**Process Substitution**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/gxDGoRA3JkD#how-important-is-this-lesson)

I spent years reading and writing bash before I understood this concept, so this lesson can be skipped. However, since I learned about **process substitution**, I use it on the command line almost every day, so I recommend you learn it at some point.

### Simple Process Substitution [#](https://www.educative.io/courses/master-the-bash-shell/gxDGoRA3JkD#simple-process-substitution)

Type this in to set files up for this lesson:

1

2

3

4

5

6

mkdir a

mkdir b

touch a/1 a/2  # Creates files 1 and 2 in folder a

touch b/2 b/3  # Creates files 2 and 3 in folder b

ls a

ls b





Type the above code into the terminal in this lesson.

You’ve created two folders with slightly different contents.

Now let’s say that you want to diff the output of ls a and ls b (a trivial but usefully simple example here). How would you do it?

Note: if you are not familiar with the diff command, you can find an introduction to it [here](https://en.wikipedia.org/wiki/Diff).

You might do it like this:

1

2

3

4

ls a > aout      # List the files in the folder 'a' and redirect output to the 'aout' file

ls b > bout      # List the files in the folder 'b' and redirect output to the 'bout' file

diff aout bout   # Diff the two outputs

rm aout bout     # Clean up the temporary output files





Type the above code into the terminal in this lesson.

That works, and there’s nothing wrong with it, but typing all that out and cleaning up the files is a bit cumbersome. There’s a much neater way that exposes a very useful technique.

Type this in:

1

diff <(ls a) <(ls b)





Type the above code into the terminal in this lesson.

That’s neater, isn’t it?

So what’s going on?

### The <() operator [#](https://www.educative.io/courses/master-the-bash-shell/gxDGoRA3JkD#the-operator)

The <() operator is conceptually similar to the $() we saw ealier. In the same way that $() substitutes the output of the process contained within it into the command, eg:

1

echo $(ls a)





Type the above code into the terminal in this lesson.

the <() operator substitutes a file containing the output of the process contained within it. You might need to stop and think about this for a second.

That means that this line:

diff <(ls a) <(ls b)

effectively becomes the command to diff two files, equivalent to the files aout and bout in the lines you typed in above.

So wherever you would normally put a filename, you can use the <() operator to save some time by dropping these in rather than creating files.

### The >() operator [#](https://www.educative.io/courses/master-the-bash-shell/gxDGoRA3JkD#the-operator-2)

Can you guess what this does? It’s similar to the <() but for me it was a lot trickier to grasp, and much more rarely seen (so feel free to skip).

See if you can work out from this line what it does:

1

tar cvf >(cat > out.tar) /tmp





Type the above code into the terminal in this lesson.

As with the <() operator, this replaces a file in a command. This time, rather than sending the output to the file, it takes input from the command that would normally go to that file reference, and feeds that input to the command in the operator.

Let’s take a step back and think about that, because it can be hard to follow.

Normally you’d write something like this:

1

2

rm -f out.tar

tar cvf out.tar /tmp





Type the above code into the terminal in this lesson.

The command is read into bash, expanded out, and the tar command accepts two arguments: a file and a folder (out.tar and the /tmp). It tars up the contents of the /tmp folder and places it in the out.tar file.

The difference in the previous command is that the contents that would normally be inputted into the file are instead fed into the command cat > out.tar.

Obviously, in this case that command is pointless - in both cases you end up with a file called out.tar that is a tar file.

Let’s say, however, that you wanted to use a different compression scheme for your tar file. You could type this:

1

tar cvf >(gzip > out.tar.gz) /tmp





Type the above code into the terminal in this lesson.

which would gzip the tarfile and place it into the out.tar.gz file.

It can reasonably be pointed out that most versions of tar offer a gzip flag (-z). that does this for you. However, some versions don’t (especially on minimal Linux distributions like busybox), so this can be a neat way of getting round that.

I have never had a need to use this mechanism in real life, but I’ve written things like this before, which are less neat (but good enough):

1

2

tar cvf out.tar /tmp

gzip out.tar





Type the above code into the terminal in this lesson.

### What You Learned [#](https://www.educative.io/courses/master-the-bash-shell/gxDGoRA3JkD#what-you-learned)

* What the <() operator is
* What the >() operator is
* How to use these operators
* How <() differs from the $() operator
* How >() works

### What Next? [#](https://www.educative.io/courses/master-the-bash-shell/gxDGoRA3JkD#what-next)

Next you will cover **subshells**, and **grouping** commands more generally.

**Subshells**

The concept of **subshells** is not a complicated one, but can lead to a little confusion at times, and occasionally is very useful.

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/gkGyN2lNM33#how-important-is-this-lesson)

As with process substitution, this is another concept I came to later in my bash career. It comes in handy fairly often, and an understanding of it helps deepen your bash wisdom.

1

VAR1='the original variable'





Type the above code into the terminal in this lesson.

Now create a subshell:

1

(





Type the above code into the terminal in this lesson.

You’ll notice the prompt has changed. Now try and echo something:

1

echo Inside the subshell





Type the above code into the terminal in this lesson.

It’s not been run. This is because the subshell’s instructions aren’t run until the parenthesis has been closed. Next we’ll try and echo the variable we created outside.

1

echo ${VAR1}





Type the above code into the terminal in this lesson.

Then update that variable to another value, and echo it again:

1

2

VAR1='the updated variable'

echo ${VAR1}





Type the above code into the terminal in this lesson.

And finally create another variable, before closing the subshell out:

1

2

VAR2='the second variable'

)





Type the above code into the terminal in this lesson.

So a subshell is a shell that inherits variables from the parent shell, and whose running is deferred until the parentheses are closed.

It will pay to think about the output and review the subshell commands to grasp what you just saw. Play with the commands and experiment until you’re comfortable with what a subshell can and can’t do.

### Subshells and Scope [#](https://www.educative.io/courses/master-the-bash-shell/gkGyN2lNM33#subshells-and-scope)

What happened to the variable you updated inside the subshell?

1

echo ${VAR1}





Type the above code into the terminal in this lesson.

Now you now know that variables can mask their parent shell’s value within the subshell. This is very handy to know if you want to do something in a slightly modified environment, as you do not have to manage variables’ previous values.

What happens if we try exporting the variable? Will that ‘export’ it to the parent shell?

1

2

( export VAR1='the first variable exported' )

echo ${VAR1}





Type the above code into the terminal in this lesson.

You will see that it does not. export does not ‘export’ the variable outside the enclosing parentheses.

### The export builtin [#](https://www.educative.io/courses/master-the-bash-shell/gkGyN2lNM33#the-export-builtin)

The export command can cause a lot of confusion here. Let’s experiment with it here to show how it works.

Note: The next code snippet puts quotes around the END delimiter. We will cover why in the next part of the course.

1

2

3

4

5

6

cat > echoes.sh << 'END'

#!/bin/bash

echo $EXPORTED

echo $NOTEXPORTED

END

chmod +x echoes.sh





Type the above code into the terminal in this lesson.

The above code creates the echoes.sh file and makes the file executable.

Now run this. What do you think will happen?

1

2

3

export EXPORTED='exported'

NOTEXPORTED='exported'

./echoes.sh





Type the above code into the terminal in this lesson.

The first variable (EXPORTED on **line 1**) is seen in the outputs of the command run in the same shell echoes.sh because it is preceded by the export builtin.

The second variables (NOTEXPORTED on **line 2**) is not exported, so is not seen by the command run from the same shell.

If the above doesn’t make sense to you, you might want to revisit Part I, where export was covered in[‘Variables’](https://www.educative.io/collection/page/5164406595911680/5419374779301888/5676808240365568).

### Subshells and Redirection [#](https://www.educative.io/courses/master-the-bash-shell/gkGyN2lNM33#subshells-and-redirection)

One often useful property of subshells is the fact that the code is treated as a single unit. You can therefore redirect output from a set of commands wholesale.

You might write code that looks like this:

1

2

echo Some output, ls to follow >> logfile

ls >> logfile





Type the above code into the terminal in this lesson.

which is fine for a couple of lines, but what if you have hundreds of lines like this? It leads to quite inelegant code. Instead, you can write:

1

2

3

4

(

echo Some output, ls to follow

ls

) >> logfile





Type the above code into the terminal in this lesson.

which is much neater and easier to manage.

### () vs {} [#](https://www.educative.io/courses/master-the-bash-shell/gkGyN2lNM33#vs)

One source of confusion is the difference between these two ways of grouping commands.

First, try replacing the ( and ) in the above listing with curlies ({ and }) and see what happens.

Then, try these:

1

2

( echo output ) >> logfile

{ echo output } >> logfile





Type the above code into the terminal in this lesson.

Hmmm. Hit CTRL+c to get out.

The curlies need a semicolon to indicate the end.

1

{ echo output ; } >> logfile





Type the above code into the terminal in this lesson.

This is because the curlies are a grouping command and need an indication of when the command has been completed. No subshell is created. The environment, variables, current working directory, indeed the entire context are the same as the surrounding code.

### Subshells and Working Directories [#](https://www.educative.io/courses/master-the-bash-shell/gkGyN2lNM33#subshells-and-working-directories)

Another useful feature of subshells is that if you change folder, then that folder change only applies within the subshell. When you return, you go back to where you were.

Let’s see this in action. First show where you are:

1

pwd





Type the above code into the terminal in this lesson.

Then, in a subshell, create a folder, move into it, and show you have moved with another call to pwd.

1

2

3

( mkdir lbthw\_subshells\_2

cd lbthw\_subshells\_2

pwd )





Type the above code into the terminal in this lesson.

Now the subshell has run, and without doing anything other than cding to a new folder, we are returned to the folder we were in before the subshell started:

1

pwd





Type the above code into the terminal in this lesson.

This is a very useful feature of subshells when you’re scripting, to save repeated cd and cd - commands.

**The Internal Field Separator**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/m7xKvm6kOkn#how-important-is-this-lesson)

Learning this concept will save you a great deal of time trying to figure out why your for loop is not working as expected, and will help you write more correct bash scripts that won’t fail.

### Files With Spaces [#](https://www.educative.io/courses/master-the-bash-shell/m7xKvm6kOkn#files-with-spaces)

Create a couple of files with spaces in their names:

1

2

3

4

echo file1 created > "Spaces in filename1.txt"

cat "Spaces in filename1.txt"

echo file2 created > "Spaces in filename2.txt"

cat "Spaces in filename2.txt"





Type the above code into the terminal in this lesson.

Note that you have to quote the filename to get the spaces in. Without the quotes, bash treats the space as a token separator. This means that it would treat the redirection as going to the file Spaces and not know what to do with the in and filenameN.txt tokens.

Now if you write a for loop over these files that just runs ls on each file in turn, you might not get what you expect. Type this in and see what happens:

1

2

3

4

for f in $(ls)

do

    ls $f

done





Type the above code into the terminal in this lesson.

Hmmm. The for loop has treated every word in the filenames as a separate token to look for with ls.

In other words, bash has treated each space as a ‘field separator’. Normally this is fine, as our for loops have items separated by spaces, like this:

1

2

3

4

for f in 1 2 3 4

do

    echo $f

done





Type the above code into the terminal in this lesson.

However, here we want the spaces to be ignored. And we can control this by setting the IFS shell variable.

### The IFS Shell Variable [#](https://www.educative.io/courses/master-the-bash-shell/m7xKvm6kOkn#the-ifs-shell-variable)

1

echo $IFS





Type the above code into the terminal in this lesson.

If you retained the default, then you will have seen nothing in the output, which isn’t very helpful. To see how it’s really set up, we can use set:

1

set | grep IFS





Type the above code into the terminal in this lesson.

You should see output like this:

    IFS=$' \t\n'

Recall that the $ before the single quotes means that the variable is showing you special characters with the backlash escape notation. Above, the IFS variable is set to: space, tab, and newline.

Bash takes the characters in that variable and will treat any of them as a field separator, which means that the original for loop you wrote above will create from these files:

    Spaces in filename1.txt  
    Spaces in filename2.txt

these list of items:

    Spaces  
    in  
    filename1.txt  
    Spaces  
    in  
    filename2.txt

What we want to do is treat the spaces like any other character. We can do this by altering the IFS variable to remove the space and re-running the for loop:

1

2

3

4

5

IFS=$'\t\n'

for f in $(ls)

do

    ls $f

done





Type the above code into the terminal in this lesson.

This might sound obscure but it’s quite common in bash to perform operations over bunches of files like this:

    find . -type f | xargs -n1 grep somestring

and in these cases you may have files lying around that have spaces in their names. Then your scripts can fail in ways that it can be difficult to debug when you don’t know about the IFS variable.

#### Code Explanation [#](https://www.educative.io/courses/master-the-bash-shell/m7xKvm6kOkn#code-explanation)

The find program used above (as the name suggests) helps you find files.

* If you just give it a folder (as above with the ‘.’ local directory) then it will return all files and folders under that folder
* The -type f flag tells find to just return files, not folders
* The xargs command runs the command given against the files piped in
* The -n1 flag tells xargs to run the command once per field piped in

### The Null Byte as Separator [#](https://www.educative.io/courses/master-the-bash-shell/m7xKvm6kOkn#the-null-byte-as-separator)

While we’re on the subject, it’s worth mentioning quickly a pattern that’s very commonly-used to get around the above scenario.

    find . -type f -print0 | xargs -0 -n1 grep somestring

By adding the -print0 flag, find no longer uses a new line as a field separator. It uses what’s called a NUL byte as the separator. The NUL byte is literally a byte of value zero. It doesn’t get displayed on the screen, but can be read by xargs as the separator if it’s given the -0 flag. This bypasses all sorts of challenges you might get fiddling with the IFS variable or dealing with spaces or other odd characters in filenames.

### What You Learned [#](https://www.educative.io/courses/master-the-bash-shell/m7xKvm6kOkn#what-you-learned)

* What the IFS shell variable does
* How to deal with filenames containing spaces and other unusual characters
* What the find and xargs programs do
* What a NUL byte is

### What Next? [#](https://www.educative.io/courses/master-the-bash-shell/m7xKvm6kOkn#what-next)

Well done! You’ve made it to the end of the ‘Scripting Bash’ section of the course. Now you are fully equipped to write and read useful shell scripts.

The next part takes what you’ve learned so far and extends it to understanding in a more advanced way how bash is used day to day on the command line