Переменные, проверка условий

Вспомним наш первый скрипт:

```
#! /bin/bash
cp -r $1 $2
rm -rf $1
```

Попробуем сделать его ещё лучше.

Первым делом давайте добавим несколько комментариев. **Комментарии** — это текст, который не интерпретируется как инструкции для выполнения. Он нужен для того, чтобы человеку, который будет читать ваш скрипт, было проще разобраться.

Комментарии в bash обозначаются символом **хеш,** или **решётка: #.**
Строка после этого символа считается комментарием и не выполняется. Давайте откроем наш скрипт и добавим пару комментариев:

```
#! /bin/bash
# Copy folder into target folder
cp -r $1 $2
# Remove old folder
rm -rf $1
```

Если мы запустим скрипт, мы увидим, что он выполняется, как и раньше.

Комментарии могут быть длиннее одной строки, для этого в начале каждой строки мы добавляем символ хеш (#).

```
#! /bin/bash
# Copy folder into target folder
cp -r $1 $2
# Remove old folder
# -r remove directory
# -f do not prompt, ignore non-existent
rm -rf $1
```

Теперь можно быстрее разобраться, что делает скрипт.

Переменные в шелл-скриптах
Переменные в шелл-скриптах немного похожи на шелл-переменные. Первые существуют, только пока работает шелл-скрипт, вторые — только пока запущен процесс шелла.

Чаще всего переменные имеют строго заданный тип, например число или строка. Это определяет, сколько памяти мы отводим для переменной, какие действия можем или не можем с ней совершать. В bash же нам потребуется самим следить за тем, чтобы не попытаться умножить строку из слов на дату.

Давайте напишем маленький пример с переменными:

```
$!/bin/bash
```

Создаём переменную «имя», в другом языке это был бы тип данных «строка», потому что у нас здесь просто набор символов:

```
name=MacBook
```

Дальше создаём переменную «возраст». Это был бы тип данных «целое число»:

```
age=2
```

И давайте используем эти переменные как слова в предложении. Для вывода текста мы используем команду echo. И в кавычках пишем предложение. Там, где нам нужны значения переменных, подставляем имя переменной со знаком доллара:

```
echo "Hi! This is $name, I am $age years old"
```

Запускаем скрипт.

Но значение переменной мы можем не только задать вручную, но и получить, например из результата выполнения команды.

Определение текущей даты

Мы уже знаем команду *date*. Она, правда, выводит слишком много информации, но, сказав *man date,* мы сможем узнать, как форматировать вывод.

Для того чтобы узнать дату, нам нужна команда:

```
date +'%d-%m'
```

Теперь, чтобы получить переменную с нужными нам данными, мы можем использовать следующую конструкцию.

Так как нам нужно только определить переменную и вывести её значение, я воспользуюсь просто консолью, не буду писать скрипт:

```
today=`date +'%d-%m'`
```

Обратите внимание, это не одиночные кавычки, а специальные символы — бэктики (обратные апострофы). Вы найдёте их рядом с тильдой на клавиатуре:

```
echo $today
```

Или вот так:

```
today=$(date +'%d-%m')
echo $today
```

Как видите, результаты полностью равнозначны.

Неудобно, что пришлось писать две строчки, можно ли из этих команд сделать однострочник?

Чаще всего однострочник можно сделать из любых подручных материалов.
Мы умеем передавать данные через пайп, но это относится только к тем случаям, когда у нас есть вывод и ввод. Здесь мы хотим проверить результат работы первой команды, вывести, что записалось, в переменную.
Такие независимые действия можно перечислять через точку с запятой, например:

```
today=$(date +'%d-%m') ; echo $today
```

Проверки перед копированием

Для разнообразных проверок по заданным условиям есть специальная команда *test*.

Она умеет сравнивать строки, числа, проверять, существуют ли файлы или папки и выставлены ли на них нужные значения.
Это одна из самых полезных для скриптов программ.
Давайте проверим самый простой пример использования:

```
test a=a
```

На самом деле, команда только что сравнила два значения и сказала нам о том, что они равны, но мы этого не видим. Мы уже сталкивались с подобными случаями, например команда *mkdir* после создания директории ничего нам не сообщала.

Но когда директорию создать не удавалось, например, из-за нехватки прав, она выводила ошибку в стандартный поток ошибок, может быть, тут такой же случай?

```
test a=b
```

Видимо, нет.

Дело в том, что когда команда завершается, у неё есть так называемый exit code — код возврата. Это число, которое говорит о том, как выполнился наш процесс: успешно или с ошибкой.

Код возврата

Узнать код возврата при помощи переменной:

```
$?
```

Это так называемый специальный параметр. Он формируется по такой же схеме, как и позиционные параметры.

$ — обозначение переменной плюс некий шаблон, вместо которого подставляется какое-то значение. Например, при помощи специального параметра

```
$*
```

Звёздочка тут будет иметь почти то же значение, как и в контексте поиска по шаблону или шелл-глоббинга.
Вместо знака вопроса подставляется код возврата последнего выполненного действия.

Ещё один пример специального параметра:

```
$#
```

С помощью него мы можем получить количество позиционных аргументов, которые были переданы.

Итак, код возврата — это число. В bash мы будем сталкиваться со следующими кодами:
0 — успешное выполнение, от 1 до 255 идут коды возврата, которые обозначают различные ошибки. Во-первых, они стандартны для большинства систем Linux, а во-вторых, наиболее часто используются. Коды возврата от двух до 128 и от 130 до 255 могут переназначаться пользователем, если ему требуется, чтобы его скрипт отдавал какой-то свой код возврата с особенным смыслом.

Пример
0. Код успешного выполнения. То есть когда процесс завершается, выполнив всё, что он хотел сделать, — он возвращает 0.

```
touch newfile
echo $?
```

1. Ошибка. Этот код используется для самых различных ошибок, например если при создании файла у нас не хватает доступа или если grep ничего не нашёл в файле.

```
grep "aaaa" newfile
echo $?
```

2. Ошибка использования встроенных команд шелла (например, отсутствие аргументов или опций). Давайте, например, попробуем вызвать команду *kill* без параметров:

```
kill
echo $?
```

126. Команда, которую мы вызвали, не может быть исполнена — ошибка доступов, или просто мы попытались исполнить неисполнимое. На нашем свежесозданном файле нет права выполнения, так что попробуем выполнить его:

```
./newfile
echo $?
```

127. Команда не найдена.

Ну здесь всё просто, давайте попробуем ввести в консоль какой-нибудь бессмысленный набор символов и нажмём enter:

```
Sssss
echo $?
```

Такой же код возврата будет, если мы попытаемся выполнить файл, для которого есть право на исполнение, но его нет в PATH (при этом не укажем ./)

Вернёмся к команде *test* и узнаем, равно ли a b.

```
test a=b
echo $?
```

Всё-таки не равно, как мы и предполагали. Зато:

```
test a=a
echo $?
```

a равно a.

Мы можем проверять строки не только на равенство (=), но и на неравенство (`!=`). Обратите внимание, что если мы проверяем на неравенство:

```
test a!=b
echo $?
```

Код возврата 0 будет отдаваться в том случае, когда строки не равны. И, соответственно, 1, когда строки будут равными.

Ещё мы можем сравнить строку с нулём. -z завершится успешно, когда строка равна нулю, и ошибкой, когда не равна:

```
test -z a
```

-n, наоборот, проверяет, что строка не равна нулю, и завершается успешно, когда мы передаём команде test какое-то слово.

Мы можем проверять, что одно число другому:
-eq (равно)
-ne (не равно)
-ge (больше либо равно)
-gt (строго больше)
-le (меньше либо равно)
-lt (строго меньше)
Легко заметить, что у этих буквенных сочетаний довольно простая логика составления, они получаются сложением первых букв: equal, not equal, greater or equal than, greater than и так далее.
Это просто сравнение двух чисел, базовая арифметика, не будем останавливаться на этом подробнее.

Проверить, что файл существует, проверить, является ли объект директорией, выставлены ли на объект флаги доступа (SUID/SGID/sticky bit), не пуст ли файл и выставлены ли на файл различные права доступа (чтение, запись, выполнение).
-e *FILEFILE* exists
-f *FILEFILE* exists and is a regular file
-d *FILEFILE* exists and is a directory
-g *FILEFILE* exists and is set-group-ID
-k *FILEFILE* exists and has its sticky bit set
-r *FILEFILE* exists and read permission is granted
-s *FILEFILE* exists and has a size greater than zero
-u *FILEFILE* exists and its set-user-ID bit is set
-w *FILEFILE* exists and write permission is granted
-x *FILEFILE* exists and execute permission is granted

Букву, которая служит ключом для нужного действия, вы всегда сможете вспомнить, набрав *man test*.

Пример проверки

```
test -k /tmp
echo $?
```

На директорию /tmp всё ещё выставлен стики бит.