<https://www.tldp.org/LDP/abs/html/io-redirection.html#FDREF>

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Chapter 20. I/O Redirection**  **Table of Contents**  20.1. [Using *exec*](https://www.tldp.org/LDP/abs/html/x17974.html)  20.2. [Redirecting Code Blocks](https://www.tldp.org/LDP/abs/html/redircb.html)  20.3. [Applications](https://www.tldp.org/LDP/abs/html/redirapps.html)  There are always three default *files* [[1]](https://www.tldp.org/LDP/abs/html/io-redirection.html" \l "FTN.AEN17884) open, stdin (the keyboard), stdout (the screen), and stderr (error messages output to the screen). These, and any other open files, can be redirected. Redirection simply means capturing output from a file, command, program, script, or even code block within a script (see [Example 3-1](https://www.tldp.org/LDP/abs/html/special-chars.html#EX8) and [Example 3-2](https://www.tldp.org/LDP/abs/html/special-chars.html#RPMCHECK)) and sending it as input to another file, command, program, or script.  Each open file gets assigned a file descriptor. [[2]](https://www.tldp.org/LDP/abs/html/io-redirection.html" \l "FTN.AEN17894) The file descriptors for stdin, stdout, and stderr are 0, 1, and 2, respectively. For opening additional files, there remain descriptors 3 to 9. It is sometimes useful to assign one of these additional file descriptors to stdin, stdout, or stderr as a temporary duplicate link. [[3]](https://www.tldp.org/LDP/abs/html/io-redirection.html" \l "FTN.AEN17906) This simplifies restoration to normal after complex redirection and reshuffling (see [Example 20-1](https://www.tldp.org/LDP/abs/html/x17974.html#REDIR1)).   |  | | --- | | COMMAND\_OUTPUT >  # Redirect stdout to a file.  # Creates the file if not present, otherwise overwrites it.  ls -lR > dir-tree.list  # Creates a file containing a listing of the directory tree.  : > filename  # The > truncates file "filename" to zero length.  # If file not present, creates zero-length file (same effect as 'touch').  # The : serves as a dummy placeholder, producing no output.  > filename  # The > truncates file "filename" to zero length.  # If file not present, creates zero-length file (same effect as 'touch').  # (Same result as ": >", above, but this does not work with some shells.)  COMMAND\_OUTPUT >>  # Redirect stdout to a file.  # Creates the file if not present, otherwise appends to it.  # Single-line redirection commands (affect only the line they are on):  # --------------------------------------------------------------------  1>filename  # Redirect stdout to file "filename."  1>>filename  # Redirect and append stdout to file "filename."  2>filename  # Redirect stderr to file "filename."  2>>filename  # Redirect and append stderr to file "filename."  &>filename  # Redirect both stdout and stderr to file "filename."  # This operator is now functional, as of Bash 4, final release.  M>N  # "M" is a file descriptor, which defaults to 1, if not explicitly set.  # "N" is a filename.  # File descriptor "M" is redirect to file "N."  M>&N  # "M" is a file descriptor, which defaults to 1, if not set.  # "N" is another file descriptor.  #==============================================================================  # Redirecting stdout, one line at a time.  LOGFILE=script.log  echo "This statement is sent to the log file, \"$LOGFILE\"." 1>$LOGFILE  echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE  echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE  echo "This statement is echoed to stdout, and will not appear in \"$LOGFILE\"."  # These redirection commands automatically "reset" after each line.  # Redirecting stderr, one line at a time.  ERRORFILE=script.errors  bad\_command1 2>$ERRORFILE # Error message sent to $ERRORFILE.  bad\_command2 2>>$ERRORFILE # Error message appended to $ERRORFILE.  bad\_command3 # Error message echoed to stderr,  #+ and does not appear in $ERRORFILE.  # These redirection commands also automatically "reset" after each line.  #======================================================================= |      |  | | --- | | 2>&1  # Redirects stderr to stdout.  # Error messages get sent to same place as standard output.  >>filename 2>&1  bad\_command >>filename 2>&1  # Appends both stdout and stderr to the file "filename" ...  2>&1 | [command(s)]  bad\_command 2>&1 | awk '{print $5}' # found  # Sends stderr through a pipe.  # |& was added to Bash 4 as an abbreviation for 2>&1 |.  i>&j  # Redirects file descriptor *i* to *j*.  # All output of file pointed to by *i* gets sent to file pointed to by *j*.  >&j  # Redirects, by default, file descriptor *1* (stdout) to *j*.  # All stdout gets sent to file pointed to by *j*. |      |  | | --- | | 0< FILENAME  < FILENAME  # Accept input from a file.  # Companion command to ">", and often used in combination with it.  #  # grep search-word <filename  [j]<>filename  # Open file "filename" for reading and writing,  #+ and assign file descriptor "j" to it.  # If "filename" does not exist, create it.  # If file descriptor "j" is not specified, default to fd 0, stdin.  #  # An application of this is writing at a specified place in a file.  echo 1234567890 > File # Write string to "File".  exec 3<> File # Open "File" and assign fd 3 to it.  read -n 4 <&3 # Read only 4 characters.  echo -n . >&3 # Write a decimal point there.  exec 3>&- # Close fd 3.  cat File # ==> 1234.67890  # Random access, by golly.  |  # Pipe.  # General purpose process and command chaining tool.  # Similar to ">", but more general in effect.  # Useful for chaining commands, scripts, files, and programs together.  cat \*.txt | sort | uniq > result-file  # Sorts the output of all the .txt files and deletes duplicate lines,  # finally saves results to "result-file". |   Multiple instances of input and output redirection and/or pipes can be combined in a single command line.   |  | | --- | | command < input-file > output-file  # Or the equivalent:  < input-file command > output-file # Although this is non-standard.  command1 | command2 | command3 > output-file |   See [Example 16-31](https://www.tldp.org/LDP/abs/html/filearchiv.html#DERPM) and [Example A-14](https://www.tldp.org/LDP/abs/html/contributed-scripts.html#FIFO).  Multiple output streams may be redirected to one file.   |  | | --- | | ls -yz >> command.log 2>&1  # Capture result of illegal options "yz" in file "command.log."  # Because stderr is redirected to the file,  #+ any error messages will also be there.  # Note, however, that the following does \*not\* give the same result.  ls -yz 2>&1 >> command.log  # Outputs an error message, but does not write to file.  # More precisely, the command output (in this case, null)  #+ writes to the file, but the error message goes only to stdout.  # If redirecting both stdout and stderr,  #+ the order of the commands makes a difference. |   **Closing File Descriptors**  n<&-  Close input file descriptor *n*.  0<&-, <&-  Close stdin.  n>&-  Close output file descriptor *n*.  1>&-, >&-  Close stdout.  Child processes inherit open file descriptors. This is why pipes work. To prevent an fd from being inherited, close it.   |  | | --- | | # Redirecting only stderr to a pipe.  exec 3>&1 # Save current "value" of stdout.  ls -l 2>&1 >&3 3>&- | grep bad 3>&- # Close fd 3 for 'grep' (but not 'ls').  # ^^^^ ^^^^  exec 3>&- # Now close it for the remainder of the script.  # Thanks, S.C. | |

# Example 20

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 20.1. Using *exec* An **exec <filename** command redirects stdin to a file. From that point on, all stdin comes from that file, rather than its normal source (usually keyboard input). This provides a method of reading a file line by line and possibly parsing each line of input using [sed](https://www.tldp.org/LDP/abs/html/sedawk.html" \l "SEDREF) and/or [awk](https://www.tldp.org/LDP/abs/html/awk.html" \l "AWKREF).  **Example 20-1. Redirecting stdin using *exec***   |  | | --- | | #!/bin/bash  # Redirecting stdin using 'exec'.  exec 6<&0 # Link file descriptor #6 with stdin.  # Saves stdin.  exec < data-file # stdin replaced by file "data-file"  read a1 # Reads first line of file "data-file".  read a2 # Reads second line of file "data-file."  echo  echo "Following lines read from file."  echo "-------------------------------"  echo $a1  echo $a2  echo; echo; echo  exec 0<&6 6<&-  # Now restore stdin from fd #6, where it had been saved,  #+ and close fd #6 ( 6<&- ) to free it for other processes to use.  #  # <&6 6<&- also works.  echo -n "Enter data "  read b1 # Now "read" functions as expected, reading from normal stdin.  echo "Input read from stdin."  echo "----------------------"  echo "b1 = $b1"  echo  exit 0 |   Similarly, an **exec >filename** command redirects stdout to a designated file. This sends all command output that would normally go to stdout to that file.   |  |  | | --- | --- | | Important | **exec N > filename** affects the entire script or *current shell*. Redirection in the [PID](https://www.tldp.org/LDP/abs/html/special-chars.html#PROCESSIDREF) of the script or shell from that point on has changed. However . . .  **N > filename** affects only the newly-forked process, not the entire script or shell.  Thank you, Ahmed Darwish, for pointing this out. |   **Example 20-2. Redirecting stdout using *exec***   |  | | --- | | #!/bin/bash  # reassign-stdout.sh  LOGFILE=logfile.txt  exec 6>&1 # Link file descriptor #6 with stdout.  # Saves stdout.  exec > $LOGFILE # stdout replaced with file "logfile.txt".  # ----------------------------------------------------------- #  # All output from commands in this block sent to file $LOGFILE.  echo -n "Logfile: "  date  echo "-------------------------------------"  echo  echo "Output of \"ls -al\" command"  echo  ls -al  echo; echo  echo "Output of \"df\" command"  echo  df  # ----------------------------------------------------------- #  exec 1>&6 6>&- # Restore stdout and close file descriptor #6.  echo  echo "== stdout now restored to default == "  echo  ls -al  echo  exit 0 |   **Example 20-3. Redirecting both stdin and stdout in the same script with *exec***   |  | | --- | | #!/bin/bash  # upperconv.sh  # Converts a specified input file to uppercase.  E\_FILE\_ACCESS=70  E\_WRONG\_ARGS=71  if [ ! -r "$1" ] # Is specified input file readable?  then  echo "Can't read from input file!"  echo "Usage: $0 input-file output-file"  exit $E\_FILE\_ACCESS  fi # Will exit with same error  #+ even if input file ($1) not specified (why?).  if [ -z "$2" ]  then  echo "Need to specify output file."  echo "Usage: $0 input-file output-file"  exit $E\_WRONG\_ARGS  fi  exec 4<&0  exec < $1 # Will read from input file.  exec 7>&1  exec > $2 # Will write to output file.  # Assumes output file writable (add check?).  # -----------------------------------------------  cat - | tr a-z A-Z # Uppercase conversion.  # ^^^^^ # Reads from stdin.  # ^^^^^^^^^^ # Writes to stdout.  # However, both stdin and stdout were redirected.  # Note that the 'cat' can be omitted.  # -----------------------------------------------  exec 1>&7 7>&- # Restore stout.  exec 0<&4 4<&- # Restore stdin.  # After restoration, the following line prints to stdout as expected.  echo "File \"$1\" written to \"$2\" as uppercase conversion."  exit 0 |   I/O redirection is a clever way of avoiding the dreaded [inaccessible variables within a subshell](https://www.tldp.org/LDP/abs/html/subshells.html#PARVIS) problem.  **Example 20-4. Avoiding a subshell**   |  | | --- | | #!/bin/bash  # avoid-subshell.sh  # Suggested by Matthew Walker.  Lines=0  echo  cat myfile.txt | while read line;  do {  echo $line  (( Lines++ )); # Incremented values of this variable  #+ inaccessible outside loop.  # Subshell problem.  }  done  echo "Number of lines read = $Lines" # 0  # Wrong!  echo "------------------------"  exec 3<> myfile.txt  while read line <&3  do {  echo "$line"  (( Lines++ )); # Incremented values of this variable  #+ accessible outside loop.  # No subshell, no problem.  }  done  exec 3>&-  echo "Number of lines read = $Lines" # 8  echo  exit 0  # Lines below not seen by script.  $ cat myfile.txt  Line 1.  Line 2.  Line 3.  Line 4.  Line 5.  Line 6.  Line 7.  Line 8. | |