# *csce215 — UNIX/Linux Fundamentals*
# *Spring 2022 — Lecture Notes: How to be lazy*

*This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.*

## (4.1) *Last time*

Last time we learned about **standard input**, **standard output**, and **standard error** and how to **redirect** and **pipe** them.

- output redirection with > and >>
- input redirection with <
- error redirection with 2> and 2>>
- pipes with |

**Today**, we will learn some ways to use the shell more efficiently, i.e. ways to **be lazy**. The key ideas will be:

- **Command line arguments** and **special characters** that influence them;
- **Lists** that connect multiple commands to be run independently; and
- **Command substitutions** that use the output of one command as arguments for another.

## (4.2) *Command line arguments*

Remember that when you type commands, you are communicating with a special program called a **shell**, whose job is to read, interpret, and execute the commands you give, usually by running other programs.

The shell passes **command line arguments** to the programs it runs.

```
$ ls -l shakespeare.txt
-rw-rw-r-- 1 jokane jokane 75223 Jun  9 09:59 shakespeare.txt
```

Command: `ls`

Arguments:

- `-l`
- `shakespeare.txt`

## (4.3) *Using the arguments*

The arguments are available to the program being run, in a form that depends on the programming language.

For example, in Java, we can do something like this:

```
class ShowArgs {
  public static int main(String[] args) {
    String s;
    for (int i = 0; i < args.length; i++) {
      s = String.format("Arg %d is [%s]", i, args[i]);
      System.out.println(s);
    }
    return 0;
  }
}
```

Some of the examples below use a similar program, written as a shell script, called `showargs`.

## (4.4) *Special characters*

Usually, the shell gets arguments by splitting the command up at spaces.

**However** most forms of punctuation are treated as **special characters** that can change this behavior.

We've seen a few special characters already:

> < | ~

There are a few others that are important to know.

## (4.5) *Wildcards*

Arguments that contain * or ? are replaced with a list of filenames that match.

- A ? matches any single character.
- A * matches any sequence of characters.

The characters * and ? are called **wildcards**.

Examples in a directory filled with C++ (cpp), header (h), and object (o) files:

```
$ ls task.*
task.cpp
task.h
task.o
```

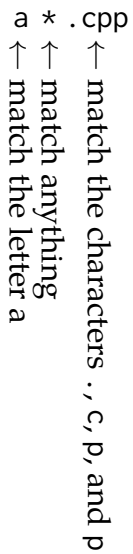
```
$ ls task.?
task.h
task.o
```

```
$ ls *.cpp | head
angle.cpp
animate-callback.cpp
animate.cpp
animate-gl.cpp
apollonius.cpp
circle.cpp
config.cpp
diffdrive.cpp
drawercolor.cpp
drawer.cpp
```

## (4.6) *Wildcard example*

Suppose we wanted a list of all files in the current directory that start with 'a' and have a cpp extension.

We can use this wildcard pattern: `a*.cpp`




```
$ ls a*.cpp
angle.cpp
animate-callback.cpp
animate.cpp
animate-gl.cpp
apollonius.cpp
```

## (4.7) *Braces*

Arguments containing **curly braces** are replaced with a list of each of the things between braces. 📖

```
$ showargs X{a,b,c}Y
Arg 1 is [XaY]
Arg 2 is [XbY]
Arg 3 is [XcY]
```

```
$ showargs X{a,b{1,2},c}Y
Arg 1 is [XaY]
Arg 2 is [Xb1Y]
Arg 3 is [Xb2Y]
Arg 4 is [XcY]
```

```
$ ls util.{cpp,h}
util.cpp
util.h
```

```
$ ls *.{cpp,h} | tail
triangle.cpp
triangle.h
unionfind.cpp
unionfind.h
uniseq.cpp
uniseq.h
util.cpp
util.h
wanderingrobots.cpp
wanderingrobots.h
```

## (4.8) *Tilde for home directory*

The tilde character (~) is replaced with the path to the user's home directory.

```
$ showargs ~
Arg 1 is [/home/jokane]
```

## (4.9) *Quotation marks*

Double quotation marks (") prevent the shell from:

- splitting into separate arguments
- expanding wildcards

```
$ showargs a b c d e f
Arg 1 is [a]
Arg 2 is [b]
Arg 3 is [c]
Arg 4 is [d]
Arg 5 is [e]
Arg 6 is [f]
```

```
$ showargs a b c "d e f"
Arg 1 is [a]
Arg 2 is [b]
Arg 3 is [c]
Arg 4 is [d e f]
```

```
$ showargs "*.cpp"
Arg 1 is [*.cpp]
```

```
$ showargs *.cpp | head -n 5
Arg 1 is [angle.cpp]
Arg 2 is [animate-callback.cpp]
Arg 3 is [animate.cpp]
Arg 4 is [animate-gl.cpp]
Arg 5 is [apollonius.cpp]
```

## (4.10) *Backslash*

Use a **backslash** (\) to force the next character to be treated normally. This is called 'escaping' the character.

```
$ showargs a b c d e f
Arg 1 is [a]
Arg 2 is [b]
Arg 3 is [c]
Arg 4 is [d]
Arg 5 is [e]
Arg 6 is [f]
```

```
$ showargs a\ b\ c\ d\ e\ f
Arg 1 is [a b c d e f]
```

```
$ showargs \*\?\ \\
Arg 1 is [*? \]
```

With escaping:

```
$ echo hello world > hello\ world.txt
$ ls
hello world.txt
$ cat "hello world.txt"
hello world
```

Without escaping:

```
$ echo hello world > hello world.txt
$ ls
hello
$ cat hello
hello world world.txt
```

## (4.11) *Connecting commands*

Use ; to separate two commands, when each one should be executed. 📖

```
$ date; whoami
Fri Jun  9 09:59:05 AM CDT 2023
jokane
```

Use && to separate two commands, when the second should be ignored if the first fails. 📖

```
$ ls main.cpp && echo Success
main.cpp
Success
$ ls fake.cpp && echo Success
ls: cannot access 'fake.cpp': No such file or directory
```

Use || to separate two commands, when the second should be executed if the first fails.

```
$ ls main.cpp || echo Fail
main.cpp
$ ls fake.cpp || echo Fail
ls: cannot access 'fake.cpp': No such file or directory
Fail
```

## (4.12) *Compile and run*

A useful pattern is:

*compile command* && *run command*

```
$ javac Hello.java && java Hello
hello, world
```

```
$ javac "Hello (Broken).java" && java Hello
Hello (Broken).java:2: error: ']' expected
    public static String mian(String[ args) {
                                      ^
Hello (Broken).java:3: error: ';' expected
        System.out.println("hello, world")
                                           ^
Hello (Broken).java:4: error: reached end of file while parsing
}
 ^
3 errors
```

## (4.13) *Command substitution*

Use $( ) to insert the *output* of one command into the *arguments* for another command.

(This can also be written with backticks (`). These are the backwards quotes that, on most keyboards, come from the shift-tilde key).

Example: Making a copy of a program.

```
$ which zoom
/usr/bin/zoom
$ cp -v /usr/bin/zoom .
'/usr/bin/zoom' -> './zoom'
$ cp -v $(which zoom) .
'/usr/bin/zoom' -> './zoom'
```

Example: Looking for a file by name.

```
$ locate pyx | grep version.py
/usr/local/lib/python3.10/dist-packages/pandas/_libs/tslibs/conversion.pyx
/usr/local/lib/python3.10/dist-packages/pandas/_libs/tslibs/tzconversion.pyx
$ cat $(locate pyx | grep version.py) | tail -n 2

    return fold
```

## (4.14) *Other special characters*

There are a few other characters that bash treats in special ways:

1. parentheses `( )`
2. square brackets `[ ]`
3. comment `#`
4. variable `$`
5. run in background `&`
6. negation `!`
7. . . .

We'll cover some of these later, but the main thing to remember is that *nearly all punctuation has a special meaning*, so you'll sometimes need to be careful. When in doubt, escape it.

## (4.15) *Sample final exam questions*

1. Which of these commands will use command substitution to copy the executable for the program `grep` to the current directory?

   A. `cp ?(grep) .`

   B. `cp (which grep) .`

   C. `cp $(grep) .`

   D. `cp $(which grep) .`

2. In a directory containing many kinds of files, which of these commands will display at most 10 files with the extension `cpp`?

   A. `ls ?.cpp | head`

   B. `ls .cpp | head`

   C. `ls *.cpp | head`

   D. `ls *.cpp | tac`

3. Which of these files would be listed by the command `ls a b c d e f ?`

   A. Six files named `a`, `b`, `c`, `d`, `e`, and `f`.

   B. A single file named `'a b c d e f g'`, with 5 spaces in its name.

   C. Two files, one named `'a b c'` and one named `'d e f'`, each with two spaces in its name.

   D. None of the above.

4. Which symbol is used to force the next character to be treated normally?

   A. `` ` ``

   B. `?`

   C. `\`

   D. `/`

5. For the command `ls hi.?` which files would be listed?

   A. `hi.java`

   B. `hi.py`

   C. `hi.`

   D. `hi.c`

6. Which of these commands will list all of the files whose name contains an opening parenthesis?

   A. `ls *(*`

   B. `ls \*\(\*`

   C. `ls *\(*`

   D. `ls \(`

7. Which of these files would be listed by the command `ls hello.* ?`

   A. `hello.py`

   B. `hello.csharp`

   C. `hello.cpp`

   D. All of the above.

8. What would the effect of the command `ls *.c` be?

   A. To list all files whose names consist of exactly one character, followed by a period, followed by a `c`.

   B. To list all files with a `c` extension.

   C. To list all files whose names do not contain a `c`.

   D. To list all files whose names contain a `c`.

9. Which of these files would be listed by the command `ls a\ b\ c\ d\ e\ f`, which has a space after each of its backslashes?

A. Six files named `a`, `b`, `c`, `d`, `e`, and `f`.

B. A single file named `'a b c d e f'`, with 5 spaces in its name.

C. Two files, one named `'a b c'` and one named `'d e f'`, each with two spaces in its name.

D. None of the above.

10. What would the effect of the command `ls r*` be?

A. To list all files, including those whose names start with `r`.

B. To list all files whose name are exactly zero or more `r` characters.

C. To list all files whose names start with `r`.

D. To list all files whose names start with `r` and have exactly two characters.

11. Which command below will list all of the files with extensions that contain exactly three characters?

A. `ls *.???`

B. `ls .???`

C. `ls *....`

D. `ls *.*`