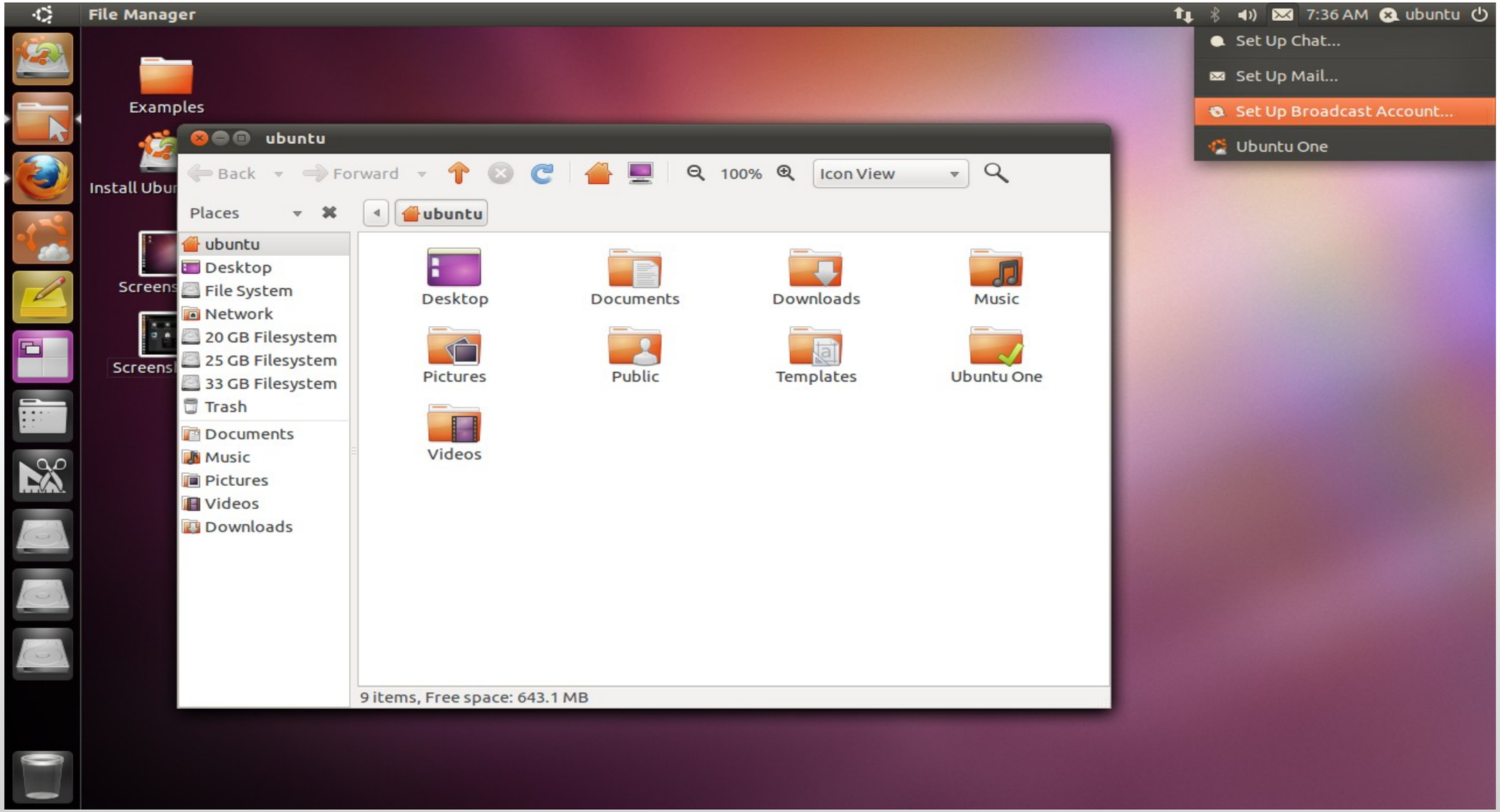
[Software Development]

Introduction to the Shell

Davide Balzarotti

Eurecom – Sophia Antipolis, France

What a Linux Desktop Installation may look like



What we need

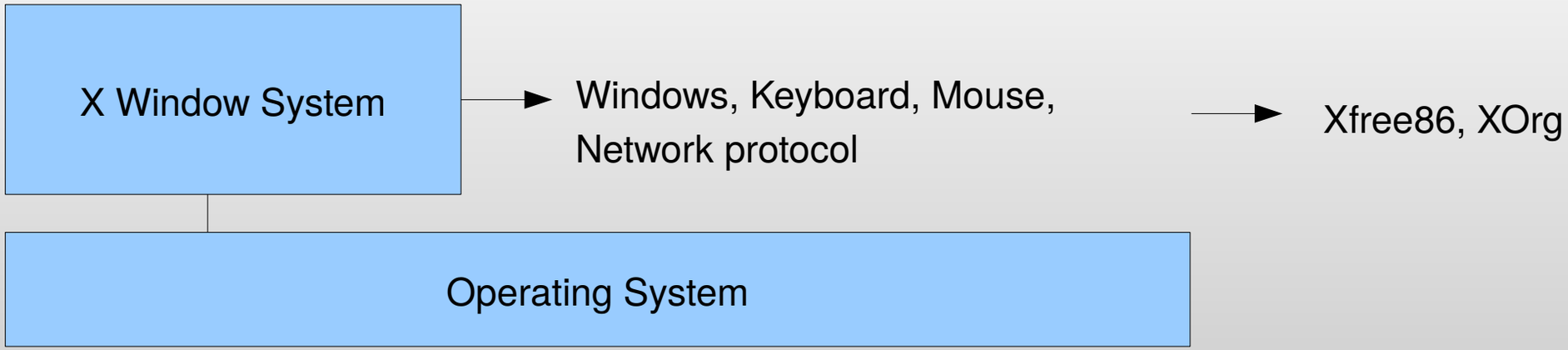
```
[/home]$ ls
vidarlo
[/home]$ cd ..
[/]$ cd etc
[/etc]$ ls
0.0.10.in-addr.arpa  csh.cshrc            gshadow-          logrotate.d      odbcinst.ini          rmt
adduser.conf         csh.login            gtk               lynx.cfg         openoffice            rpc
adjtime              csh.logout           host.conf         magic            opt                   screenrc
aliases              db.cache             hostname          mailcap          pam.conf              securetty
alternatives         debconf.conf         hosts             mailcap.order    pam.d                 security
apm                  debian_version       hosts.allow       mailname         passwd                services
apt                  default              hosts.deny        mail.rc          passwd-               shadow
asterisk             defoma               hotplug           manpath.config   perl                  shadow-
at.deny              deluser.conf         hotplug.d         mdadm            ppp                   shells
bakipkungfu          dhclient.conf        identd.conf       mediaprm         printcap              skel
bash.bashrc          dhclient-script      identd.key        mime.types       profile               squid
bash_completion      dictionaries-common  inetd.conf        mkinitrd         protocols             ssh
bash_completion.d    discover.conf        init.d            modprobe.d       python2.3             sudoers
bind                 discover.conf-2.6    inittab           modules          raidtab               sysctl.conf
blkid.tab            discover.d           inputrc           modules.conf     rc0.d                 syslog.conf
blkid.tab.old        dpkg                 ipkungfu          modules.conf.old rc1.d                 terminfo
calendar             emacs                issue             modutils         rc2.d                 timezone
chatscripts          emacs21              issue.net         motd             rc3.d                 ucf.conf
chkrootkit.conf      email-addresses      kernel-img.conf   mtab             rc4.d                 updatedb.conf
complete.tcsh        environment          ldap              mtools.conf      rc5.d                 vidarlo.net.hosts
console              exim4                ld.so.cache       Muttrc           rc6.d                 w3m
console-tools        fdmount.conf         ld.so.conf        mysql            rc.d                  wgetrc
cron.d               fonts                locale.alias      nanorc           rcS.d                 #wvdial.conf#
cron.daily           fstab                locale.gen        network          reportbug.conf        wvdial.conf
cron.hourly          groff                localtime         networks         resolvconf            wvdial.conf~
cron.monthly         group                logcheck          nsswitch.conf    resolv.conf           X11
crontab              group-               login.defs        ODBCDataSources  resolv.conf~          xpilot
cron.weekly          gshadow              logrotate.conf    odbc.ini         resolv.conf.pppd-backup
[/etc]$
```

Few Words about the Graphic Interface

Unlike in Windows, the graphic interface is just a program,
and it is NOT part of the operating system

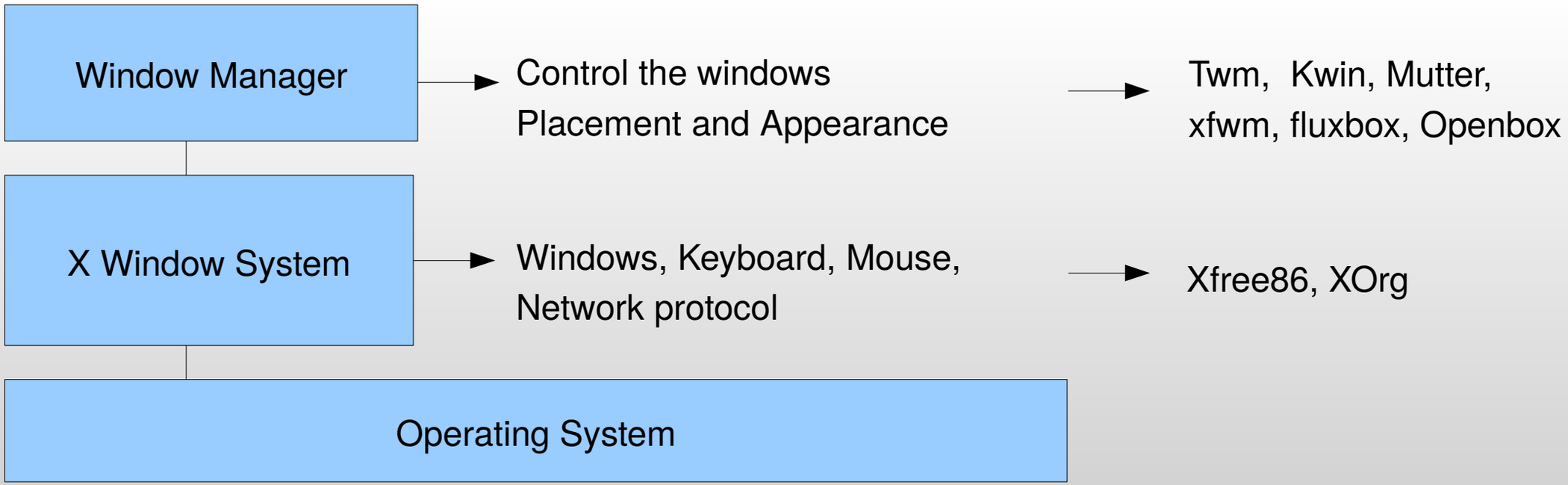
Few Words about the Graphic Interface

Unlike in Windows, the graphic interface is just a program,
and it is NOT part of the operating system



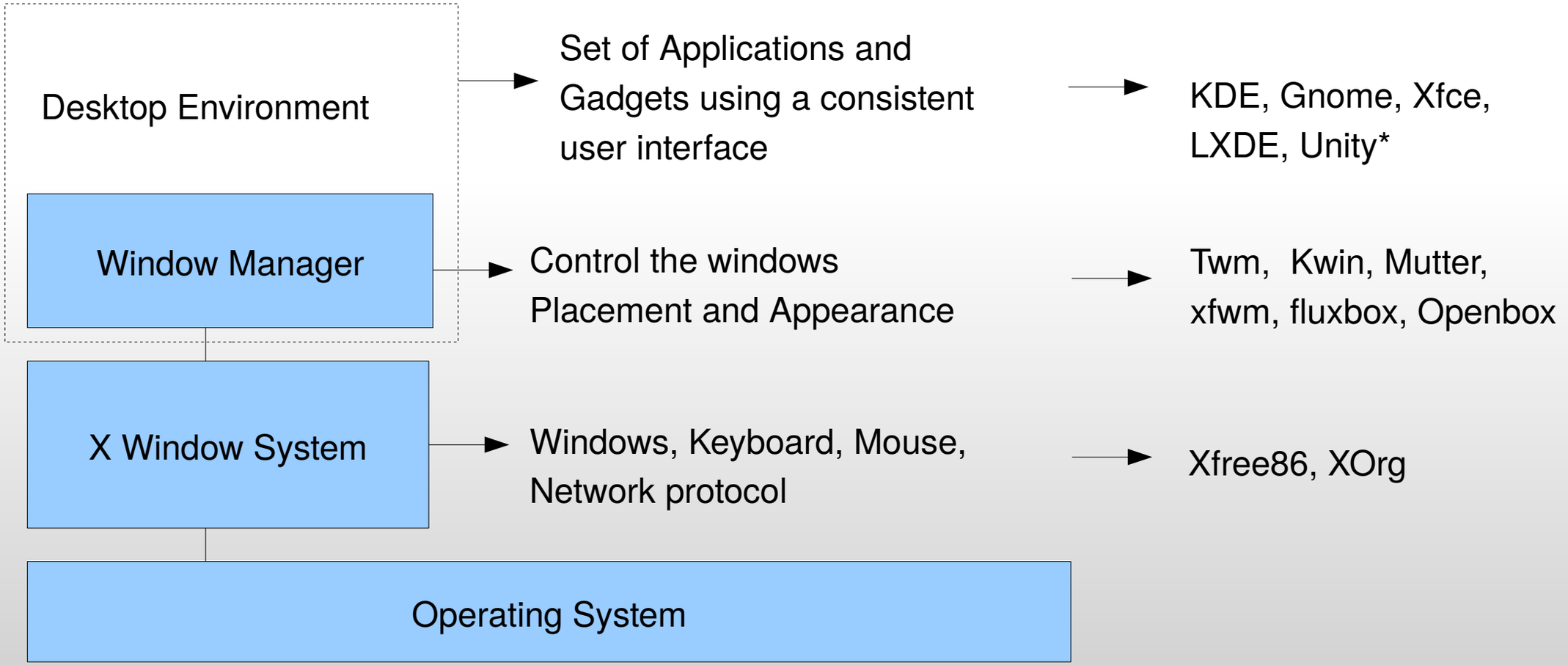
Few Words about the Graphic Interface

Unlike in Windows, the graphic interface is just a program,
and it is NOT part of the operating system



Few Words about the Graphic Interface

Unlike in Windows, the graphic interface is just a program, and it is NOT part of the operating system



*(technically a desktop shell)

Fancy, but not Required

- You can have a Linux system without a graphic interface
  - Almost always the case for servers
  - Probably not a good idea for desktops
- Since the graphic interface is just a program, you can start it, stop it, replace it, uninstall it..
- We will use the command line to control the system and do our job
  - Press CTRL-ALT-F1 to (temporarily) switch back to a console
  - Or open a terminal window (Xterm, Eterm, Konsole...) in the graphic environment
- Inside your terminal you interact with a program that is responsible to interpret your commands: the shell

The Shell

- The Shell is the program you use to communicate with the system
- The Unix shell is both a command interpreter and a programming language
  - As a language, the shell provides variables, control flow constructs, functions …
- A shell may be used interactively or non-interactively
  - In interactive mode, it interprets and executes the commands that the user types on the keyboard
  - When executing non-interactively, it reads and executes the commands from a file (shell script)

Executing Commands

- Each shell provides a small set of built-in commands (*builtins*) that implement functionalities either impossible or inconvenient to obtain with separate utilities

- When the user types a command...

  1. the shell first checks if it is a built-in command and, if so, it executes it
  2. If the command name is an absolute path beginning with / (like `/bin/ls`) the corresponding program is executed
  3. If the command is neither built-in, nor specified with an absolute path name, the shell looks in its search PATH for an executable file with the given name

- When a shell has to execute an external command

  - It spawns (fork) an identical subprocess
  - It executes (exec) the command inside the new process

Ready to Take Your Orders

```
balzarot:/usr> ls -l -a
```

Command Prompt:
Shows some (configurable) information to the user and tells him that the shell is ready to take commands

Ready to Take Your Orders

User command:
Commands are case sensitive

Ready to Take Your Orders

```
balzarot:/usr> ls -l -a
```

Command Parameters:
Space-separated list of parameters

Ready to Take Your Orders

```
balzarot:/usr> ls -l -a
total 168
drwxr-xr-x  12 root root  4096 2008-09-24 22:09 .
drwxr-xr-x  21 root root  4096 2008-07-18 17:47 ..
drwxr-xr-x   2 root root 40960 2009-08-29 18:55 bin
drwxr-xr-x   2 root root  4096 2009-01-25 17:41 games
drwxr-xr-x  37 root root 12288 2009-07-26 13:05 include
drwxr-xr-x 141 root root 69632 2009-08-29 18:55 lib
drwxr-xr-x   4 root root  4096 2009-07-24 13:55 lib32
drwxr-xr-x  11 root root  4096 2008-08-20 21:50 local
drwxr-xr-x   2 root root 12288 2009-07-26 17:49 sbin
drwxr-xr-x 211 root root  4096 2009-08-17 19:51 share
drwxrwsr-x   5 root src   4096 2008-09-24 22:09 src
drwxr-xr-x   3 root root  4096 2008-04-22 20:43 X11R6

balzarot:/usr>
```

[B]ourne [A]gain [SH]ell

- There are many shells
  - Bourne Shell (sh)
  - Korn Shell (ksh)
  - Z Shell (zsh)
  - C Shell (csh)
  - Bourne Again Shell (bash)
  - Mud Shell (mudsh)
  - …
- Bash is the shell developed by the GNU Project
  - It is the default shell on most systems built on top of the Linux kernel as well as on Mac OS X

Shell & Environment Variables

- Every Unix process runs in a specific environment
  - The environment is defined by an array of strings, each defining a variable with its assigned value
  - When a new program is executed, it inherits the environment from its parent (the process that created it)

Shell & Environment Variables

- Every Unix process runs in a specific environment
  - The environment is defined by an array of strings, each defining a variable with its assigned value
  - When a new program is executed, it inherits the environment from its parent (the process that created it)

- The shell also has its *own* variables
  - When BASH starts, it copies all the environment variables to local variables and set them to be automatically exported to the environment
  - If a new shell variable is defined, it must be explicitly "*exported*" to the environment in order to be seen from any forked subprocesses

Variables Use

Assignment	`varname=value (no spaces!)`
Deletion	`unset varname`
Use	`$varname`
Export to the Environment	`export varname`
List	`set (shell variables)` `printenv (environment variables)`

- Plenty of predefined variables

  `$SHELL, $PATH, $USER, $HOME, $PS1...`

- Special variables (can be referenced but not assigned)

  `$?` = return code of the last executed command

The $PATH Variable

- The PATH is a colon ":" separated list of directories that the shell use to locate the commands to execute

```
balzarot:~> echo $PATH
/usr/local/bin:/usr/sbin:/usr/bin:/bin
```

- The current directory (.) is NOT in the PATH for very good security reasons

The $PATH Variable

- The PATH is a colon ":" separated list of directories that the shell use to locate the commands to execute

```
balzarot:~> echo $PATH
/usr/local/bin:/usr/sbin:/usr/bin:/bin
```

- The current directory (.) is NOT in the PATH for very good security reasons

```
balzarot:~> PATH=.:$PATH
balzarot:~> ls /tmp/bad_dir
  ls*
balzarot:~> cd /tmp/bad_dir
balzarot:~> ls
  All your files are belong to us!!
balzarot:~>
```

Expansions

Expansions are performed on the command line after it has been split into tokens

- Tilde expansion
  - Replace `~` with the user home directory
  - Replace `~jack` with jack's home directory
- Shell parameter expansion
  - Replace `$varname` with the value of variable `varname`
- Command substitution
  - Replace `$(cmd)` or `` `cmd` `` with the output of cmd
- Process substitution
  - Replace `<(cmd)` with a temporary filename that contains the standard output of `cmd`

Expansions

- Filename expansion
  - Replace each word containing the characters ‘*’, ‘?’, ‘[]’, and '{}' with an alphabetically sorted list of file names matching the pattern
    - '*' matches any string (including an empty one)
    - '?' matches any character
    - '[...]' matches any of the enclosed characters
    - '{..,..} matches any of the enclosed (comma-separated) strings

Quoting

- Single quotes preserves the literal value of each character within the quotes (no expansion applied)

- Double quotes preserves the literal value of all characters within the quotes, with the exception of ‘$’, ‘`’, ‘\’ (no *filename* and *tilde* expansions)

Examples

```
> echo 'Hello $USER'
Hello $USER

> echo “Hello $USER”
Hello balzarot
```

Examples

```
> echo 'Hello $USER'
Hello $USER

> echo “Hello $USER”
Hello balzarot

> echo “Today is `date`”
Today is Sep 30 22:57:36 CEST 2009
```

Examples

```
> echo 'Hello $USER'
Hello $USER

> echo “Hello $USER”
Hello balzarot

> echo “Today is `date`”
Today is Sep 30 22:57:36 CEST 2009

> echo “List of text files: “ *.{txt,tex}
List of txt files: quotes.txt reviews.tex doc.txt
```

Examples

```
> echo 'Hello $USER'
Hello $USER

> echo “Hello $USER”
Hello balzarot

> echo “Today is `date`”
Today is Sep 30 22:57:36 CEST 2009

> echo “List of text files: “ *.{txt,tex}
List of txt files: quotes.txt reviews.tex doc.txt

> echo “My home directory is ” ~
My home directory is /home/balzarot
```

Examples

```
> echo 'Hello $USER'
Hello $USER

> echo “Hello $USER”
Hello balzarot

> echo “Today is `date`”
Today is Sep 30 22:57:36 CEST 2009

> echo “List of text files: “ *.{txt,tex}
List of txt files: quotes.txt reviews.tex doc.txt

> echo “My home directory is ” ~
My home directory is /home/balzarot

> echo <(ls)
/dev/fd/63

> cat <(ls)
quotes.txt reviews.tex doc.txt foo.c
```

Input and Output

- When a program starts, it inherits from its parent three open streams:
  - The standard input (or **`stdin`**)
  - The standard output (or **`stdout`**)
  - The standard error (or **`stderr`**)
- By default
  - The standard input is connected to the keyboard
  - The standard output and error are connected to the terminal screen

Input and Output

- When a program starts, it inherits from its parent three open streams:
  - The standard input (or **`stdin`**)
  - The standard output (or **`stdout`**)
  - The standard error (or **`stderr`**)
- By default
  - The standard input is connected to the keyboard
  - The standard output and error are connected to the terminal screen
- When a program ends, it returns a positive integer value (that is then stored in the `$?` variable)
  - 0 if the operation was successful
  - > 0 otherwise (the program documentation usually reports the possible return codes and their meaning)

Input/Output Redirection

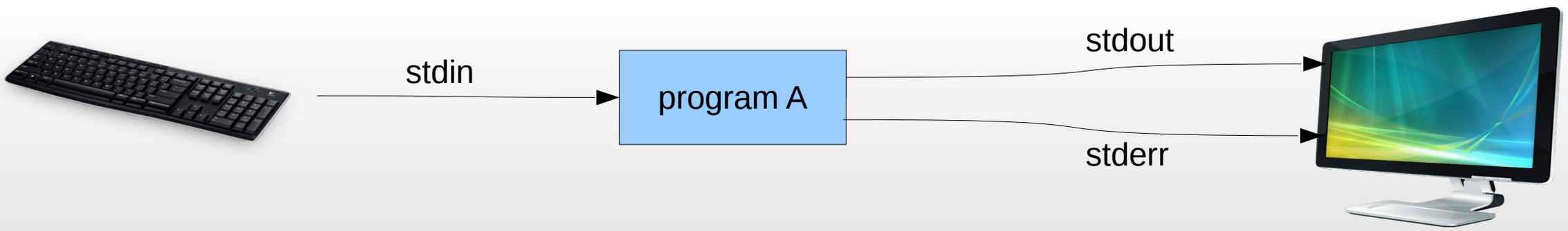
Send stdout (of prg) to a file	`prg > file`
Append stdout to a file	`prg >> file`
Send stderr to a file	`prg 2> file`
Append stderr to a file	`prg 2>> file`

Read stdin from a file	`prg < file`
Pipe stdout of prg1 to stdin of prg2	`prg1 | prg2`

Send stdout and stderr to a file	`prg > file 2>&1`
Append stdout and stderr to a file	`prg >> file 2>&1`
Pipe stdout and stderr of prg1 to stdin of prg2	`prg1 2>&1 | prg2`

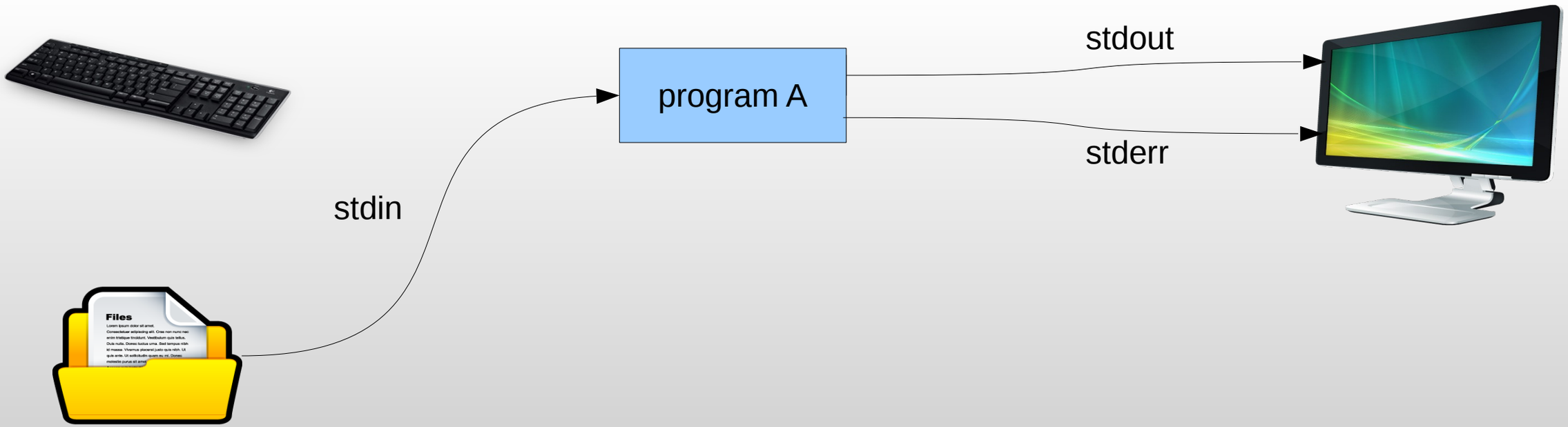
Example

```
> ./program_A
```



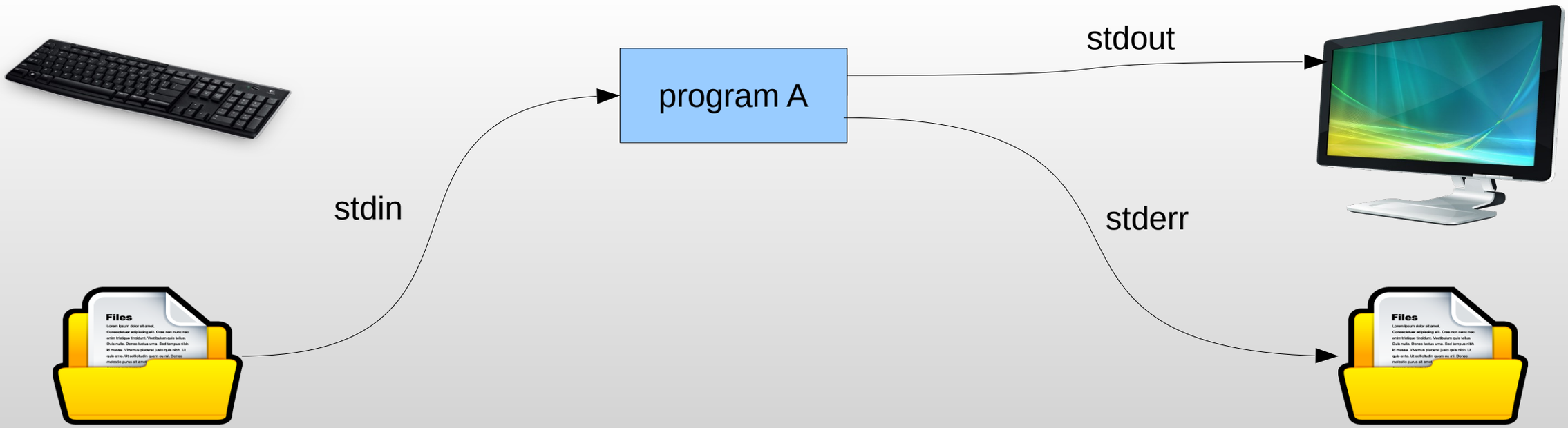
Example

```
> ./program_A < file_x
```



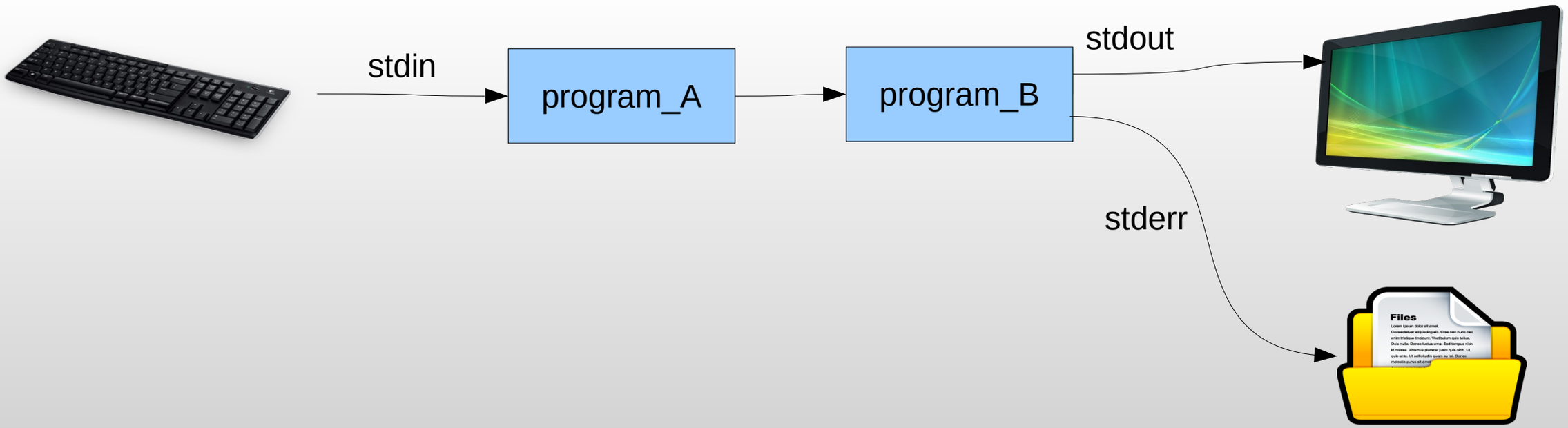
Example

```
> ./program_A < file_x 2> file_y
```



Example

```
> ./program_A | ./program_B 2> file_y
```



Advanced Redirection

- The operator `n>&m` rearranges the file descriptors making file descriptor `n` point to the same file as file descriptor `m`
  - The order matters !!
    - `cmd1 2>&1 > file`
    - `cmd1 > file 2>&1`

Advanced Redirection

- The operator `n>&m` rearranges the file descriptors making file descriptor `n` point to the same file as file descriptor `m`
  - The order matters !!
    - `cmd1 2>&1 > file`
    - `cmd1 > file 2>&1`
  - Pipe the standard error of a command to the standard input of another
    - `cmd1 3>&2 2>&1 1>&3 | cmd2`

Advanced Redirection

- The operator `n>&m` rearranges the file descriptors making file descriptor `n` point to the same file as file descriptor `m`
  - The order matters !!
    - `cmd1 2>&1 > file`
    - `cmd1 > file 2>&1`
  - Pipe the standard error of a command to the standard input of another
    - `cmd1 3>&2 2>&1 1>&3 | cmd2`
- Multiple redirections can be combined on the same line
  - `prog < input_file > output_file`
  - `prog > output_file 2>> errors_file`

warning: `prog < file > file` does not work because before executing the command `file` is open in read and write (opening a file in write mode empties the file)

Combining Commands

- Simple commands:
  - One command followed by its arguments
- Pipes
  - `cmd1 | cmd2 | cmd3 ...`
- Command lists
  - `cmd1 ; cmd2` – executes cmd1 and then cmd2
  - `cmd1 && cmd2` – cmd2 is executed if, and only if, cmd1 returns an exit status of zero (i.e., if cmd1 succeeded)
  - `cmd1 || cmd2` – cmd2 is executed if, and only if, cmd1 returns a non zero exit status (i.e., if cmd1 failed)
- Compound Commands
  - A list of commands with something (a test or a loop) around them

Combining Commands

- Commands can be grouped between brackets and the output of the entire group redirected or piped
  - `(cat file1; echo “end of file”) | cmd`
  - `(cat file1; echo “end of file”) > file`
- Streams can be suppressed by redirecting them to `/dev/null`
- Redirect the standard output of a command to the argument list of another command

```
cmd1 | xargs cmd2
```

For Loop

- Repeats a list of commands for each value in a list

```
for var in <ss_list>; do cmd1; cmd2; ... ; done
```

- How to use it
  - Combined with file name expansion
    - `for doc in *.txt; do cat $doc; done`
  - For each word in a file
    - `for word in $(cat file); do echo $word; done`
  - Traditional C way (using the `seq` command)
    - `for number in `seq 1 10`; do echo $number; done`

While Loop

- Repeats a list of commands, as long as the command controlling the while loop executes successfully (exit status equal to zero)

```
while test_cmd; do cmd1; cmd2; ...; done
```

- How to use it
  - For each line in a file (using the `read` builtin command)

```
cat file | while read line; do echo $line; done
```

Useful Bash Shortcuts

`crtl-r` – search in the command history

`crtl-l` – clear the screen

`ctrl-c` – kill the current process

`ctrl-z` – suspend the current process

`ctrl-s` – stop the output to the screen

`ctrl-q` – re-enable the output to the screen