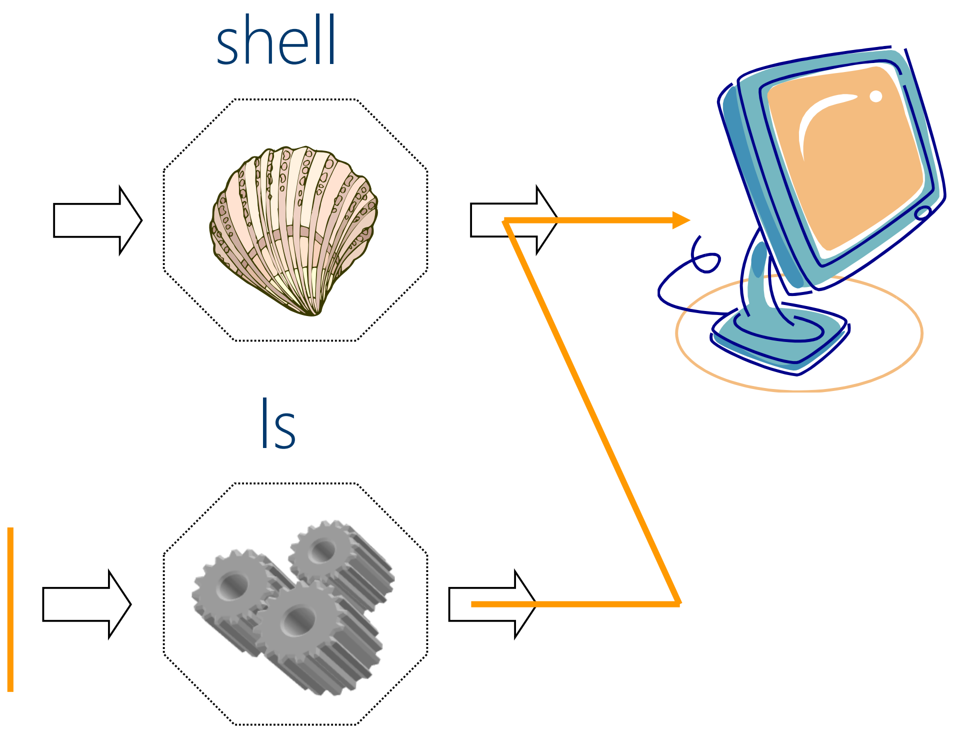


We know that…

Normally, standard output is directed to a display:

We know that…

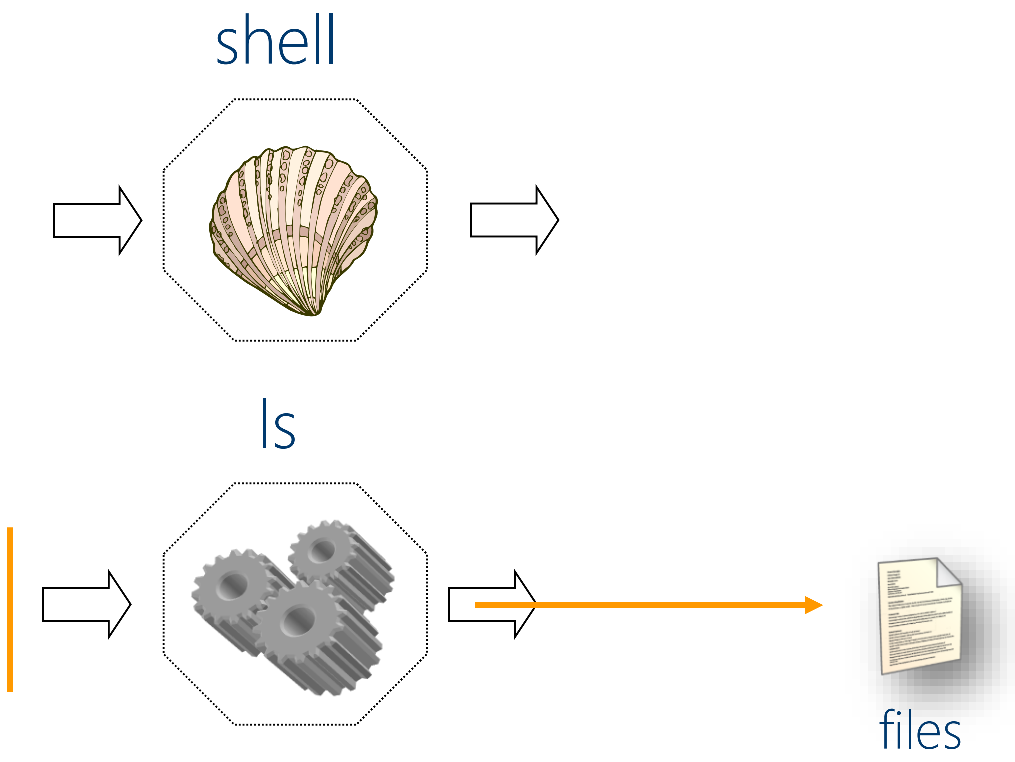
Normally, standard output is directed to a display:



We know that...

Normally, standard output is directed to a display:

But we have redirected it to a file instead:



But what happens with error messages?

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or
directory
```

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or
directory
```

No files are listed in *files*, as you might expect.

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or
directory
```

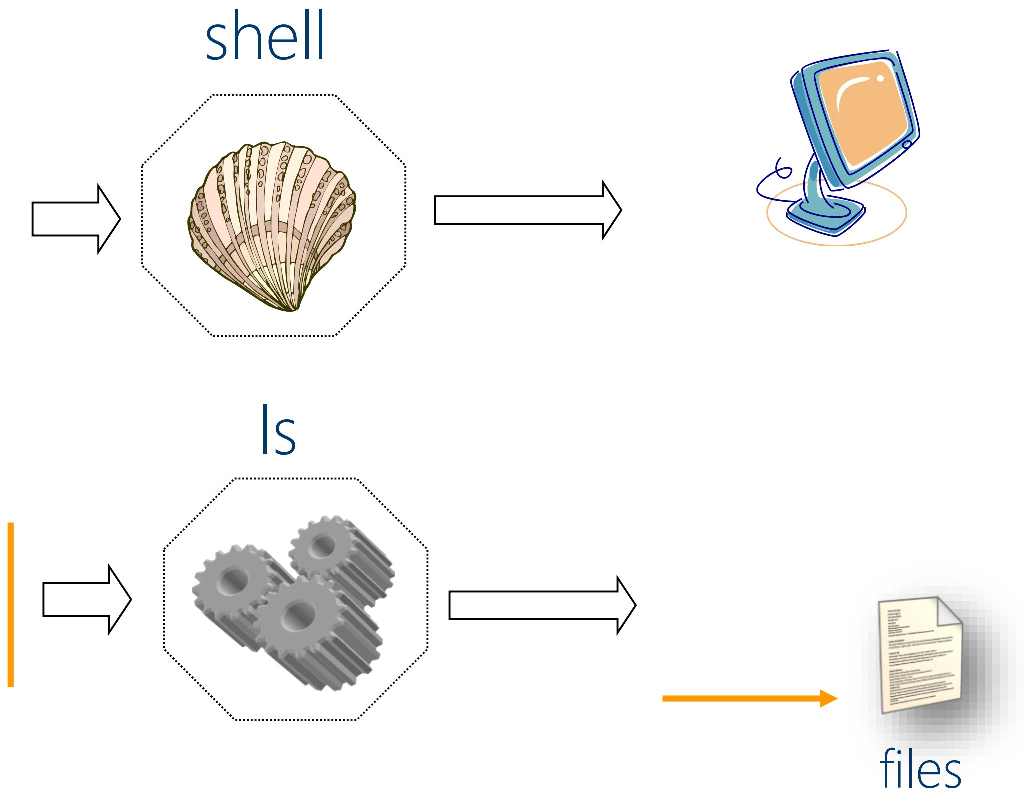
No files are listed in *files*, as you might expect.

But why isn't the error message in *files*?

This is because error messages are sent to the standard error (stderr), separate to stdout

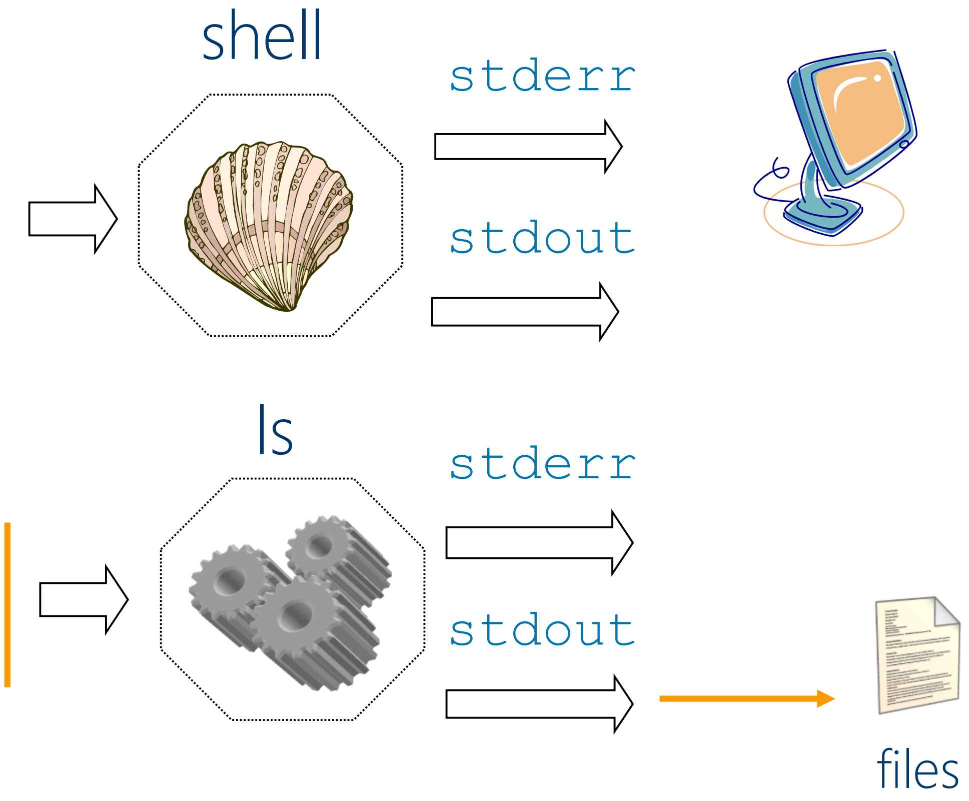
This is because error messages are sent to the standard error (stderr), separate to stdout

So what was happening with the previous example?



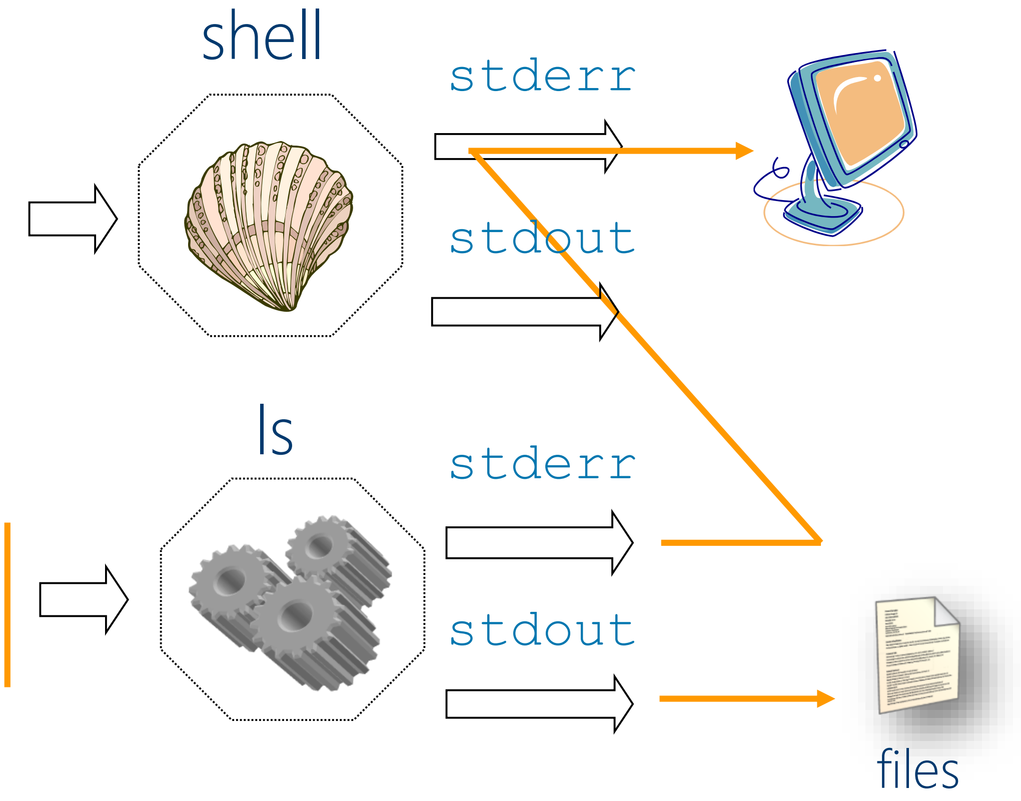
This is because error messages are sent to the standard error (stderr), separate to stdout

So what was happening with the previous example?



This is because error messages are sent to the standard error (stderr), separate to stdout

So what was happening with the previous example?



We can capture standard error as well as standard output

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Redirect as before, but with a slightly different operator

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in error-log

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in error-log

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2>
error-log
```

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in error-log

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2>
error-log
```

We can use both `stdout` and `stderr` redirection – at the same time

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in error-log

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2>
error-log
```

Which would give us contents of */usr* in *files* as well.

So why a ‘2’ before the ‘>’ ?

So why a '2' before the '>' ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2>
error-log
```

So why a ‘2’ before the ‘>’ ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2>
error-log
```

Refers to stdout

Refers to stderr

So why a ‘2’ before the ‘>’ ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2> error-log
```

To just redirect both to the same file we can also do:

```
$ ls /usr /some/nonexistent/path &> everything
```

With ‘&’ denoting both stdout and stderr

So why a '2' before the '>' ?

Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2>
error-log
```

To just redirect both to the same file we can also do:

```
$ ls /usr /some/nonexistent/path &>
everything
```

With '&' denoting both stdout and stderr
We can also use append for each of these too:

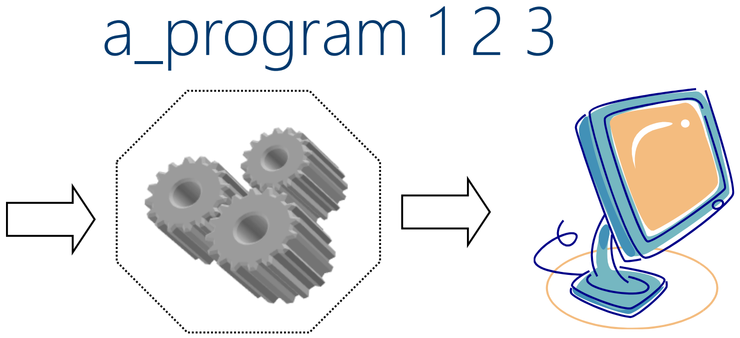
```
$ ls /usr /some/nonexistent/path 1>> files
2>> error-log
```

>	1>	Redirect stdout to a file
	2>	Redirect stderr to a file
	&>	Redirect both stdout and stderr to the same file

>	1>	Redirect stdout to a file
	2>	Redirect stderr to a file
	&>	Redirect both stdout and stderr to the same file
>>	1>>	Redirect and append stdout to a file
	2>>	Redirect and append stderr to a file
	&>>	Redirect and append both stdout and stderr to a file

We’ve seen how pipes and filters work with using a single program on some input data…

We've seen how pipes and filters work with using a single program on some input data…

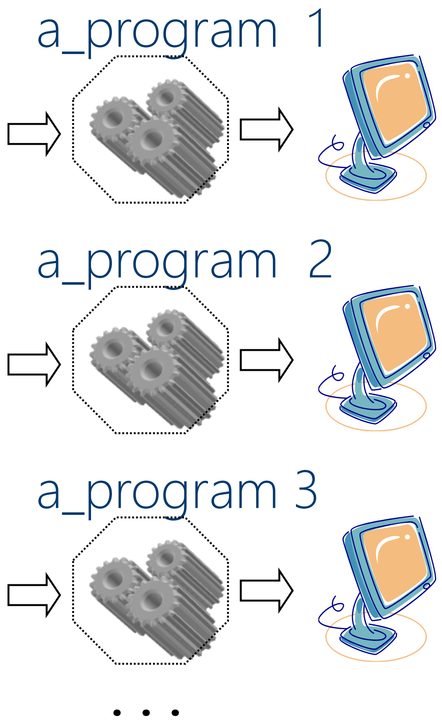


We've seen how pipes and filters work with using a single program on some input data…

But what about running the same program *separately*, for each input?

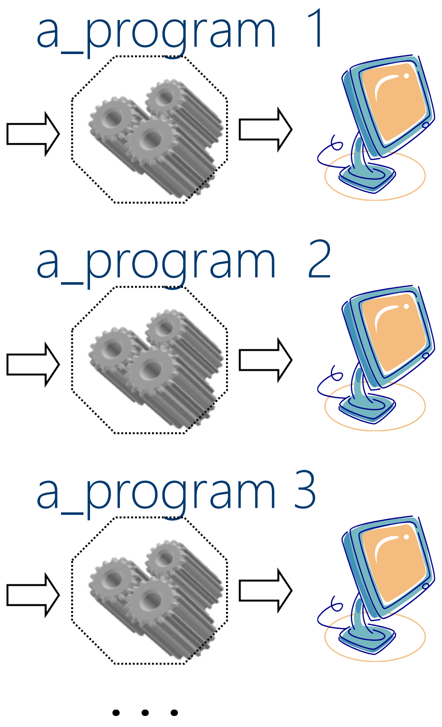
We've seen how pipes and filters work with using a single program on some input data...

But what about running the same program *separately*, for each input?



We've seen how pipes and filters work with using a single program on some input data...

But what about running the same program *separately*, for each input?

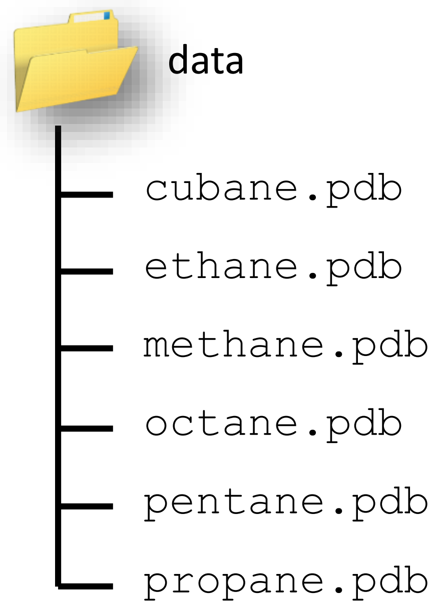


We can use *loops* for this...

So what can we do with loops?

So what can we do with loops?

Let’s go back to our first set of pdb files, and assume we want to compress each of them

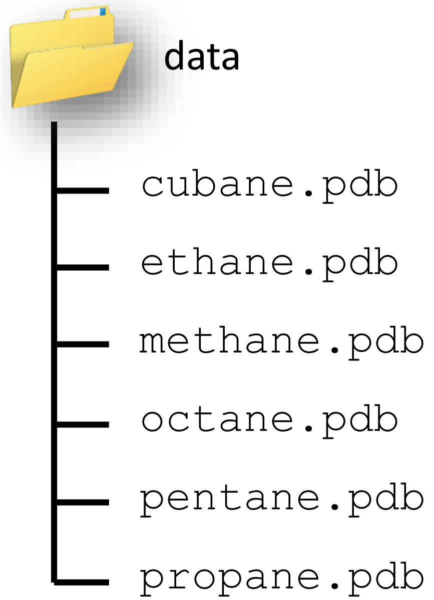


So what can we do with loops?

Let’s go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated
73%)
```



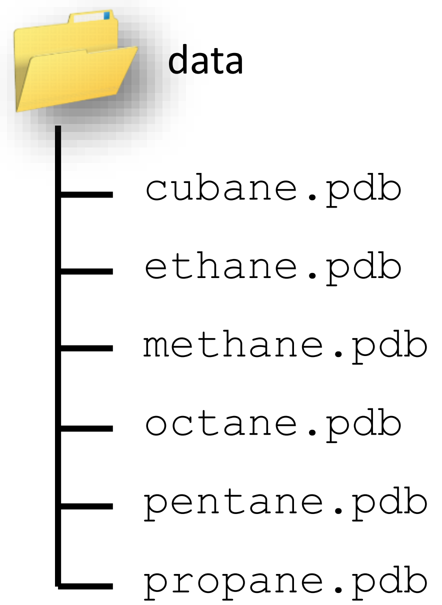
So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

← typical output from the zip command



So what can we do with loops?

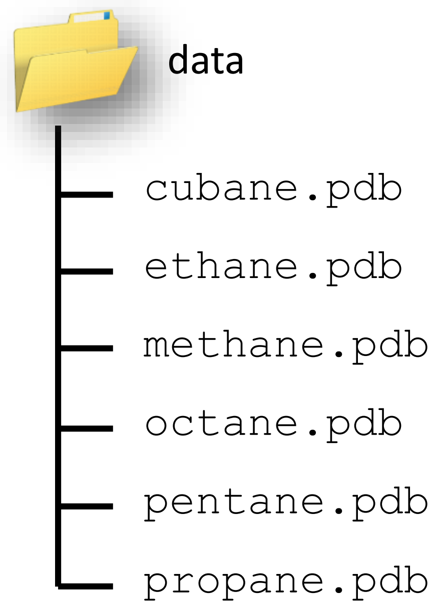
Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated
73%)
```

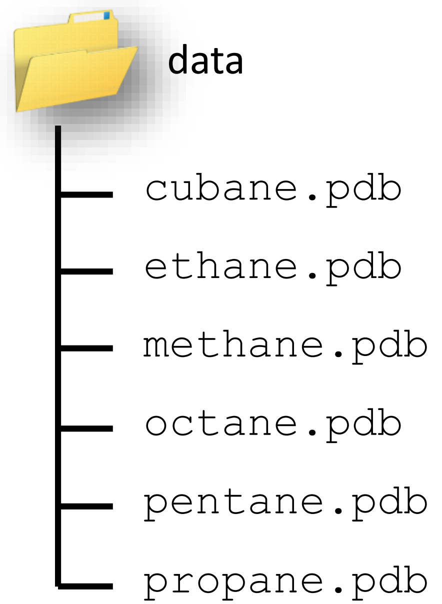
The zip file we wish to create

typical output from the zip command



So what can we do with loops?

Let’s go back to our first set of pdb files, and assume we want to compress each of them



We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated
73%)
```

typical output from the zip command

The zip file we wish to create

The file(s) we wish to add to the zip file

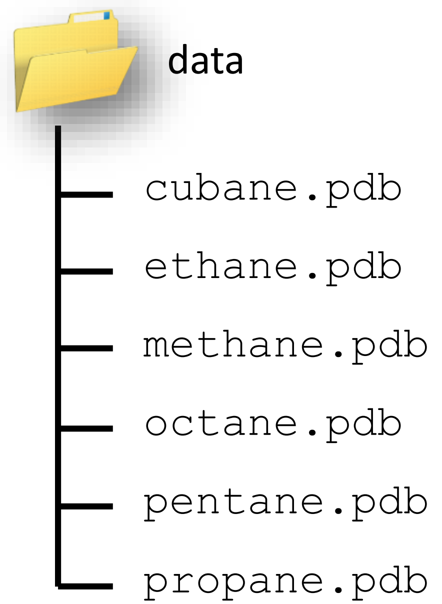
So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated
73%)
```

Not efficient for many files



Using a loop, we can iterate over each file, and run zip on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

For each pdb file in this directory...

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

Run this command

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

This is the end of the loop

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

The semicolons separate each part of the loop construct

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

This expands to a list of every pdb file

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

This expands to a list of every pdb file

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

We reference the ‘file’ variable, and use ‘.’ to add the zip extension to the filename

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$  for file in *.pdb; do zip $file.zip $file; done
```

We reference the ‘file’ variable again

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$  for file in *.pdb; do zip $file.zip $file; done
   adding: cubane.pdb (deflated 73%)
   adding: ethane.pdb (deflated 70%)
   adding: methane.pdb (deflated 66%)
   adding: octane.pdb (deflated 75%)
   adding: pentane.pdb (deflated 74%)
   adding: propane.pdb (deflated 71%)
```

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$  for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
  adding: methane.pdb (deflated 66%)
  adding: octane.pdb (deflated 75%)
  adding: pentane.pdb (deflated 74%)
  adding: propane.pdb (deflated 71%)
```

In one line, we've ended up with all files zipped

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$  for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
  adding: methane.pdb (deflated 66%)
  adding: octane.pdb (deflated 75%)
  adding: pentane.pdb (deflated 74%)
  adding: propane.pdb (deflated 71%)
```

In one line, we've ended up with all files zipped

```
$  ls *.zip
cubane.pdb.zip     methane.pdb.zip  pentane.pdb.zip
ethane.pdb.zip     octane.pdb.zip   propane.pdb.zip
```

Now instead, what if we wanted to output the first line of each pdb file?

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

```
==> cubane.pdb <==
COMPND      CUBANE

==> ethane.pdb <==
COMPND      ETHANE

==> methane.pdb <==
COMPND      METHANE
…
```

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

```
==> cubane.pdb <==
COMPND      CUBANE

==> ethane.pdb <==
COMPND      ETHANE

==> methane.pdb <==
COMPND      METHANE
…
```

← head produces this (it's not in the file)

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

```
==> cubane.pdb <==
COMPND      CUBANE

==> ethane.pdb <==
COMPND      ETHANE

==> methane.pdb <==
COMPND      METHANE
…
```

head produces this (it's not in the file)

this is actually the first line in this file!

Now instead, what if we wanted to output the first line of each pdb file?

We could use `head -1 *.pdb` for that, but it would produce:

```
==> cubane.pdb <==
COMPND      CUBANE

==> ethane.pdb <==
COMPND      ETHANE

==> methane.pdb <==
COMPND      METHANE
…
```

head produces this (it's not in the file)

this is actually the first line in this file!

Perhaps we only want the actual first lines…

However, using a loop:

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
```

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
```

We use $file as we did before, but this time with the head command

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
COMPND      CUBANE
COMPND      ETHANE
COMPND      METHANE
COMPND      OCTANE
COMPND      PENTANE
COMPND      PROPANE
```

What if we wanted this list sorted in reverse afterwards?

What if we wanted this list sorted in reverse afterwards?

Simple!

```
$ (for file in ls *.pdb; do head -1 $file; done) | sort -r
```

What if we wanted this list sorted in reverse afterwards?

Simple!

```
$(for file in ls *.pdb; do head -1 $file; done) | sort -r
```

Using a pipe, we can just add this on the end

What if we wanted this list sorted in reverse afterwards?

Simple!

```
$ (for file in ls *.pdb; do head -1 $file; done) | sort -r
COMPND      PROPANE
COMPND      PENTANE
COMPND      OCTANE
COMPND      METHANE
COMPND      ETHANE
COMPND      CUBANE
```

`zip`	Create a compressed zip file with other files in it
`for …; do … done;`	Loop over a list of data and run a command once for each element in the list

Checkpoint 5

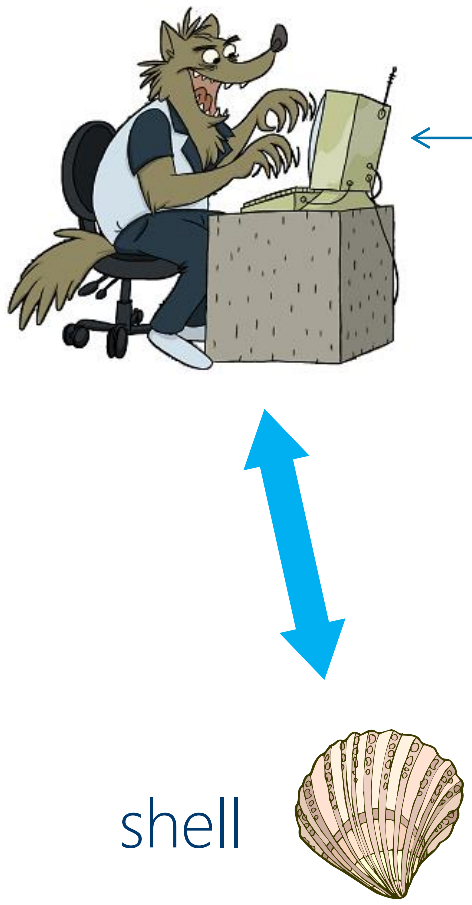
- Please complete the *What is Model Survey Results*:
  - Blackboard > Course Content > Week 3 (Sept. 28- Oct. 2) > Monday Sept. 28 > Checkpoint 5

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

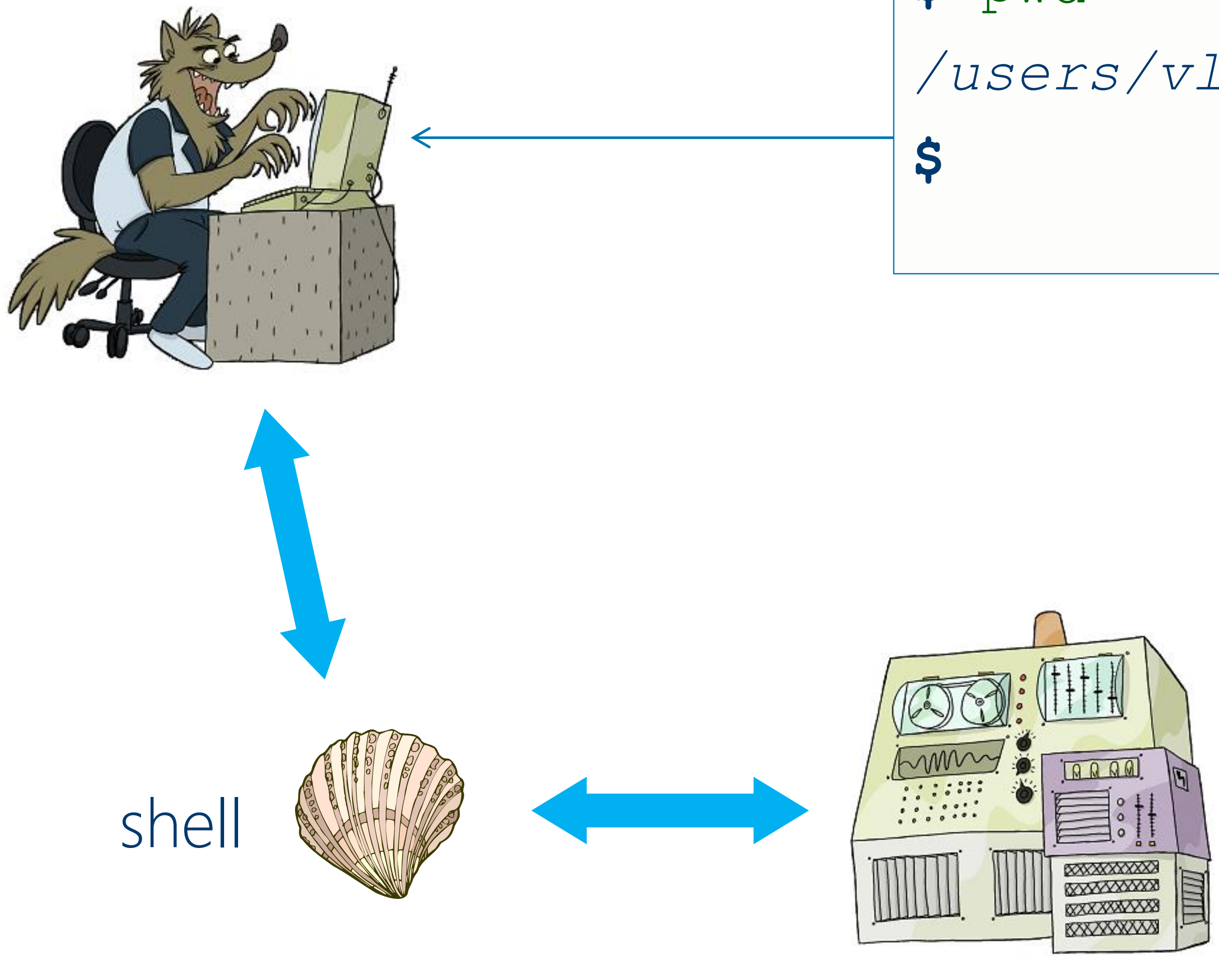
The Unix Shell

The Secure Shell

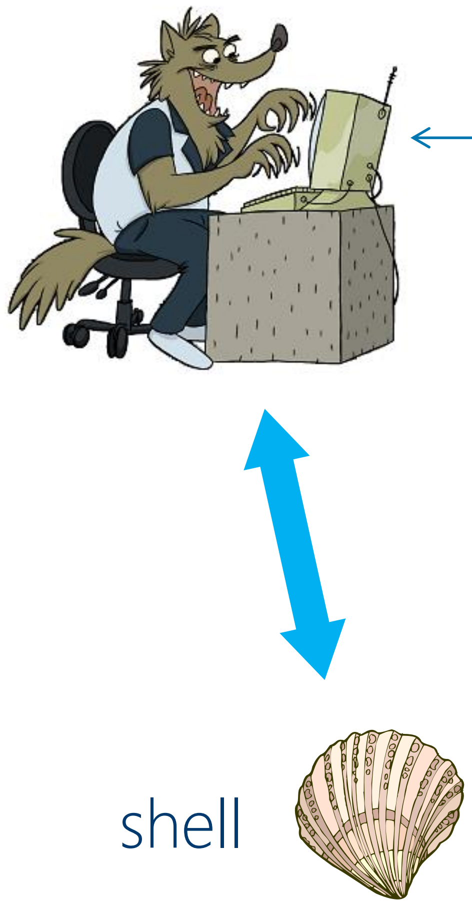
Created by Elango Cheran



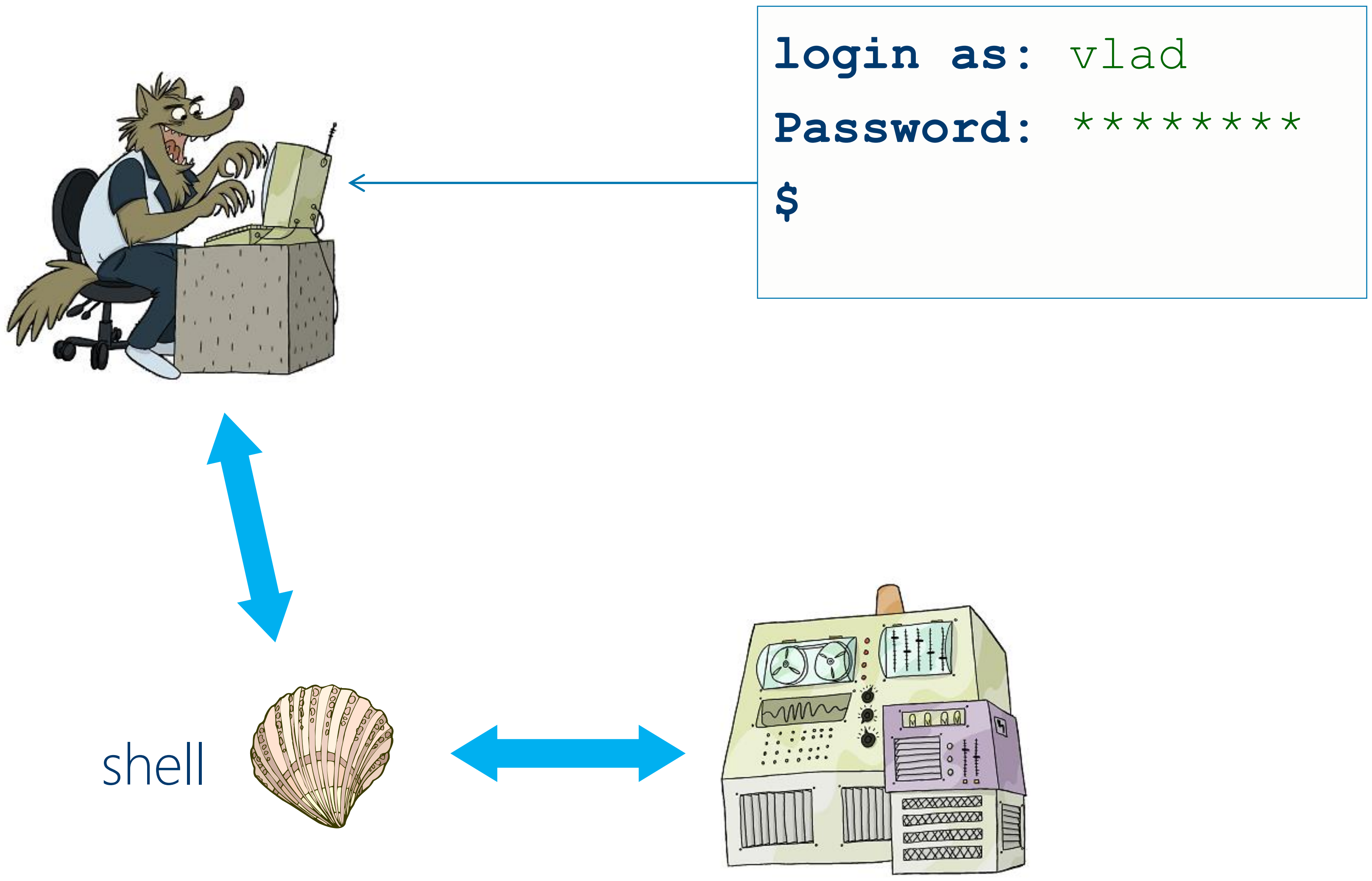
```
$ pwd
```



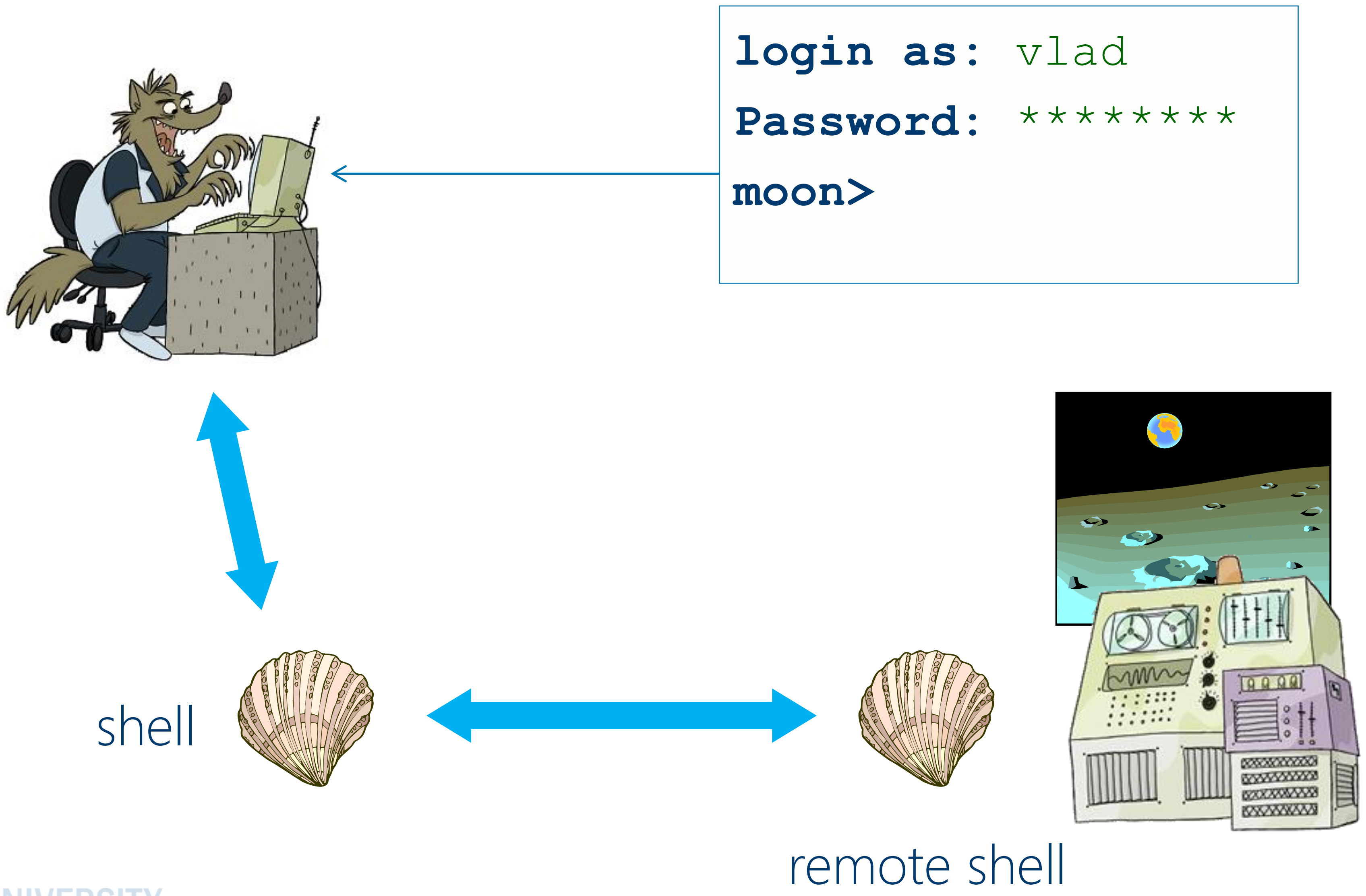
```
$ pwd
/users/vlad
$
```



```
login as: vlad
Password: ********
```

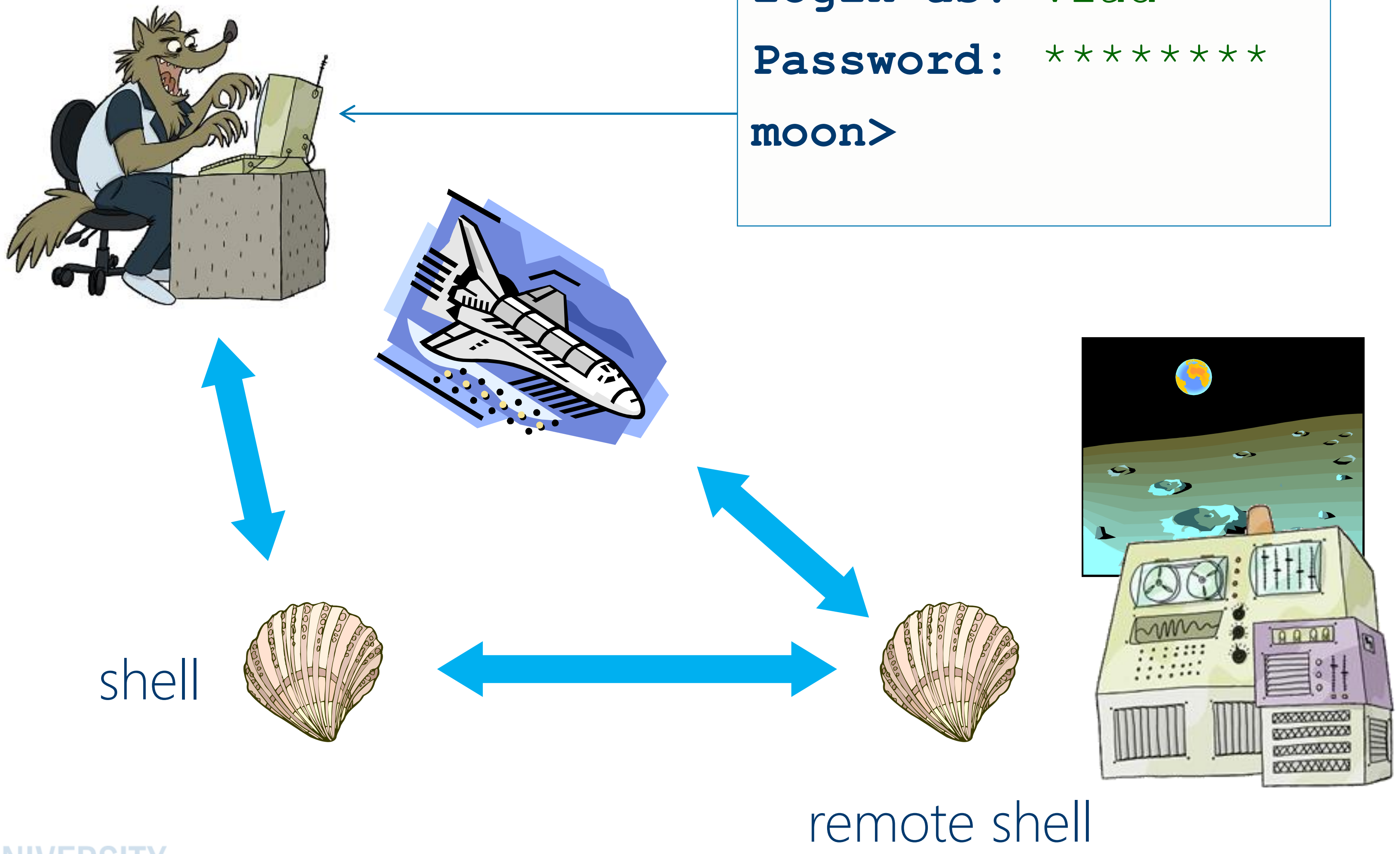


```
login as: vlad
Password: ********
$
```



```
login as: vlad
Password: ********
moon>
```

```
login as: vlad
Password: ********
moon>
```



```
$ pwd
/users/vlad
$ ssh vlad@moon
Password:
```

```
$ pwd
/users/vlad
$ ssh vlad@moon
Password: ***
Access denied
Password:
```

```
$ pwd
/users/vlad
$ ssh vlad@moon
Password: ***
Access denied
Password: ********
moon> pwd
/home/vlad
moon> ls -F
bin/      cheese.txt   dark_side/   rocks.cfg
```

```
$ pwd
/users/vlad
$ ssh vlad@moon
Password: ***
Access denied
Password: ********
moon> pwd
/home/vlad
moon> ls -F
bin/     cheese.txt   dark_side/   rocks.cfg
moon> exit
$ pwd
/users/vlad
```

```
$ ssh vlad@moon
Password: ********
moon> pwd
/home/vlad
moon> ls -F
bin/      cheese.txt   dark_side/   rocks.cfg
moon> exit
$ pwd
/users/vlad
$ ls -F
bin/         data/      mail/       music/
notes.txt    papers/    pizza.cfg   solar/
solar.pdf    swc/
```

```
$ scp vlad@moon:/home/vlad/cheese.txt
    vlad@earth:/users/vlad
```

source file...

```
$ scp vlad@moon:/home/vlad/cheese.txt
        vlad@earth:/users/vlad
```

source file...

...to destination directory

```
$ scp vlad@moon:/home/vlad/cheese.txt
          vlad@earth:/users/vlad
```

source file…

…to destination directory

source and destination are written as

user@computer:path

```
$ scp vlad@moon:/home/vlad/cheese.txt
        vlad@earth:/users/vlad
Password: ********
cheese.txt                 100%  9  1.0 KB/s 00:00
```

```
$ scp vlad@moon:/home/vlad/cheese.txt
        vlad@earth:/users/vlad
$ scp -r vlad@moon:/home/vlad/dark_side
        vlad@earth:/users/vlad
```

`-r` indicates a directory and its contents

```
$ scp -r vlad@moon:/home/vlad/dark_side
         vlad@earth:/users/vlad
$ scp -r vlad@moon:/home/vlad/dark_side
         /users/vlad
$ pwd
/users/vlad
$ scp -r vlad@moon:/home/vlad/dark_side
         .
```

same destination path

```
$ ssh vlad@moon
Password: ********
moon> df -h
Filesystem   Size  Used  Avail  Use% Mounted On
/dev/sda1    7.9G  2.1G  5.5G   28%  /
/dev/sda2    791G  150G  642G   19%  /home
moon> df -h > usage.txt
moon> exit
$ scp vlad@moon:/home/vlad/usage.txt .
Password: ********
usage.txt                   100%  134  1.0 KB/s 00:00
```

```
$ ssh vlad@moon 'df -h'
Password: ********
Filesystem    Size  Used  Avail  Use% Mounted On
/dev/sda1     7.9G  2.1G  5.5G   28%  /
/dev/sda2     791G  150G  642G   19%  /home
```

```
$ ssh vlad@moon 'df -h'
Password: ********
Filesystem    Size  Used  Avail  Use% Mounted On
/dev/sda1     7.9G  2.1G  5.5G   28%  /
/dev/sda2     791G  150G  642G   19%  /home
$ ssh vlad@moon 'df -h' >> usage.log
Password: ********
$ ls -F
bin/          data/      mail/      music/
notes.txt     papers/    pizza.cfg  solar/
solar.pdf     swc/       usage.log  usage.txt
```

same result

character stream

```
$ echo "open sesame, please"  |   ssh
    vlad@moon  'cat > magic.txt'
Password: ********
```

character stream

remote shell receives
stream from pipe

```
$ echo "open sesame, please"  |  ssh
    vlad@moon  'cat > magic.txt'
Password: ********
```

redirection within
remote shell

remote command receives
input piped to `ssh`
cat repeats input stream
as output

```
$ ssh vlad@moon 'ls -F /home/vlad'
Password: ********
bin/     cheese.txt    dark_side/    rocks.cfg
$ echo "open sesame, please"  |  ssh
   vlad@moon  'cat > magic.txt'
Password: ********
$ ssh vlad@moon 'ls -F /home/vlad'
Password: ********
bin/     cheese.txt    dark_side/    magic.txt
rocks.cfg
```

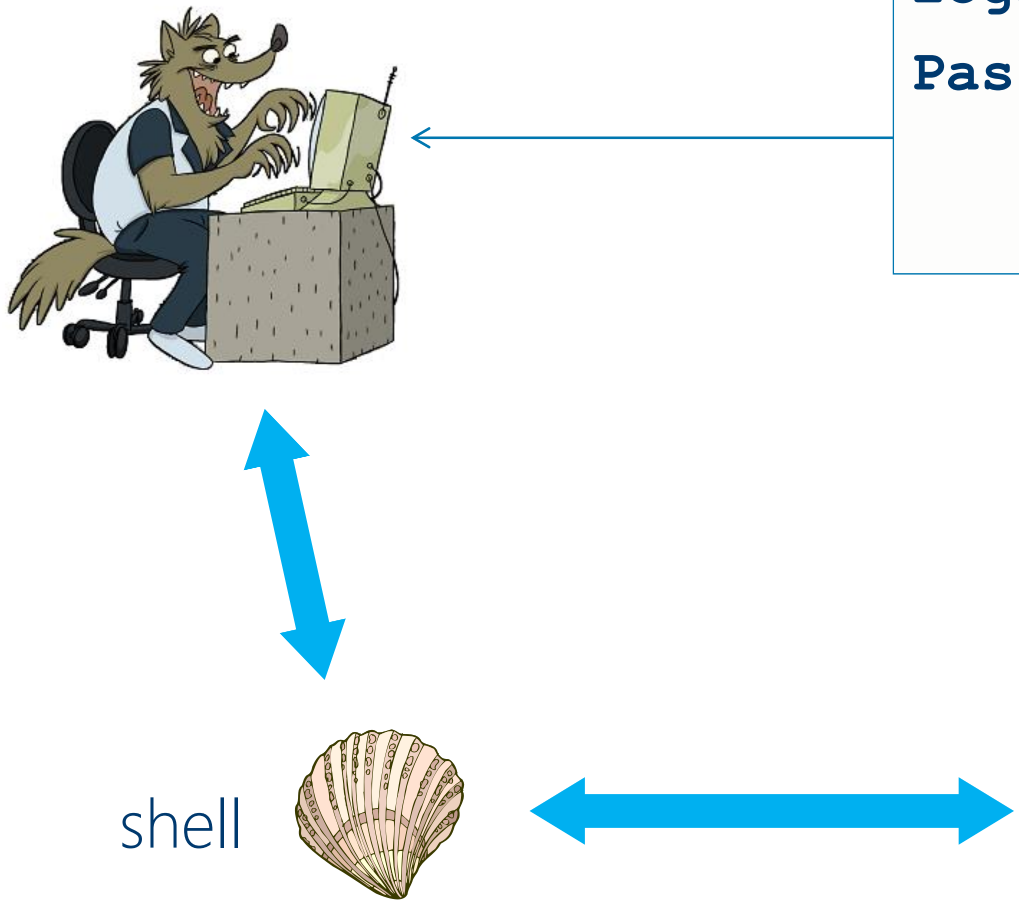
before

after

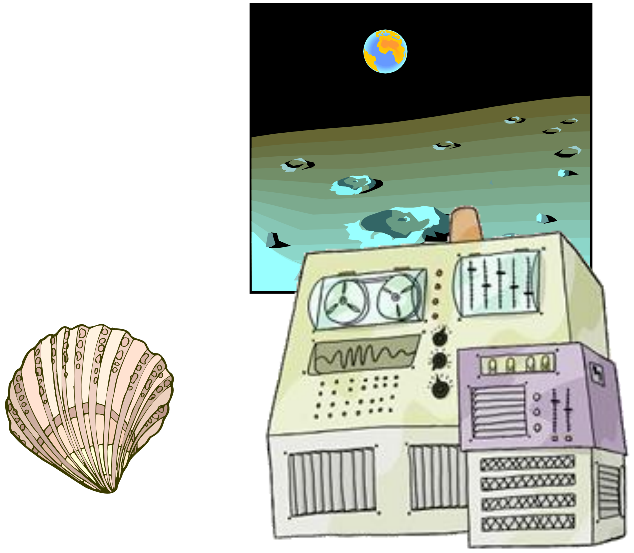
```
$ ssh vlad@moon 'ls -F /home/vlad'
Password: ********
bin/    cheese.txt   dark_side/   rocks.cfg
$ echo "open sesame, please"  |  ssh
   vlad@moon  'cat > magic.txt'
Password: ********
$ ssh vlad@moon 'ls -F /home/vlad'
Password: ********
bin/    cheese.txt   dark_side/   magic.txt
rocks.cfg
$ scp vlad@moon:/home/vlad/magic.txt .
Password: ********
```

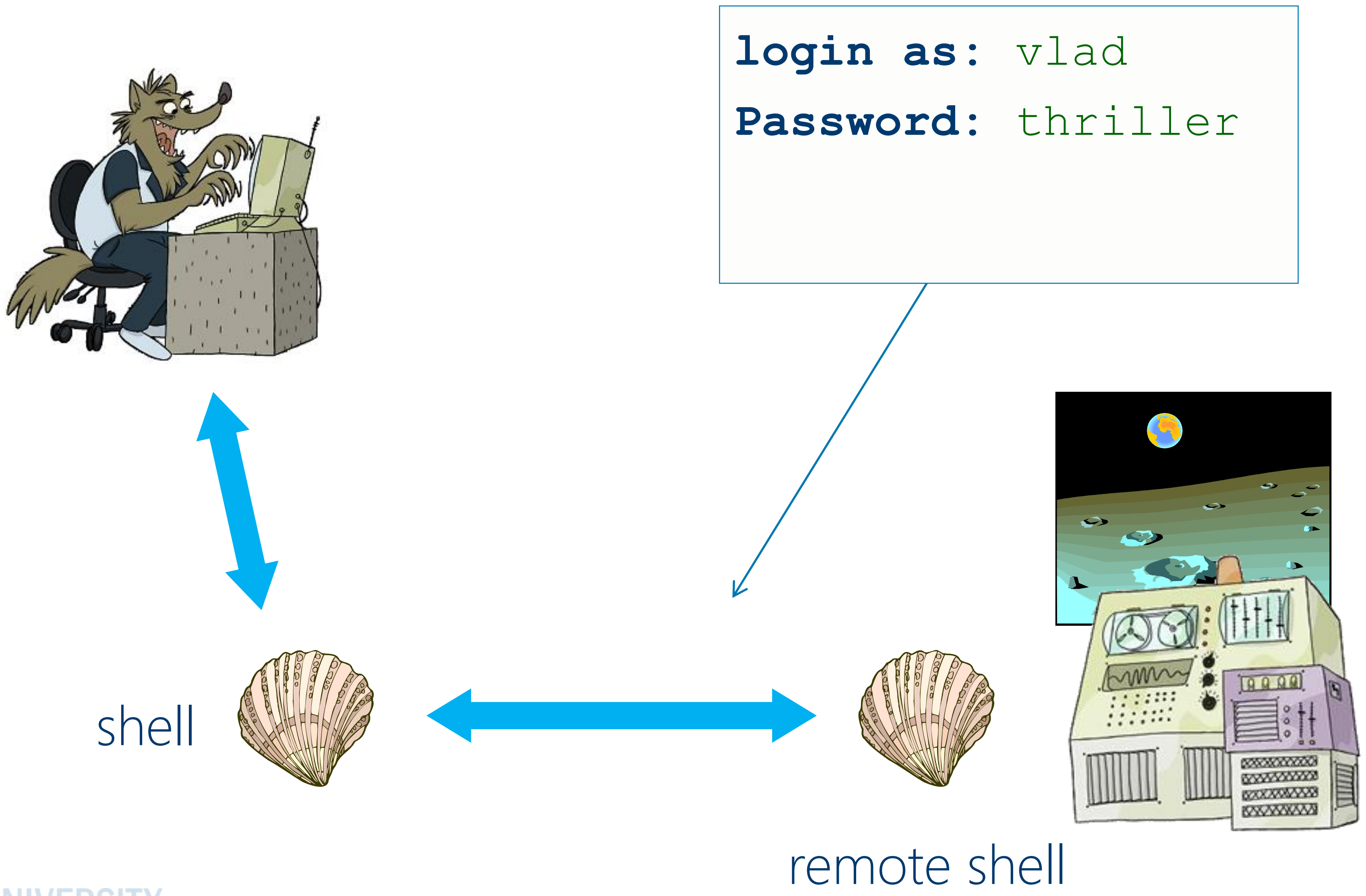
before

after

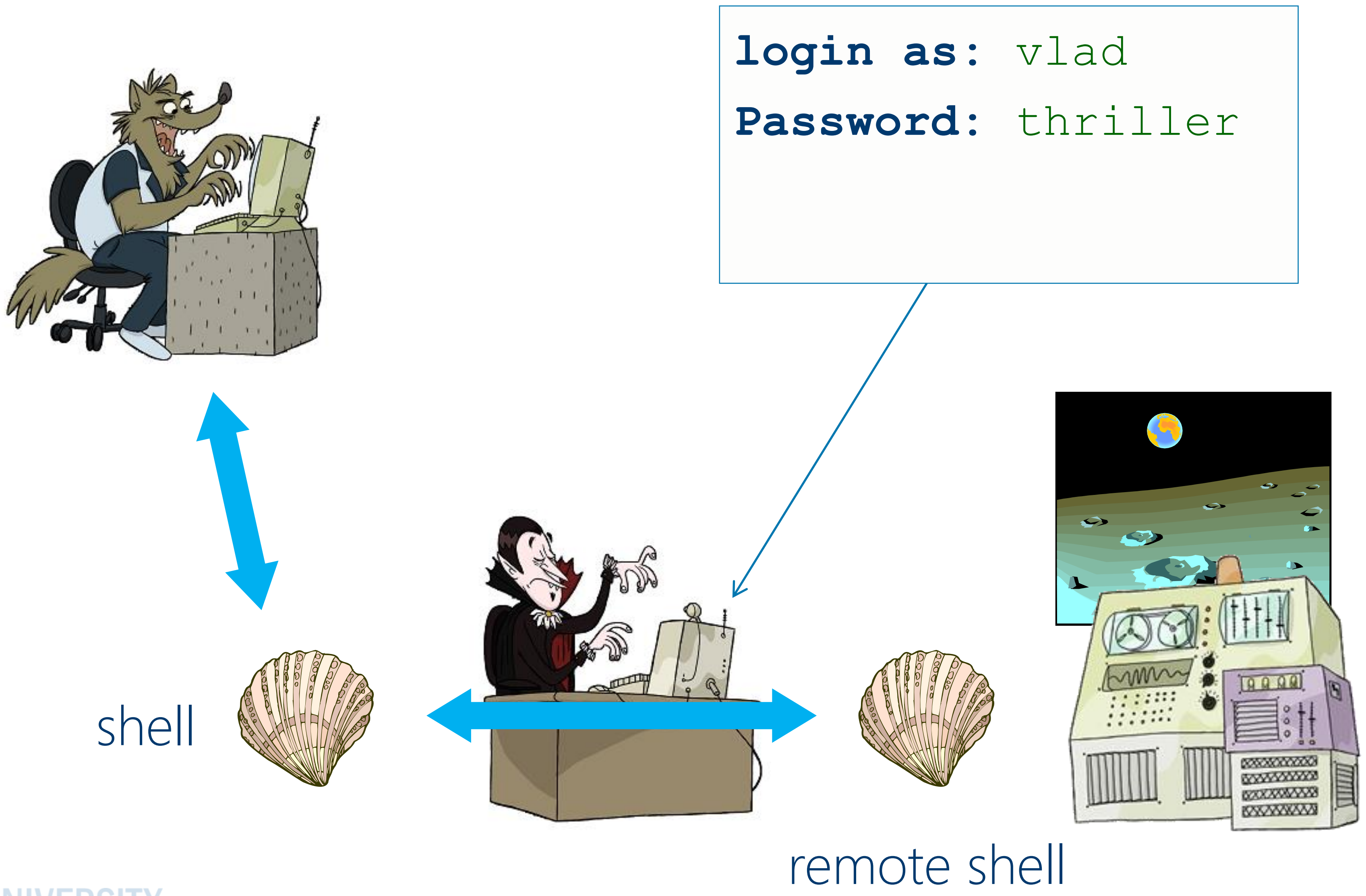


```
login as:  vlad
Password:  ********
```

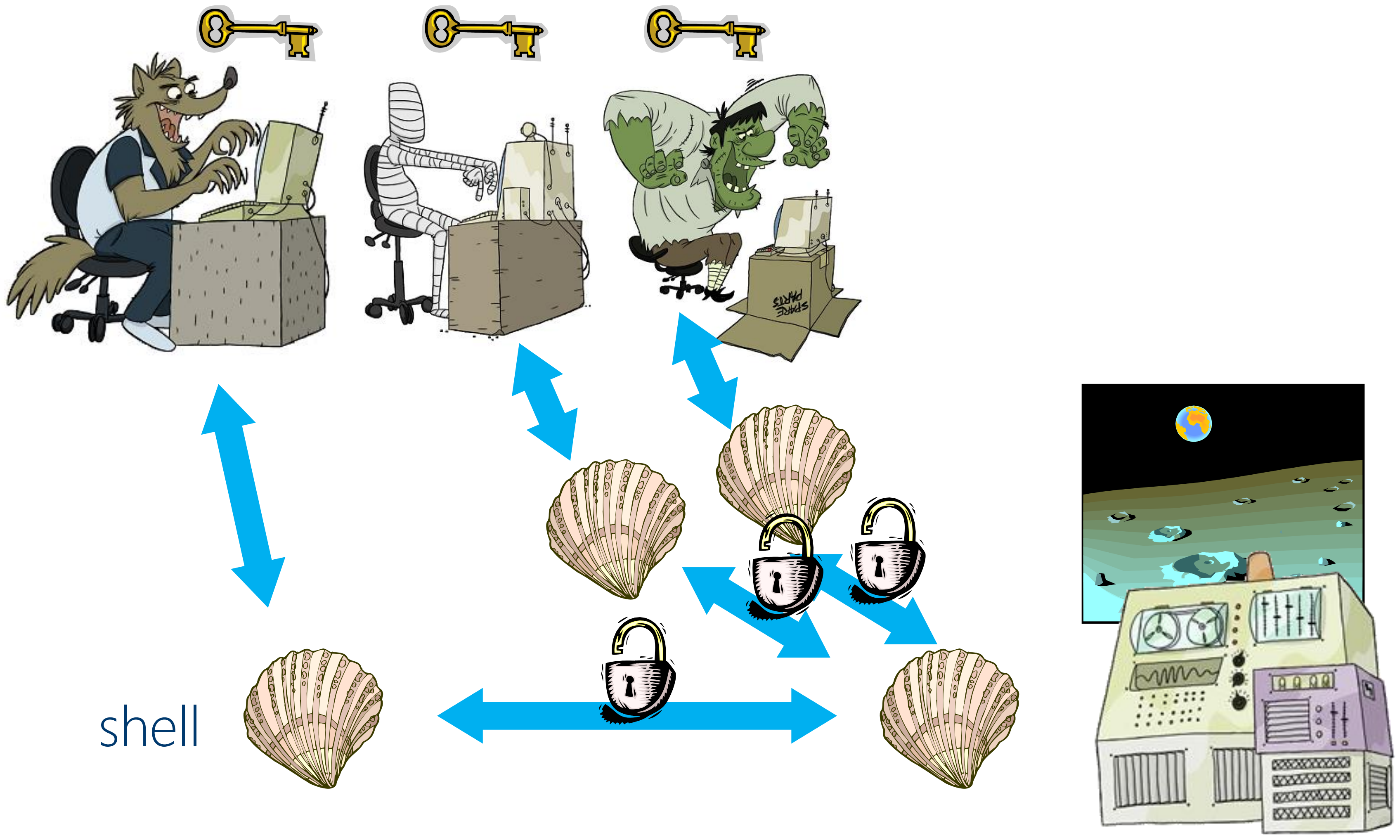


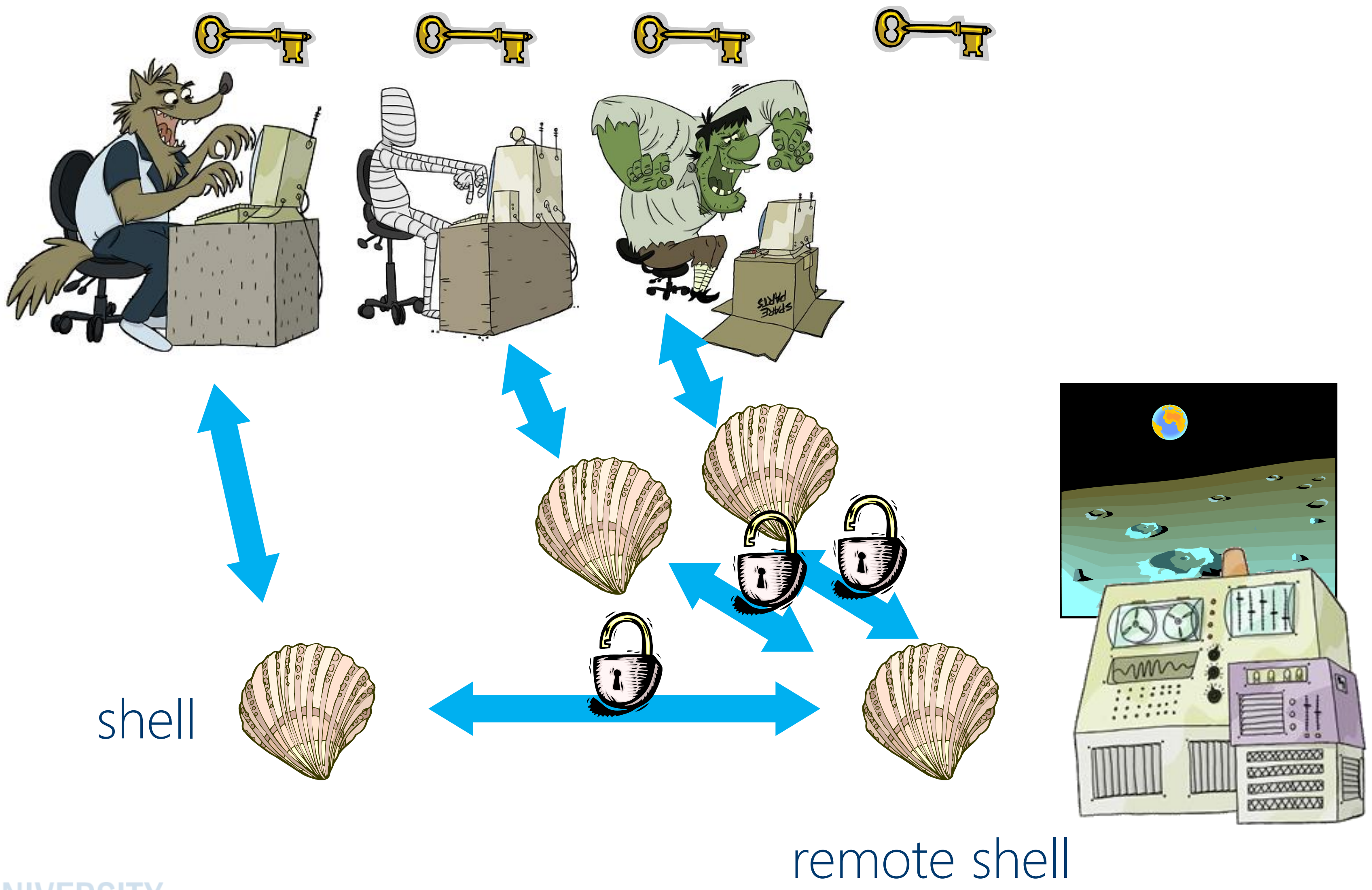


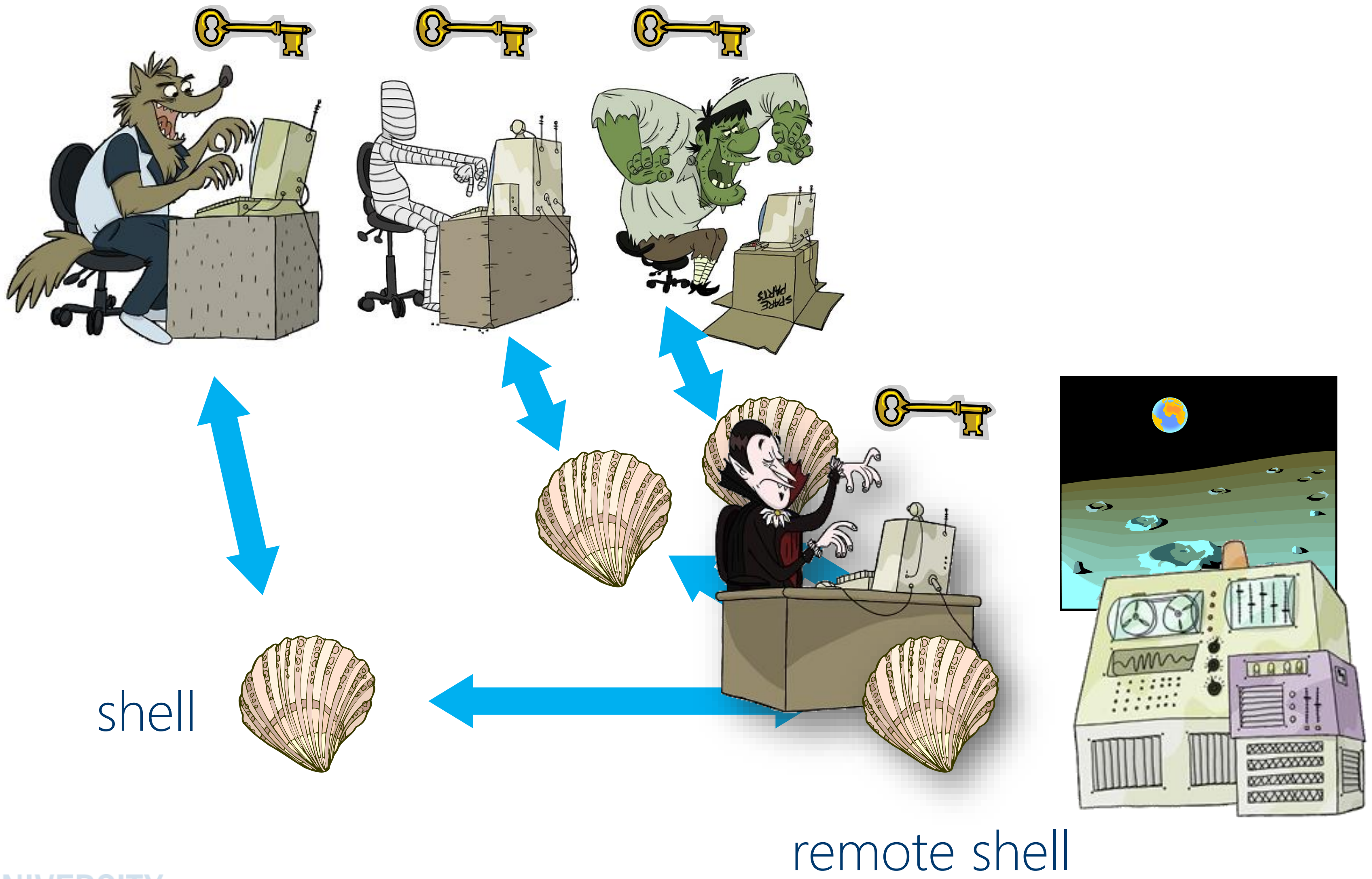
```
login as: vlad
Password: thriller
```

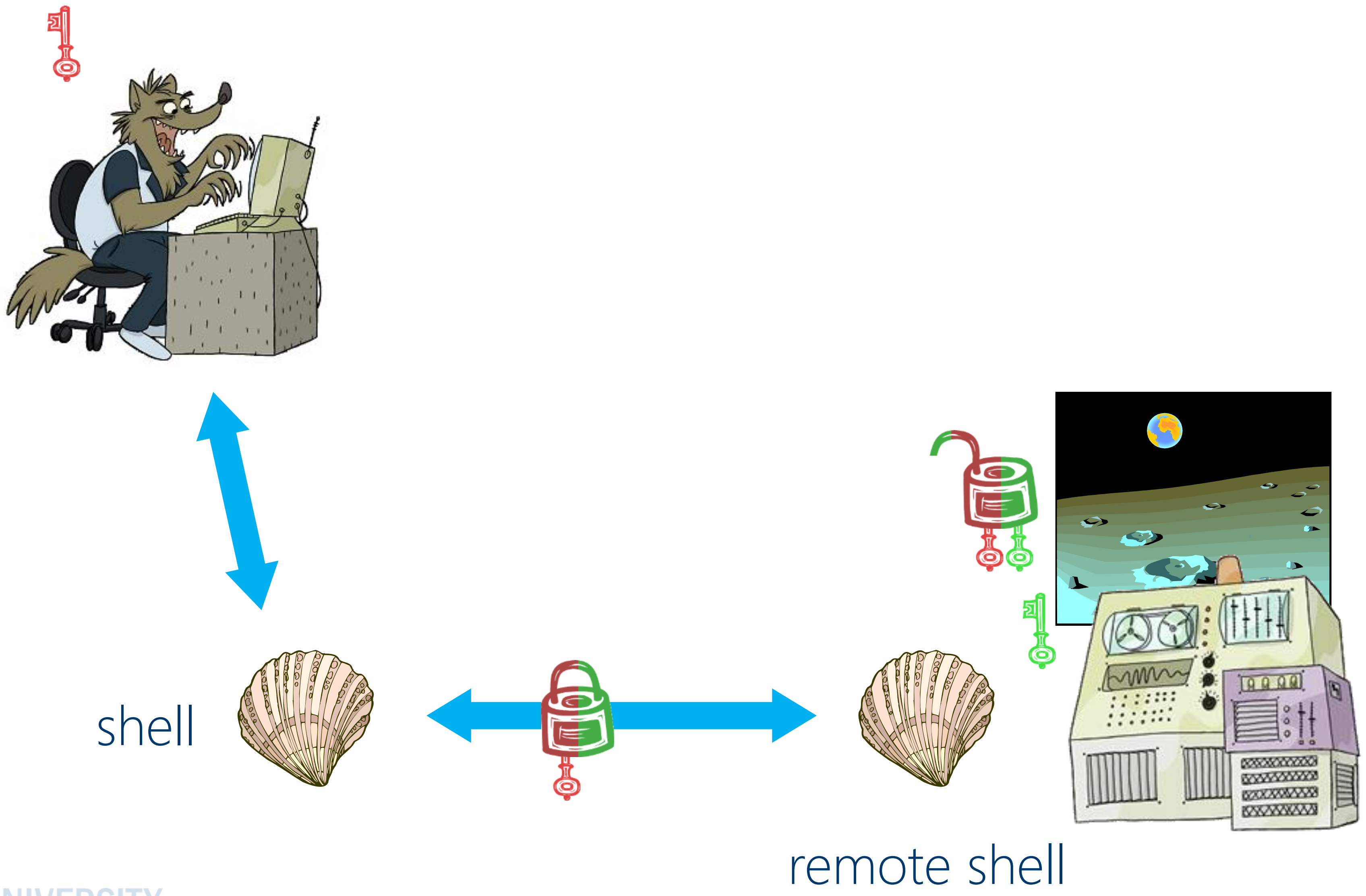


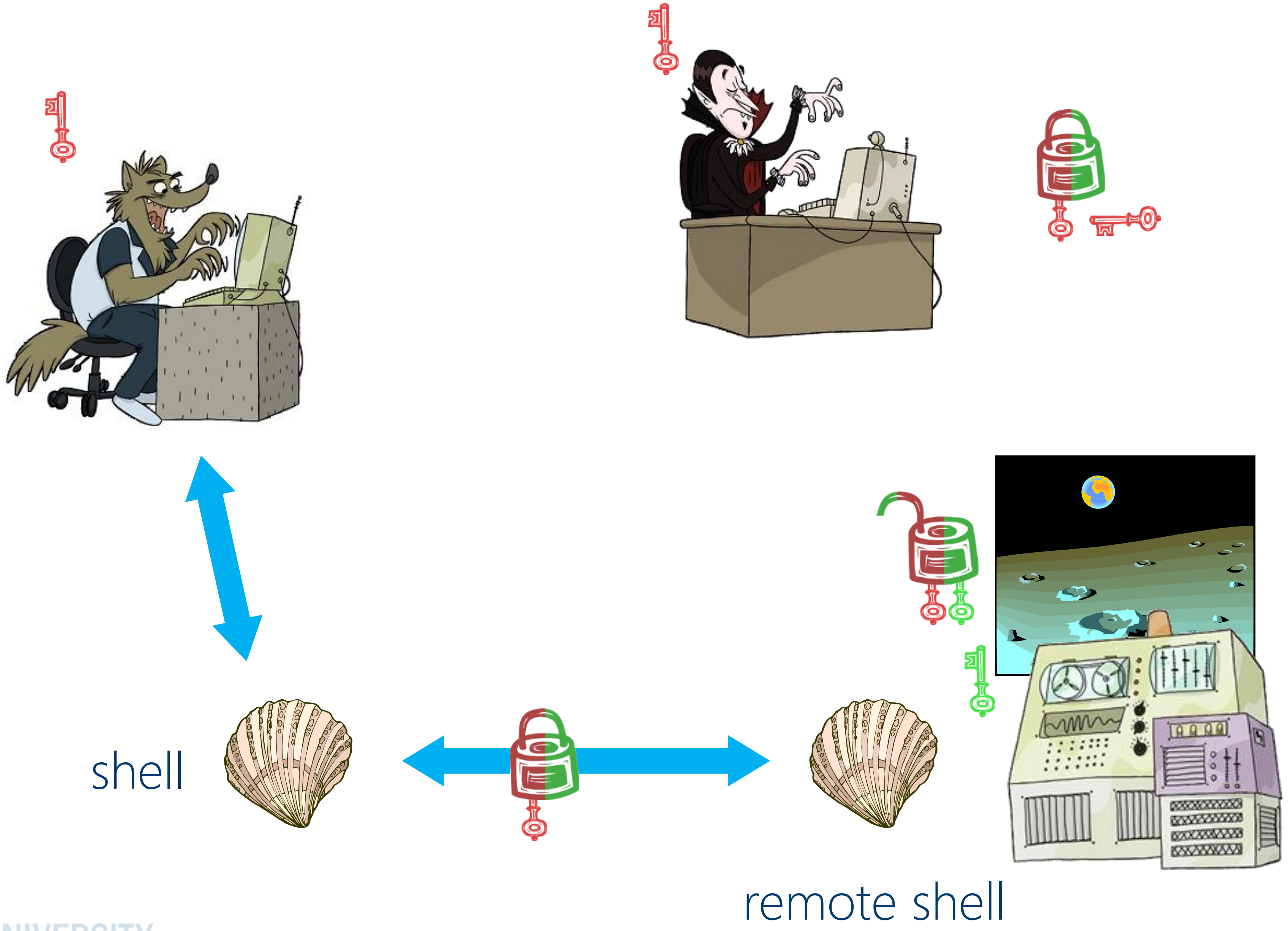
```
login as: vlad
Password: thriller
```

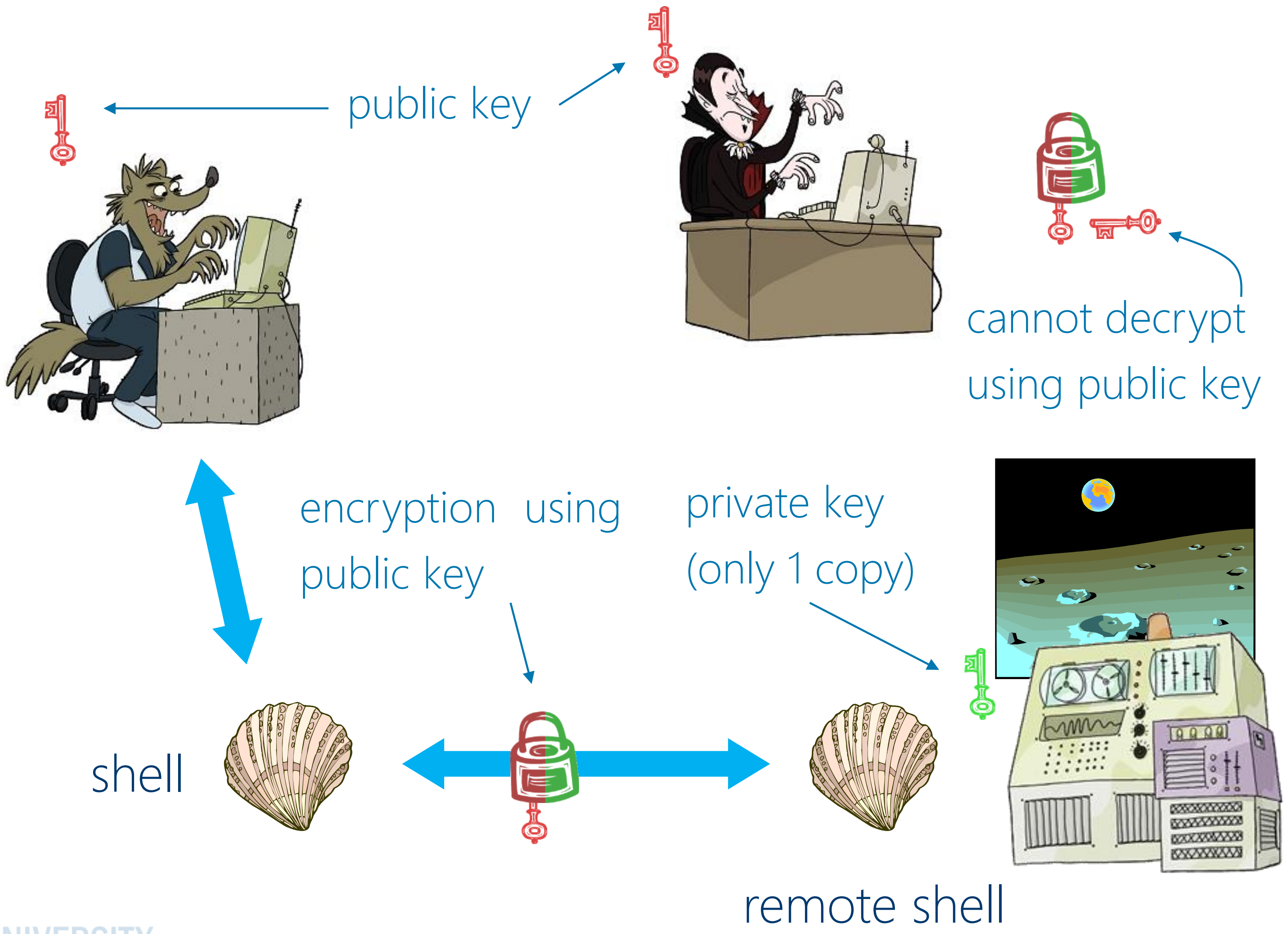

shell
remote shell

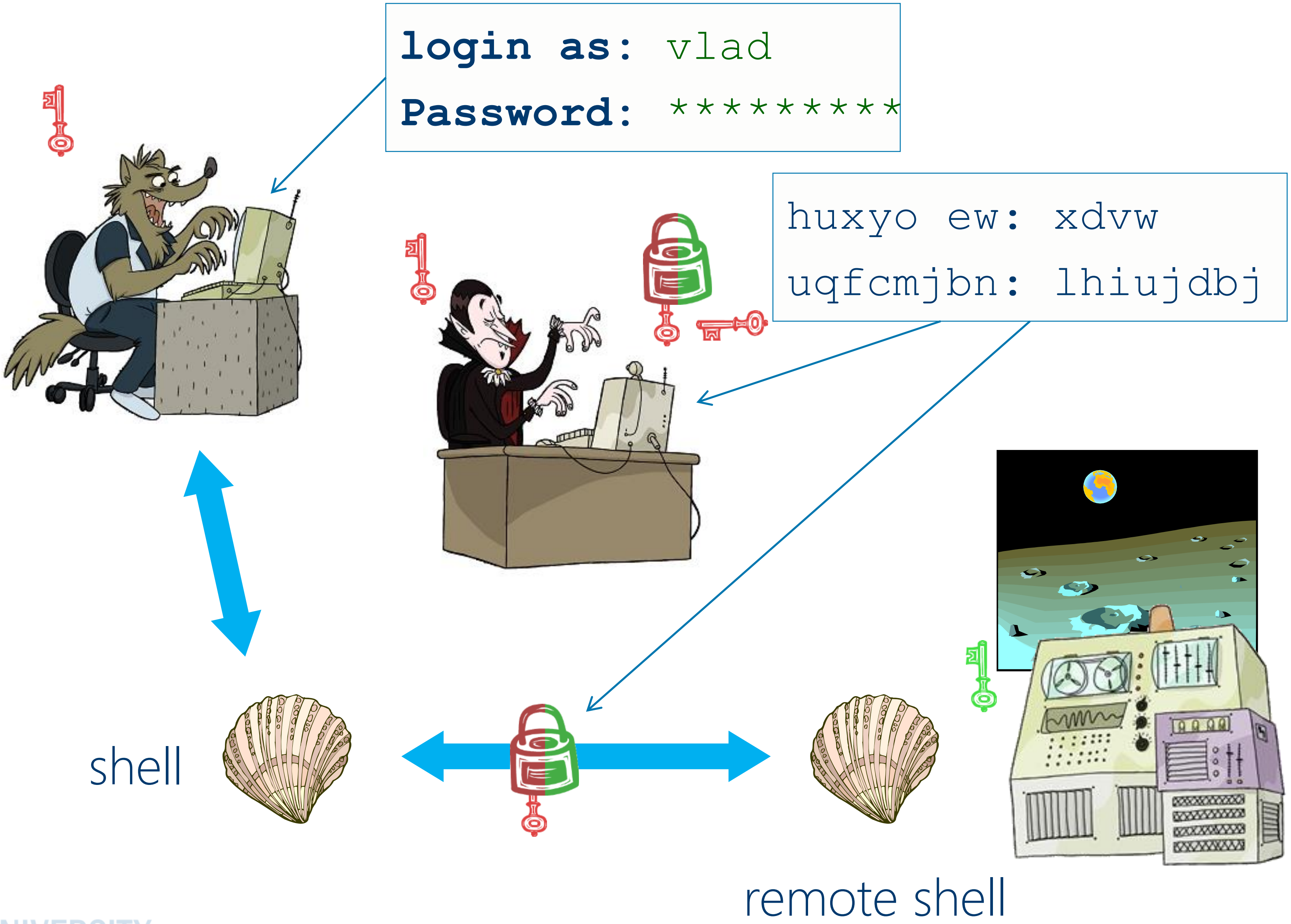

shell
remote shell
SPARE PARTS


SPARE PARTS
shell
remote shell


shell
remote shell


shell
remote shell


public key
cannot decrypt
using public key
encryption using
public key
private key
(only 1 copy)
shell
remote shell



```
login as: vlad
Password: *********
```

```
huxyo ew: xdvw
uqfcmjbn: lhiujdbj
```

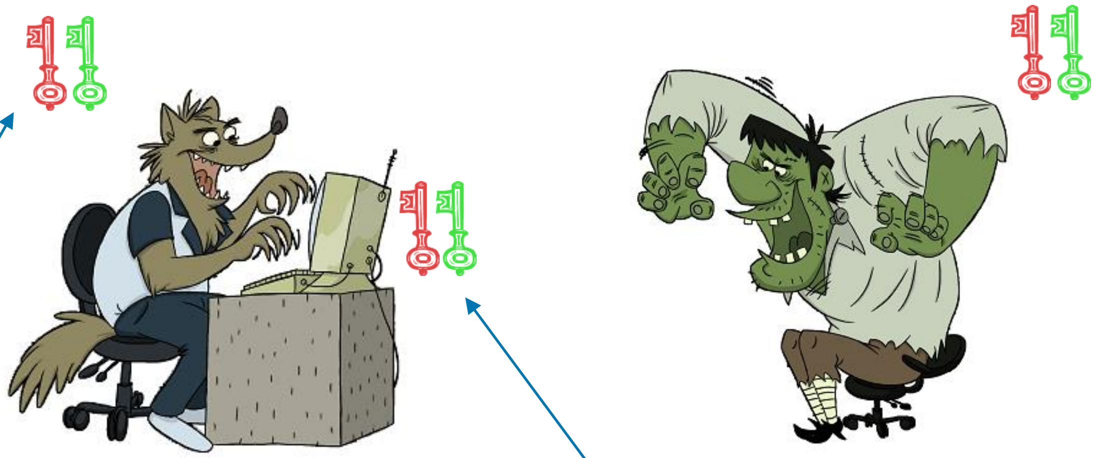
```
$ ssh vlad@moon
The authenticity of host ‘moon (10.1.2.3)‘
 can't be established.
RSA key fingerprint is
 f1:68:f5:90:47:dc:a8:e9:62:df:c9:21:f0:8b:c5:39.
Are you sure you want to continue connecting
 (yes/no)? yes
Warning: Permanently added ‘moon,10.1.2.3' (RSA)
 to the list of known hosts.
Password: ********
moon>
```

```
while true:
    ...
    if time.mins == 30:
        ssh vlad@moon 'df -h' >> usage.log
    ...
```

```
while true:
  ...
  if time.mins == 30:
    ssh vlad@moon 'df -h' >> usage.log
  ...
```

```
$ ssh vlad@moon 'df -h' >> usage.log
Password:
Connection closed by 10.1.2.3
$
```

$\longleftarrow$ waited too long



user key pair 2

user key pair 1

host key pair

only 1 copy

default if user has no user key pair

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key
 (/users/vlad/.ssh/id_rsa):
Enter passphrase (empty for no
 passphrase): ********
Enter same passphrase again: ********
Your identification has been saved in
 /users/vlad/.ssh/id_rsa.
Your public key has been saved in
 /users/vlad/.ssh/id_rsa.pub.
The key fingerprint is:
d3:1a:27:38:aa:54:e8:a5:03:db:79:2f:b2:c3:c9:3d
```

$\longleftarrow$ press enter

```
$ ssh vlad@moon
Enter passphrase for key
 '/users/vlad/.ssh/id_rsa': ********
moon>
```

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key
 (/users/vlad/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
 /users/vlad/.ssh/id_rsa.
Your public key has been saved in
 /users/vlad/.ssh/id_rsa.pub.
The key fingerprint is:
d3:1a:27:38:aa:54:e8:a5:03:db:79:2f:b2:c3:c9:3d
```

press enter

```
$ scp ~/.ssh/id_rsa.pub vlad@moon
Password: ********
$ ssh vlad@moon
Password: ********
moon> cat id_rsa.pub >> ~/.ssh/authorized_keys
moon> exit

$ cat ~/.ssh/id_rsa.pub | ssh vlad@moon
 'cat >> ~/.ssh/authorized_keys'
Password: ********

$ ssh-copy-id vlad@moon
Password: ********
```

```
$ ssh vlad@moon
moon>
```

```
while true:
  ...
  if time.mins == 30:
    ssh vlad@moon 'df -h' >> usage.log
  ...
```

Checkpoint 6

- Please answer the *Self-Assessment* :
  - Blackboard > Course Content > Week 3 (Sept. 28- Oct. 2) > Monday Sept. 28 > Checkpoint 6

QOTD

- All models are wrong, but some are useful

Geroge Box
(1919–2013)
British statistician