

Linux I/O Redirection Operators

LinuXpert Systems

No.5, Dr. Ambedkar St, Rathinapuram,

Nanganallur, Chennai – 600 061.

Phone: 98841 65649, 90808 01382

E-mail: baskar@linuxpert.in

TABLE OF CONTENTS

1. Understanding Linux Standard I/O 1
2. What Is Redirection ? 3
3. The Three Basic Redirection Operators 3
4. Operator > — Output Redirection 4
5. How > Works 4
6. Important Property of > 5
7. Creating a New File with > 6
8. Using > with **echo** 6
9. Using > with **date** 7
10. Using > with **pwd** 7
11. Using > with **who** 8
12. Operator >> — Append Redirection 8
13. Example of >> 8
14. Difference Between > and >> 9
15. >> Creates a File If It Doesn't Exist 10
16. Practical Example of >> 10
17. Using **date** with >> 11
18. Using **ls** with >> 11
19. Operator < — Input Redirection 11
20. Example of < with **cat** 12
21. Why Use < ? 12
22. Example with **wc** 13
23. Comparing **wc** filename and **wc** < filename 13
24. Another Example with **sort** 14
25. Combining Input and Output Redirection 14
26. Example 15

27. Combining >> with < 16
28. Redirection and File Descriptors 16
29. Standard Error Is Different 17
30. Redirecting Errors 17
31. Redirecting Both Output and Error 18
32. Redirection Does Not Mean Piping 18
33. Redirection with **grep** 19
34. Redirection with **df** 19
35. Redirection with **ps** 19
36. Redirection with **history** 20
37. Using Redirection in Shell Scripts 20
38. Practical System Administration Example 20
39. Redirection with Configuration Information 21
40. Important Difference: > vs >> 21
41. Important Difference: > vs < 22
42. Important Difference: >> vs > 22
43. Common Mistakes 23
44. Redirection Order Matters 24
45. Using Redirection with Logs 24
46. Using Redirection for Reports 24
47. Redirection with **cat** 25
48. Input Redirection with **cat** 26
49. Redirection in Automation 26
50. Summary 27

Linux I/O Redirection Operators

Linux provides **I/O (Input/Output) redirection** to control where commands receive their input from and where they send their output. Instead of always displaying output on the terminal, we can redirect it to files, or use files as input.

The **three fundamental redirection operators** commonly taught in Linux are:

1. `>` — Redirect standard output and **overwrite** a file
2. `>>` — Redirect standard output and **append** to a file
3. `<` — Redirect standard input **from a file**

These operators are extremely important in Linux because they allow commands to be combined with files and other commands efficiently.

---

1. Understanding Linux Standard I/O

Before learning redirection operators, it is important to understand the three standard streams used by Linux programs.

Every Linux process normally has three standard file descriptors:

File Descriptor	Name	Purpose
`0`	Standard Input (`stdin`)	Input to a command
`1`	Standard Output (`stdout`)	Normal output from a command
`2`	Standard Error (`stderr`)	Error messages from a command

Standard Input

Standard input is normally the keyboard.

For example:

```
cat
```

When you type:

```
Hello Linux
```

and press Enter, `cat` receives that text through **standard input**.

Standard Output

Standard output normally goes to the terminal.

For example:

```
ls
```

might display:

```
Documents
Downloads
Music
Pictures
Videos
```

This is standard output.

Standard Error

Errors are normally displayed on the terminal as well, but they use a separate stream.
For example:

```
ls /does-not-exist
```

may produce:

```
ls: cannot access '/does-not-exist': No such file or directory
```

This is **standard error**, file descriptor 2.

2. What Is Redirection?

Redirection means changing the normal destination or source of a command's input/output.

Normally:

```
Keyboard
   |
   v
Command
   |
   v
Terminal
```

With output redirection:

```
Command
   |
   v
File
```

With input redirection:

```
File
   |
   v
Command
```

Therefore, redirection allows Linux commands to work with files without requiring the user to manually type or copy information.

---

3. The Three Basic Redirection Operators

The three operators are:

```
>     Output redirection — overwrite
>>    Output redirection — append
<     Input redirection
```

Let's examine each one in detail.

4. Operator > — Output Redirection

The **>** operator redirects **standard output** from the terminal into a file.

Basic syntax

```
command > filename
```

For example:

```
ls > files.txt
```

Instead of displaying the output of `ls` on the screen, Linux writes it to:

```
files.txt
```

You can then see the contents using:

```
cat files.txt
```

---

5. How > Works

Suppose the directory contains:

```
file1.txt
file2.txt
file3.txt
```

Running:

```
ls
```

displays:

```
file1.txt
file2.txt
file3.txt
```

But:

```
ls > files.txt
```

does not display the directory listing.

Instead, it creates:

```
files.txt
```

containing:

```
file1.txt
file2.txt
file3.txt
```

The data flow becomes:

```
ls
 |
 | standard output
 v
files.txt
```

---

6. Important Property of >

The most important feature of > is:

> **If the destination file already exists, its previous contents are overwritten.**

For example:

```
echo "First line" > test.txt
```

Now:

```
cat test.txt
```

produces:

```
First line
```

Now execute:

```
echo "Second line" > test.txt
```

The old contents are replaced.

The file now contains:

```
Second line
```

It does **not** contain:

```
First line
Second line
```

because > overwrites the existing file.

7. Creating a New File with >

The > operator can also create a new file.

For example:

```
echo "Linux Operating System" > linux.txt
```

If `linux.txt` does not exist, Linux creates it.

Check it:

```
cat linux.txt
```

Output:

```
Linux Operating System
```

Therefore, > can perform two operations:

1. Create a new file
2. Overwrite an existing file

---

8. Using > with `echo`

The `echo` command is frequently used with output redirection.

For example:

```
echo "Hello" > hello.txt
```

The file contains:

```
Hello
```

Another example:

```
echo "Welcome to Linux" > welcome.txt
```

The file contains:

```
Welcome to Linux
```

9. Using > with `date`

You can store the current date in a file:

```
date > today.txt
```

Check:

```
cat today.txt
```

Example output:

```
Mon Aug 24 08:30:00 IST 2026
```

If you run the command again:

```
date > today.txt
```

the previous date is replaced.

---

10. Using > with `pwd`

The `pwd` command displays the current working directory.

Normally:

```
pwd
```

might produce:

```
/home/bharath
```

Using redirection:

```
pwd > location.txt
```

Now:

```
cat location.txt
```

produces:

```
/home/bharath
```

11. Using > with who

You can save currently logged-in users:

```
who > users.txt
```

Then:

```
cat users.txt
```

This is useful when you want to preserve command output for later examination.

---

12. Operator >> — Append Redirection

The >> operator also redirects standard output to a file.

However, there is one major difference:

> **>> adds new output to the end of an existing file instead of replacing its contents.**

Syntax:

```
command >> filename
```

---

13. Example of >>

First:

```
echo "First line" > test.txt
```

The file contains:

```
First line
```

Now:

```
echo "Second line" >> test.txt
```

The file becomes:

```
First line
Second line
```

The previous content is preserved.

14. Difference Between > and >>

This is one of the most important concepts in Linux redirection.

Using >

```
echo "Apple" > fruits.txt
```

Contents:

```
Apple
```

Then:

```
echo "Banana" > fruits.txt
```

Contents become:

```
Banana
```

`Apple` is lost.

Using >>

```
echo "Apple" > fruits.txt
```

Contents:

```
Apple
```

Then:

```
echo "Banana" >> fruits.txt
```

Contents become:

```
Apple
Banana
```

So:

```
>   = overwrite
>>  = append
```

15. >> Creates a File If It Doesn't Exist

The >> operator can also create a new file.

For example:

```
echo "Linux" >> linux.txt
```

If `linux.txt` does not exist, Linux creates it.

If it already exists, the text is added to the end.

Therefore:

```
>   → create or overwrite
>>  → create or append
```

---

16. Practical Example of >>

Suppose you are maintaining a daily log.

You could use:

```
echo "System started" >> system.log
```

Later:

```
echo "User logged in" >> system.log
```

Later:

```
echo "Backup completed" >> system.log
```

The file might contain:

```
System started
User logged in
Backup completed
```

This is why >> is commonly used for **log files**.

17. Using `date` with >>

You can create a simple log:

```
date >> dates.log
```

Run it multiple times:

```
date >> dates.log
date >> dates.log
date >> dates.log
```

The file will contain multiple date entries.

This is useful for recording events.

---

18. Using `ls` with >>

For example:

```
ls >> directory.log
```

The directory listing is appended to the file.

If you run the command again, another listing is added.

---

19. Operator < — Input Redirection

The **<** operator redirects **standard input** from a file.

Syntax:

```
command < filename
```

Normally a command may receive input from the keyboard.

With **<**, the command receives input from a file instead.

The data flow becomes:

```
File
 |
 | standard input
 v
Command
```

20. Example of < with `cat`

Suppose:

```
cat > names.txt
```

and you enter:

```
Bharath
Ravi
Kumar
```

After pressing `Ctrl+D`, the file contains:

```
Bharath
Ravi
Kumar
```

Now:

```
cat < names.txt
```

The `cat` command receives its input from `names.txt`.

The result is:

```
Bharath
Ravi
Kumar
```

---

21. Why Use < ?

For many commands, < may not be necessary because the command can accept a filename directly.

For example:

```
cat names.txt
```

and:

```
cat < names.txt
```

usually produce the same visible result.

However, < becomes particularly useful when working with commands that expect input from **standard input** rather than a filename argument.

22. Example with `wc`

The `wc` command counts lines, words, and characters.

Suppose:

```
names.txt
```

contains:

```
Bharath
Ravi
Kumar
```

You can run:

```
wc < names.txt
```

The file becomes standard input to `wc`.

You may get output similar to:

```
3 3 20
```

The values represent counts of:

```
lines words bytes
```

---

23. Comparing `wc filename` and `wc < filename`

These two commands are related:

```
wc names.txt
```

and:

```
wc < names.txt
```

The first tells `wc` to open the file named `names.txt`.

The second redirects the file into standard input.

This can affect the way the filename is displayed in the output.

For example, `wc names.txt` may show:

```
3 3 20 names.txt
```

while:

```
wc < names.txt
```

may show:

```
3 3 20
```

This distinction becomes important when constructing shell scripts and command pipelines.

---

24. Another Example with `sort`

Suppose `names.txt` contains:

```
Ravi
Bharath
Kumar
Arun
```

You can use:

```
sort < names.txt
```

Output:

```
Arun
Bharath
Kumar
Ravi
```

The input comes from the file.

---

25. Combining Input and Output Redirection

One of the powerful features of Linux is that redirection operators can be combined.

For example:

```
sort < names.txt > sorted.txt
```

This means:

1. Read input from `names.txt`
2. Sort it
3. Write the result to `sorted.txt`

Data flow:

```
names.txt
    |
    v
  sort
    |
    v
sorted.txt
```

This is a very important Linux concept.

---

26. Example

Suppose `names.txt` contains:

```
Zebra
Apple
Mango
Banana
```

Run:

```
sort < names.txt > sorted.txt
```

Then:

```
cat sorted.txt
```

Output:

```
Apple
Banana
Mango
Zebra
```

The original file remains unchanged.

27. Combining >> with <

You can also combine input and append redirection.

For example:

```
sort < names.txt >> sorted.log
```

This means:

```
names.txt
   |
   v
 sort
   |
   v
sorted.log
```

The result is appended to `sorted.log`.

---

28. Redirection and File Descriptors

Linux internally associates standard streams with file descriptor numbers.

The important ones are:

```
0 = stdin
1 = stdout
2 = stderr
```

Therefore:

```
command > output.txt
```

is effectively associated with:

```
command 1> output.txt
```

Similarly:

```
command >> output.txt
```

is equivalent to:

```
command 1>> output.txt
```

The `1` means standard output.

29. Standard Error Is Different

The three basic operators discussed here primarily concern standard input and standard output.

Remember:

```
0 → stdin
1 → stdout
2 → stderr
```

For example:

```
ls /abc > output.txt
```

If `/abc` does not exist, the error message may still appear on the terminal.

Why?

Because:

```
> output.txt
```

redirects **stdout**, not stderr.

The error stream is separate.

---

30. Redirecting Errors

Although the basic three operators are:

```
>
>>
<
```

Linux also allows file-descriptor-specific redirection.

For example:

```
command 2> error.txt
```

means:

> Redirect standard error to `error.txt`.

Example:

```
ls /does-not-exist 2> error.txt
```

Now the error is stored in:

```
error.txt
```

31. Redirecting Both Output and Error

A command may produce both normal output and errors.

For example:

```
command > output.txt 2> error.txt
```

This separates them:

```
stdout → output.txt
stderr → error.txt
```

This is very useful when troubleshooting Linux commands.

---

32. Redirection Does Not Mean Piping

Redirection and piping are related but different.

Redirection

```
ls > files.txt
```

The output goes to a file.

```
ls → file
```

Pipe

```
ls | sort
```

The output of one command becomes the input of another command.

```
ls → sort
```

The pipe operator is:

```
|
```

It is not one of the three basic redirection operators.

33. Redirection with `grep`

Suppose you want to save matching lines:

```
grep "error" system.log > errors.txt
```

The matching lines are stored in:

```
errors.txt
```

If you want to keep adding results:

```
grep "error" system.log >> errors.txt
```

---

34. Redirection with `df`

You can save disk-space information:

```
df -h > disk_usage.txt
```

Then:

```
cat disk_usage.txt
```

This is useful for creating reports.

---

35. Redirection with `ps`

You can save running-process information:

```
ps aux > processes.txt
```

Then examine it:

```
less processes.txt
```

36. Redirection with `history`

You can save shell history:

```
history > command_history.txt
```

This creates a file containing previously executed commands.

You can later search it:

```
grep "koha" command_history.txt
```

---

37. Using Redirection in Shell Scripts

Redirection is extremely important in shell scripting.

For example:

```
#!/bin/bash

echo "Backup started" > backup.log
echo "Backup completed" >> backup.log
```

After execution, `backup.log` contains:

```
Backup started
Backup completed
```

This allows scripts to create logs automatically.

---

38. Practical System Administration Example

Suppose you want to save memory information:

```
free -h > memory.txt
```

You can later inspect it:

```
cat memory.txt
```

Or append another report:

```
free -h >> memory.txt
```

This creates a simple history of memory measurements.

39. Redirection with Configuration Information

You might want to save information about your Linux system:

```
uname -a > system_info.txt
```

Then append:

```
df -h >> system_info.txt
free -h >> system_info.txt
```

The file now contains information from multiple commands.

---

40. Important Difference: > vs >>

Feature	>	>>
Purpose	Output redirection	Output append
Existing file	Overwritten	Preserved
New file	Created	Created
Existing contents	Removed	Kept
New output	Written	Added at end
Common use	New reports	Logs

Example

```
echo A > file.txt
echo B > file.txt
```

Result:

```
B
```

But:

```
echo A > file.txt
echo B >> file.txt
```

Result:

```
A
B
```

41. Important Difference: > vs <

These operators work in opposite directions.

>

Output goes **from command to file**:

```
Command → File
```

Example:

```
ls > files.txt
```

<

Input goes **from file to command**:

```
File → Command
```

Example:

```
sort < names.txt
```

Therefore:

```
>  = output direction
<  = input direction
```

---

42. Important Difference: >> vs >

Think about a notebook.

> is similar to saying:

> "Erase the old contents and write this new information."

>> is similar to saying:

> "Keep everything already written and add this information at the bottom."

This simple analogy makes the difference easy to remember.

43. Common Mistakes

Mistake 1: Accidentally overwriting a file

Suppose:

`important.txt`

contains important information.

Running:

```
echo "new information" > important.txt
```

will replace the contents.

If you intended to add information, use:

```
echo "new information" >> important.txt
```

Mistake 2: Expecting > to redirect errors

This:

```
ls /wrong > output.txt
```

does not necessarily put the error into `output.txt`.

That's because the error is sent through file descriptor `2`.

Use:

```
ls /wrong 2> error.txt
```

for error output.

Mistake 3: Confusing < and >

Remember:

```
< : input comes from the file
> : output goes to the file
```

44. Redirection Order Matters

When using multiple redirections, their order can matter.

For example:

```
command > output.txt 2>&1
```

means redirect stdout to `output.txt`, then redirect stderr to wherever stdout currently goes.

This results in both streams going to the same file.

The important concept is that shell redirections are processed from **left to right**.

---

45. Using Redirection with Logs

One of the most common practical applications is logging.

For example:

```
./backup.sh >> backup.log
```

Every execution adds output to the log instead of destroying previous information.

This is much safer for repeated operations.

For example:

```
echo "Backup started" >> backup.log
date >> backup.log
echo "Backup finished" >> backup.log
```

---

46. Using Redirection for Reports

Suppose you need a system report:

```
echo "SYSTEM REPORT" > report.txt
uname -a >> report.txt
df -h >> report.txt
free -h >> report.txt
who >> report.txt
```

The first command creates the report.

The remaining commands append information.

The final file contains information from all commands.

47. Redirection with `cat`

`cat` is particularly useful for demonstrating redirection.

Create a file

```
cat > example.txt
```

Type:

```
Linux
Unix
Ubuntu
```

Press:

```
Ctrl+D
```

to signal end-of-input.

The text is stored in `example.txt`.

Append to the file

```
cat >> example.txt
```

Enter:

```
Debian
Fedora
```

Press:

```
Ctrl+D
```

Now the file contains:

```
Linux
Unix
Ubuntu
Debian
Fedora
```

48. Input Redirection with `cat`

You can read the file through standard input:

```
cat < example.txt
```

This produces:

```
Linux
Unix
Ubuntu
Debian
Fedora
```

The data flow is:

```
example.txt
     |
     v
   stdin
     |
     v
    cat
     |
     v
   stdout
     |
     v
  terminal
```

---

49. Redirection in Automation

Redirection allows Linux commands to operate without continuous interaction.

For example:

```
date >> backup.log
tar -czf backup.tar.gz /home/user/Documents >> backup.log 2>&1
```

A script can run these commands automatically and store the results.

This is especially useful with:

- Cron jobs
- Backup scripts
- Monitoring scripts
- Installation scripts
- System administration
- Log collection

---

50. Summary

Linux provides several mechanisms for controlling command input and output. The three fundamental redirection operators are:

1. >

Redirects standard output to a file.

```
command > file
```

If the file exists, it is **overwritten**.

---

2. >>

Redirects standard output to the end of a file.

```
command >> file
```

Existing contents are **preserved** and new output is appended.

---

3. <

Redirects standard input from a file.

```
command < file
```

The command receives its input from the specified file instead of the keyboard.

Quick Revision Table

Operator	Meaning	Example	Effect
`>`	Output redirection	`ls > files.txt`	Create/overwrite
`>>`	Append output	`ls >> files.txt`	Create/append
`<`	Input redirection	`sort < names.txt`	File → command

Easy way to remember

```
>     Command ──────► File

>>    Command ──────► File (append)

<     File ─────────► Command
```

The three operators form the foundation of Linux command-line I/O. Once you understand them, concepts such as **file descriptors, error redirection, pipes (|), command chaining, shell scripting, logging, and automation** become much easier to understand.