Bash Basics 1: Intro & Commands

These slides introduce the basics of working with **Bash**:

1. **Intro & Commands**
   - **shells and CLI**
   - **GNU project**
   - **command interpreter**
   - **types of commands in Bash**
   - **commands syntax**
2. Expansions & Metachars
3. Variables & Quoting
4. Interactive Use
5. Redirections & Locale

Bash

A **shell** is a piece of software that provides a **command line interface** (**CLI**).

A CLI allows users to type commands to be run by the OS.

The default Linux **shell** is **Bash** (**Bourne Again SHell**).

It is a GNU extension of the **Bourne shell** (`sh`), one of the first UNIX shells.

Bash is an excellent shell both for **interactive** use and for **shell scripting** (programming).

The Bash executable is typically: `/bin/bash`

Linux systems do not have a true Bourne shell, but rather use Bash in "legacy mode" (when invoked as `sh`).

Source Code and Further Information

Bash is open source software, provided by the GNU project:
`http://www.gnu.org/software/bash`

The *Bash Reference Manual* can be found at:
`http://www.gnu.org/software/bash/manual`

The *Bash Prompt HOWTO* can be found at:
`http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO`

Command Interpreter

A shell does not simply take the line you type and run it.

Bash first "interprets" the line as follows:
1. the input line is broken into **tokens** ("**words**" + "**operators**")
2. **aliases** are applied (i.e., **macros**, possibly rewriting commands)
3. the tokens are *parsed* to identify individual **commands**
4. a sequence of **shell expansions** are performed (possibly rewriting commands)
5. I/O **redirections** are performed

After this processing, the command(s) that result are **executed** (each simple command is run in a separate **subprocess**).

Bash *waits* for all command(s) to terminate before prompting for the next input line.

Commands

Bash supports several types of **commands**:

- **simple commands** – a single command/executable
- **pipelines** – sequence of simple commands run *in parallel*, the *output* of each command becomes the *input* of the next
- **lists** – multiple simple commands or pipelines, run *sequentially*, and possibly *conditionally* (based on success)
- **compound commands** – conditional, looping, and grouping commands

Commands (contd.)

A command line is typically terminated by typing *enter/return*, which indicates **newline/end-of-line**.

Unlike *C-family languages*, in Bash the *newline is meaningful*, so you don't need to put semicolons at the end of every Bash line.

A command line can also be terminated with a "**&**" (*ampersand*) before typing enter/return.

This causes the command be run "in the **background**," which means it is run **asynchronously** with the shell, in a subprocess.

"**Backgrounding a command**" is often done when the command starts a GUI program, since this will allow the shell to continue to be used (the shell will immediately prompt for further input).

Simple Command Syntax

The standard syntax for a **simple command** is:
`command [option...] [arg...]`

`command` is either:
- the *name* of the command; or
- a **pathname** (**path**) of an executable.

If only a command *name* is given, the shell locates the executable by looking through the directories in the **shell search path** for a file with the specified name, which is executable by the user.

The shell search path is maintained in the **PATH environment variable**, as a *colon*-separated list of directory pathnames:
`printenv PATH` ⇒ `/usr/bin:/bin:/usr/local/bin:`...

Simple Command Syntax (contd.)

`[option...]` indicates that one or more **options** may be specified after the command, but that options are *optional*.

`[arg...]` indicates that one or more **command arguments** (also called **operands**) can be supplied after any options, but that arguments may be optional.

If supplied, command **options** are typically in one of *two forms*:

- *old/short style* – *single letters*, prefixed with a *single dash* (`-`)
- *new/long style* – words, prefixed with a *double dash* (`--`)

Examples:
```
ls -a -d *
ls --all --directory *
```

Simple Command Syntax (contd.)

With old style options:

- multiple options may be listed together:
  `ls -ad *`
- if an option requires an **argument**, the argument may be put right after the option letter or separated by whitespace:
  `ls -adw80 *`

With new style options:

- if an option requires an argument, it is done using "=":
  `ls --all --directory --width=80 *`

Old and new options may be mixed:
`ls -ad --width=80 *`

Dash or Hyphen or Minus Sign?

What should the ASCII character 0x2D (which has a standard printed representation of "-") be called?

The term *"dash"* was used above, but many people call it *hyphen* and some *minus sign*.

SUS does refer to it as *hyphen*, but mainly as "the `'-'` character."

In English, there is a difference between a *dash* and a *hyphen*, both in purpose and in typical printed representation.

However, when using 0x2D to introduce options, the character is functioning neither as an English dash nor hyphen (and most certainly not as an arithmetic operator).

We tend to prefer the single syllable "dash," but these are not English dashes any more than those are "periods" in "`www.cs.siu.edu`".

Pipelines

One of the most powerful aspects of Linux/UNIX shells, is the ability to construct unique, complex commands by connecting together *sequences of simple commands* into a **pipeline**.

This is accomplished by using **pipes**, a Linux/UNIX mechanism for **interprocess communication**.

The shell pipeline symbol is: | (the *vertical bar*).

The basic format of a shell pipeline command is:
`simple_command1 | simple_command2`

What this does is to setup a pipe that passes the *output* from `simple_command1` to become the *input* to `simple_command2`.

Pipelines (contd.)

More precisely, for `simple_command1 | simple_command2` :

- the shell creates a **pipe** object
- one **subprocess** is created to run `simple_command1`
- in that subprocess, **standard output** is **redirected** to the **write end** of the pipe
- a second **subprocess** is created to run `simple_command2`
- in that subprocess, **standard input** is **redirected** to the **read end** of the pipe
- the two commands are run **concurrently** in the two subprocesses, while the shell waits

Pipelines (contd.)

For a pipe to work, the individual commands must *read from* **standard input** and *write to* **standard output**.

Commands of this type are called **filters**.

Many standard filter commands are available for use:
`cut`, `grep`, `head`, `nl`, `sort`, `split`, `tail`, `tr`, etc.

Multiple pipelines may be used to string together more than two filter commands.

In this way, complex and unique functionality can easily be constructed by users from multiple relatively simple programs.

Pipelines (contd.)

Pipeline examples:

- `grep printf lab2.c | wc -l` (count `printf`'s in `lab2.c`)
- `ls -l *.c | grep Jun | wc -l` (count C files modified in June)

Several commands can function as "normal commands" or as filters, depending on how they are called:

- `grep [OPTIONS] PATTERN [FILE...]`
- reads from `FILE` argument(s) if any are given, else reads from **standard input**

Lists

A **list** command consists of multiple simple commands or pipelines separated by one of:

- `;` (semicolon) – commands are run *sequentially*
- `&&` (double ampersand) – each command is run only if the previous *"succeeded"*
- `||` (double vertical bar) – each command is run only if the previous *"failed"*

Success/failure refers to the **exit status** of a command.

"`&&`" is effectively *"short-circuit and"* while "`||`" is effectively *"short-circuit or"*.

Compound Commands

Compound commands is the general name for Bash' **grouping commands** and **control constructs**.

Grouping commands simply allow a **list** to be treated as a *unit* (for redirection):

- `(` *list* `)` – run *list* in subshell
- `{` *list*`; }` – run *list* in current shell context

Control Constructs

Shells typically provide a variety of **control constructs**, though syntax varies among shells.

Bash **conditional** constructs:
if-then, if-then-else, if-then-elseif, case (switch)

Bash **looping** constructs:
while, until, for, map (through a list)

Control constructs are mainly used in **shell scripts** (shell language programs).

However they are often useful in interactive commands:

```
  for f in *.txt; do mv "$f" "${f%txt}text"; done
```

(change the extensions of all `.txt` files in the CWD to `.text`, something that cannot be done with a single `mv` command)

Shell Scripts

A **shell script** is a program written in a shell language and placed into an executable file.

Scripts are often used to automate tasks that involve a sequence of commands (so can be run by cron).

E.g., convert mp3's to wav's in some directory:

```
#!/bin/bash

cd "$1"  #cd to directory given as first command-line argument

for f in *.{mp3,MP3}; do
  lame --decode "$f" "${f%.*}.wav"
done

exit 0
```

Bash Basics 2: Expansions & Metachars

These slides introduce the basics of working with **Bash**:

1. Intro & Commands
2. **Expansions & Metachars**
   - **shell expansions**
   - **metacharacters for expansions**
   - **parameters**
   - **filename expansion**
3. Variables & Quoting
4. Interactive Use
5. Redirections & Locale

Expansions and Metacharcters

A key aspect of a shell "interpreting" a command line prior to executing it is the application of the various **shell expansions**.

Expansions make use of a number of **metacharacters**: characters that have special interpretations.

Expansions are a key *productivity feature*, since they save typing.

The main **expansion phases** are (in order):

1. **brace expansion**
2. **tilde expansion**
3. **parameter expansion**
4. **command substitution**
5. **arithmetic expansion**
6. **filename expansion** (also known as **globbing**)

Brace Expansion

Brace expansion ({str_1,str_2,*etc*} or {int_1..int_2}) allows *multiple words* containing *alternative* substrings to be generated:

- `echo a{bc,def,ghi}j` produces:
  `abcj adefj aghij` (a list of three words)
- `mv file.{txt,text}` produces:
  `mv file.txt file.text`
- `rm test.save{01..04}` produces:
  `rm test.save01 test.save02 test.save03 test.save04`
- `cp {a..g}* ~/save` produces:
  `cp a* b* ... f* g* ~/save`
  (Single characters effectively interpreted as ASCII integers.)

(Note that **concatenation** happens automatically in Bash via the adjacent placement of words.)

Tilde Expansion

Tilde expansion (~*username*) allows a users home directory to be compactly specified as part of a path:

- `~smith/Docs/foo` is replaced by `/home/smith/Docs/foo`
- `~` alone is replaced by the current user's home directory:
  e.g., `~/.bashrc` is replaced by `/home/carver/.bashrc`
- `~` is often used to indicate any user's home directory

Parameters

Before we can discuss **parameter expansion**, we must be clear on just what "**parameters**" are.

In Bash, a **parameter** is an *"entity that stores values."*

There are three types of parameters:

- **variables** – denoted by *names*
- **positional parameters** – denoted by *numbers*
- **special parameters** – denoted by *special characters*

Parameters (contd.)

Parameter examples:

- variables (possible examples): `x`, `file_ext`, etc.
- environment variables: `PATH`, `PWD`, etc.
- positional parameters: `1`, `2`, etc.
- special parameters: `*`, `@`, `?`, `#`, etc.

The positional parameters represent the arguments given to a program/command (or a function) on the command line, with `1` representing the first argument, `2` the second, and so forth.

Parameters (contd.)

Special parameters represent various things, such as:

- `*` – all the CL arguments (`"*"` gives args as one word)
- `@` – all the CL arguments (`"@"` gives args as separate words)
- `#` – the number of CL arguments
- `?` – **exit status** of the last command
- `$` – shell PID

Parameter Expansion

In most programming languages, when a variable is used in an expression or on the RHS of an assignment, the *variable name is replaced with its value*—i.e., the variable is *implicitly* **evaluated** and its value substituted.

This is *not* the case for Bash: variables (and other parameters) must be *explicity evaluated* if their values are desired.

Parameter expansion retrieves a parameter's value:

- `${param}` is replaced by the value stored in parameter `param`
- `$param` can also be used if end of parameter word is clear

Parameter Expansion (contd.)

The parameter expansion syntax also accepts a number of *operators* that can be used to manipulate a retrieved value.

Here are some examples showing the use of select operators: (assume `file` is a variable with value "`test.text.save`")

- `${file%.*}` `file`'s value minus the *final* extension: `test.text`
- `${file%%.*}` `file`'s value minus *all* extensions: `test`
- `${file#*.}` `file`'s value minus everything up to the *first* "`.`": `text.save`
- `${file##*.}` `file`'s value minus everything up to the *final* "`.`": `save`

Command Substitution

Command substitution (`$(`*cmd*`)`) executes a command and substitutes its result (output to *standard output*) in its place:

- `$(ls *.txt)` is replaced by a list of all the "`.txt`" files in the CWD
- `` `ls *.txt` `` is alternative syntax ("**backticks**")

Most commonly used in shell scripts to avoid having to use *temporary variables* to store command results.

Can be useful when working interactively to avoid need to do cutting-and-pasting or retyping results:

```
ls -l $(which mv)
```

Arithmetic Expansion

Arithmentic expansion (`$((`*cmd*`))`) evaluates an arithmetic expression and substitutes its result in its place:

- `$((1 + 2))`
- `$((x + 2))` (note: do not need to write `$x`)
- `$((x++))`

Note: an artihmetic expansion like "`$((x++))`" must be used differently in Bash than the expression "`x++;`" in C-family languages.

While "`x++;`" is a fine expression in C, having "`$((x++))`" on a line will cause the resulting number to be interpreted as a command: e.g., `bash: 1: command not found`

Filename Expansion

Filename expansion uses several metacharacters to do *pattern matching on filenames*, allowing *multiple files* to be specified compactly.

Note that the filename expansion metachars appear similar to those used in **regular expressions**, but they are *not* the same!

Filename expansion metacharacters:

- `*` – matches any string, including the null/empty string
- `?` – matches any single character (but not none)
- `[...]` – matches any single enclosed character: `[yY]` `[abcd]` `[0123456789]`

Filename Expansion (contd.)

`[...]` accepts additional notation:

- a *dash* can be used to specify a range of characters:
  - `[0-9]` for `[0123456789]`
  - `[A-Z]` for upper case alphabetic chars (C locale)
  - `[a-zA-Z]` for all alphabetic chars (C locale)
- `[:`*class*`:]` character class notation:
  `[[:alpha:]]  [[:upper:]]  [[:digit:]]`

Can combine notations:

- `[[:upper:][:digit:]]`
- `[ab[:upper:]0-3]`

Filename Expansion (contd.)

Filename pattern example:

- `cp i*-[1-5]??.{txt,text} ~/tmp`
- will copy these files:
  - `i-123.text` (`*` matches empty string, `?`'s 23)
  - `iabc-5ab.txt` (`*` matches abc, `?`'s ab)
- will not copy these files:
  - `i123.txt` (no dash)
  - `abc-123.txt` (doesn't start with i)
  - `ia-923.txt` (9 is not in range 1-5)
  - `ia-12.txt` (second `?` has no match)
  - `i-123.text.save` (does not end in "`txt`" or "`text`")

Bash Basics 3: Variables & Quoting

These slides introduce the basics of working with **Bash**:

1. Intro & Commands
2. Expansions & Metachars
3. **Variables & Quoting**
   - **variables vs. parameters**
   - **assignment and evaluation**
   - **values**
   - **environment variables and exporting**
   - **quoting**
   - **word splitting**
4. Interactive Use
5. Redirections & Locale

Variables

Variables are one type of Bash **parameter** (*"entity that stores values"*).

Variables are used more in **shell scripts** than interactively, but are occasionally useful on the CL to avoid retyping arguments.

Users may need access to **environment variables** on the CL, and environment variables are accessible as shell variables.

Variables (contd.)

Variables in Bash are usually created via an **assignment**:

- `var=value` (*lack of whitespace is required!*)
- `var=` or `var=""` (assigns empty string value)

A variable's value is retrieved by explicity **evaluating it** (via **parameter expansion**):

- `${var}`
- `$var` (if end of variable name is clear)

Values are effectively stored as *strings*, which may be interpreted as other things (e.g., numbers) by using appropriate syntax.

Variables (contd.)

Variables can be declared using the `declare` builtin.

Variables are generally `declare`'d only if they are to be *arrays*.

They can also be `declare`'d to hold *integers*, which can simplify use of variables in arithmetic expressions.

Can "delete" a variable with the `unset` command.

Environment Variables

When Bash starts, all **environment variables** are automatically added as **shell variables**.

This means you can obtain the values of environment variables by evaluating the corresponding shell variables: `echo $PATH`

Regular shell variables are *not inherited* by any commands run in the shell (recall commands are normally run in subprocesses).

The `export` builtin can be used to cause regular shell variables to be accessible as *environment variables* in subsequent commands.

Quoting

Quoting can be used to disable the special treatment of characters or words, including:

- expansion **metacharacters**
- **word delimiter/separator** characters (e.g., whitespace)
- Bash **reserved words**

There are three quoting mechanisms:

• double quotes:	`"filename with spaces"`
• single quotes:	`'filename with spaces'`
• escapes:	`filename\ with\ spaces`

Quoting (contd.)

Quoting has two main interactive uses:

- avoiding filenames *containing whitespace* from being misinterpreted as *multiple arguments*:
  - e.g., `ls "Linux Talk 1"`
  - e.g., `ls 'Linux Talk 1'`
  - e.g., `ls Linux\ Talk\ 1`
- allowing *metacharacters* to be *passed* into a program (without being removed by the shell):
  - e.g., `myrename "*.txt"`
  - (want program to receive "`*.txt`" as its argument, not a list of `.txt` files)

Quoting (contd.)

The different types of quoting behave differently:

- *single quotes* preserve the literal value of characters (but a single quoted string cannot contain a single quote, even if it is escaped)
- *double quotes* preserve the literal value of most characters, but maintain the *special meaning of*: "`$`", "`` ` ``", "`\`", and "`!`"
- can get the *literal value* of these characters by *escaping them*: `"micro\$oft"`
- can include double quote(s) inside a double-quoted string by *escaping them*:
  `echo "type the command: \"test\" next"`

Quoting (contd.)

It is virtually always a good idea to *enclose variable evaluations inside double quotes*:

- e.g., `"$var"`
- otherwise there can be problems if `var` contains whitespace or is unset

It can sometimes be fairly difficult to understand the result of combining multiple quoting mechanisms, or to figure out how to ensure metachars get passed as arguments to programs.

Word Splitting

As mentioned above, one use of quoting is to suppress filenames from being split into multiple "words."

Bash *tokenizes* its input as a first step in its processing, and does similar "**word splitting**" during the expansion phases, just *prior* to *filename expansion*.

Sometimes it is critical to understand when/where word splitting will occur in order to know if quoting is required.

For example, in a variable **assignment** statement, `name=[value]`, `value` is not subject to word splitting (or filename expansion).

Thus, there is no need to write `x="$y"` rather than `x=$y`, since the result of parameter expansion on `y` will not be word split.

Word Splitting (contd.)

The shell variable `IFS` contains a string of the characters that can serve as word **delimiters/separators** in command lines.

By default, `IFS` is "$\langle space\rangle\langle tab\rangle\langle newline\rangle$" meaning that any sequence of spaces/tabs/newlines serves to separate words.

It is sometimes desirable to change `IFS`.

For example, if one wants to work with lists of filenames that can contain spaces, it might be desirable to split only on newlines.

This can be done with: `IFS=$'\n'`
(note special "C-string" notation to get a $\langle newline\rangle$)

Bash Basics 4: Interactive Use

These slides introduce the basics of working with **Bash**:

1. Intro & Commands
2. Expansions & Metachars
3. Variables & Quoting
4. **Interactive Use**
   - **features specifically for interactive use**
   - **aliases and functions**
   - **startup files (to make changes permanent)**
   - **terminal control characters**
5. Redirections & Locale

Features for Interactive Use

Bash has a number of features designed specifically to enhance user productivity for *interactive use*:

- **command-line editing**
- **tab completion**
- **command history**
- **prompt** customization

Command-Line Editing

The current command line can be *edited*:
(including one retrieved from the history)

- $\langle backspace \rangle$ deletes character to left of cursor
- $\langle delete \rangle$ and ctrl-d delete character to right of cursor
- left and right arrow keys move cursor
- ctrl-a moves to start of line
- ctrl-e moves to end of line
- alt-$\langle backspace \rangle$ deletes word to left of cursor
- alt-$\langle delete \rangle$ deletes word to right of cursor
- ctrl-u erases command-line to left of cursor
- ctrl-k erases command-line to right of cursor

Tab Completion

When the user types a $\langle tab \rangle$ on the command line, Bash will try to complete the current word, as:

- a variable, if starts with $
- a username, if starts with ~
- a command, alias, or function
- a filename/directory (if no other matches are found)

If the current word is ambiguous, Bash will display the set of matches (or beep if there are no matches).

Command History

Bash maintains a **history** of the recent commands that have been executed, and you can view and recall these commands to avoid retyping them.

The `history` command lists the entire history.

The *up arrow* moves backward through history, while *down arrow* moves forward.

You can also *search* through the history:

- ctrl-r does a reverse search
- ⟨*escape*⟩ terminates the search
- ctrl-g cancels the search

The Prompt

The Bash **prompt** is controlled by **shell variable** `PS1`, and is highly customizable.

Examples:

• `PS1="> "`	prompt:	`>`
• `PS1="\w> "`	prompt like:	`~/courses/306>`
• `PS1="\h:\w> "`	prompt like:	`pc01:~/courses/306>`

The *default prompt* string is: `"\s-\v\$ "` (Bash version + "`$` ")

See the *Bash Prompt HOWTO* for complete details and options.

Aliases

Aliases are a type of **macro** facility in the shell:
"Aliases allow a string to be substituted for a word when it is used as the first word of a simple command."

Aliases are applied as part of tokenizing the command line.

An alias is defined with the `alias` command:
`alias ll="ls -l"`

The `alias` command (with no arguments) will list all the currently defined aliases.

By default, aliases are applied only if the shell is *interactive*.

Aliases (contd.)

Aliases allow users to:

- Use certain command options by default:
  - `alias rm="rm -i"`
  - `alias ls="ls -F"`
- Give special/short names to commands with commonly used options and/or arguments:
  - `alias ll="ls -l"`
  - `alias up="cd .."`
- Give "better" names to standard commands:
  - `alias dir="ls"`
  - `alias filesize="du -s"`

Aliases (contd.)

Alias definitions must be placed in a Bash startup file in order to be permanently available (see below).

Many users have a separate file of alias definitions that is loaded each time they login.

Most distributions also define various aliases in the system-wide Bash startup file.

The `unalias` command can be used to disable an alias.

Functions

In Bash, aliases cannot take arguments (as they can in csh), so **functions** may have to be used instead.

A function can be defined using any of the following syntax variations:

- `function` *name* `() {` *...body...* `}`
- `function` *name* `{` *...body...* `}`
- *name* `() {` *...body...* `}`

Functions are called and executed just like simple commands.

Inside the function body, arguments to the function call can be accessed using the parameters $1, $2, etc.

Functions (contd.)

Simple example function:

```
function dos2unix ()
{
  tr -d "\r" < "$1" > "$2"
}
```

Called as:

```
dos2unix file.dos file.unix
```

Cannot be done as an alias, since ">" must be inserted into the middle of the file argument list.

Functions (contd.)

Since functions can contain arbitrary shell code, they can be much more complex than aliases:

```
#Compile C source file using GCC:
function comp ()
{
  if [[ ($# -lt 1) || ($# -gt 2) ]]; then
    echo "Usage: comp C_SOURCE_FILE [OBJECT_FILE]"
    return 1
  fi

  if [[ $# == 1 ]]; then
    gcc -Wall -std=gnu99 -o "${1%.*}" "$1"
  else
    gcc -Wall -std=gnu99 -o "$2" "$1"
  fi
}
```

Functions (contd.)

Defined like aliases at shell startup, functions like this provide an alternative to having large numbers of small shell script files.

You can check whether "*name*" names a function by using the command `type`:

- `type -t` *name*
- `type` *name* (shows the function definition)

Functions can also be used as **subroutines** in shell scripts, to name a sequence of repeatedly called shell code statements.

Startup Files

Bash reads from one or more **initialization files** when it starts.

Customization (e.g., setting the prompt, defining aliases) must be done via these files in order to be *permanent*.

If invoked as a "**login shell**," reads:

- `/etc/profile`
- followed by first found of:
  `~/.bash_profile` or `~/.bash_login` or `~/.profile`

If not a login shell, reads:

- `~/.bashrc`

User customizations typically go in `~/.bashrc`

Special Terminal Control Characters

Though actually a **terminal driver** rather than shell feature, there are *three special characters* that are frequently used when working interactively with shells:

- **ctrl-c** – sends an "**interrupt signal**" (`SIGINT`) that will usually *terminate* a running command
- **ctrl-\** – sends a "**quit signal**" (`SIGQUIT`) that will usually terminate a running command
- **ctrl-d** – causes an **end of file** return when a program is reading from **standard input** (`STDIN`)

Bash Basics 5: Redirections & Locale

These slides introduce the basics of working with **Bash**:

1. Intro & Commands
2. Expansions & Metachars
3. Variables & Quoting
4. Interactive Use
5. **Redirections & Locale**
   - **input and output redirection**
   - **locale and collation (sorting) order**

Redirections

Prior to actual execution of an interpreted command line, input and output to/from the commands may be **redirected**.

For example, instead of having output go to the terminal, it might be redirected to a file (so it can be saved).

Redirections are done in the shell, so commands/programs are usually unaware of it: a command writes to standard output as normal, but the output ends up in a file.

We have two kinds of redirections:

- **output redirection** – affects output/writing
- **input redirection** – affects input/reading

Redirections (contd.)

Output redirection:

- syntax: *command* `>FILE`
- syntax: *command* `>>FILE` (for appending)
- causes *command* output (to **standard output**) to be written/appended to `FILE`

Input redirection:

- syntax: *command* `<FILE`
- causes *command* input (from **standard input**) to be read from `FILE`

Redirections (contd.)

Examples:

- `grep printf lab2.c > lab2-printfs`
- `grep printf *.c >> labs-printfs`
- `ls -l *.c | grep Jun | wc -l > count`
- `grep printf < lab2.c`
  (same effect as: `grep printf lab2.c`)
- `tr -d'\r' < dosfile > unixfile`
  (`tr` only reads from stdin and writes to stdout, so must use redirection to apply to files)

Redirections (contd.)

Can redirect **standard error** output as well:

- redirect standard error output:
  command `2> FILE`
- redirect standard output and standard error:
  command `&> FILE`
  (this is shorthand for: *command* `>FILE 2>&1`)

Redirections are generally applied to standard input, standard output, and/or standard error.

They can be applied to arbitrary **file descriptors** however:

- syntax: *command* `[N]>FILE`
- syntax: *command* `[N]<FILE`
- `[N]` indicates optional FD (integer)

Locale and Collation Order

Internationalization (**i18n**) affects some aspects of filename expansion and other shell elements.

Internationalization settings are controlled by **locale** settings, which are maintained as *environment variables*, most having the prefix "`LC_`".

The `locale` command will print a list of the locale environment variables and their values.

An important variable for Bash is `LC_COLLATE`, which determines the **collation (sort) order** for characters.

Locale and Collation Order (contd.)

Collation order affects the meaning of character ranges in the `[...]` filename expansion.

For example, we expect `[A-Z]` to match any uppercase letter, since the ASCII collation order is abc...xyzABC...XYZ.

However, with some newer distros, the default collation order can be aAbBcC...xXyYzZ.

This causes `[A-Z]` to be equivalent to `[AbBcC...xXyYzZ]`, i.e., to match all letters but "a".

This could be disastrous if a user executes a command such as "`rm [A-Z]*.data`" as files may unexpectedly be deleted.

Locale and Collation Order (contd.)

Collation order may be set to standard ASCII order by running the following command:

```
export LC_COLLATE=C
```

A collate setting that will cause order issues is:

```
LC_COLLATE=en_US.UTF-8
```

The locale settings can affect other things, such as the *date format* used by `ls` or other programs.