COMP 2718: The File System: Part 3

By: Dr. Andrew Vardy

Adapted from the notes of Dr. Rod Byrne

Outline

File System Navigation — Chapter 4 of TLCL

We'll now cover material from chapter 4 of the textbook. The following commands will be introduced:

- `mkdir`: Create directories
- `cp`: Copy files and directories
- `mv`: Move/rename files and directories
- `rm`: Remove files and directories
- `ln`: Create hard and symbolic links
- `touch`: Change file access time / creates file

The functionality of these commands are also well-captured in graphical file managers that are suitable for easy tasks. But the command line excels for more complex tasks.

Globbing (a.k.a. Wildcards)

Globbing is the use of special characters to select filenames based on patterns.

Wildcard	Meaning
`*`	Matches any characters (including 0 characters)
`?`	Matches any single character
`[`*`characters`*`]`	Matches any character that is a member of the set *characters*
`[!`*`characters`*`]`	Matches any character that is not a member of the set *characters*
`[[:`*`class`*`:]]`	Matches any character that is a member of the specified *class*

Character classes

Character Class	Meaning
`[:alnum:]`	Matches any alphanumeric character
`[:alpha:]`	Matches any alphabetic character
`[:digit:]`	Matches any numeral
`[:lower:]`	Matches any lowercase letter
`[:upper:]`	Matches any uppercase letter

Examples

Assume the following current directory contents:

```
ab       abc      de.txt       h1.class    h3.class
```

```
$ ls a*
```

```
ab  abc
```

```
$ ls *[ct]    # ends in c or t
```

```
abc de.txt
```

```
$ ls *c*
```

```
abc     h1.class    h3.class
```

Note that * is interpreted as any number of characters, including *zero*.

```
$ ls ??
```

```
ab
```

```
$ ls *.class

h1.class   h3.class

$ rm d*

$ ls

ab      abc     h1.class   h3.class

$ rm *s

$ ls

ab  abc
```

Examples with character classes

Assume the following current directory contents:

```
012 10.txt  ABC xyz

$ ls [[:upper:]]*   # Any file beginning with upper-case

ABC

$ ls [![:digit:]]*  # Any file NOT beginning with a digit

ABC     xyz

$ ls *[[:digit:]t]  # Any file ending in a digit or 't'

012     10.txt
```

Hard and symbolic Links

See slides entitled “Hard & Symbolic Links”

File Manipulation Commands

We review below the major file manipulation commands and show common options:

`mkdir` - Create Directories

```
mkdir dir1...
```

Where `...` indicates that the argument could be repeated, for example:

```
mkdir dir1 dir2 dir3
```

The only common option I know about is `-p` which creates the necessary parent directories. For example:

```
mkdir -p /tmp/A/B/C
```

Assuming `/tmp` exists but not `A`, `B`, or `C`, this creates the directories `A`, `B`, and `C`.

cp - Copy Files and Directories

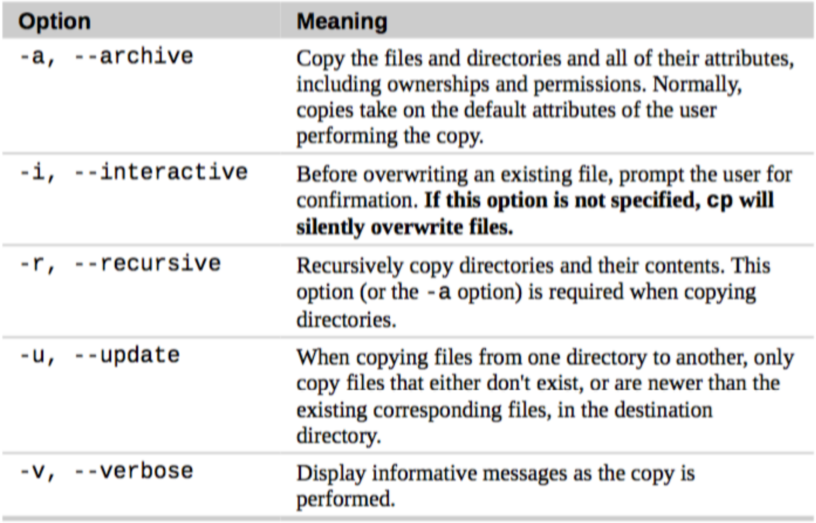
```
cp item1 item2
```

Copies the file/directory `item1` to file/directory `item2`.

```
cp item... directory
```

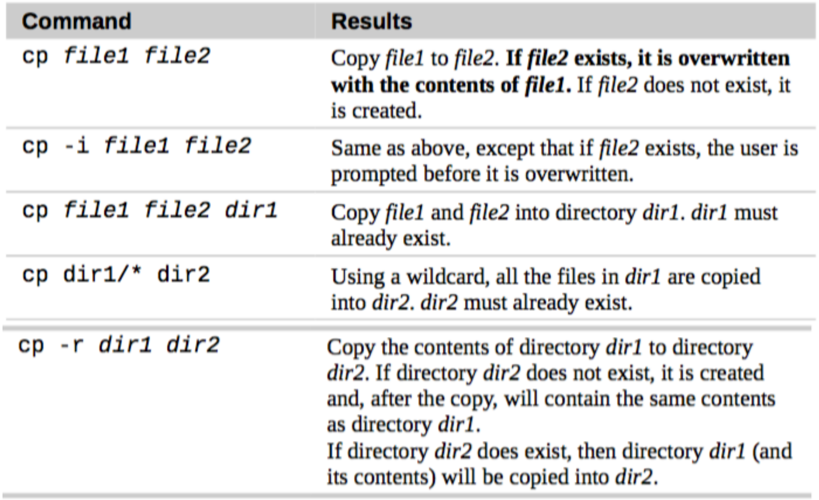
Copies multiple items (files or directories) into the directory.

Common options for cp



Option	Meaning
`-a, --archive`	Copy the files and directories and all of their attributes, including ownerships and permissions. Normally, copies take on the default attributes of the user performing the copy.
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, cp will silently overwrite files.**
`-r, --recursive`	Recursively copy directories and their contents. This option (or the `-a` option) is required when copying directories.
`-u, --update`	When copying files from one directory to another, only copy files that either don't exist, or are newer than the existing corresponding files, in the destination directory.
`-v, --verbose`	Display informative messages as the copy is performed.

Examples of using cp



Command	Results
`cp file1 file2`	Copy *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created.
`cp -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`cp file1 file2 dir1`	Copy *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`cp dir1/* dir2`	Using a wildcard, all the files in *dir1* are copied into *dir2*. *dir2* must already exist.
`cp -r dir1 dir2`	Copy the contents of directory *dir1* to directory *dir2*. If directory *dir2* does not exist, it is created and, after the copy, will contain the same contents as directory *dir1*. If directory *dir2* does exist, then directory *dir1* (and its contents) will be copied into *dir2*.

`mv` - Move and Rename Files

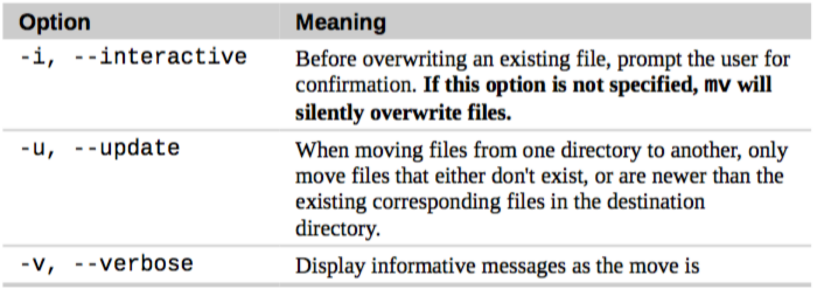
```
mv item1 item2
```

Moves file/directory `item1` to `item2`.

```
mv item... directory
```

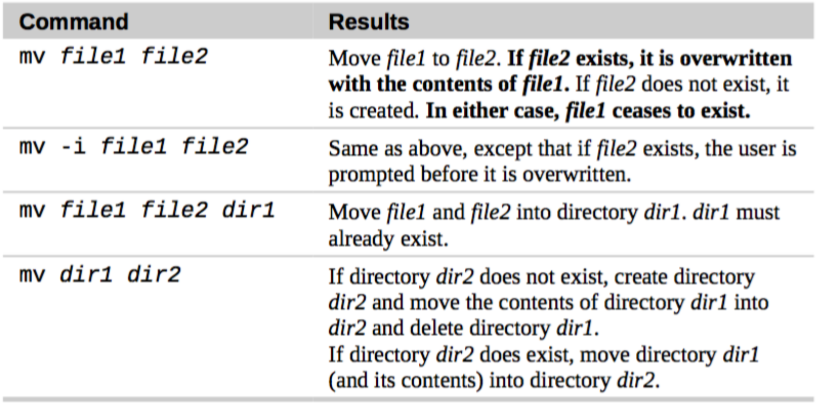
Moves multiple items to the given directory.

Common options for `mv`



Option	Meaning
`-i, --interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, `mv` will silently overwrite files.**
`-u, --update`	When moving files from one directory to another, only move files that either don't exist, or are newer than the existing corresponding files in the destination directory.
`-v, --verbose`	Display informative messages as the move is

Examples of using `mv`



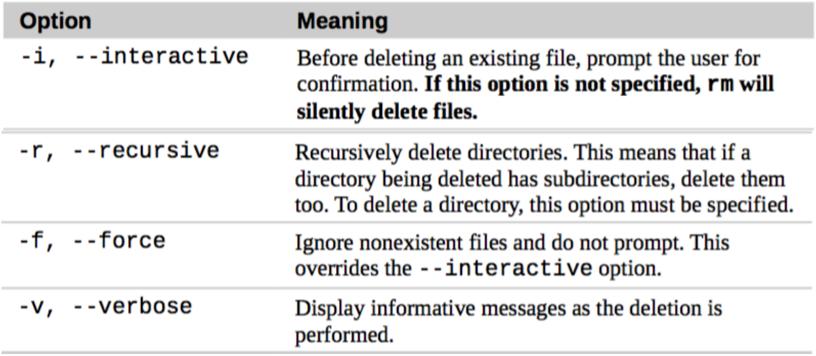
Command	**Results**
`mv file1 file2`	Move *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created. **In either case, *file1* ceases to exist.**
`mv -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`mv file1 file2 dir1`	Move *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`mv dir1 dir2`	If directory *dir2* does not exist, create directory *dir2* and move the contents of directory *dir1* into *dir2* and delete directory *dir1*. If directory *dir2* does exist, move directory *dir1* (and its contents) into directory *dir2*.

`rm` - Remove Files and Directories

```
rm item...
```

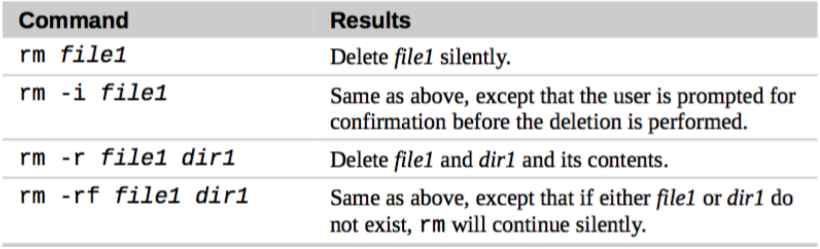
Removes `item` (or items) whether they are files or directories.

Common options for `rm`



Option	Meaning
`-i, --interactive`	Before deleting an existing file, prompt the user for confirmation. **If this option is not specified, `rm` will silently delete files.**
`-r, --recursive`	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
`-f, --force`	Ignore nonexistent files and do not prompt. This overrides the `--interactive` option.
`-v, --verbose`	Display informative messages as the deletion is performed.

Examples of using `rm`



Command	Results
`rm` *`file1`*	Delete *file1* silently.
`rm -i` *`file1`*	Same as above, except that the user is prompted for confirmation before the deletion is performed.
`rm -r` *`file1 dir1`*	Delete *file1* and *dir1* and its contents.
`rm -rf` *`file1 dir1`*	Same as above, except that if either *file1* or *dir1* do not exist, `rm` will continue silently.

BE CAREFUL WITH `rm`!

Unless you implement it yourself, there is no undelete command on the command-line. Be especially careful when using `rm` and globbing. The following is intended to delete all of the html files in the current directory:

```
rm *.html
```

But what if you type the following by accident:

```
rm * .html
```

Firstly, `rm` will **delete all of the files in the current directory** (Ahh!). Then it will complain there is no file called `.html`.
But the damage can be much worse. . .

. . . if you incorporate `-r` for recursive deletion. Lets say you have directories A.1, A.2, A.3 that you want to completely delete. You could type the following:

```
rm -r A.*
```

But if you typed the following you would wipe out everything in your current directory (very bad if cur. dir. is your home—even worse if its /).

```
rm -r A. *
```

A good solution is to first use `ls` in place of `rm` to give a listing of the files that will be deleted:

```
ls -r A. *
```

[Huge listing of files appears and the mistake is realized.]

```
rm -r A.*
```

touch - Change File Access Time / Creates File

Sets both the modification and access times of files. By default it will set both to the current time.

```
$ ls -l test1.txt

-rw-rw----  1 av  staff  15 21 Jan 11:58 test1.txt

[2 minutes later]

$ touch test1.txt
$ ls -l test1.txt

-rw-rw----  1 av  staff  15 21 Jan 12:00 test1.txt
```

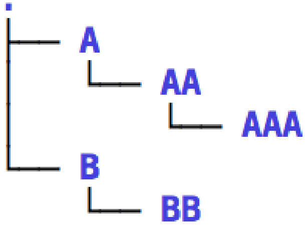
`touch` will also create an empty file if the given arguments are non-existent files. We will use this in some examples below. . .

General File Manipulation Examples

Lets use `mkdir` and `touch` to create a set of directories and files:

```
$ mkdir -p A/AA/AAA B/BB
$ ls -R                # Lists all contents recursively

A   B

./A:
AA

./A/AA:
AAA

./A/AA/AAA:

./B:
BB

./B/BB:
```

Alternatively, the program `tree -C` can be used to display the same information in a tree-like format:



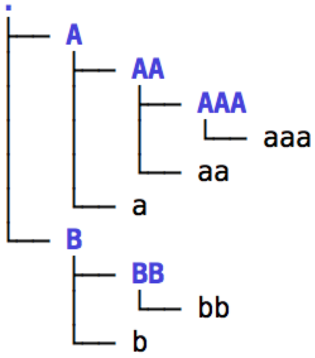
```
.
├── A
│   └── AA
│       └── AAA
└── B
    └── BB
```

Remember this was generated with:

```
$ mkdir -p A/AA/AAA B/BB
```

Lets add some files with `touch`:

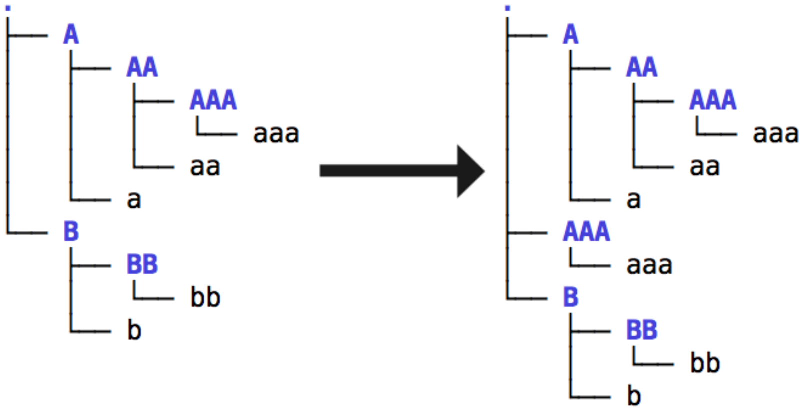
```
$ touch A/a A/AA/aa A/AA/AAA/aaa B/b B/BB/bb
```



```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   └── bb
    └── b
```

Copy recursively from `A/AA/AAA` to the current directory. This shows why the symbol `.` is needed!

```
$ cp -r A/AA/AAA .
```

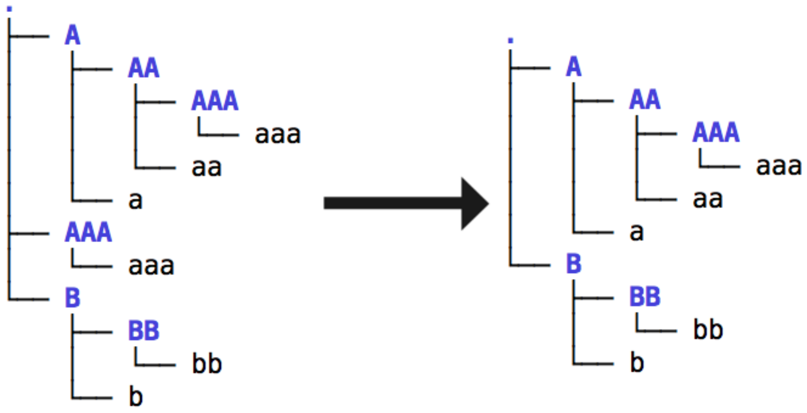


```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   └── bb
    └── b
```

```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
├── AAA
│   └── aaa
└── B
    ├── BB
    │   └── bb
    └── b
```

Lets undo what we just did.

```
$ rm -r AAA
```

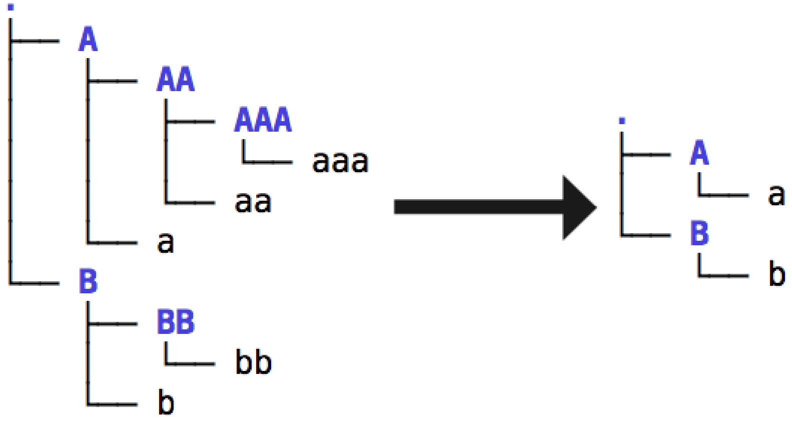


```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
├── AAA
│   └── aaa
└── B
    ├── BB
    │   └── bb
    └── b
```

```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   └── bb
    └── b
```

Remove all two-character dir's and files contained in any subdir of the current dir:

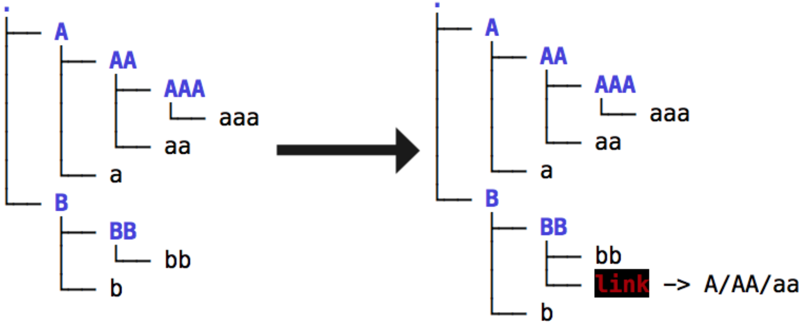
```
$ rm -r */??
```



Bad example of creating a symbolic link

Recreate original structure again, then create a symbolic link from to aa in BB:

```
$ rm -r *
$ mkdir -p A/AA/AAA B/BB
$ touch A/a A/AA/aa A/AA/AAA/aaa B/b B/BB/bb

$ ln -s A/AA/aa B/BB/link
```

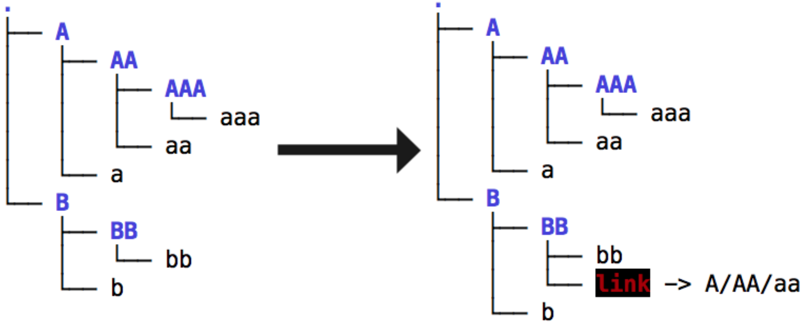


```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   └── bb
    └── b
```

```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   ├── bb
    │   └── link -> A/AA/aa
    └── b
```

[Repeated from last slide]

```
$ ln -s A/AA/aa B/BB/link
```



```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   └── bb
    └── b
```

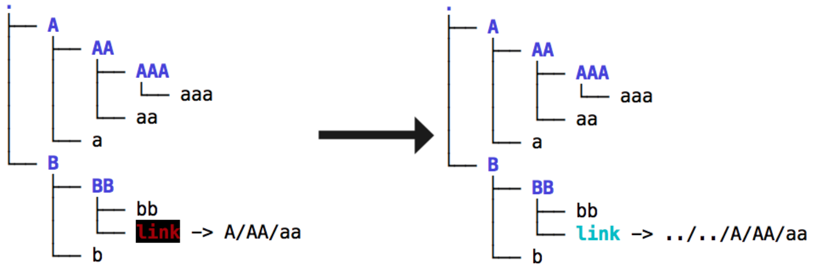
```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   ├── bb
    │   └── link -> A/AA/aa
    └── b
```

Note that the relative pathname `A/AA/aa` is actually stored in the link. This link is **broken** from the start! Why? Because it assumes that `B/BB` should contain `A/AA/aa` which is **not the case**.

Good example of creating a symbolic link

Remove previous link, then change to the `B/BB` directory and create the link there with an appropriate relative path:

```
$ rm B/BB/link
$ cd B/BB
$ ln -s ../../A/AA/aa link
$ cd ../..
```



```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   ├── bb
    │   └── link -> A/AA/aa
    └── b
```

```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   ├── bb
    │   └── link -> ../../A/AA/aa
    └── b
```

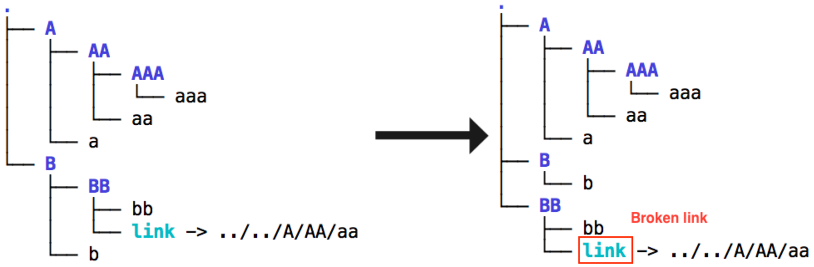
The link is now valid (see change in colour).

Lets make sure that aa actually contains something:

```
$ echo "STUFF" > A/AA/aa  # Redirection (covered soon)

$ cat A/AA/aa             # Displays file (covered soon)

STUFF

$ cat B/BB/link

STUFF
```

If we alter the directory structure it can break the link.

```
$ mv B/BB .
```



```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
└── B
    ├── BB
    │   ├── bb
    │   └── link -> ../../A/AA/aa
    └── b
```

```
.
├── A
│   ├── AA
│   │   ├── AAA
│   │   │   └── aaa
│   │   └── aa
│   └── a
├── B
│   └── b
└── BB
    ├── bb
    └── link -> ../../A/AA/aa
```

```
$ cat BB/link
[nothing prints]
```