

Bash-скрипты, часть 6: функции и разработка библиотек

<https://likegeeks.com/bash-functions/>

- Блог компании RUVDS.com,
- Настройка Linux,
- Серверное администрирование
- [Перевод](#)

[Bash-скрипты: начало](#)

[Bash-скрипты, часть 2: циклы](#)

[Bash-скрипты, часть 3: параметры и ключи командной строки](#)

[Bash-скрипты, часть 4: ввод и вывод](#)

[Bash-скрипты, часть 5: сигналы, фоновые задачи, управление сценариями](#)

[Bash-скрипты, часть 6: функции и разработка библиотек](#)

[Bash-скрипты, часть 7: sed и обработка текстов](#)

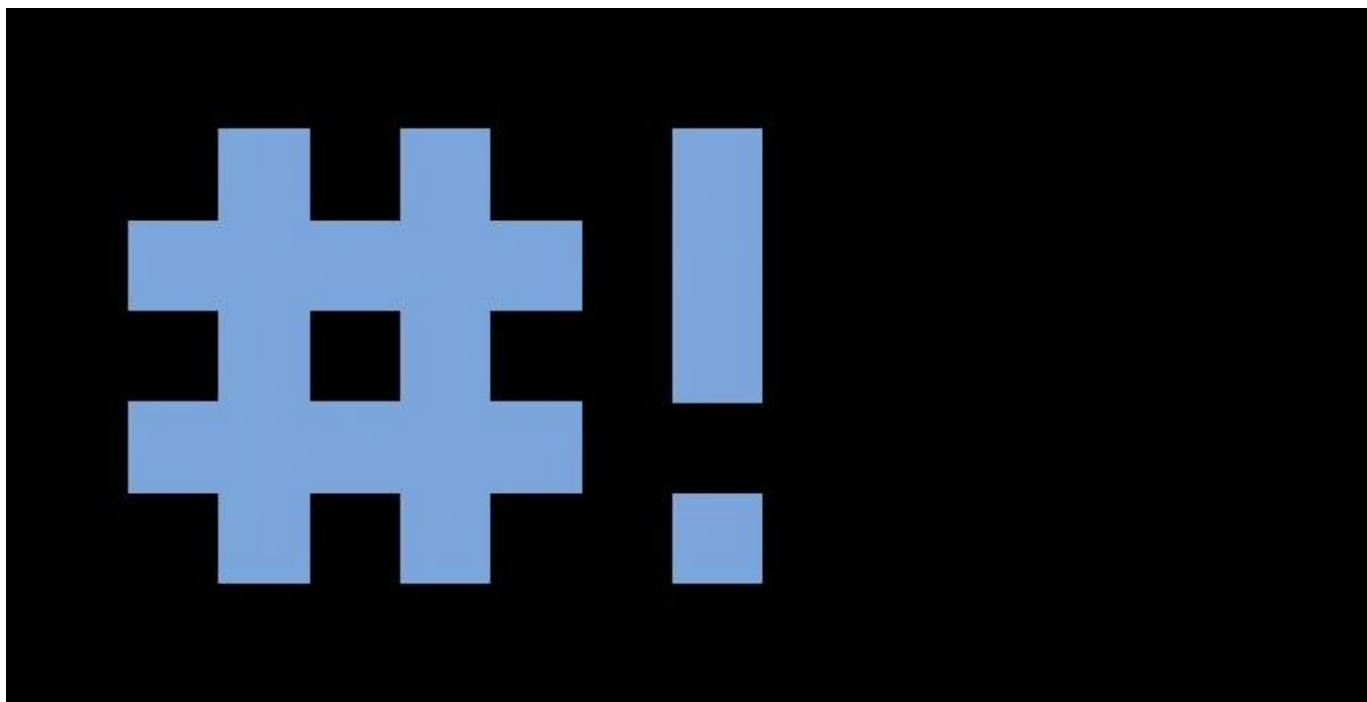
[Bash-скрипты, часть 8: язык обработки данных awk](#)

[Bash-скрипты, часть 9: регулярные выражения](#)

[Bash-скрипты, часть 10: практические примеры](#)

[Bash-скрипты, часть 11: expect и автоматизация интерактивных утилит](#)

Занимаясь разработкой bash-скриптов, вы рано или поздно столкнётесь с тем, что вам периодически приходится использовать одни и те же фрагменты кода. Постоянно набирать их вручную скучно, а копирование и вставка — не наш метод. Как быть? Хорошо бы найти средство, которое позволяет один раз написать блок кода и, когда он понадобится снова, просто сослаться на него в скрипте.



Оболочка bash предоставляет такую возможность, позволяя создавать функции. Функции bash — это именованные блоки кода, которые можно повторно использовать в скриптах.

Объявление функций

Функцию можно объявить так:

```
functionName {  
  
}
```

Или так:

```
functionName () {  
  
}
```

Функцию можно вызвать без аргументов и с аргументами.

Использование функций

Напишем скрипт, содержащий объявление функции и использующий её:

```
#!/bin/bash
```

```
function myfunc {
```

```
    echo "This is an example of using a function"
```

```
}
```

```
count=1
```



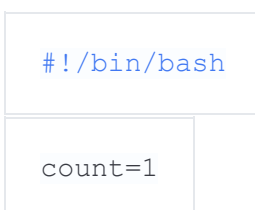
Здесь создана функция с именем `myfunc`. Для вызова функции достаточно указать её имя.

```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh  
This is an example of using a function  
This is an example of using a function  
This is an example of using a function  
This is the end of the loop  
This is an example of using a function  
End of the script  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The screenshot shows a terminal window titled `likegeeks@likegeeks-VirtualBox ~/Desktop`. The terminal displays the execution of a script `./myscript.sh`. The output of the script is: `This is an example of using a function` (repeated three times), `This is the end of the loop`, and `This is an example of using a function` (repeated once). The prompt `likegeeks@likegeeks-VirtualBox ~/Desktop $` is shown at the bottom.

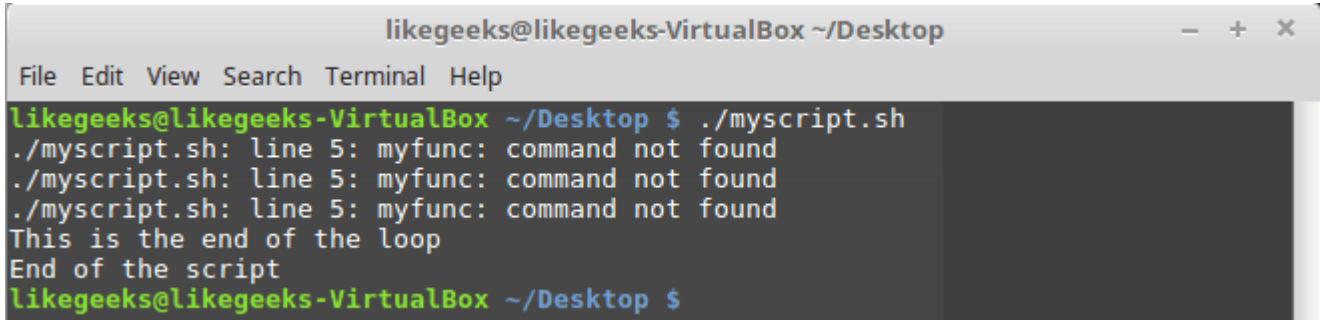
Результаты вызова функции

Функцию можно вызывать столько раз, сколько нужно. Обратите внимание на то, что попытавшись использовать функцию до её объявления, вы столкнётесь с ошибкой. Напишем демонстрирующий это скрипт:



```
while [ $count -le 3 ]  
  
do  
  
myfunc  
  
count=$(( $count + 1 ))  
  
done  
  
echo "This is the end of the loop"  
  
function myfunc {  
  
echo "This is an example of using a function"  
  
}  
  
echo "End of the script"
```

Как и ожидается, ничего хорошего после его запуска не произошло.



```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh  
./myscript.sh: line 5: myfunc: command not found  
./myscript.sh: line 5: myfunc: command not found  
./myscript.sh: line 5: myfunc: command not found  
This is the end of the loop  
End of the script  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Попытка воспользоваться функцией до её объявления

Придумывая имена для функций, учитывайте то, что они должны быть уникальными, иначе проблем не избежать. Если вы переопределите ранее объявленную функцию, новая функция будет вызываться вместо старой без каких-либо уведомлений или сообщений об ошибках. Продемонстрируем это на примере:

```
#!/bin/bash
```

```
function myfunc {
```

```
echo "The first function definition"
```

```
}
```

```
myfunc
```

```
function myfunc {
```

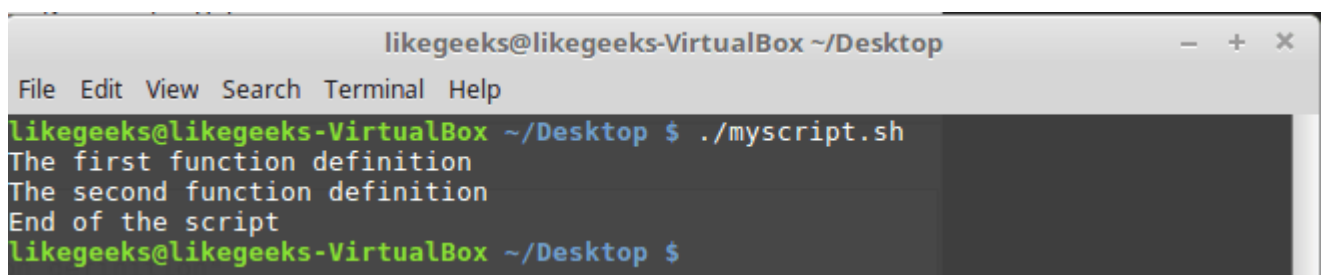
```
echo "The second function definition"
```

```
}
```

```
myfunc
```

```
echo "End of the script"
```

Как видно, новая функция преспокойно затёрла старую.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The first function definition
The second function definition
End of the script
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Переопределение функции

Использование команды return

Команда `return` позволяет задавать возвращаемый функцией целочисленный код завершения. Есть два способа работы с тем, что является результатом вызова функции. Вот первый:

```
#!/bin/bash
```

```
function myfunc {
```

```
read -p "Enter a value: " value
```

```
echo "adding value"
```

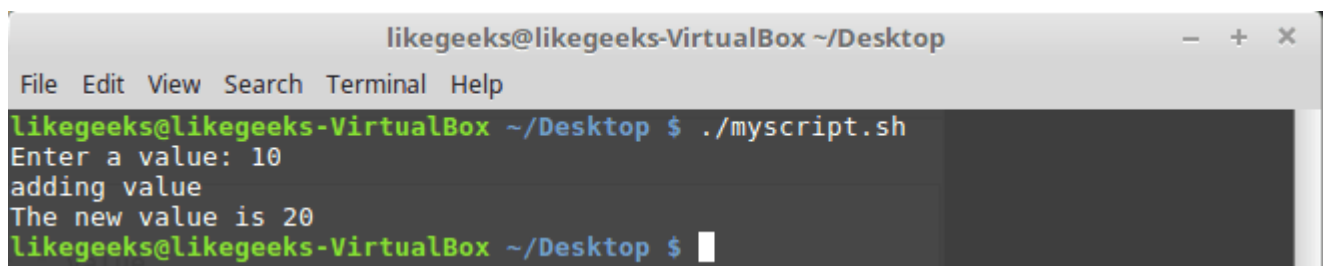
```
return $(( $value + 10 ))
```

```
}
```

```
myfunc
```

```
echo "The new value is $?"
```

Команда `echo` вывела сумму введённого числа и числа 10.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script './myscript.sh'. The output is as follows: 'Enter a value: 10', 'adding value', and 'The new value is 20'. The prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' is visible at the end of each line.

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
adding value
The new value is 20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Вывод значения, возвращаемого функцией

Функция `myfunc` добавляет 10 к числу, которое содержится в переменной `$value`, значение которой задаёт пользователь во время работы сценария. Затем она возвращает результат, используя команду `return`. То, что возвратила функция, выводится командой `echo` с использованием переменной `$?`. Если вы выполните любую другую команду до извлечения из переменной `$?` значения, возвращённого функцией, это значение будет утеряно. Дело в том, что данная переменная хранит код возврата последней выполненной команды.

Учтите, что максимальное число, которое может вернуть команда `return` — 255. Если функция должна возвращать большее число или строку, понадобится другой подход.

Запись вывода функции в переменную

Ещё один способ возврата результатов работы функции заключается в записи данных, выводимых функцией, в переменную. Такой подход позволяет обойти ограничения команды `return` и возвращать из функции любые данные. Рассмотрим пример:

```
#!/bin/bash

function myfunc {

    read -p "Enter a value: " value

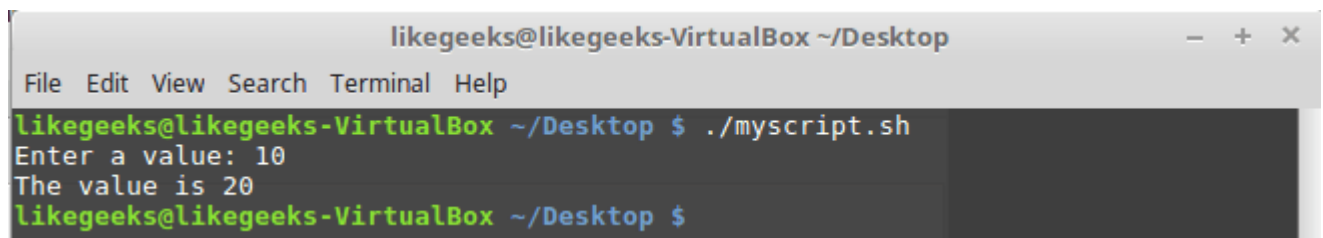
    echo $(( $value + 10 ))

}

result=$( myfunc)

echo "The value is $result"
```

Вот что получится после вызова данного скрипта.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command `./myscript.sh` being executed. The output is 'Enter a value: 10' followed by 'The value is 20'. The prompt returns to `likegeeks@likegeeks-VirtualBox ~/Desktop $`.

Запись результатов работы функции в переменную

Аргументы функций

Функции `bash` можно воспринимать как небольшие фрагменты кода, которые позволяют экономить время и место, избавляя нас от необходимости постоянно вводить с клавиатуры или копировать одни и те же наборы команд. Однако, возможности функций гораздо шире. В частности, речь идёт о передаче им аргументов.

Функции могут использовать стандартные позиционные параметры, в которые записывается то, что передаётся им при вызове. Например, имя функции хранится в параметре `$0`, первый переданный ей аргумент — в `$1`, второй — в `$2`,

и так далее. Количество переданных функции аргументов можно узнать, обратившись к переменной \$#. Если вы знакомы с [третьей частью](#) этого цикла материалов, вы не можете не заметить, что всё это очень похоже на то, как скрипты обрабатывают переданные им параметры командной строки.

Аргументы передают функции, записывая их после её имени:

```
myfunc $val1 10 20
```

Вот пример, в котором функция вызывается с аргументами и занимается их обработкой:

```
#!/bin/bash
```

```
function addnum {
```

```
if [ $# -eq 0 ] || [ $# -gt 2 ]
```

```
then
```

```
echo -1
```

```
elif [ $# -eq 1 ]
```

```
then
```

```
echo $(( $1 + $1 ))
```

```
else
```

```
echo $(( $1 + $2 ))
```

```
fi
```

```
}
```



```
echo -n "Adding 10 and 15: "
```

```
value=$(addnum 10 15)
```

```
echo $value
```

```
echo -n "Adding one number: "
```

```
value=$(addnum 10)
```

```
echo $value
```

```
echo -n "Adding no numbers: "
```

```
value=$(addnum)
```

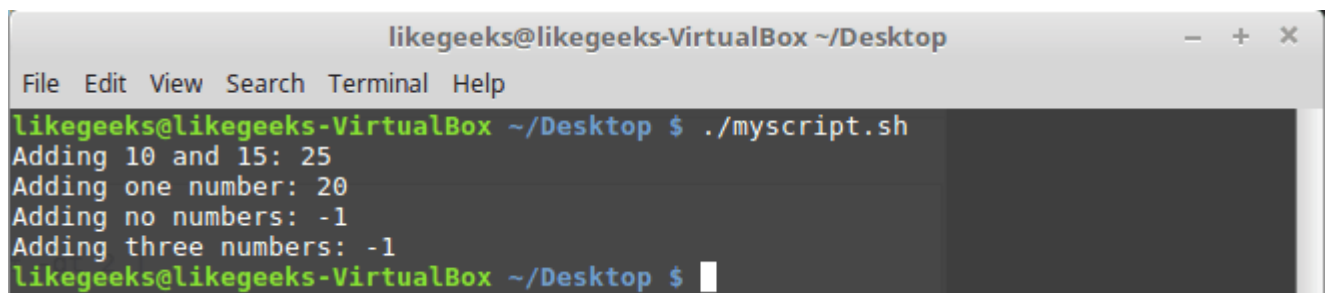
```
echo $value
```

```
echo -n "Adding three numbers: "
```

```
value=$(addnum 10 15 20)
```

```
echo $value
```

Запустим скрипт.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command './myscript.sh' being executed. The output is: 'Adding 10 and 15: 25', 'Adding one number: 20', 'Adding no numbers: -1', and 'Adding three numbers: -1'. The prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' is visible at the bottom.

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Adding 10 and 15: 25
Adding one number: 20
Adding no numbers: -1
Adding three numbers: -1
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Вызов функции с аргументами

Функция `addnum` проверяет число переданных ей при вызове из скрипта аргументов. Если их нет, или их больше двух, функция возвращает значение `-1`. Если параметр всего один, она прибавляет его к нему самому и возвращает результат. Если параметров два, функция складывает их.

Обратите внимание на то, что функция не может напрямую работать с параметрами, которые переданы скрипту при его запуске из командной строки. Например, напомним такой сценарий:

```
#!/bin/bash

function myfunc {

echo $(( $1 + $2 ))

}

if [ $# -eq 2 ]

then

value=$(( myfunc))

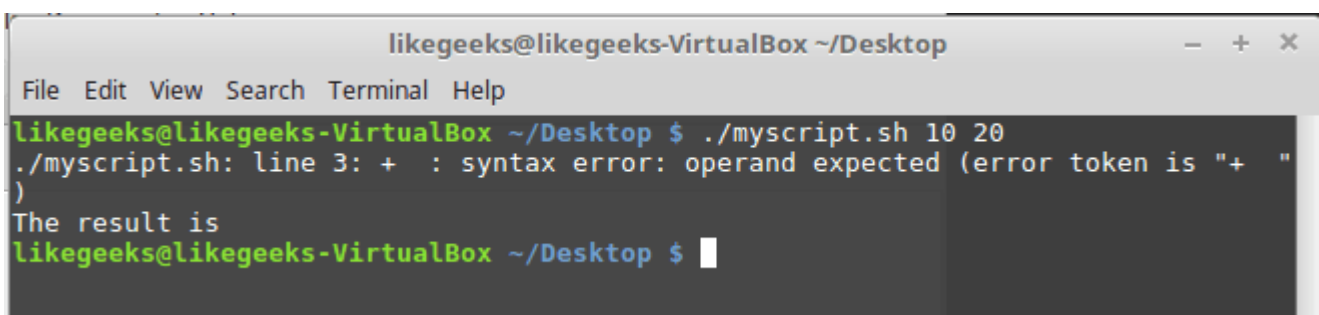
echo "The result is $value"

else

echo "Usage: myfunc a b"

fi
```

При его запуске, а точнее, при вызове объявленной в нём функции, будет выведено сообщение об ошибке.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the command './myscript.sh 10 20' being executed. The output is a syntax error: './myscript.sh: line 3: + : syntax error: operand expected (error token is "+ "') followed by 'The result is'. The prompt returns to the shell: 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

Функция не может напрямую использовать параметры, переданные сценарию

Вместо этого, если в функции планируется использовать параметры, переданные скрипту при вызове из командной строки, надо передать их ей при вызове:

```
#!/bin/bash

function myfunc {

    echo $(( $1 + $2 ))

}

if [ $# -eq 2 ]

then

    value=$(myfunc $1 $2)

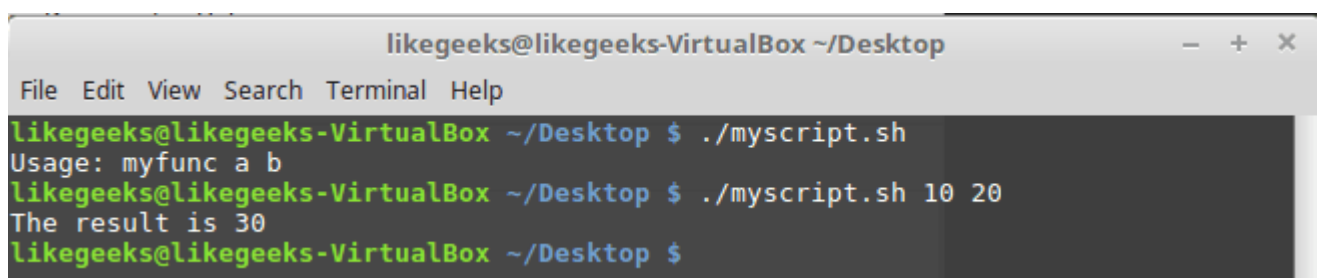
    echo "The result is $value"

else

    echo "Usage: myfunc a b"

fi
```

Теперь всё работает правильно.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Usage: myfunc a b
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh 10 20
The result is 30
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Передача функции параметров, с которыми запущен скрипт

Работа с переменными в функциях

Переменные, которыми мы пользуемся в сценариях, характеризуются областью видимости. Это — те места кода, из которых можно работать с этими переменными. Переменные, объявленные внутри функций, ведут себя не так, как те переменные, с которыми мы уже сталкивались. Они могут быть скрыты от других частей скриптов.

Существуют два вида переменных:

- Глобальные переменные.
- Локальные переменные.

Глобальные переменные

Глобальные переменные — это переменные, которые видны из любого места `bash`-скрипта. Если вы объявили глобальную переменную в основном коде скрипта, к такой переменной можно обратиться из функции.

Почти то же самое справедливо и для глобальных переменных, объявленных в функциях. Обращаться к ним можно и в основном коде скрипта после вызова функций.

По умолчанию все объявленные в скриптах переменные глобальны. Так, к переменным, объявленным за пределами функций, можно без проблем обращаться из функций:

```
#!/bin/bash
```

```
function myfunc {
```

```
value=$(( $value + 10 ))
```

```
}
```

```
read -p "Enter a value: " value
```

```
myfunc
```

```
echo "The new value is: $value"
```

Вот что выведет этот сценарий.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
The new value is: 20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Обращение к глобальной переменной из функции

Когда переменной присваивается новое значение в функции, это новое значение не теряется когда скрипт обращается к ней после завершения работы функции. Именно это можно видеть в предыдущем примере.

Что если такое поведение нас не устраивает? Ответ прост — надо использовать локальные переменные.

Локальные переменные

Переменные, которые объявляют и используют внутри функции, могут быть объявлены локальными. Для того, чтобы это сделать, используется ключевое слово `local` перед именем переменной:

```
local temp=$(( $value + 5 ))
```

Если за пределами функции есть переменная с таким же именем, это на неё не повлияет. Ключевое слово `local` позволяет отделить переменные, используемые внутри функции, от остальных переменных. Рассмотрим пример:

```
#!/bin/bash
```

```
function myfunc {
```

```
    local temp=$(( $value + 5 ))
```

```
    echo "The Temp from inside function is $temp"
```

```
}
```

```
temp=4
```

```
myfunc
```

```
echo "The temp from outside is $temp"
```

Запустим скрипт.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The Temp from inside function is 5
The temp from outside is 4
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Локальная переменная в функции

Здесь, когда мы работаем с переменной `$temp` внутри функции, это не влияет на значение, назначенное переменной с таким же именем за её пределами.

Передача функциям массивов в качестве аргументов

Попробуем передать функции в качестве аргумента массив. Сразу хочется сказать, что работать такая конструкция будет неправильно:

```
#!/bin/bash
```

```
function myfunc {
```

```
echo "The parameters are: $@"
```

```
arr=$1
```

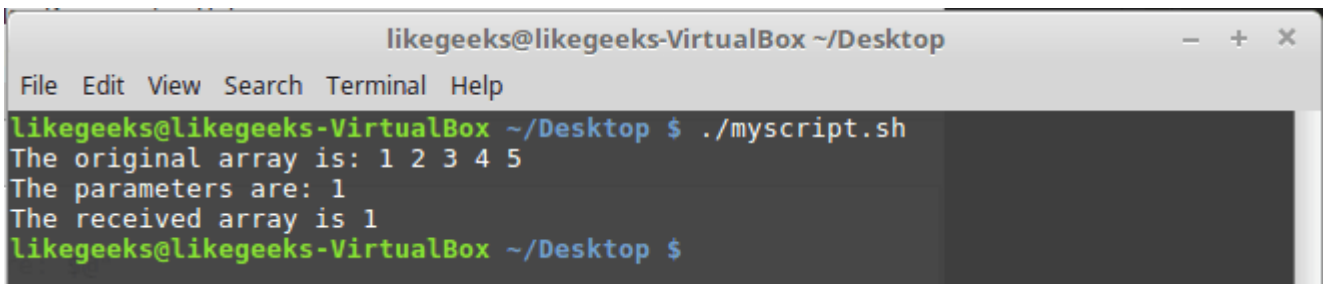
```
echo "The received array is ${arr[*]}"
```

```
}
```

```
myarray=(1 2 3 4 5)
```

```
echo "The original array is: ${myarray[*]}"
```

```
myfunc $myarray
```



The screenshot shows a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal output is as follows:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The original array is: 1 2 3 4 5
The parameters are: 1
The received array is 1
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Неправильный подход к передаче функциям массивов

Как видно из примера, при передаче функции массива, она получит доступ лишь к его первому элементу.

Для того, чтобы эту проблему решить, из массива надо извлечь имеющиеся в нём данные и передать их функции как самостоятельные аргументы. Если надо, внутри функции полученные ей аргументы можно снова собрать в массив:

```
#!/bin/bash
```

```
function myfunc {
```

```
    local newarray
```

```
    newarray=("$@")
```

```
    echo "The new array value is: ${newarray[*]}"
```

```
}
```

```
myarray=(1 2 3 4 5)
```

```
echo "The original array is ${myarray[*]}"
```

```
myfunc ${myarray[*]}
```

Запустим сценарий.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Сборка массива внутри функции

Как видно из примера, функция собрала массив из переданных ей аргументов.

Рекурсивные функции

Рекурсия — это когда функция сама себя вызывает. Классический пример рекурсии — функция для вычисления факториала. Факториал числа — это произведение всех натуральных чисел от 1 до этого числа. Например, факториал 5 можно найти так:

$$5! = 1 * 2 * 3 * 4 * 5$$

Если формулу вычисления факториала написать в рекурсивном виде, получится следующее:

$$x! = x * (x-1)!$$

Этой формулой можно воспользоваться для того, чтобы написать рекурсивную функцию:

```
#!/bin/bash
```

```
function factorial {
```

```
if [ $1 -eq 1 ]
```

```
then
```

```
echo 1
```



```
else
```

```
local temp=$(( $1 - 1 ))
```

```
local result=$(factorial $temp)
```

```
echo $(( $result * $1 ))
```

```
fi
```

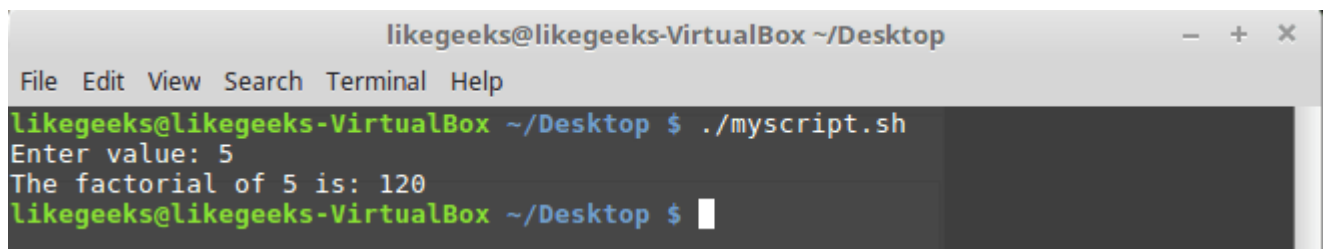
```
}
```

```
read -p "Enter value: " value
```

```
result=$(factorial $value)
```

```
echo "The factorial of $value is: $result"
```

Проверим, верно ли работает этот скрипт.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command './myscript.sh' being executed. The output is 'Enter value: 5' followed by 'The factorial of 5 is: 120'. The prompt returns to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

Вычисление факториала

Как видите, всё работает как надо.

Создание и использование библиотек

Итак, теперь вы знаете, как писать функции и как вызывать их в том же скрипте, где они объявлены. Что если надо использовать функцию, тот блок кода, который она собой представляет, в другом скрипте, не используя копирование и вставку?

Оболочка `bash` позволяет создавать так называемые библиотеки — файлы, содержащие функции, а затем использовать эти библиотеки в любых скриптах, где они нужны.

Ключ к использованию библиотек — в команде `source`. Эта команда используется для подключения библиотек к скриптам. В результате функции, объявленные в библиотеке, становятся доступными в скрипте, в противном же случае функции из библиотек не будут доступны в области видимости других скриптов.

У команды `source` есть псевдоним — оператор «точка». Для того, чтобы подключить файл в скрипте, в скрипт надо добавить конструкцию такого вида:

```
. ./myscript
```

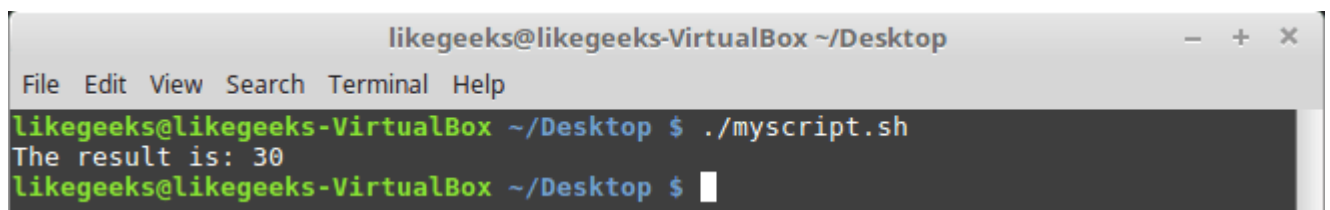
Предположим, что у нас имеется файл `myfuncs`, который содержит следующее:

```
function addnum {  
  
    echo $(( $1 + $2 ))  
  
}
```

Это — библиотека. Воспользуемся ей в сценарии:

```
#!/bin/bash  
  
. ./myfuncs  
  
result=$(addnum 10 20)  
  
echo "The result is: $result"
```

Вызовем его.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command `./myscript.sh` being executed, followed by the output `The result is: 30`. The prompt `likegeeks@likegeeks-VirtualBox ~/Desktop $` is visible at the bottom.

```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh  
The result is: 30  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Использование библиотек

Только что мы использовали библиотечную функцию внутри скрипта. Всё это

замечательно, но что если мы хотим вызвать функцию, объявленную в библиотеке, из командной строки?

Вызов bash-функций из командной строки

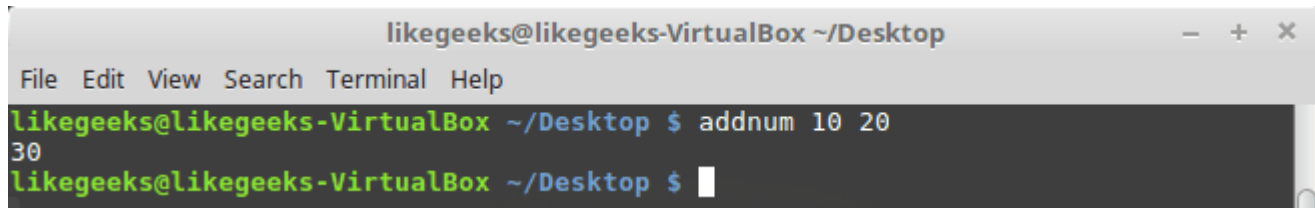
Если вы освоили [предыдущую часть](#) из этой серии, вы, вероятно, уже догадываетесь, что функцию из библиотеки можно подключить в файле `.bashrc`, используя команду `source`. Как результат, вызывать функцию можно будет прямо из командной строки.

Отредактируйте `.bashrc`, добавив в него такую строку (путь к файлу библиотеки в вашей системе, естественно, будет другим):

```
. /home/likegeeks/Desktop/myfuncs
```

Теперь функцию можно вызывать прямо из командной строки:

```
$ addnum 10 20
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command 'addnum 10 20' being entered and executed, resulting in the output '30'. The prompt changes from '\$' to '\$ ' after the command is executed.

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ addnum 10 20
30
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Вызов функции из командной строки

Ещё приятнее то, что такая вот библиотека оказывается доступной всем дочерним процессам оболочки, то есть — ей можно пользоваться в bash-скриптах, не заботясь о подключении к ним этой библиотеки.

Тут стоит отметить, что для того, чтобы вышеприведённый пример заработал, может понадобится выйти из системы, а потом войти снова. Кроме того, обратите внимание на то, что если имя функции из библиотеки совпадёт с именем какой-нибудь стандартной команды, вместо этой команды будет вызываться функция. Поэтому внимательно относитесь к именам функций.

Итоги

Функции в bash-скриптах позволяют оформлять блоки кода и вызывать их в скриптах. А наиболее часто используемые функции стоит выделить в библиотеки, которые можно подключать к скриптам, используя оператор `source`. Если же среди

ваших функций найдутся такие, без которых вы прямо таки жить не можете — библиотеки с ними можно подключить в файле `.bashrc`. Это позволит удобно пользоваться ими в командной строке или в других скриптах. Главное — чтобы имена ваших функций не совпадали с именами встроенных команд.

На сегодня это всё. В следующий раз поговорим об утилите `sed` — мощном средстве обработки строк.