Linux shell scripting – "Getting started"*

David Morgan

*based on chapter by the same name in Classic Shell Scripting by Robbins and Beebe

What's a script?

- text file containing commands
- executed as a unit
- "command" means a body of code from somewhere
- from where?
  - the alias list, in the shell's memory
  - the keywords, embedded in the shell
  - the functions that are in shell memory
  - the builtins, embedded in the shell code itself
  - a "binary" file, outboard to the shell

Precedence of selection for execution

- aliases
- keywords (a.k.a. reserved words)
- functions
- builtins
- files (binary executable and script)
  - hash table
  - PATH

Keywords (a.k.a. reserved words)

RESERVED WORDS

Reserved words are words that have a special meaning to the shell. The following words are recognized as reserved when unquoted and either the first word of a simple command … or the third word of a case or for command:

! case do done elif else esac fi for function if in select then until while { } time [[ ]]

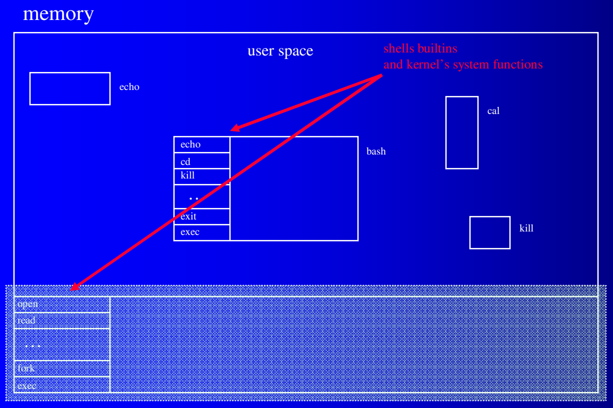
bash man page

bash builtin executables

source	continue	fc	popd	test
alias	declare	fg	printf	times
bg	typeset	getopts	pushd	trap
bind	dirs	hash	pwd	type
break	disown	help	read	ulimit
builtin	echo	history	readonly	umask
cd	enable	jobs	return	unalias
caller	eval	kill	set	unset
command	exec	let	shift	wait
compgen	exit	local	shopt	
complete	export	logout	suspend	

* code for a bash builtin resides in file /bin/bash, along with the rest of the builtins plus the shell program as a whole. Code for a utility like ls resides in its own dedicated file /bin/ls, which holds nothing else. Being "builtin" is a matter of placement. Other shells too may contain builtins. Their list may overlap with this one. Identically named builtins may not behave identically.

bash builtins

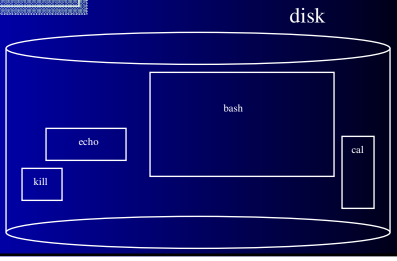


A builtin is a command contained within the Bash tool set, literally built in. This is either for performance reasons -- builtins execute faster than external commands, which usually require forking off a separate process -- or because a particular builtin needs direct access to the shell internals.

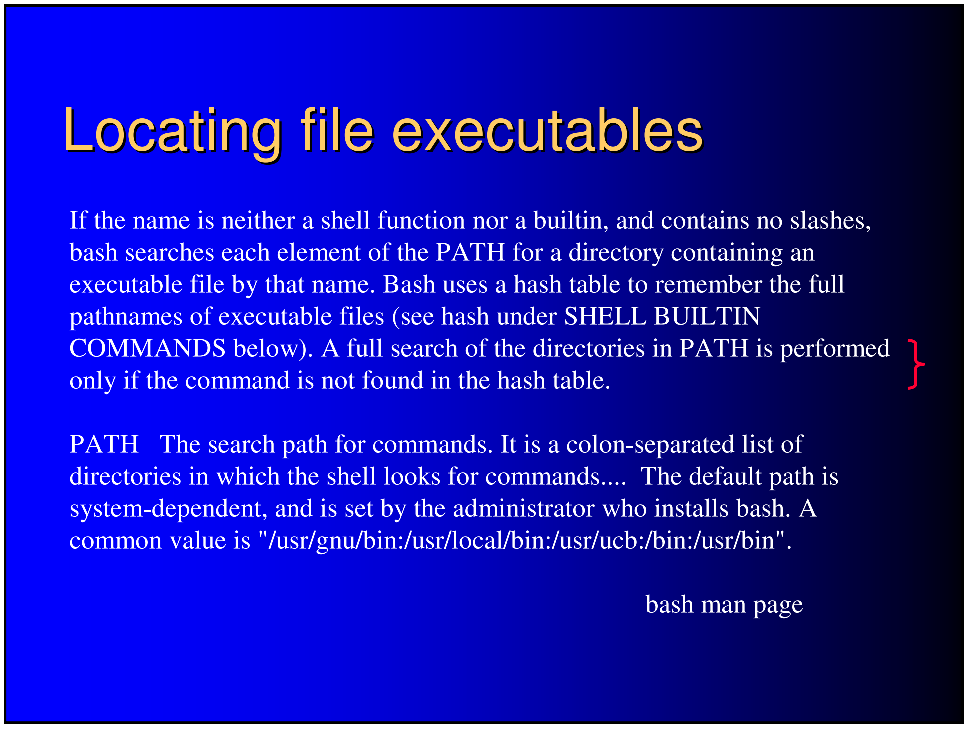
Advanced Bash-Scripting Guide

Shell and kernel each (separately) carry something built in to them– shell "builtins" and kernel "system functions"

kernel space



What reasons for builtins?
- some directly manipulate shell itself, can't achieve function outside shell
  e.g., cd, exit, exec
- efficiency
  e.g., echo, printf
- emergency role
  e.g. kill, if you've run out of free PIDs
  echo, if you've accidentally deleted /bin/echo
  echo *, if you've accidentally deleted /bin/ls



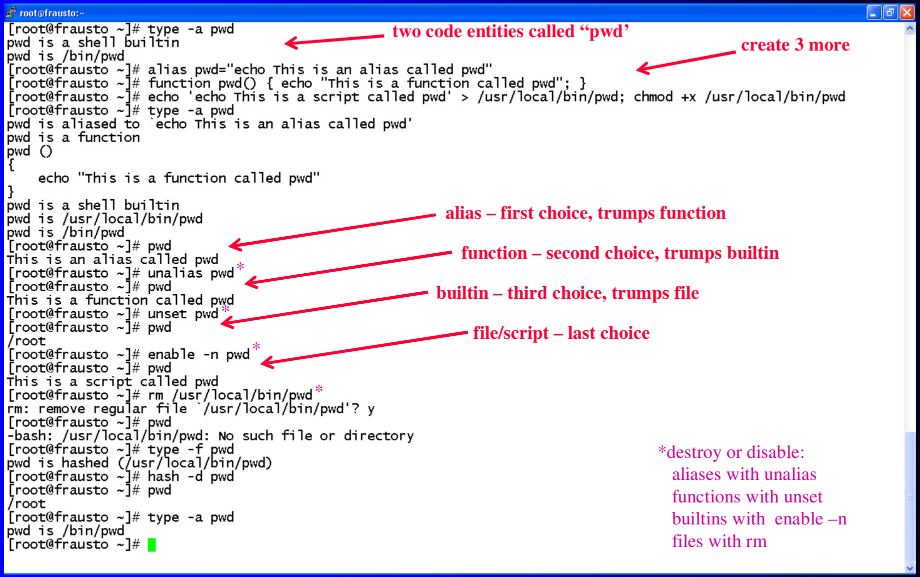
Locating file executables

If the name is neither a shell function nor a builtin, and contains no slashes, bash searches each element of the PATH for a directory containing an executable file by that name. Bash uses a hash table to remember the full pathnames of executable files (see hash under SHELL BUILTIN COMMANDS below). A full search of the directories in PATH is performed only if the command is not found in the hash table. }

PATH   The search path for commands. It is a colon-separated list of directories in which the shell looks for commands.... The default path is system-dependent, and is set by the administrator who installs bash. A common value is "/usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin".

bash man page

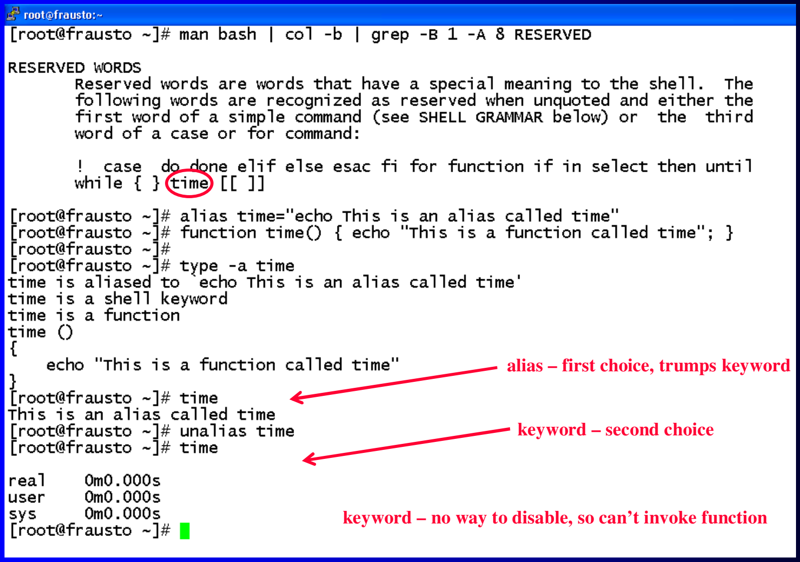
Example – lots of "pwd"s


root@frausto:~
two code entities called "pwd'
create 3 more
alias – first choice, trumps function
function – second choice, trumps builtin
builtin – third choice, trumps file
file/script – last choice

```
[root@frausto ~]# type -a pwd
pwd is a shell builtin
pwd is /bin/pwd
[root@frausto ~]# alias pwd="echo This is an alias called pwd"
[root@frausto ~]# function pwd() { echo "This is a function called pwd"; }
[root@frausto ~]# echo 'echo This is a script called pwd' > /usr/local/bin/pwd; chmod +x /usr/local/bin/pwd
[root@frausto ~]# type -a pwd
pwd is aliased to `echo This is an alias called pwd'
pwd is a function
pwd ()
{
    echo "This is a function called pwd"
}
pwd is a shell builtin
pwd is /usr/local/bin/pwd
pwd is /bin/pwd
[root@frausto ~]# pwd
This is an alias called pwd
[root@frausto ~]# unalias pwd*
[root@frausto ~]# pwd
This is a function called pwd
[root@frausto ~]# unset pwd*
[root@frausto ~]# pwd
/root
[root@frausto ~]# enable -n pwd*
[root@frausto ~]# pwd
This is a script called pwd
[root@frausto ~]# rm /usr/local/bin/pwd*
rm: remove regular file `/usr/local/bin/pwd'? y
[root@frausto ~]# pwd
-bash: /usr/local/bin/pwd: No such file or directory
[root@frausto ~]# type -f pwd
pwd is hashed (/usr/local/bin/pwd)
[root@frausto ~]# hash -d pwd
[root@frausto ~]# pwd
/root
[root@frausto ~]# type -a pwd
pwd is /bin/pwd
[root@frausto ~]#
```

*destroy or disable:
  aliases with unalias
  functions with unset
  builtins with enable –n
  files with rm

Example – lots of “time”s


root@frausto:~
alias – first choice, trumps keyword
keyword – second choice
keyword – no way to disable, so can’t invoke function

```
[root@frausto ~]# man bash | col -b | grep -B 1 -A 8 RESERVED

RESERVED WORDS
       Reserved words are words that have a special meaning to the shell.  The
       following words are recognized as reserved when unquoted and either the
       first word of a simple command (see SHELL GRAMMAR below) or  the  third
       word of a case or for command:

       !  case  do done elif else esac fi for function if in select then until
       while { } time [[ ]]

[root@frausto ~]# alias time="echo This is an alias called time"
[root@frausto ~]# function time() { echo "This is a function called time"; }
[root@frausto ~]#
[root@frausto ~]# type -a time
time is aliased to `echo This is an alias called time'
time is a shell keyword
time is a function
time ()
{
    echo "This is a function called time"
}
[root@frausto ~]# time
This is an alias called time
[root@frausto ~]# unalias time
[root@frausto ~]# time

real    0m0.000s
user    0m0.000s
sys     0m0.000s
[root@frausto ~]#
```

tip – in scripts control which identically named code you use

eg time (keyword or file?)

if you want the file
- provide its fully-qualified path name, or
- precede it with the built-in command “command”

if you want the keyword, just “time”

eg echo “hello” (builtin or file?)

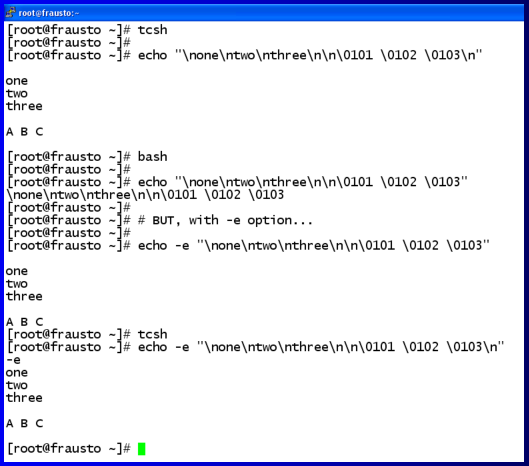
if you want the file
- provide its fully-quailified name or
- precede it with the built-in command “command” or
- disable (then later re-enable?) the builtin

if you want the builtin just “echo” or “builtin echo”

Simple output with echo

- common and convenient for output
- portability headache, different historical versions

example:

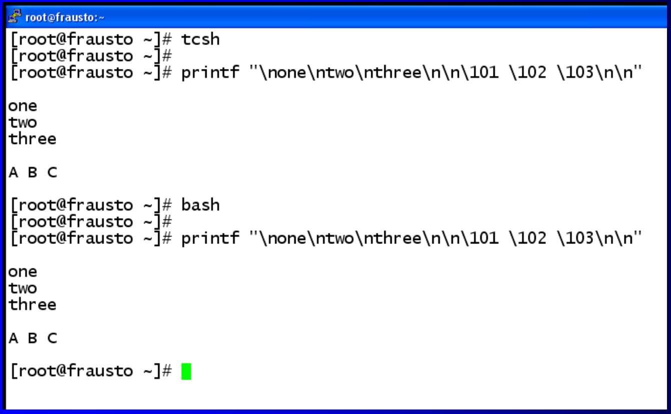


```
[root@frausto ~]# tcsh
[root@frausto ~]#
[root@frausto ~]# echo "\none\ntwo\nthree\n\n\0101 \0102 \0103\n"

one
two
three

A B C

[root@frausto ~]# bash
[root@frausto ~]#
[root@frausto ~]# echo "\none\ntwo\nthree\n\n\0101 \0102 \0103"
\none\ntwo\nthree\n\n\0101 \0102 \0103
[root@frausto ~]#
[root@frausto ~]# # BUT, with -e option...
[root@frausto ~]#
[root@frausto ~]# echo -e "\none\ntwo\nthree\n\n\0101 \0102 \0103"

one
two
three

A B C
[root@frausto ~]# tcsh
[root@frausto ~]# echo -e "\none\ntwo\nthree\n\n\0101 \0102 \0103\n"
-e
one
two
three

A B C

[root@frausto ~]#
```

same syntax, output differs per shell

Simple output with printf

- modeled after C printf( ) function
- portable across platforms
- preferred for portability



```
[root@frausto ~]# tcsh
[root@frausto ~]#
[root@frausto ~]# printf "\none\ntwo\nthree\n\n\101 \102 \103\n\n"

one
two
three

A B C

[root@frausto ~]# bash
[root@frausto ~]#
[root@frausto ~]# printf "\none\ntwo\nthree\n\n\101 \102 \103\n\n"

one
two
three

A B C

[root@frausto ~]#
```

same syntax, same output, both shells

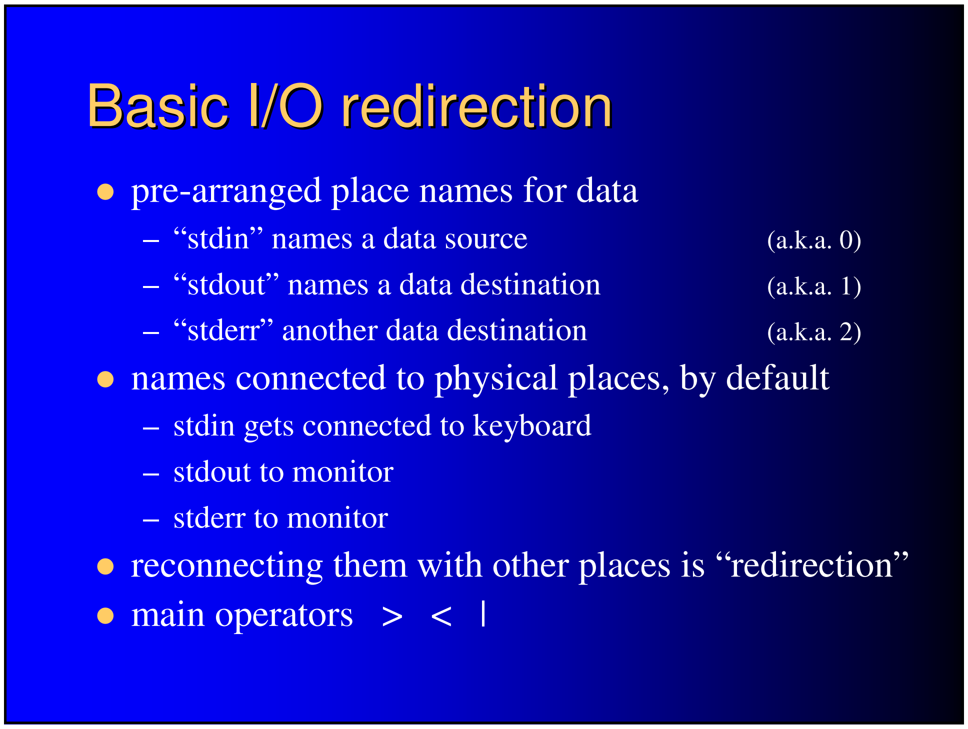
printf *format-string* [*arguments*]

- format string contains
  - literals
  - escape sequences
  - format specifiers
- specifiers apply to arguments

Standard I/O

"*Standard I/O* is perhaps the most fundamental concept in the Software Tools philosophy. The idea is that programs should have a data source, a data sink (where data goes, and a place to report problems. These are referred to by the names *standard input*, *standard output*, and *standard error*, respectively. A program should neither know, nor care, what kind of device lies behind its input and outputs: disk files, terminals, tape drives, network connections, or even another running program! A program can expect these standard places to be already open and ready to use when it starts up."

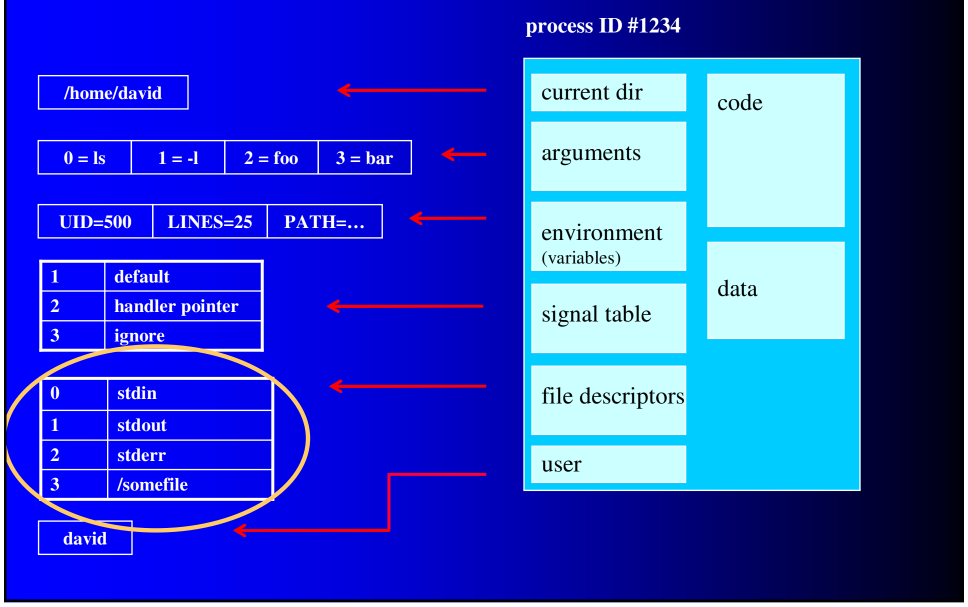
Classic Shell Scripting, Robbins and Beebe, O'Reilly, p. 18

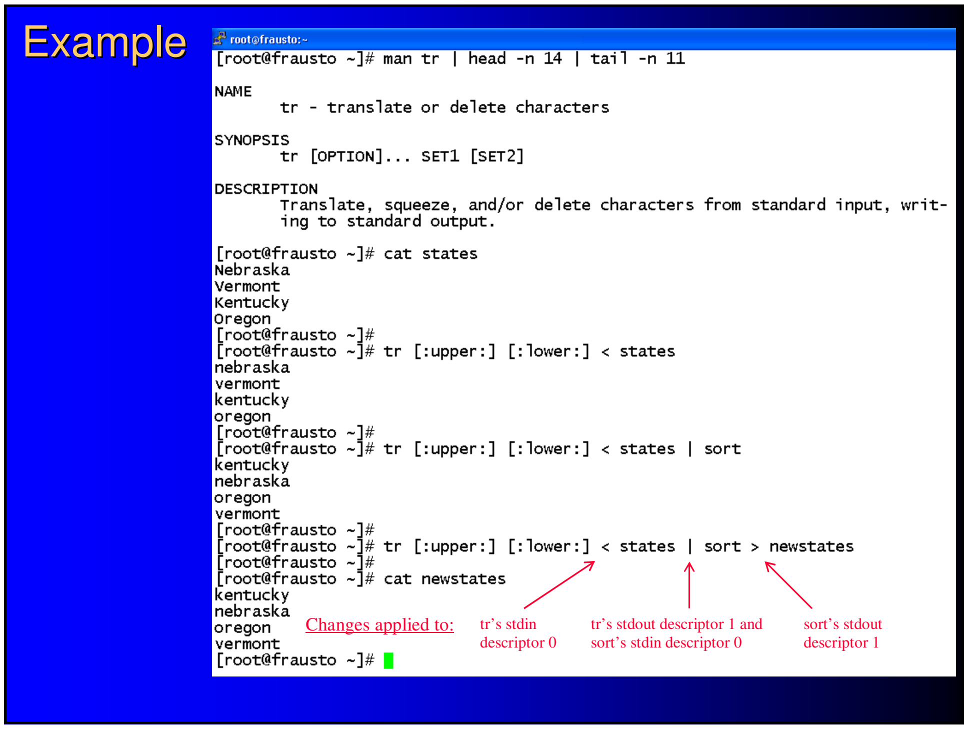


Basic I/O redirection

- pre-arranged place names for data
  - "stdin" names a data source (a.k.a. 0)
  - "stdout" names a data destination (a.k.a. 1)
  - "stderr" another data destination (a.k.a. 2)
- names connected to physical places, by default
  - stdin gets connected to keyboard
  - stdout to monitor
  - stderr to monitor
- reconnecting them with other places is "redirection"
- main operators > < |

I/O descriptors in a unix process

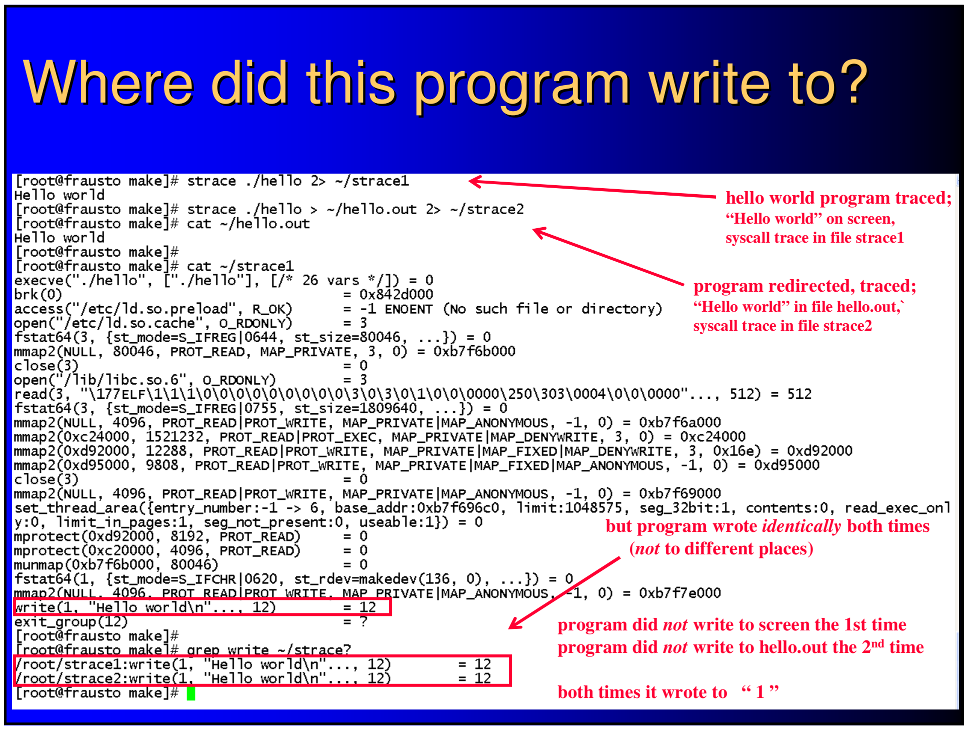




Example

```
[root@frausto ~]# man tr | head -n 14 | tail -n 11

NAME
       tr - translate or delete characters

SYNOPSIS
       tr [OPTION]... SET1 [SET2]

DESCRIPTION
       Translate, squeeze, and/or delete characters from standard input, writ-
       ing to standard output.

[root@frausto ~]# cat states
Nebraska
Vermont
Kentucky
Oregon
[root@frausto ~]#
[root@frausto ~]# tr [:upper:] [:lower:] < states
nebraska
vermont
kentucky
oregon
[root@frausto ~]#
[root@frausto ~]# tr [:upper:] [:lower:] < states | sort
kentucky
nebraska
oregon
vermont
[root@frausto ~]#
[root@frausto ~]# tr [:upper:] [:lower:] < states | sort > newstates
[root@frausto ~]#
[root@frausto ~]# cat newstates
kentucky
nebraska
oregon
vermont
[root@frausto ~]#
```

Changes applied to: tr's stdin descriptor 0 — tr's stdout descriptor 1 and sort's stdin descriptor 0 — sort's stdout descriptor 1



Where did this program write to?

```
[root@frausto make]# strace ./hello 2> ~/strace1
Hello world
[root@frausto make]# strace ./hello > ~/hello.out 2> ~/strace2
[root@frausto make]# cat ~/hello.out
Hello world
[root@frausto make]#
[root@frausto make]# cat ~/strace1
execve("./hello", ["./hello"], [/* 26 vars */]) = 0
brk(0)                                  = 0x842d000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=80046, ...}) = 0
mmap2(NULL, 80046, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f6b000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0000\250\303\0004\0\0\0000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1809640, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f6a000
mmap2(0xc24000, 1521232, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xc24000
mmap2(0xd92000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16e) = 0xd92000
mmap2(0xd95000, 9808, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xd95000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f69000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7f696c0, limit:1048575, seg_32bit:1, contents:0, read_exec_onl
y:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xd92000, 8192, PROT_READ)     = 0
mprotect(0xc20000, 4096, PROT_READ)     = 0
munmap(0xb7f6b000, 80046)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f7e000
write(1, "Hello world\n"..., 12)        = 12
exit_group(12)                          = ?
[root@frausto make]#
[root@frausto make]# grep write ~/strace?
/root/strace1:write(1, "Hello world\n"..., 12)          = 12
/root/strace2:write(1, "Hello world\n"..., 12)          = 12
[root@frausto make]#
```

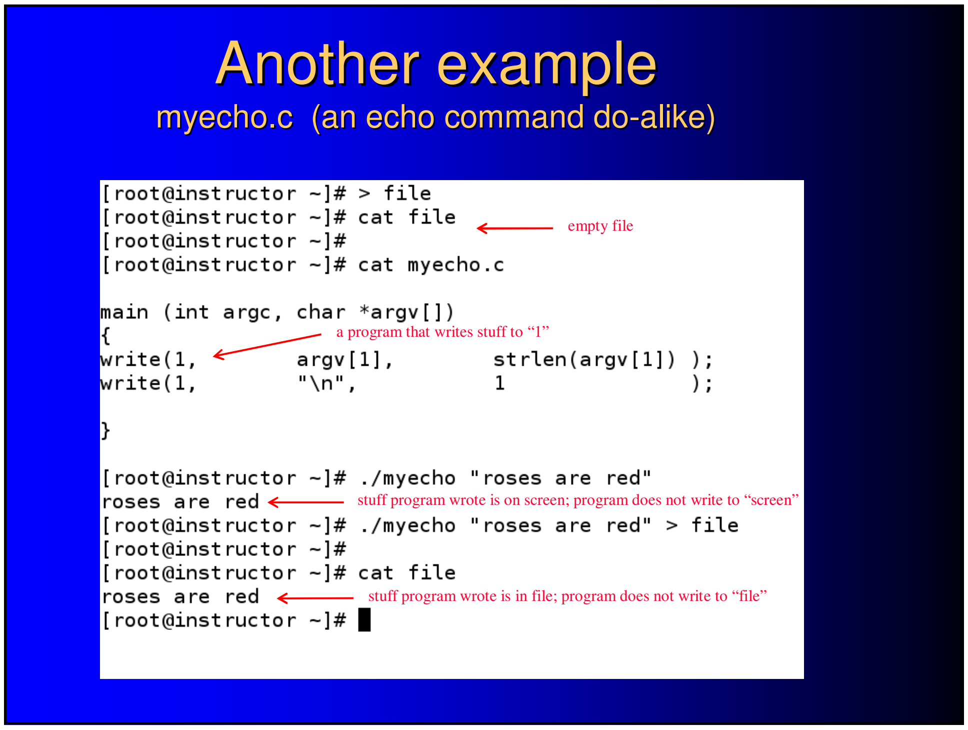
hello world program traced;
"Hello world" on screen,
syscall trace in file strace1

program redirected, traced;
"Hello world" in file hello.out,`
syscall trace in file strace2

but program wrote *identically* both times
(*not* to different places)

program did *not* write to screen the 1st time
program did *not* write to hello.out the 2nd time

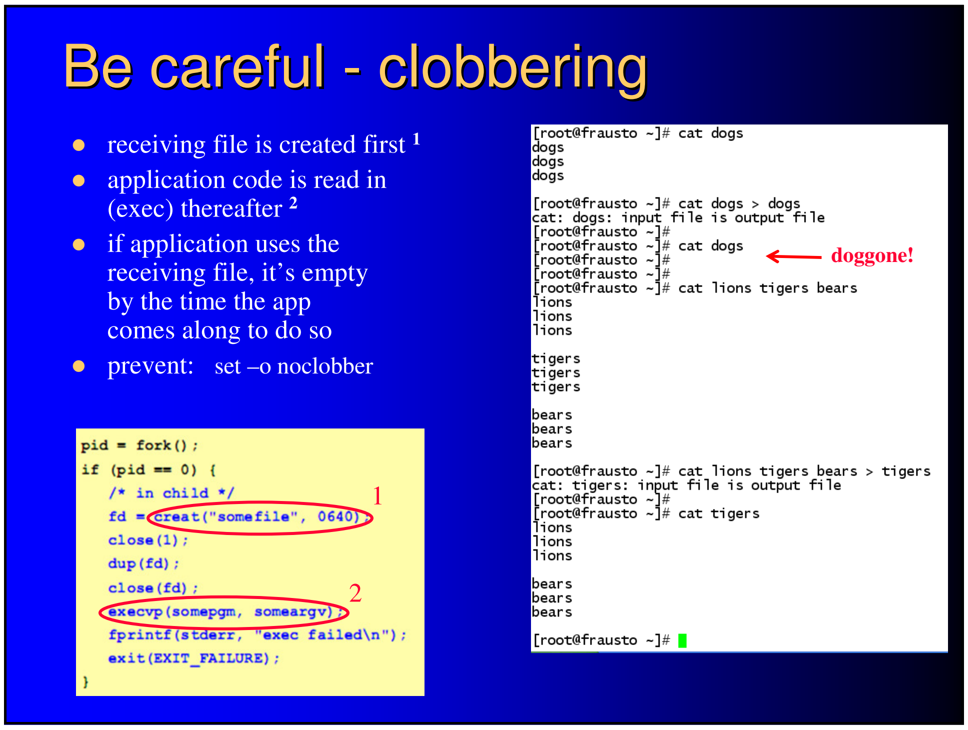
both times it wrote to " 1 "



Another example

myecho.c (an echo command do-alike)

```
[root@instructor ~]# > file
[root@instructor ~]# cat file          empty file
[root@instructor ~]#
[root@instructor ~]# cat myecho.c

main (int argc, char *argv[])
{                     a program that writes stuff to "1"
write(1,        argv[1],        strlen(argv[1]) );
write(1,        "\n",           1               );

}

[root@instructor ~]# ./myecho "roses are red"
roses are red     stuff program wrote is on screen; program does not write to "screen"
[root@instructor ~]# ./myecho "roses are red" > file
[root@instructor ~]#
[root@instructor ~]# cat file
roses are red     stuff program wrote is in file; program does not write to "file"
[root@instructor ~]# █
```



Be careful - clobbering

- receiving file is created first [1]
- application code is read in (exec) thereafter [2]
- if application uses the receiving file, it's empty by the time the app comes along to do so
- prevent: set –o noclobber

```
pid = fork();
if (pid == 0) {
   /* in child */                        1
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);                        2
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
```

```
[root@frausto ~]# cat dogs
dogs
dogs
dogs

[root@frausto ~]# cat dogs > dogs
cat: dogs: input file is output file
[root@frausto ~]#
[root@frausto ~]# cat dogs          doggone!
[root@frausto ~]#
[root@frausto ~]#
[root@frausto ~]# cat lions tigers bears
lions
lions
lions

tigers
tigers
tigers

bears
bears
bears

[root@frausto ~]# cat lions tigers bears > tigers
cat: tigers: input file is output file
[root@frausto ~]#
[root@frausto ~]# cat tigers
lions
lions
lions

bears
bears
bears

[root@frausto ~]# █
```

i/o descriptor manipulation

- preceding examples demonstrate C
- shell also provides syntax functionally similar
- uses “exec” with no command argument to do the job

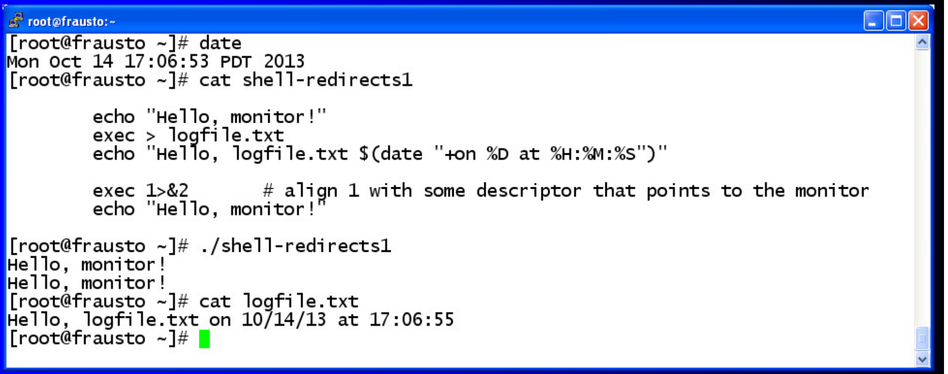
exec [-cl] [-a name] [command [arguments]]

If command is specified, it replaces the shell. No new process is created. ... [BUT!!! Check it out!!] If command is not specified, any redirections take effect in the current shell…

--bash man page

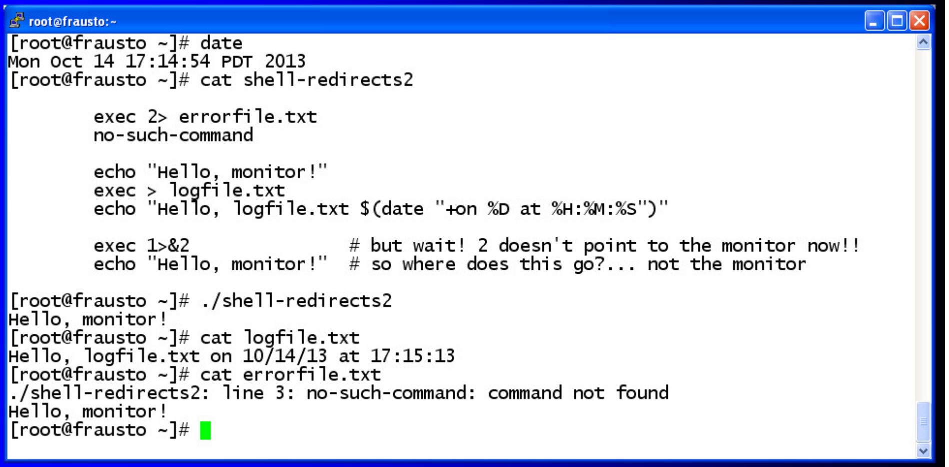
● redirect stdout (i.e., descriptor 1) to file:	exec > filename
● redirect descriptor n to file:	exec n> filename
● redirect descriptor n to m: “make n point, also, to wherever m does at the moment”	exec n>&m
● close descriptor n	n>&-

Redirect output to file, and back



```
[root@frausto ~]# date
Mon Oct 14 17:06:53 PDT 2013
[root@frausto ~]# cat shell-redirects1

        echo "Hello, monitor!"
        exec > logfile.txt
        echo "Hello, logfile.txt $(date "+on %D at %H:%M:%S")"

        exec 1>&2       # align 1 with some descriptor that points to the monitor
        echo "Hello, monitor!"

[root@frausto ~]# ./shell-redirects1
Hello, monitor!
Hello, monitor!
[root@frausto ~]# cat logfile.txt
Hello, logfile.txt on 10/14/13 at 17:06:55
[root@frausto ~]#
```

Redirect output to file, and *whoops!*

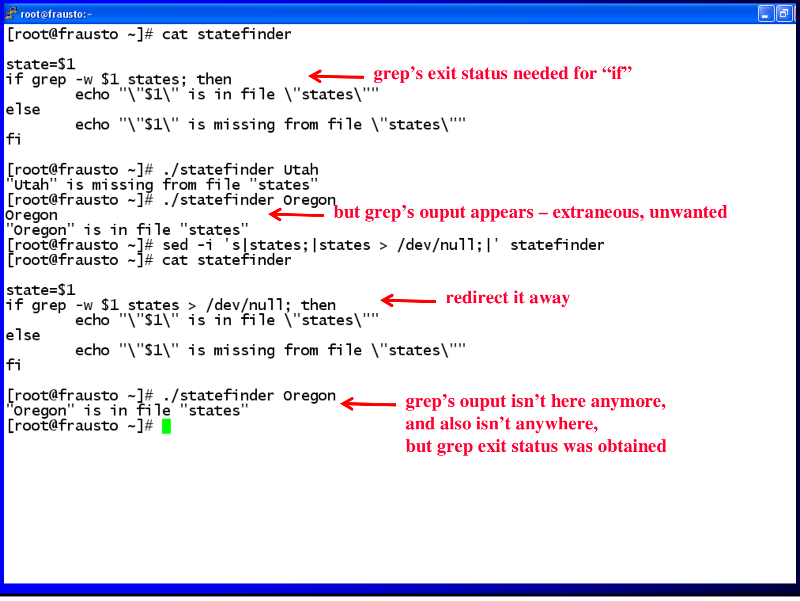


```
[root@frausto ~]# date
Mon Oct 14 17:14:54 PDT 2013
[root@frausto ~]# cat shell-redirects2

        exec 2> errorfile.txt
        no-such-command

        echo "Hello, monitor!"
        exec > logfile.txt
        echo "Hello, logfile.txt $(date "+on %D at %H:%M:%S")"

        exec 1>&2               # but wait! 2 doesn't point to the monitor now!!
        echo "Hello, monitor!"  # so where does this go?... not the monitor

[root@frausto ~]# ./shell-redirects2
Hello, monitor!
[root@frausto ~]# cat logfile.txt
Hello, logfile.txt on 10/14/13 at 17:15:13
[root@frausto ~]# cat errorfile.txt
./shell-redirects2: line 3: no-such-command: command not found
Hello, monitor!
[root@frausto ~]# 
```

Special files: /dev/null

- bit bucket
  - program that writes to it experiences success
  - nothing done in practice with what's written
- allows getting exit status without output

Special files: /dev/null



```
[root@frausto ~]# cat statefinder

state=$1
if grep -w $1 states; then
        echo "\"$1\" is in file \"states\""
else
        echo "\"$1\" is missing from file \"states\""
fi

[root@frausto ~]# ./statefinder Utah
"Utah" is missing from file "states"
[root@frausto ~]# ./statefinder Oregon
Oregon
"Oregon" is in file "states"
[root@frausto ~]# sed -i 's|states;|states > /dev/null;|' statefinder
[root@frausto ~]# cat statefinder

state=$1
if grep -w $1 states > /dev/null; then
        echo "\"$1\" is in file \"states\""
else
        echo "\"$1\" is missing from file \"states\""
fi

[root@frausto ~]# ./statefinder Oregon
"Oregon" is in file "states"
[root@frausto ~]#
```

grep’s exit status needed for “if”

but grep’s ouput appears – extraneous, unwanted

redirect it away

grep’s ouput isn’t here anymore,
and also isn’t anywhere,
but grep exit status was obtained