filename: bash_reading-input-into-array-5P-multif_20241217.txt
https://linuxopsys.com/bash-readarray-with-examples

Bash readarray with Examples
March 24, 2023

In every programming language, the term array means a collection of items that holds some data. The data would be in an indexed manner.

The unique property bash array is that it can save different types of elements. In this tutorial, we are going to learn about readarray.

Bash readarray
From Bash version 4, storing the contents in an array has become straightforward. Now you can easily read contents into the array. The readarray utility simply read lines from the standard input into the indexed array. It can also be read from the file descriptor by making use of the -u flag. The output is passed to the readarray command which, in turn, reads those contents and stores them in the array.

Here's the syntax:

```
readarray [-d delimiter] [-n count] [-O origin] [-s count] [-t] [-u fd] myArr
```

where,

-d delimiter: Specifies a delimiter character to use instead of the newline character.

-n: Copy at most the specified number of lines. All the lines are copied if the count is set to 0.

-O: Assigning to the array at index origin. By default the index kept 0.

-s: Let go of the first count lines read.

-t: Remove a trailing delimiter from each line read. The default value is a newline.

-u: Read lines from file descriptor fd instead of the standard input.

myArr: Name of the array to be declared.

Bash readarray Examples
Populating the array using the readarray command can be done either by reading the file, the command output or the standard input.

1. Reading from a File
Let's assume there is a sample.txt file that with the following contents:

```
[sample.txt]
France
Singapore
Peru
Ghana
Mexico
Australia
```

While reading from a file, the '<' symbol will first redirect the standard input to sample.txt and then the array will be created based on the contents of the file.

Example:

```
#!/bin/bash
readarray -t myArr  < sample.txt
echo ${myArr[0]}        #Index 0 of the array will contain the first line of sample.txt
echo ${myArr[1]}        #Index 1 of the array will contain the second line of sample.txt
```

The file sample.txt is simply fed to the readarray command. Each line of the file is stored in the array myArr.

```
$> ./readArrayScript.1.sh
France
Singapore
```

2. From command output
In this case, the output of any command is passed as an input to the readarray command. The input is then stored in the declared array.

Example:

```
#!/bin/bash
readarray -t myArr < <(seq 5)
echo ${myArr[0]}        #Index 0 of the array will contain 1
echo ${myArr[1]}        #Index 1 of the array will contain 2
```

The < <(seq 5) redirects its output to the standard input. The process substitution <(seq 5) is redirecting the output of 'seq 5' to the standard input.This makes the output of the command appear like a file. Redirecting this file to readarray is done using the first redirection operator (<). Thus, the readarray command can read the output of the command and populate the array.

```
$> bash readArrayScript.2.sh
1
```

2

3. Reading line from standard input
   Here, the input from the standard input is read by the readarray utility. This is then stored in the
   array variable defined.

   Example:
```
#!/bin/bash
readarray myArr  <<< $(cat sample.txt)
echo ${myArr[0]}        #Index 0  will contain the line till the first character 'n' is found
echo ${myArr[1]}        #Index 1 will contain after the first 'n' till the second 'n' is found
```

   This method makes use of a here string to supply the contents of the file as a standard input to the
   readarray command. The lines from the standard input fill the indexed array.
   output of script - readarray reading line from standard input output of script - readarray reading
   line from standard input

```
$> bash readArrayScript.3.sh
France
Singapore
```

4. Working with Delimiters
   The readarray command can take a delimiter symbol which indicates the beginning or the end of a data
   item. With this feature, one can decide when the next element of the array is to be populated.

   Example:
```
#!/bin/bash
readarray -d "n" myArr  < sample.txt
echo ${myArr[0]}        #Index 0  will contain the line till the first character 'n' is found
echo ${myArr[1]}        #Index 1 will contain after the first 'n' till the second 'n' is found
```

   The delimiter in this example is the character 'n'. The number of elements in the array will depend
   on the number of times the character 'n' appears in the sample.txt file. The readarray command will
   keep on adding data to that particular array element until the delimiter character 'n' is not found.
   The moment it finds 'n' in the file, it will increment the array index and assign data from the next
   character onwards, until the next 'n' is not found. This process will continue until the end of the
   file.

```
$> bash readArrayScript.4.sh
Fran
ce Sin
```

5. Accessing Array Elements in Bash readarray
   Bash arrays can be referenced using the index number. Each element of the array can also be
   accessed using a loop. Furthermore, all the array elements can be accessed together in one go. Bash
   also gives the option to access the array elements from a particular element.

   Example:
```
#!/bin/bash
readarray -t myArr  < sample.txt
echo ${myArr[0]}        #Index 0 of the array will contain the first line of sample.txt
echo ${myArr[1]}        #Index 1 of the array will contain the second line of sample.txt
echo ${myArr[4]}        #Index 4 of the array will contain the fifth line of sample.txt

echo "-----------------Printing values using a loop-------------------"
echo "Method 1:"
for i in ${myArr[@]}
do
        echo $i
done

echo "Method 2:"
for (( i=0; i<${#myArr[@]}; i++ ))
do
        echo ${myArr[$i]}
done

echo "-----------------Printing all the values--------------------"
echo "Method 1:"
echo ${myArr[*]}
echo "Method 2:"
echo ${myArr[@]}

echo "-----------------Printing values from a particular index say 3--------------------"
echo "Method 1:"
echo ${myArr[*]:3}
echo "Method 2:"
echo ${myArr[*]:3}
```

   Assuming the array myArr has been populated, the array elements can be accessed using their indexes.
   The default index of an array is numeric. The echo command will display each array value
   individually.

   The array can be iterated over a loop as well and individual elements can be referenced using the
   iterator.

The entire array can be printed using a single echo command by referencing the array index with the "*" or "@" symbol.

The array elements can be displayed from a particular starting index as well..

```
$> bash readArrayScript.5.sh
France
Singapore
Mexico
-----------------Printing values using a loop-------------------
Method 1:
France
Singapore
Peru
Ghana
Mexico
Australia
Method 2:
France
Singapore
Peru
Ghana
Mexico
Australia
-----------------Printing all the values-------------------
Method 1:
France Singapore Peru Ghana Mexico Australia
Method 2:
France Singapore Peru Ghana Mexico Australia
-----------------Printing values from a particular index say 3-------------------
Method 1:
Ghana Mexico Australia
Method 2:
Ghana Mexico Australia
```

Conclusion

Take away points we learned here:

* The Bash readarray utility is a simple, single line command for storing data in the form of an array.
* The output of any command is fed as an input to the array, thereby populating it.
* The contents of a file can be stored in an array directly, too.
* The array elements can be effectively accessed using the index numbers.

---

https://www.baeldung.com/linux/reading-output-into-array

Reading Output of a Command Into an Array in Bash
March 18, 2024

1. Overview

When we write shell scripts, we often call a command and save the output into a variable for further processing. Sometimes, we want to save a multi-line output into a Bash array.

In this tutorial, we'll discuss some common pitfalls of doing this and address how to do it in the right way.

2. Common Pitfalls

First of all, let's define our problem. We're going to execute a command and save its multi-line output into a Bash array. Each line should be an element of the array.

At first glance, the problem looks simple. We can put a command substitution between parentheses to initialize an array:

```
my_array=( $(command) )
```

Let's take the seq command as an example and try if the above approach works:

```
$> seq 5
1
2
3
4
5
$> my_array=( $(seq 5) )
$> declare -p my_array
declare -a my_array=([0]="1" [1]="2" [2]="3" [3]="4" [4]="5")
```

Great, it works!

We use the Bash built-in declare with the -p option to examine the array. It shows that the array has been initialized as we expected.

Well, so far, so good. However, this is not a stable solution. Let's see what's wrong with it.

2.1. Output May Contain Spaces

The output of a command can often include spaces. Let's change the seq command a little bit and check if our solution still works:

```
$> seq -f 'Num %g' 5
Num 1
Num 2
Num 3
Num 4
Num 5
$> my_array=( $(seq -f 'Num %g' 5) )
$> declare -p my_array
declare -a my_array=([0]="Num" [1]="1" [2]="Num" [3]="2" [4]="Num" [5]="3" [6]="Num" [7]="4" [8]="Num" [9]="5")
```

The spaces in the output break our solution. The output above tells us, the my_array now has ten elements, instead of five.

The fix may come to mind immediately: set the IFS to a newline character, so that a whole line can be assigned to an array element.

Let's try if it can fix the problem:

```
$> IFS=$'\n'
$> my_array=( $(seq -f 'Num %g' 5) )
$> declare -p my_array
declare -a my_array=([0]="Num 1" [1]="Num 2" [2]="Num 3" [3]="Num 4" [4]="Num 5")
```

Yes! That fixed it!
Unfortunately, the solution is still fragile, even though it handled spaces correctly.

Let's see what problem it still has.

2.2. Output May Contain Wildcard Characters
Some output of a command may contain wildcard characters such as *, [...] or ?, and so on.

Let's change the seq command once again and create a couple of files under our working directory:

```
$> seq -f 'Num*%g' 5
Num*1
Num*2
Num*3
Num*4
Num*5

$> touch Number.app.log.{4..5}
$> ls -1
Number.app.log.4
Number.app.log.5
```

Now, let's check if our solution can still convert the output into an array correctly:

```
$> my_array=( $(seq -f 'Num*%g' 5) )
$> declare -p my_array
declare -a my_array=([0]="Num*1" [1]="Num*2" [2]="Num*3" [3]="Number.app.log.4" [4]="Number.app.log.5")
```

Oops! The last two elements are filled by the two filenames instead of the expected "Num*4" and "Num*5". This is because if the wildcard characters match some filenames in our working directory, the filename will be picked instead of the original string.

Well, we can do a quick fix to disable the filename globbing by set -f. However, it's not wise to fix a fragile technique by changing the IFS and set -f.

Next, let's take a look at more proper ways to solve the problem.

3. Using the readarray Command
readarray is a built-in Bash command. It was introduced in Bash ver.4.

We can use the readarray built-in to solve the problem:
freestar

```
$> readarray -t my_array < <(seq 5)
$> declare -p my_array
declare -a my_array=([0]="1" [1]="2" [2]="3" [3]="4" [4]="5")

$> readarray -t my_array < <(seq -f 'Num %g' 5)
$> declare -p my_array
declare -a my_array=([0]="Num 1" [1]="Num 2" [2]="Num 3" [3]="Num 4" [4]="Num 5")

$> ls -1
Number.app.log.4
Number.app.log.5

$> readarray -t my_array < <(seq -f 'Num*%g' 5)
$> declare -p my_array
declare -a my_array=([0]="Num*1" [1]="Num*2" [2]="Num*3" [3]="Num*4" [4]="Num*5")
```

The output above shows that readarray -t my_array < <(COMMAND) can always convert the output of the COMMAND into the my_array correctly. This works no matter if the COMMAND output contains spaces or wildcard characters.

Now, let's understand why it works.

The readarray reads lines from the standard input into an array variable: my_array. The -t option will remove the trailing newlines from each line.

We used the < <(COMMAND) trick to redirect the COMMAND output to the standard input. The <(COMMAND) is called process substitution. It makes the output of the COMMAND appear like a file. Then, we redirect the file to standard input using the < FILE.

Thus, the readarray command can read the output of the COMMAND and save it to our my_array.

4. Using the read Command
We've seen that by using the readarray command, we can conveniently solve this problem. Since the readarray command was introduced in Bash ver.4, it is not available if we are working with an older Bash version.

The Bash shell has another built-in command: read, it reads a line of text from the standard input and splits it into words.

We can solve the problem using the read command:
```
IFS=$'\n' read -r -d '' -a my_array < <( COMMAND && printf '\0' )
```

Let's test it and see if it will work on different cases:
```
$> IFS=$'\n' read -r -d '' -a my_array < <( seq 5 && printf '\0' )
$> declare -p my_array
declare -a my_array=([0]="1" [1]="2" [2]="3" [3]="4" [4]="5")

$> IFS=$'\n' read -r -d '' -a my_array < <( seq -f 'Num %g' 5 && printf '\0' )
$> declare -p my_array
declare -a my_array=([0]="Num 1" [1]="Num 2" [2]="Num 3" [3]="Num 4" [4]="Num 5")

$> IFS=$'\n' read -r -d '' -a my_array < <( seq -f 'Num*%g' 5 && printf '\0' )
$> declare -p my_array
declare -a my_array=([0]="Num*1" [1]="Num*2" [2]="Num*3" [3]="Num*4" [4]="Num*5")
```

The output shows it works with our examples as well. The command looks a little bit longer than the readarray one, but it's not hard to understand either.

Let's break it down to explain what it does:
* COMMAND && printf '\0': Here we append a null byte '\0' to the output of the COMMAND so that later read will stop reading here
* < <(COMMAND && printf '\0'): This is not new to us. We redirect the COMMAND output together with the trailing null byte to the standard input
* IFS=$'\n': We've learned this as well, we set the IFS to a newline character so that the read command will read a whole line from the stream
* read -r: The -r option tells the read command not to interpret the backslashes as escape sequences
* -d ": We let the read command stop reading at a null byte
* -a my_array: This is straightforward, we tell the read command to populate the array my_array while reading

It's worthwhile to mention that the IFS variable change will only set the variable for the read statement. It won't interfere with the current shell environment.

5. Conclusion
In this article, we've solved the problem: How to save the output of a command into a Bash array.

The readarray command will be the most straightforward solution to that problem if we're working with a Bash newer than Ver. 4.

If we have to work with an older Bash, we can still solve the problem using the read command.

Apart from that, we've also seen some common pitfalls, which we should pay attention to when we write shell scripts.

---
https://stackoverflow.com/questions/53091505/add-elements-to-an-existing-array-bash

Add Elements To An Existing Array Bash

I need to process some data in several steps.

I first create a list of managed policies in AWS and put that list into an array:
```
readarray -t managed_policies < <(aws iam list-attached-user-policies --user-name tdunphy  --output json \
--profile=company-nonprod | jq -r '.AttachedPolicies[].PolicyArn')
```

But then I need to add more information to that array later on in my script. My question is, can I use readarray to do this? Or do I just need to ammend the data to the list using the command such as:
```
managed_policies+=($(aws iam list-attached-group-policies --group-name grp-cloudops --profile=compnay-nonprod \
| jq -r '.AttachedPolicies[].PolicyArn'))
```

***
I suggest to use option -O to append to your array:
```
mapfile -t -O "${#managed_policies[@]}" managed_policies < <(aws ...)
```

***
You can also do this:

```
readarray -t managed_policies < <(aws ...)
readarray -t tmp < <(aws ...)
managed_policies+=("${tmp[@]}")
```

***
   Assuming you could make the calls to aws immediately (and the example you give suggest you can, since
   both sets of arguments are hard-coded), you only need a single call to readarray. (The function is
   just some unnecessary refactoring.)

```
get_policies () {
        aws iam list-attached-user-policies --output json --profile company-nonprod "$@"
}

readarray -t managed_policies < <(
        { get_policies --user-name tdunphy
          get_policies --group-name grp-cloudops
        } | jq -r '.AttachedPolicies[].PolicyArn')
```

   (Off-topic, but it's also possible you can omit the call to jq by using appropriate arguments to aws
   to adjust the output appropriately.)

---
https://stackoverflow.com/questions/26634978/how-to-use-readarray-in-bash-to-read-lines-from-a-file-into-a-2d-array

How to use 'readarray' in bash to read lines from a file into a 2D array

Let's say I have a text file 'demo.txt' who has a table in it like this:

```
1 2 3
4 5 6
7 8 9
```

   Now, I want to read each line separately using the 'readarray' command in bash, so I write:

```
readarray myarray < demo.txt
```

   The problem is that it doesn't work. If I try to print 'myarray' with:

```
echo $myarray
```

   I get:

```
1 2 3
```

   Also, if I write:

```
echo ${myarray[1]}
```

   I get:

```
4 5 6
```

   Instead of:

```
2
```

   as I expected. Why is that? How can accesses each line separately and in that line get access to each
   member?

***
   This is the expected behavior. readarray will create an array where each element of the array is a
   line in the input.

   If you want to see the whole array you need to use

```
echo "${myarray[@]}"
```

   as echo "$myarray will only output myarray[0], and ${myarray[1]} is the second line of the data.

   What you are looking for is a two-dimensional array. See for instance
   [https://stackoverflow.com/questions/16487258/how-to-declare-2d-array-in-bash]this.

   If you want an array with the content of the first line, you can do like this:

```
$> read -a arr < demo.txt
$> echo ${arr[0]}
1
$> echo ${arr[1]}
2
$> echo ${arr[2]}
3
```

***

```
readarray rows < demo.txt

for row in "${rows[@]}";do
        row_array=(${row})
        first=${row_array[0]}
        echo ${first}
done
```

***
   To expand on Damien's answer (and because I can't submit comments yet...) you just iterate on read.
   What I mean is something like the following

```
exec 5<demo.txt
for i in $(seq 1 ${numOfLinesInFile})
do
read -a arr -u 5
        for j in $(seq 0 ${numOfColumnsMinus1})
        do
                echo ${arr[$j]}
        done
done
```

I hope you found a solution already (sorry to bump...). I stumbled upon this page while helping teach a friend and figured others may do the same.

***

How can accesses each line separately and in that line get access to each member?

Per the Bash Reference Manual, Bash provides one-dimensional indexed and associative array variables. So you cannot expect matrix[1][2] or similar to work. However, you can emulate matrix access using a bash associative arrays, where the key denotes a multiple dimension.

For example, matrix[1,2] uses the string "1,2" as the associative array key denoting the 1st row, 2nd column. Combining this with readarray:

```
typeset -A matrix
function load() {
        declare -a a=( $2 )
        for (( c=0; c < ${#a[@]}; c++ ))
        do
                matrix[$1,$c]=${a[$c]}
        done
}
readarray -C load -c 1 <<< $'1 2 3\n4 5 6\n7 8 9'
declare -p matrix
```

***

Sorry to bump but I believe there's an easy and very clean solution for your request:

```
$> cat demo.txt
1 2 3
4 5 6
7 8 9
$> while read line;do IFS=' ' myarray+=(${line}); done < demo.txt
$> declare -p myarray
declare -a myarray='([0]="1" [1]="2" [2]="3" [3]="4" [4]="5" [5]="6" [6]="7" [7]="8" [8]="9")'
```

***

Since 3 out of 5 answers ignore the OPs request to use readarray I'm guessing no one will downvote me for adding another that also fails to use readarray.

Paste the following code unaltered into an Ubuntu bash command line (not tried in any other environment)

Code

```
# Create test array
echo -e "00 01 02 03 04
10 11 12 13 14
20 21 22 23 24
30 31 32 33 34" > original.txt;

# Reformat test array as a declared bash variable.
sed 's/^/"/g;  s/$/"/g;  1s/^/declare my2d=(\n/;  $s/$/\n);/' original.txt > original.sh;

# Source the bash variable.
source original.sh;

# Get a row.
declare row2=(${my2d[2]});

# Get a cell.
declare cell3=${row2[3]};
echo -e "Cell [2, 3] holds [${cell3}]";
```

Output

```
Cell [2, 3] holds [23]
```

Explanation

The four sed groups do the following:

1. s/^/"/g; - prepend double quotes to each line
2. s/$/"/g; - append double quotes to each line
3. 1s/^/declare my2d=(\n/; - prepend declare my2d=(to file
4. $s/$/\n); - append);to file

Note

This gets too messy to be worth using if your array elements have whitespace in them

---
https://www.geeksforgeeks.org/mapfile-command-in-linux-with-examples/

mapfile Command in Linux With Examples
05 Sep, 2024

The 'mapfile' command, also known as 'readarray', is a powerful built-in feature of the Bash shell used for reading input lines into an array variable. Unlike using a loop with read, mapfile reads lines directly from standard input or command substitution (< <(command)) rather than from a pipe, making it faster and more convenient for handling arrays. If no array name is provided, 'mapfile' defaults to using the variable MAPFILE to store the data.

What does the 'mapfile' Command do?
The 'mapfile' command is particularly useful when you need to read data line by line into an array, allowing you to manipulate each line individually or collectively with ease. This command is especially beneficial when dealing with data processing, file reading, and output capturing tasks in scripts.
Syntax: mapfile [array]

Alternatively, we can use read array [arrayname] instead of mapfile.

mapfile Command Examples in Linux

```
[example.txt]
GEEKS1
GEEKS2
GEEKS3
```

Example 1. Reading an array from a file:
```
$> mapfile MYFILE < example.txt
$> echo ${MYFILE[@]}
GEEKS1 GEEKS2 GEEKS3

$> echo ${MYFILE[0]}
GEEKS1
```

Example 2. Capture the output into an array:
```
$> mapfile GEEKSFORGEEKS < <(printf "Item 1\nItem 2\nItem 3\n")
$> echo  ${GEEKSFORGEEKS[@]}
Item 1 Item 2 Item 3
```

Here, Item1, Item2, and Item 3 have been stored in the array GEEKSFORGEEKS.

Example 3. Strip newlines and store item using -t:
```
$> mapfile -t GEEKSFORGEEKS < <(printf "Item 1\nItem 2\nItem 3\n")
$> printf "%s\n" "${GEEKSFORGEEKS[@]}"
Item 1
Item 2
Item 3
```

Example 4. Read the specified number of lines using -n:
```
$> mapfile -n 2 GEEKSFORGEEKS < example.txt
$> echo  ${GEEKSFORGEEKS[@]}
GEEKS1 GEEKS2
```

It reads at most 2 lines. If 0 is specified then all lines are considered.

Conclusion
The mapfile command in Bash is a useful tool for reading lines into arrays from standard input or command substitution. It makes handling data in scripts easier by eliminating the need for loops and provides various options for managing input, like stripping newlines, reading a set number of lines, or using callbacks for processing. Mapfile and its options can enhance your data handling capabilities in Bash, leading to more powerful and flexible scripting.

---
https://www.computerhope.com/unix/bash/mapfile.htm

Bash mapfile builtin command
09/10/2024

On Unix-like operating systems, mapfile is a builtin command of the Bash shell. It reads lines from standard input into an indexed array variable.

Syntax

```
mapfile [-n count] [-O origin] [-s count] [-t] [-u fd]
        [-C callback [-c quantum]] [array]
```

Options
The mapfile builtin command takes the following options:
-n count Read a maximum of count lines. If count is zero, all available lines are copied.
-O origin Begin writing lines to array array at index number origin. The default value is zero.
-s count Discard the first count lines before writing to array.
-t If any line ends in a newline, strip it.

-u fd Read lines from file descriptor fd rather than standard input.
-C callback Execute/evaluate a function/expression, callback, every time quantum lines are read. The default value of quantum is 1, unless otherwise specified with -c.
-c quantum Specify the number of lines, quantum, after which function/expression callback should be executed/evaluated if specified with -C.
array The name of the array variable where lines should be written. If array is omitted, the default variable MAPFILE is the target.

Notes
The command name readarray may be used as an alias for the command name mapfile, with no difference in operation.

If the -u option is specified, mapfile reads from file descriptor fd instead of standard input.

If array is not specified, the default variable MAPFILE is used as the target array variable.

The mapfile command is not very portable. That is, to ensure your script can run on a wide array of systems, it's not recommended to use mapfile. It's provided primarily as a convenience. The same functionality can be achieved using a read loop, although in general mapfile performs faster.

Exit status
The mapfile command returns 0 for success, or 1 if anything goes wrong, e.g., an invalid option is provided, or the target variable is read-only or not an array.

Examples
The mapfile command reads input line by line, and puts each line in an array variable. Let's provide it with multiple lines of input.

We can use printf to do this. It's an easy way to print text with newlines.

In our printf format string, we can include "\n" (a backslash immediately followed by a lowercase n) to create a newline. ("\n" is a metacharacter, a sequence of characters representing another character that can't be typed literally, such as the Enter key. For a complete list of bash metacharacters, see quoting in bash.)

This printf command prints three lines of text:

```
$> printf "Line 1\nLine 2\nLine 3\n"
 Line 1
 Line 2
 Line 3
```

We want to use mapfile to put each of these lines in its own element of an array.

By default, mapfile reads from standard input, so you might be tempted to pipe the output of printf to mapfile like this:

```
$> printf "Line 1\nLine 2\nLine 3\n" | mapfile
```

You would expect the default array variable MAPFILE to contain the values from these lines. But if you check the value of MAPFILE:

```
$> echo "${MAPFILE[@]}"
[a blank line]
```

The variable is empty. Why?

Each command in a pipeline executes in a subshell - an instance of bash, executed as a child process. Each subshell has its own environment, and its own lexical scope - the variables that make up the environment of each subshell in the pipeline do not carry over to others. In other words, there are no environmental side effects shared from one element of a pipeline to the next. In our example above, mapfile works correctly, and sets the values of MAPFILE, but when the command's subshell terminates, the variable MAPFILE vanishes.

You can see this if you echo the value of MAPFILE inside a subshell that also contains the mapfile command, by enclosing both in parentheses:

```
$> printf "Line 1\nLine 2\nLine 3\n" | ( mapfile; echo "${MAPFILE[@]}" )
 Line 1
 Line 2
 Line 3
```

In the above command, echo prints all the elements of array variable MAPFILE, separated by a space. The space appears at the beginning of lines 2 and 3 because of the newlines in our data. Our explicit subshell, expressed with parentheses, preserves the value of MAPFILE long enough for us to see the values.

We can fix the line breaks by stripping them with -t:

```
$> printf "Line 1\nLine 2\nLine 3\n" | ( mapfile -t; echo "${MAPFILE[@]}" )
Line 1 Line 2 Line 3
```

(We can put the line breaks back in our output if we use printf - we'll do that in subsequent examples.)

So, mapfile is working, but the array variable is inaccessible to the parent shell. Normally, however, you will want the MAPFILE variable to persist for subsequent commands. You can accomplish that with process substitution.

Using mapfile with process substitution

With process substitution, we can redirect output to mapfile without using a pipeline.
```
$> mapfile -t < <(printf "Line 1\nLine 2\nLine 3")
```

Let's look at the individual parts of this command:
mapfile -t Mapfile takes input from standard input, and strip newlines (-t) from the end of each line. This is (usually) what you want: only the text of the line is stored in your array element, and the newline character is discarded.
< The first < is a redirection character. It expects to be followed by a file name, or file descriptor. The contents of that file are redirected to the standard input of the preceding command.
<(...) These characters indicate process substitution, which returns a file descriptor. The commands inside the parentheses are executed, and their output is assigned to this file descriptor. In any bash command, you can use process substitution like a file name.

When you run the whole command, mapfile silently reads our three lines of text, and places each line into individual elements of the default array variable, MAPFILE.

We can verify this using printf to print the elements of the array.
```
$> printf "%s" "${MAPFILE[@]}"
```

The first argument, "%s" is the printf format string. The second argument, "${MAPFILE[@]}", is expanded by bash. All elements ("@") of array MAPFILE are expanded to individual arguments. (For more information, see: Referencing array elements in bash.)
```
Line 1Line 2Line 3
```

As you can see, our three lines of text are printed right next to each other. That's because we stripped the newlines with -t, and printf doesn't output newlines by default.

To specify that printf print a newline after each line, use \n in the format string:
```
$> printf "%s\n" "${MAPFILE[@]}"
Line 1
Line 2
Line 3
```

To access individual elements of the array, replace @ with an index number. The numbering is zero-based, so 0 indexes the first element, 1 indexes the second, etc:
```
$> printf "%s\n" "${MAPFILE[0]}"
Line 1
$> printf "%s\n" "${MAPFILE[2]}"
Line 3
```

---
https://medium.com/@linuxadminhacks/understanding-mapfile-command-in-linux-9a13a2e2008a

Understanding mapfile Command in linux
Jan 23, 2024; Generated By: Adobe Firefly AI

In Linux, the mapfile command is a built-in command in the Bash shell. It is used to read lines from a file and assign each line to an element in an array. The mapfile command is also known as readarray.

Here is the basic syntax of the mapfile command:
```
mapfile [options] arrayname
mapfile [-d delim] [-n count] [-O origin] [-s count] [-t] [-u fd] [-C callback] [-c quantum] [array]
```

Let's break down the options used in the command:
- -d delim: The first character of delim is used to terminate each input line, rather than newline. If delim is the empty string, mapfile will terminate a line when it reads a NUL character.
- -n count: Copy at most count lines. If count is 0, all lines are copied.
- -O origin: Begin assigning to array at index origin. The default index is 0.
- -s count: Discard the first count lines read.
- -t: Remove a trailing newline from each line read.
- -u fd: Read lines from file descriptor fd instead of the standard input.
- -C callback: Evaluate callback each time quantum lines are read. The -c option specifies quantum.
- -c quantum: Specify the number of lines read between each call to callback.

Basic Example
Here is an example of how to use the mapfile command:
```
$> mapfile lines < file.txt
$> echo ${lines[@]}
$> echo ${lines[0]}
```

In the context of the mapfile command, lines is simply a placeholder for an array variable name. When you execute the command mapfile lines < file.txt, the content of file.txt is read into the lines array. After this operation, you can access the elements of lines using ${lines[index]} where index is the position of the element in the array.

For instance, after executing the command, you could print the first element of the lines array with echo ${lines[0].

In Bash scripting, ${lines[@]} is a form of array expansion. It represents all elements of the lines array. The @ symbol is used to represent all indices of the array.

When used without quotes around it, it expands to a list of words split by the first character of the IFS (Internal Field Separator) special variable. Each word is a separate argument

It's important to note that lines is just a name chosen for the array in this example. You could use any valid variable name in place of lines.

How to specify a starting index

When using the mapfile command in Linux, you can specify a starting index for the elements in the array using the -O option followed by the desired index number.

Here's the syntax:

```
$> mapfile -O index arrayname
```

Here are examples:

```
$> mapfile -O 1 lines < file.txt
```

The lines array will start filling from index 1. Therefore, the first line of file.txt will be stored in lines[1].

```
$> mapfile -O 5 lines < file.txt
$> echo ${lines[0]}
$> echo ${lines[1]}
$> echo ${lines[5]}
```

This command will read the content of file.txt into an array named lines, starting from index 5. This means the first line of file.txt will be stored in lines[5], the second line will be stored in lines[6], and so on.

An Array of Disks

```
$> mapfile -t disks < <(lsblk -o PATH | grep -v "PATH")
$> echo ${disks[@]}
```

This command is doing several things:

* lsblk -o PATH: This part of the command lists the block devices in your system and outputs the path of each device. The -o PATH option tells lsblk to only display the path of each block device.
* | grep -v "PATH": This pipes (|) the output of the lsblk -o PATH command to grep -v "PATH", which filters out lines containing the string "PATH". The -v option inverts the match, meaning it will exclude lines that contain "PATH".
* mapfile -t disks <: This reads the output from the preceding commands into an array called disks. The -t option removes a trailing newline from each line read.

So, in summary, this command is reading the paths of all block devices in your system (excluding the header line that says "PATH") into an array called disks.

The -t option in the mapfile command is used to remove a trailing newline from each line read. This means that when mapfile reads each line from the input, it doesn't include the newline character (\n) at the end of each line. This can be useful when you want to preserve the exact formatting of the input lines in the array.

Here's an example:

```
$> mapfile -t myarray < file.txt
```

In this example, the content of file.txt is read into the myarray array, and the -t option ensures that the trailing newline characters are removed from each line.

If you were to print the contents of myarray afterwards, you would see that each line is immediately followed by the next line, with no extra newline characters in between. This can be useful when you want to process or manipulate the lines individually later in your script.

---

https://medium.com/@kuldeepkumawat195/mapfile-command-in-linux-with-examples-58d5f1e7c3ac

mapfile Command in Linux With Examples

The mapfile command, also known as readarray, is a powerful utility in the Bash shell that efficiently reads lines from standard input into an array. As a versatile tool, it is often preferred over traditional read loops due to its speed and convenience. Below, we delve into the syntax, usage, and examples of the mapfile command, highlighting its importance in simplifying array operations in Linux.

Syntax of the mapfile Command

The syntax for the mapfile command is straightforward:

```
mapfile [array]
```

Key Options and Their Functions

* -d delim: By default, mapfile reads each line of a file into an array element, ending with a new line. The -d option allows you to specify a different delimiter. If delim is an empty string, mapfile will terminate a line upon encountering a NUL character.
* -n count: This option restricts the number of lines copied into the array. If count is set to 0, mapfile reads all lines.
* -O origin: Begin assigning array elements from the specified index origin. The default index is 0.
* -s count: Skip the first count lines of the input file.
* -t: Automatically removes the trailing newline character from each line read.
* -u fd: Reads lines from the specified file descriptor fd instead of the standard input.
* -C callback: Executes a callback function each time a specified number of lines (quantum) are read. This is used in combination with the -c option.

     * -c quantum: Defines the number of lines read between each call to the callback function.

   Alternatively, you can use the readarray command, which is functionally identical:
readarray [arrayname]

Key Points:
     * Input Source: mapfile reads from standard input.
     * Default Array Variable: If no array name is specified, the MAPFILE variable is used by default.
     * Substitution Requirement: mapfile requires input from substitution (< <) rather than a pipe.

Practical Examples of mapfile

Example 1: Reading an Array from a File
   The mapfile command can read the contents of a file directly into an array. This is particularly
   useful for processing files where each line needs to be stored as an element in an array.
$> mapfile MYFILE < example.txt
$> echo ${MYFILE[@]}
$> echo ${MYFILE[0]}

   Explanation:
     * mapfile MYFILE < example.txt reads each line of example.txt into the MYFILE array.
     * `echo ${MYFILE[@]}prints all elements of theMYFILE` array in a single line.
     * echo ${MYFILE[0]} outputs the first element of the array, which corresponds to the first line of
       example.txt.

Example 2: Capturing Command Output into an Array
   You can use mapfile to capture the output of a command into an array. This method is particularly
   effective when dealing with command outputs that generate multiple lines.
$> mapfile test < <(printf "Item 1\nItem 2\nItem 3\n")
$> echo ${test[@]}

   Explanation:
     * mapfile test< <(printf "Item 1\nItem 2\nItem 3\n") captures the output of printf into the test
       array.
     * echo ${test[@]} displays all the array elements.

Example 3: Stripping Newlines with the -t Option
   The -t option in mapfile strips the trailing newline characters from each line before storing them in
   the array. This is useful when you need to cleanly handle input data without the additional newline
   characters.
$> mapfile -t test < <(printf "Item 1\nItem 2\nItem 3\n")
$> printf "%s\n" "${test[@]}"

   Explanation:
     * mapfile -t test< <(printf "Item 1\nItem 2\nItem 3\n") reads the output into an array while
       stripping newlines.
     * printf "%s\n" "${test[@]}" prints each element of the array on a new line.

Example 4: Reading a Specified Number of Lines with -n
   The -n option allows you to specify the number of lines to read from the input. This feature is
   valuable when you need to limit the number of lines processed.
$> mapfile -n 2 test < example.txt
$> echo ${test[@]}

   Explanation:
     * mapfile -n 2 test < example.txt reads only the first two lines from example.txt into the test
       array.
     * echo ${test[@]} prints the array contents.

Example 5: Specifying a Starting Index
   You can control the starting index of the array elements using the -O option. For instance, to start
   the array at the index 1:
$> mapfile -O 1 lines < example.txt

   This command fills the array lines, starting at the index 1, with the content of example.txt.

Example 6: Skipping Lines
   To skip a specified number of lines at the beginning of the input, use the -s option:
$> mapfile -s 3 lines < example.txt

   This command skips the first three lines of example.txt and stores the subsequent lines in the lines
   array.

Real-World Application: Array of Disks
   A practical application of the mapfile command is reading the paths of block devices into an array:
$> mapfile -t disks < <(lsblk -o PATH | grep -v "PATH")
$> echo ${disks[@]}

Breakdown:
     * lsblk -o PATH: Lists block devices, showing their paths.
     * grep -v "PATH": Filters out the header row containing "PATH".
     * mapfile -t disks: Reads the filtered output into the disks array, stripping any trailing
       newlines.

   So, in summary, this command is reading the paths of all block devices in your system (excluding the
   header line that says "PATH") into an array called disks.

Additional Options and Considerations
* Handling Large Files: When dealing with large files, mapfile can efficiently handle reading lines into arrays, making it a preferable alternative to looping constructs.
* Error Handling: The command returns 1 if it fails to execute and 0 upon success, enables you to implement robust error-handling routines in your scripts.

Conclusion
The mapfile command in Linux is a highly efficient tool for reading and managing arrays directly from input sources. By understanding and leveraging its various options, such as -t for stripping newlines or -n for limiting the number of lines read, you can simplify array handling in your scripts. Whether you're processing files or capturing command output, mapfile offers a more elegant and performance-oriented alternative to traditional methods.

---
https://stackoverflow.com/questions/11426529/reading-output-of-a-command-into-an-array-in-bash

Reading output of a command into an array in Bash

I need to read the output of a command in my script into an array. The command is, for example:
$> ps aux | grep | grep | x

and it gives the output line by line like this:
10
20
30

I need to read the values from the command output into an array, and then I will do some work if the size of the array is less than three.

***
The other answers will break if output of command contains spaces (which is rather frequent) or glob characters like *, ?, [...].

To get the output of a command in an array, with one line per element, there are essentially 3 ways:
1. With Bash >= 4 use mapfile-it's the most efficient:
```
mapfile -t my_array < <( my_command )
```

2. Otherwise, a loop reading the output (slower, but safe):
```
my_array=()
while IFS= read -r line; do
        my_array+=( "$line" )
done < <( my_command )
```

3. As suggested by Charles Duffy in the comments (thanks!), the following might perform better than the loop method in number 2:
```
IFS=$'\n' read -r -d '' -a my_array < <( my_command && printf '\0' )
```

Please make sure you use exactly this form, i.e., make sure you have the following:
+ IFS=$'\n' on the same line as the read statement: this will only set the environment variable IFS for the read statement only. So it won't affect the rest of your script at all. The purpose of this variable is to tell read to break the stream at the EOL character \n.
+ -r: this is important. It tells read to not interpret the backslashes as escape sequences.
+ -d '': please note the space between the -d option and its argument ''. If you don't leave a space here, the '' will never be seen, as it will disappear in the quote removal step when Bash parses the statement. This tells read to stop reading at the nil byte. Some people write it as -d $'\0', but it is not really necessary. -d '' is better.
+ -a my_array tells read to populate the array my_array while reading the stream.
+ You must use the printf '\0' statement after my_command, so that read returns 0; it's actually not a big deal if you don't (you'll just get an return code 1, which is okay if you don't use set -e - which you shouldn't anyway), but just bear that in mind. It's cleaner and more semantically correct. Note that this is different from printf '', which doesn't output anything. printf '\0' prints a null byte, needed by read to happily stop reading there (remember the -d '' option?).

If you can, i.e., if you're sure your code will run on Bash >= 4, use the first method. And you can see it's shorter too.

If you want to use read, the loop (method 2) might have an advantage over method 3 if you want to do some processing as the lines are read: you have direct access to it (via the $line variable in the example I gave), and you also have access to the lines already read (via the array ${my_array[@]} in the example I gave).

Note that mapfile provides a way to have a callback eval'd on each line read, and in fact you can even tell it to only call this callback every N lines read; have a look at help mapfile and the options -C and -c therein. (My opinion about this is that it's a little bit clunky, but can be used sometimes if you only have simple things to do - I don't really understand why this was even implemented in the first place!).

Now I'm going to tell you why the following method:
```
my_array=( $( my_command) )
```

is broken when there are spaces:
```
$> # I'm using this command to test:
$> echo "one two"; echo "three four"
```

```
one two
three four

$> # Now I'm going to use the broken method:
$> my_array=( $( echo "one two"; echo "three four" ) )
$> declare -p my_array
declare -a my_array='([0]="one" [1]="two" [2]="three" [3]="four")'

$> # As you can see, the fields are not the lines
$
$> # Now look at the correct method:
$> mapfile -t my_array < <(echo "one two"; echo "three four")
$> declare -p my_array
declare -a my_array='([0]="one two" [1]="three four")'

$> # Good!

   Then some people will then recommend using IFS=$'\n' to fix it:
$> IFS=$'\n'
$> my_array=( $(echo "one two"; echo "three four") )
$> declare -p my_array
declare -a my_array='([0]="one two" [1]="three four")'
$> # It works!

   But now let's use another command, with globs:
$> echo "* one two"; echo "[three four]"
* one two
[three four]
$> IFS=$'\n'
$> my_array=( $(echo "* one two"; echo "[three four]") )
$> declare -p my_array
declare -a my_array='([0]="* one two" [1]="t")'
$> # What?

   That's because I have a file called t in the current directory... and this filename is matched by the
   glob [three four]... at this point some people would recommend using set -f to disable globbing: but
   look at it: you have to change IFS and use set -f to be able to fix a broken technique (and you're
   not even fixing it really)! when doing that we're really fighting against the shell, not working with
   the shell.
$> mapfile -t my_array < <( echo "* one two"; echo "[three four]")
$> declare -p my_array
declare -a my_array='([0]="* one two" [1]="[three four]")'

   here we're working with the shell!

***
   You can use
my_array=( $(<command>) )

   to store the output of command <command> into the array my_array.

   You can access the length of that array using
my_array_length=${#my_array[@]}

   Now the length is stored in my_array_length.

Handling expansion and spaces issues

   To avoid expansion problems you can set the IFS env var as follows
     * as suggested by @smac89 in the comments and other answers

IFS=$'\n' my_array=( $(<command>) )

***
   Here is a simple example. Imagine that you are going to put the files and directory names (under the
   current folder) to an array and count them. The script would be like;
my_array=( `ls` )
my_array_length=${#my_array[@]}
echo $my_array_length

   Or, you can iterate over this array by adding the following script:
for element in "${my_array[@]}"
do
        echo "${element}"
done

   Please note that this is the core concept and the input must be sanitized before the processing, i.e.
   removing extra characters, handling empty Strings, and etc. (which is out of the topic of this
   thread).

***
   It helps me all the time suppose you want to copy whole list of directories into current directory
   into an array

bucketlist=($(ls))
#then print them one by one
```

```
for bucket in "${bucketlist[@]}"; do
        echo " here is bucket: ${bucket}"
done
```

---

https://ss64.com/bash/mapfile.html

mapfile

Read lines from the standard input into the indexed array variable array, or from file descriptor fd if the -u option is supplied. The variable MAPFILE is the default array. The command name readarray may be used as an alias for mapfile, with no difference in operation.

Syntax

```
      mapfile [-d delim] [-n count] [-O origin] [-s count] [-t] [-u fd] [-C callback] [-c quantum] [array]
    readarray [-d delim] [-n count] [-O origin] [-s count] [-t] [-u fd] [-C callback] [-c quantum] [array]
```

Key

-d The first character of delim is used to terminate each input line, rather than newline. If delim is the empty string, mapfile will terminate a line when it reads a NUL character.

-n Copy at most count lines. If count is 0, all lines are copied.

-O Begin assigning to array at index origin. The default index is 0.

-s Discard the first count lines read.

-t Remove a trailing newline from each line read.

-u Read lines from file descriptor fd instead of the standard input.

-C Evaluate callback each time quantum lines are read. The -c option specifies quantum.

-c Specify the number of lines read between each call to callback.

If -C is specified without -c, the default quantum is 5000.
When callback is evaluated, it is supplied the index of the next array element to be assigned as an additional argument. callback is evaluated after the line is read but before the array element is assigned.

If not supplied with an explicit origin, mapfile will clear array before assigning to it.

mapfile returns successfully unless an invalid option or option argument is supplied, array is invalid or unassignable, or if array is not an indexed array.

mapfile can't do anything that couldn't be done using read and a loop. A read loop is far more portable but is significantly slower tham mapfile.

mapfile is a BASH shell builtin, to display your local syntax from the bash prompt type: help mapfile

Examples

Capture the output of a script inner.sh and store it in an array called myarray:

```
  mapfile -t myarray < <(./inner.sh)
```

The last part of this <(./inner.sh)expands to a filename, however mapfile normally expects stdin, we can redirect the filename to stdin with a < character, but the two <'s have to be separated with a space so that they don't get interpreted as <<.

Print the text Alpha, Beta, Gamma as 3 separate lines, when fed into mapfile these will be saved in the array 'GreekArray':

```
  mapfile -t GreekArray < <(printf "Alpha\nBeta\nGamma")
```

In both these examples the -t option removes the newline characters so that only text is stored in each array element. Discarding the newline character is usually what you need.

"If you feel safe in the area you're working in, you're not working in the right area. Always go a little further into the water than you feel you're capable of being in. Go a little bit out of your depth. And when you don't feel that your feet are quite touching the bottom, you're just about in the right place to do something exciting." ― David Bowie

---

https://stackoverflow.com/questions/6980090/how-to-read-from-a-file-or-standard-input-in-bash

How to read from a file or standard input in Bash

The following Perl script (my.pl) can read from either the file in the command line arguments or from standard input (STDIN):

```
while (<>) {
   print($_);
}
```

   perl my.pl will read from standard input, while perl my.pl a.txt will read from a.txt. This is very
   handy.

   Is there an equivalent in Bash?

***
   The following solution reads from a file if the script is called with a file name as the first
   parameter $1 and otherwise from standard input.
```
while read line
do
        echo "$line"
done < "${1:-/dev/stdin}"
```

   The substitution ${1:-...} takes $1 if defined. Otherwise, the file name of the standard input of the
   own process is used.

***
   Perhaps the simplest solution is to redirect standard input with a merging redirect operator:
```
#!/bin/bash
less <&0
```

   Standard input is file descriptor zero. The above sends the input piped to your bash script into
   less's standard input.

***
   Here is the simplest way:
```
#!/bin/sh
cat -
```

   Usage:
```
$> echo test | sh my_script.sh
test
```

   To assign stdin to the variable, you may use: STDIN=$(cat -) or just simply STDIN=$(cat) as operator
   is not necessary (as per @mklement0 comment).

   To parse each line from the standard input, try the following script:

```
#!/bin/bash
while IFS= read -r line; do
        printf '%s\n' "$line"
done
```

   To read from the file or stdin (if argument is not present), you can extend it to:

```
#!/bin/bash
file=${1--} # POSIX-compliant; ${1:--} can be used either.
while IFS= read -r line; do
        printf '%s\n' "$line" # Or: env POSIXLY_CORRECT=1 echo "$line"
done < <(cat -- "$file")
```

 Notes:
     * read -r - Do not treat a backslash character in any special way. Consider each backslash to be part of
       the input line.

     * Without setting IFS, by default the sequences of Space and Tab at the beginning and end of the lines
       are ignored (trimmed).

     * Use printf instead of echo to avoid printing empty lines when the line consists of a single -e, -n or
       -E. However there is a workaround by using env POSIXLY_CORRECT=1 echo "$line" which executes your
       external GNU echo which supports it. See: How do I echo "-e"?

***
   I think this is the straightforward way:
```
$> cat reader.sh
#!/bin/bash
while read line; do
        echo "reading: ${line}"
done < /dev/stdin

$> cat writer.sh
#!/bin/bash
for i in {0..5}; do
        echo "line ${i}"
done

$> ./writer.sh | ./reader.sh
reading: line 0
reading: line 1
reading: line 2
reading: line 3
reading: line 4
reading: line 5
```

***
   The echo solution adds new lines whenever IFS breaks the input stream. @fgm's answer can be

   modified a bit:
```
cat "${1:-/dev/stdin}" > "${2:-/dev/stdout}"
```

***

   The Perl loop in the question reads from all the file name arguments on the command line, or from standard input if no files are specified. The answers I see all seem to process a single file or standard input if there is no file specified.

   Although often derided accurately as UUOC (Useless Use of cat), there are times when cat is the best tool for the job, and it is arguable that this is one of them:

```
cat "$@" |
while read -r line
do
        echo "$line"
done
```

   The only downside to this is that it creates a pipeline running in a sub-shell, so things like variable assignments in the while loop are not accessible outside the pipeline. The bash way around that is Process Substitution:

```
while read -r line
do
        echo "$line"
done < <(cat "$@")
```

   This leaves the while loop running in the main shell, so variables set in the loop are accessible outside the loop.

***

   Perl's behavior, with the code given in the OP can take none or several arguments, and if an argument is a single hyphen - this is understood as stdin. Moreover, it's always possible to have the filename with $ARGV. None of the answers given so far really mimic Perl's behavior in these respects. Here's a pure Bash possibility. The trick is to use exec appropriately.

```
#!/bin/bash

(($#)) || set -- -
while (($#)); do
        { [[ $1 = - ]] || exec < "$1"; } &&
        while read -r; do
                printf '%s\n' "$REPLY"
        done
        shift
done
```

   Filename's available in $1.

   If no arguments are given, we artificially set - as the first positional parameter. We then loop on the parameters. If a parameter is not -, we redirect standard input from filename with exec. If this redirection succeeds we loop with a while loop. I'm using the standard REPLY variable, and in this case you don't need to reset IFS. If you want another name, you must reset IFS like so (unless, of course, you don't want that and know what you're doing):

```
while IFS= read -r line; do
        printf '%s\n' "$line"
done
```

***

   More accurately...
```
while IFS= read -r line ; do
        printf "%s\n" "$line"
done < file
```

***

Stream approach
   Minor revisions to earlier answers:
     * Use cat, not less. It's faster and you don't need pagination.
     * Use $1 to read from first argument file (if present) or $* to read from all files (if present). If these variables are empty, read from stdin (like cat does)

```
#!/bin/bash
cat $* | ...
```

File approach
   Writing into a named pipe is a bit more complicated, but this allows you to treat stdin (or files) like a single file:
     * Create pipe with mkfifo.
     * Parallelize the writing process. If the named pipe is not read from, it may block otherwise.
     * For redirecting stdin into a subprocess (as necessary in this case), use <&0 (unlike what others have been commenting, this is not optional here).

```
#!/bin/bash
mkfifo /tmp/myStream
cat $* <&0 > /tmp/myStream &           # separate subprocess (!)
AddYourCommandHere /tmp/myStream       # process input like a file,
rm /tmp/myStream                       # cleaning up
```

```
File approach: Variation
   Create named pipe only if no arguments are given. This may be more stable for reading from files as
   named pipes can occasionally block.
#!/bin/bash
FILES=$*
if echo $FILES | egrep -v . >&/dev/null; then # if $FILES is empty
        mkfifo /tmp/myStream
        cat <&0 > /tmp/myStream &
        FILES=/tmp/myStream
fi

AddYourCommandHere $FILES     # do something ;)
if [ -e /tmp/myStream ]; then
        rm /tmp/myStream
fi

   Also, it allows you to iterate over files and stdin rather than concatenate all into a single stream:
for file in $FILES; do
        AddYourCommandHere $file
done

***
#!/usr/bin/bash

if [ -p /dev/stdin ]; then
        #for FILE in "$@" /dev/stdin
        for FILE in /dev/stdin
        do
                while IFS= read -r LINE
                do
                        echo "$@" "$LINE"   #print line argument and stdin
                done < "$FILE"
        done
else
        printf "[ -p /dev/stdin ] is false\n"
        #dosomething
fi

   Running:
echo var var2 | bash std.sh

   Result:
var var2

   Running:
bash std.sh < <(cat /etc/passwd)

   Result:
root:x:0:0::/root:/usr/bin/bash
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin

***
   The following works with standard sh (tested with Dash on Debian) and is quite readable, but
   that's a matter of taste:
if [ -n "$1" ]; then
        cat "$1"
else
        cat
fi | commands_and_transformations

   Details: If the first parameter is non-empty then cat that file, else cat standard input. Then the
   output of the whole if statement is processed by the commands_and_transformations.

***
   I combined all of the above answers and created a shell function that would suit my needs. This is
   from a Cygwin terminal of my two Windows 10 machines where I had a shared folder between them. I need
   to be able to handle the following:
     * cat file.cpp | tx
     * tx < file.cpp
     * tx file.cpp

   Where a specific filename is specified, I need to used the same filename during copy. Where input
   data stream has been piped through, then I need to generate a temporary filename having the hour
   minute and seconds. The shared mainfolder has subfolders of the days of the week. This is for
   organizational purposes.

   Behold, the ultimate script for my needs:
tx ()
{
        if [ $# -eq 0 ]; then
                local TMP=/tmp/tx.$(date +'%H%M%S')
                while IFS= read -r line; do
                        echo "$line"
                done < /dev/stdin > $TMP
```

```
                cp $TMP //$OTHER/stargate/$(date +'%a')/
                rm -f $TMP
        else
                [ -r $1 ] && cp $1 //$OTHER/stargate/$(date +'%a')/ || echo "cannot read file"
        fi
}
```

   If there is any way that you can see to further optimize this, I would like to know.

***

   The code ${1:-/dev/stdin} will just understand the first argument, so you can use this:

```
ARGS='$*'
if [ -z "$*" ]; then
        ARGS='-'
fi
eval "cat -- $ARGS" | while read line
do
        echo "$line"
done
```

***

Reading from stdin into a variable or from a file into a variable.

   Most examples in the existing answers use loops that immediately echo each of line as it is read from stdin. This might not be what you really want to do.

   In many cases you need to write a script that calls a command which only accepts a file argument. But in your script you may want to support stdin also. In this case you need to first read full stdin and then provide it as a file.

   Let's see an example. The script below prints the certificate details of a certificate (in PEM format) that is passed either as a file or via stdin.

```
# print-cert script

content=""
while read line
do
        content="$content$line\n"
done < "${1:-/dev/stdin}"

# Remove the last newline appended in the above loop
content=${content%\\n}

# Keytool accepts certificate only via a file, but in our script we fix this.
keytool -printcert -v -file <(echo -e $content)


# Read from file

cert-print mycert.crt

# Owner: CN=....
# Issuer: ....
# ....

# Or read from stdin (by pasting)

cert-print
#..paste the cert here and press enter
# Ctl-D

# Owner: CN=....
# Issuer: ....
# ....

# Or read from stdin by piping to another command (which just prints the cert(s) ). In this case we use openss
l to fetch directly from a site and then print its info.

echo "" | openssl s_client -connect www.google.com:443 -prexit 2>/dev/null \
| sed -n -e '/BEGIN\ CERTIFICATE/,/END\ CERTIFICATE/ p' \
| cert-print

# Owner: CN=....
# Issuer: ....
# ....
```

***

   This one is easy to use on the terminal:

```
$> echo '1\n2\n3\n' | while read -r; do echo $REPLY; done
1
2
3
```

***

   With...

```
while read line
do
        echo "$line"
done < "${1:-/dev/stdin}"
```

I got the following output:

```
Ignored 1265 characters from standard input. Use "-stdin" or "-" to tell how to handle piped
input.
```

Then decided with for:

```
Lnl=$(cat file.txt | wc -l)
echo "Last line: $Lnl"
nl=1

for num in `seq $nl +1 $Lnl`;
do
        echo "Number line: $nl"
        line=$(cat file.txt | head -n $nl | tail -n 1)
        echo "Read line: $line"
        nl=$[$nl+1]
done
```

***

Just test the number of arguments to your script, and test whether the first argumetn ($1) is a file. If false, use the stdin -:

```
#!/bin/bash
[ $# -ge 1 -a -f "$1" ] && input="$1" || input="-"
cat $input
```

***

@gniourf_gniourf's answer is the most correct one but uses a lot of bashisms and misses a corner case. Since this question is the top Google result, here is a POSIX compliant version:

```
#!/bin/sh

exec 3<&0

if [ $# -eq 0 ]; then
        set -- -
fi

for f in "$@"; do
        if { [ "$f" = - ] && exec <&3; } || exec < "$f"; then
                while IFS= read -r line; do
                        printf '%s\n' "$line"
                done
        fi
done
```

or if you want to be terse:

```
#!/bin/sh
exec 3<&0
[ $# -eq 0 ] && set -- -
for f; do
        { { [ "$f" = - ] && exec <&3; } || exec < "$f"; } &&
        while IFS= read -r line; do
                printf '%s\n' "$line"
        done
done
```

***

I don't find any of these answers acceptable. In particular, the accepted answer only handles the first command line parameter and ignores the rest. The Perl program that it is trying to emulate handles all the command line parameters. So the accepted answer doesn't even answer the question.

Other answers use Bash extensions, add unnecessary 'cat' commands, only work for the simple case of echoing input to output, or are just unnecessarily complicated.

However, I have to give them some credit, because they gave me some ideas. Here is the complete answer:

```
#!/bin/sh

if [ $# = 0 ]
then
        DEFAULT_INPUT_FILE=/dev/stdin
else
        DEFAULT_INPUT_FILE=
fi

# Iterates over all parameters or /dev/stdin
for FILE in "$@" $DEFAULT_INPUT_FILE
do
        while IFS= read -r LINE
        do
                # Do whatever you want with LINE here.
                echo $LINE
```

```
        done < "$FILE"
done
```
