

10. Основы разработки shell-сценариев для командного интерпретатора BASH

1. Введение

1.1. Задачи, решаемые с помощью shell-сценариев

Shell-сценарии (или как их еще называют, «скрипты»), позволяют автоматизировать деятельность системного администратора, избавив его от выполнения рутинной повторяющейся работы. С помощью них могут решаться задачи, связанные с мониторингом и обслуживанием системы, выполняться операции по резервному копированию и восстановлению данных, а также многие другие. И даже если вы не планируете создавать собственные shell-сценарии, крайне желательно понимать синтаксис языка BASH и уметь вносить изменения в готовые программы, ведь ряд системных сценариев, участвующих в процессе инициализации системы, написаны именно на BASH и чтобы их модифицировать, требуются навыки BASH-программиста.

В то же время существует ряд задач, для решения которых использование shell не совсем оправдано. Подумать об альтернативе для BASH стоит, если

- особенно важна скорость исполнения;
- задача связана с выполнением математических вычислений, с плавающей запятой, вычислений с повышенной точностью, комплексных чисел и т.п.
- структурирование является жизненной необходимостью (контроль за типами переменных, прототипами функций и т.п.);
- предполагается работа с многомерными массивами;
- необходимо работать со структурами данных, такими как связанные списки или деревья;
- необходимо предоставить графический интерфейс с пользователем (GUI);
- необходим прямой доступ к аппаратному обеспечению;
- необходимо использовать внешние библиотеки;
- требуется создать программу с «закрытым» кодом (скрипты представляют собой исходные тексты программ, доступные для всеобщего обозрения);

Если выполняется хотя бы одно из вышеперечисленных условий¹, то лучше обратиться к высокоуровневым компилирующим языкам, таким как C, C++. Но это вовсе не означает, что задачу нельзя решить с помощью shell.

¹ Также стоит обратить внимание на размер сценария. Если он превышает 500 строк, то скорее всего, есть смысл реализовать поставленную задачу на другом языке.

1.2. Перед началом разработки

Перед тем, как приступить к написанию сценария, необходимо

- чётко сформулировать поставленную задачу;
- разбить её на подзадачи;
- для каждой подзадачи определить алгоритм для её решения.

Никогда не приступайте к разработке сценария, не проделав трёх приведённых выше шагов, при мало-мальски сложной задаче вы зайдёте в тупик.

1.3. Конечно же, “Hello, World”

Давайте напишем сценарий, выводящий традиционную фразу “Hello, World!”

Для этого создадим файл, содержащий внутри следующие строки:

```
#!/bin/bash

# Тестовый сценарий

echo "Hello, World"
exit 0
```

Рис. 1.1. Сценарий “Hello, World”

При сохранении сценария может возникнуть соблазн назвать его “test”. Однако, этого делать не следует. Дело в том, что у нас в системе уже есть исполняемый файл с именем “test”. Убедиться в этом можно, введя команду “which test”. И появление ещё одного такого же исполняемого файла может привести к проблемам. Поэтому, выбирая имя, проверяйте, не занято ли оно другой программой.

Для сценариев рекомендуется использовать расширение файлов “.sh” Оно необходимо исключительно для удобства и на функциональность самого сценария никакого влияния не оказывает.

Давайте разберём, что означают набранные нами строчки? Символ “#” в тексте сценария означает “комментарий” (т.е. некая информация, которую хотел от себя добавить разработчик). Всё, что стоит справа от него, интерпретатор игнорирует.

Однако, есть одно исключение: если в первой строке файла присутствует последовательность “#!” то эта строка указывает, какая программа является обработчиком нашего сценария. (Т.е. кто его будет интерпретировать при запуске). В англоязычной литературе для обозначения этой строки применяется термин “shebang”, в русскоязычной литературе устоявшегося термина нет, поэтому будем пользоваться просто словом “обработчик”.

Нужно ли указывать обработчик при написании сценария? Если попробовать обойтись без этой строки, то обнаружится, что сценарий работает и так. Это потому, что наш сценарий написан на BASH и BASH же является обработчиком по умолчанию. Однако, если сценарий окажется перенесён в систему, где BASH не будет указан обработчиком по умолчанию, работать без строки “shebang” он не будет. Тоже самое произойдет, если обработчик по умолчанию BASH, но сценарий написан на другом языке, скажем “Perl”. Чтобы исключить проблемы такого рода, строку с указанием обработчика рекомендуется добавлять всегда².

При разработке сценариев желательно не пренебрегать комментариями и в начале сценария вставлять две-три строчки, поясняющие, для чего именно предназначен сценарий и есть ли у него какие-либо особенности в работе. Это поможет в дальнейшем вашим коллегам, и, возможно, вам самим. Дело в том, что если у вас возникнет необходимость внести изменения в сложный сценарий, который вы писали год или два назад, то, скорее всего, быстро вспомнить, как именно он работает, не получится. Комментарии смогут в этом помочь.

Следом за указанием обработчика идет команда `echo`, которая отправляет переданную ей в качестве параметра строку в стандартный вывод (т.е. отображает на экране) и команда `exit`. `exit` инициализирует специальную переменную “\$?”, которая хранит код возврата завершившейся программы. Нулевое значение этой переменной означает, что команда завершилась успешно. Ненулевое (конкретное значение будет зависеть от реализации программы) означает, что при выполнении команды возникли ошибки. Сравните:

```
[demo@localhost ~] $ ls /tmp
```

² Существует рекомендация вместо “#!/bin/bash” писать “#!/usr/bin/env bash”. Это связано с повышением переносимости скриптов. В случае, если BASH расположен не в директории “/bin” в первом случае скрипт не будет работать.

```
[demo@localhost ~] $ echo $?  
0
```

и

```
[demo@localhost ~] $ ls /zzzzz  
[demo@localhost ~] $ echo $?  
1
```

Таким образом, наш сценарий всегда завершается с кодом возврата, равным нулю (успех).

В самом сценарии код возврата можно применять в конструкциях “команда_1 && команда_2” и “ команда_1 || команда_2”. В первом случае команда_2 выполнится только в том случае, если команда_1 завершилась успешно (вернула “0”). Во втором случае - наоборот: команда_2 выполнится только в том случае, если команда_1 завершилась неуспешно. Например:

```
mountpoint /mnt > /dev/null || mount /dev/sdb1 /mnt
```

Рис. 1.2

Команда mountpoint вернет значение “0” (успех) если директория /mnt уже является точкой монтирования (а значит, монтировать туда ещё одну файловую систему нельзя). В противном случае мы получим “1” и благодаря конструкции “||” у нас выполнится команда mount.

1.4. Условия, необходимые для запуска сценария

Для запуска программы (в том числе и сценария) необходимо, чтобы на файл был установлен атрибут «право на исполнение», обозначаемый буквой «x»³. При этом расширение файла может быть, в общем-то, любым, операционная система не опирается на расширения файлов при их запуске. Однако, чтобы не было путаницы, лучше присваивать расширение “.sh”

³ На самом деле одного права на исполнение будет недостаточно. Для того, чтобы исполнять сценарий, интерпретатор должен его читать. Поэтому для успешного выполнения необходимо сочетание атрибутов «право на чтение» и «право на исполнение».

Без права на исполнение запустить сценарий также возможно, передав его в качестве параметра программе-обработчику, например:

```
[demo@localhost ~]$ /bin/bash /service/backup.sh
```

Добавить или удалить атрибуты можно с помощью команды `chmod`, например, так:

```
[demo@localhost service]$ chmod u+x backup.sh  
[demo@localhost service]$ ./backup.sh
```


Обратите внимание, что при запуске сценария мы использовали комбинацию символов `./`. В местах команды, где подразумевается указание пути, символ `.` используется для обозначения текущей директории. Почему нельзя просто ввести команду `service.sh` как, например, в случае, с командой `id`? Дело в том, исполняемый файл команды `id` находится в директории `/bin`. А наш сценарий, согласно примеру - в директории `/service`. При вводе команды без указания пути интерпретатор проверяет, является ли команда встроенной и если нет, то осуществляет поиск её исполняемого файла. Причем ищет он, разумеется, не во всех директориях (это было бы слишком долго, накладно и неправильно с точки зрения безопасности), а лишь в тех директориях, список которых хранится в переменной окружения `$PATH`:

```
[root@localhost ~]# echo $PATH  
/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
```

Рис.1.3. Вывод значений переменной окружения `$PATH`

Переменная `$PATH` содержит список директорий, разделённых двоеточием. Т.е. если нам нужно запускать нашу команду без указания пути, то нужно либо скопировать её в одну из перечисленных директорий, либо добавить директорию `/service` к списку. Например, так:

```
[root@localhost ~]# export PATH=$PATH:/service
```



Данная команда сохранит своё действие до перезагрузки системы. Если вы хотите, чтобы эффект сохранился после перезагрузки нужно отредактировать файл `.bash_profile`, расположенный в домашней директории пользователя

Упражнение 1.1. Обработчик сценария

В первой строчке сценария вы видите следующую конструкцию: “#!/bin/expect”
Является ли этот сценарий сценарием, написанным на языке BASH?

- ☐ Да
- ☐ Нет

Упражнение 1.2. Применение конструкции "&&"

В каком случае мы увидим на экране слово «GOOD» при использовании следующей конструкции:

```
$ ping -c2 195.34.32.11 &> /dev/null && echo "GOOD"
```

- ☐ Если сетевой узел с адресом 195.34.32.11 доступен
- ☐ Если сетевой узел с адресом 195.34.32.11 не доступен

Зачем понадобилось указывать параметр “-c2” команде ping? _____

Упражнение 1.3. Использование переменной \$PATH

В системе есть сценарий с именем getstat.sh, лежащий в директории /service
Возможно ли запустить его командой “getstat.sh” (т.е. без указания пути) если переменная \$PATH содержит следующее значение:

```
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin
```

- ☐ Да
- ☐ Нет

Упражнение 1.4. Интерпретация прав доступа

Сценарий backup.sh имеет следующие права доступа:


```
-rwxr-x--x. 1 monty backing 67 Feb 2 15:11 backup.sh
```

- Сможет ли его запустить владелец? _____
- Сможет ли его запустить пользователь chip, являющийся участником группы backing? _____
- Сможет ли его запустить пользователь dale, не являющийся участником группы backing? _____

Ответы к заданиям

Упражнение 1.1.

В первой строчке сценария вы видите следующую конструкцию: “#!/bin/expect”
Является ли этот сценарий сценарием, написанным на языке BASH?

- ☐ Да
- ☐ Нет

Упражнение 1.2.

В каком случае мы увидим на экране слово «GOOD» при использовании следующей конструкции:

```
$ ping -c2 195.34.32.11 &> /dev/null && echo "GOOD"
```

- ☐ Если сетевой узел с адресом 195.34.32.11 доступен
- ☐ Если сетевой узел с адресом 195.34.32.11 не доступен

Зачем понадобилось указывать параметр “-c2” команде ping? Без указания этого параметра команда ping будет работать до тех пор, пока ее не прервут. В то время, как с данным параметром она самостоятельно завершиться после двух опросов.

Упражнение 1.3.

В системе есть сценарий с именем getstat.sh, лежащий в директории /service
Возможно ли запустить его командой “getstat.sh” (т.е. без указания пути) если переменная \$PATH содержит следующее значение:

```
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin
```

- ☐ Да
- ☐ Нет

Упражнение 1.4.

Сценарий backup.sh имеет следующие права доступа:

```
-rwxr-x--x. 1 student backing 67 Feb 2 15:11 backup.sh
```

- Сможет ли его запустить владелец? Да
- Сможет ли его запустить пользователь chip, являющийся участником группы backing? Да

Сможет ли его запустить пользователь dale, не являющийся участником группы backing? Нет, т.к. одного права на исполнения недостаточно, необходимо также наличие права на чтение.

2. Использование переменных

Переменная - это именованная (обладающая собственным именем) область памяти, используемая для доступа к данным. В свою очередь, данные, расположенные в переменной называются значением этой переменной.

В своём сценарии мы можем использовать три категории переменных:

- Переменные среды (environment variables) Иногда их также называют переменные окружения. Эти переменные не требуют предварительной инициализации. Их значения задаются, например, сценариями оболочки (в частности, /etc/profile). Они содержат какую-либо информацию о настройках системы и могут использоваться всеми запускаемыми процессами.
- Переменные интерпретатора. В нашем случае - это переменные BASH. Они также не требуют предварительной инициализации. Их можно использовать для получения информации о настройках самого BASH, например, данные о его версии.
- Собственные переменные. Переменные, объявленные в тексте сценария для хранения каких-либо данных. Это особенно удобно, когда некоторое значение многократно используется в тексте сценария. В этом случае нет необходимости подставлять его в команду каждый раз, достаточно единожды объявить соответствующую переменную.

Таблица 2.1. Некоторые переменные окружения

Переменная	Значение
\$HOME	Расположение домашнего каталога текущего пользователя
\$USER	Имя текущего пользователя
\$PATH	Список директорий для поиска исполняемых программ
\$SHELL	Командный интерпретатор по-умолчанию
\$BASH	Расположение командного интерпретатора BASH
\$PWD	Текущая директория
\$EDITOR	Путь к файлу, вызываемому в качестве редактора по-умолчанию (например, такими программами, как crontab). Если не установлена, то используется редактор vi
HISTSIZE	Размер истории команд (по умолчанию 1000)
MAIL	Путь к файлу, содержащему очередь почтовых сообщений пользователя
\$HOSTNAME	Локальное имя хоста

Узнать, присутствует ли в системе переменная окружения с указанным именем, можно с помощью команды `printenv`, например:

```
[demo@localhost ~]$ printenv HOME      # существующая переменная
/home/demo
[demo@localhost ~]$ printenv NONEXIST  # несуществующая переменная
[demo@localhost ~]$
```

Рис. 2.1 Использование `printenv`

Команда `printenv`, запущенная без параметров, отобразит список всех имеющихся в системе переменных окружения.

Если необходимо, вы можете добавлять к имеющимся переменным окружения свои собственные, используя файл `/etc/environment` Формат этого файла простой: каждая строка представляет собой пару "имя=значение" Например: `"R_COUNTER=150"`

Список всех переменных окружения, известных конкретному процессу находится в файле `environ`. Прочитать этот файл можно, к примеру, так:

```
[demo@localhost ~]$ sleep 300 &  # создадим новый процесс
[1]  4245
[demo@localhost ~]$ cat /proc/4245/environ | tr '\0' '\n'
XDG_VTNR=1
XDG_SESSION_ID=1
HOSTNAME=localhost.localdomain
...
```

Рис. 2.2. Получение списка переменных окружения процесса

Таблица 2.2. Некоторые переменные BASH

Переменная	Значение
\$BASH_VERSION	Версия BASH
\$HISTIGNORE	Разделенный двоеточием список команд, которые не должны сохраняться в истории команд. Команды должны быть указаны в том виде, в котором они будут вводиться. Если параметры у команды всё время разные, то все возможные значения можно указать с помощью звёздочки, например, "ls *" Если звёздочка будет единственным символом, хранящимся в переменной HISTIGNORE, то никакие команды не будут попадать в историю.
\$PS1	Переменная, содержимое которой определяет, как будет выглядеть приглашение командной строки (Prompt String One).

\$PS2	Используется, когда интерпретатору необходимо указать вам, что он продолжает ждать ввода на новой строке. По умолчанию это ">".
\$PS3	Переменная определяет текст приглашения, выводимого командой "select" (рассматривается далее)
\$PS4	Задаёт символ, который отображается при запуске сценария в режиме трассировки (рассматривается далее). По умолчанию это "+".

Полный перечень этих переменных можно найти в man-странице по BASH (man 1 bash), раздел - "Shell Variables"

2.1. Имена переменных

При добавлении переменной в сценарий, нужно определиться с её именем. Имя, с одной стороны не должно быть слишком длинным (писать в сценарии каждый раз что-то типа "\$INSTANCE_OF_TEMPORARY_DATA" вы довольно скоро устанете) и слишком коротким (понять, что хранит переменная с именем "\$A" можно только, изучив предварительно текст сценария). Присваивая имя, вы должны помнить о следующих правилах:

- Имя переменной чувствительно к регистру. Т.е. \$HOST и \$Host - разные переменные.
- Имя переменной может содержать латинские буквы, цифры и знак подчеркивания, при этом начинаться оно может с латинской буквы либо знака подчеркивания.

Иногда, объявляя переменную, стоит убедиться, не была ли ранее объявлена переменная с таким именем? Это можно сделать с помощью команды `declare`⁴, вызванной с опцией `-p`:

```
[demo@localhost ~]$ K1=100
[demo@localhost ~]$ declare -p K1  # Переменная K1 была объявлена ранее
```

⁴ Команда `printenv` для этих целей не подходит. В вашем окружении могут присутствовать переменные, которые объявлены, но при этом переменными окружения не являются. Например, если выполнить команду `echo $PPID`, то вы увидите, что переменная проинициализирована, однако `printenv | grep PPID` её не покажет.

```
declare -- K="100"  
[demo@localhost ~] declare -p K2    # Переменная K2 ранее не объявлялась  
-bash: declare: K2: not found
```

Рис. 2.3 Проверка существования переменной

Как видно из рисунка 2.3, переменная объявляется довольно просто. Достаточно указать её имя и справа от знака равенства - её значение.



Обратите внимание на то, что ни слева, ни справа от знака равенства не должно стоять пробелов.

K1 =100 # приведёт к ошибке

K1= 100 # приведёт к ошибке

K1 = 100 # приведёт к ошибке

Второй способ объявить переменную - это использовать команду "declare"⁵. Вместо "K1=100" можно написать "declare K1=100". Ссылаться на объявленную ранее переменную можно по её имени, перед которым ставится знак "\$", обозначающий, что символы справа - это имя переменной:

```
[demo@localhost ~]$ echo $K1  
100
```

Рис. 2.4.Обращение к переменной

Если переменную необходимо использовать в строке без пробелов, то её имя требуется обозначить фигурными скобками, иначе интерпретатор не сможет понять, где оканчивается её имя. Например:

```
[demo@localhost ~]$ NAME=John  
[demo@localhost ~]$ echo Your Name Is $NAME Right?  
Your Name Is John Right?  
[demo@localhost ~]$ echo YourNameIs$NAMERight?  
YourNameIs  
[demo@localhost ~]$ echo YourNameIs${NAME}Right?  
YourNameIsJohnRight?
```

⁵ Иногда при объявлении переменных в сценариях встречается команда typeset. В BASH она абсолютно идентична по своим возможностям команде declare, но при этом считается устаревшей. Однако, она используется в некоторых других оболочках, например, в ksh. Если вам нужна совместимость именно с этой оболочкой, используйте typeset. Если в переносимости сценария необходимости нет - используйте declare.

```
[demo@localhost ~]$ echo YourNamels"$NAME"Right?      # тоже сработает
YourNamelsJohnRight?
```

Рис. 2.5.

При этом, такое объявление не подразумевает указание типа переменной, как это происходит, скажем, при объявлении переменной в языке С. Дело в том, что в отличие от некоторых других языков (например, упомянутого выше С), BASH поддерживает всего два типа данных: целочисленные и строковые. Различает он их довольно просто: если значение переменной состоит только из цифр - переменная считается целочисленной. Если же среди символов присутствуют какие-либо знаки, например, точка, запятая, а также любой другой алфавитный символ, кроме символов, обозначающих знак числа ("-" или "+") то такая переменная считается строкой.

Команда `declare` обладает рядом полезных возможностей. С помощью неё можно сделать переменную доступной только для чтения:

```
[demo@localhost ~]$ declare A=10
[demo@localhost ~]$ declare -r B=10
[demo@localhost ~]$ A=20
[demo@localhost ~]$ B=20
bash: B: readonly variable
```

Рис.2.6. Переменная только для чтения

В этом случае значение переменной будет невозможно изменить (а также удалить переменную командой `unset`). Это предусмотрено на случай, когда необходимо застраховаться от ситуации, когда значение переменной сменится, например, из-за некорректно написанного кода.

Также с помощью `declare` была предпринята попытка (впрочем, довольно слабая) объявлять переменные определенного типа. Например, с помощью опции `-i` можно указать, что это целочисленная переменная (integer). Теперь, если попытаться присвоить такой переменной значение с плавающей точкой, то это приведёт к ошибке.

```
[demo@localhost ~]$ declare -i C
[demo@localhost ~]$ C=100
[demo@localhost ~]$ C=7.7
bash: 7.7: syntax error: invalid arithmetic operator
[demo@localhost ~]$ C="hello"
```



```
[demo@localhost ~]$ echo $C  
0
```

Рис.2.7. Объявление целочисленной переменной

Однако, переменной, объявленной как целочисленная, по-прежнему можно присвоить букву или слово. Ошибки это не вызовет, просто её значение будет равным нулю.

У каждой переменной есть два признака: время жизни и область видимости. Если переменная объявлена в тексте сценария, то её время жизни длится с момента объявления и до окончания работы сценария. (Либо до уничтожения переменной с помощью специальной, предназначенной для этого командой "unset"). Как только сценарий завершил работу, объявленные в нём переменные будут уничтожены.

Что касается области видимости переменной, то переменная, объявленная внутри сценария считается глобальной, но только в пределах этого самого сценария. Вне его пределов она недоступна, поэтому в общесистемном смысле объявляемые переменные являются локальными, если не предпринимать никаких специальных действий.

Для того, чтобы сделать переменную "видимой" во всех дочерних процессах, её следует объявлять со специальным словом "export". Поставим эксперимент. Объявим переменную A, затем запустим ещё один экземпляр командного интерпретатора и попробуем в нём отобразить значение переменной:

```
[demo@localhost ~]$ A=100  
[demo@localhost ~]$ echo $A  
100  
[demo@localhost ~]$ bash  
[demo@localhost ~]$ echo $A
```

Рис. 2.8. Проверка видимости локальной переменной

В результате мы не увидим значения переменной, т.к. её область видимости ограничена той копией bash, в которой мы объявили переменную A. В том, что мы сейчас находимся в другом экземпляре интерпретатора, можно убедиться с помощью команды pstree.

Завершим дочерний shell и объявим еще одну переменную - B, но теперь уже глобальную:

```
[demo@localhost ~]$ exit
[demo@localhost ~]$ export B=200
[demo@localhost ~]$ echo $B
200
[demo@localhost ~]$ bash
[demo@localhost ~]$ echo $B
200
```

Рис. 2.9. Проверка видимости глобальной переменной

Как видно из рисунка 2.9, значение переменной стало глобальным. Для единообразия можно объявлять глобальные переменные с помощью команды `declare`. Вместо `"export B=200"` можно написать: `"declare -x B=200"`.

Задание 2.1. Именование переменных

Укажите, какие из объявленных ниже переменных имеют неправильное имя?

☐ _HST=test.example.com

☐ MY-VAR=10

☐ ВРЕМЯ=17.00

☐ 1ST=100

☐ versioN=4.2

☐ SUM#=-1

Задание 2.2. Переменные окружения

Определите, присутствует ли в вашей системе переменная окружения с именем MAIL? Каково её значение? _____

Задание 2.3. Область видимости переменной

В командной строке определили переменную Z:

```
$ Z=100
```

Затем был запущен ещё один шелл

```
$ bash
```

И в нём значение переменной Z было переопределено как

```
$ Z=200
```

Затем вложенный shell был закрыт командой "exit"

Что мы теперь получим на экране, если выполним команду

```
$ echo $Z
```

☐ 100

☐ 200

☐ 0

☐ ничего

☐ Сообщение о том, что переменная не существует

Задание 2.4. Типы переменных

В сценарии была объявлена переменная:

K=5.0

С точки зрения BASH эта переменная будет

☐ числом

☐ строкой

Ответы к заданиям

Задание 2.1.

Укажите, какие из объявленных ниже переменных имеют неправильное имя?

☐ _HST=test.example.com

☐ MY-VAR=10

☐ ВРЕМЯ=17.00

☐ 1ST=100

☐ versioN=4.2

☐ SUM#=-1

Задание 2.2.

Определите, присутствует ли в вашей системе переменная окружения с именем MAIL? Каково её значение?

```
[demo@localhost ~]$ prinenv | grep MAIL
```

Задание 2.3.

В командной строке определили переменную Z:

```
$ Z=100
```

Затем был запущен ещё один шелл

```
$ bash
```

И в нём значение переменной Z было переопределено как

```
$ Z=200
```

Затем вложенный shell был закрыт командой “exit”

Что мы теперь получим на экране, если выполним команду

```
$ echo $Z
```

☐ 100

- ☐ 200
- ☐ 0
- ☐ ничего
- ☐ Сообщение о том, что переменная не существует

Задание 2.4.

В сценарии была объявлена переменная:

K=5.0

С точки зрения BASH эта переменная будет

- ☐ числом
- ☐ строкой

Переменная будет считаться строкой, т.к. числами BASH считает только те переменные, которые не содержат ничего, кроме цифр и знака, а в данном случае в значении присутствует точка.

3. Организация диалога с пользователем: вывод данных

3.1. Команда "echo"

Самый простой способ отобразить на экране какую-либо информацию - это воспользоваться командой "echo". Всё, что передается ей в качестве параметров и не распознается, как опции или специализированные символы, команда "echo" выводит на экран (на самом деле она выводит в стандартный поток вывода, но об этом чуть позже). Например:

```
#!/bin/bash

# Сценарий, выводящий на экран приглашение

echo "Wellcome to test script"
exit 0
```

Рис. 3.1

По умолчанию, завершив вывод строки, команда "echo" добавляет в конец символ перевода строки. Отключить это поведение можно с помощью опции "-n".

```
[demo@localhost ~]$ echo "Hello, World"
Hello, World
[demo@localhost ~]$ echo -n "Hello, World"
Hello, World[demo@localhost ~]$
```

Рис. 3.2

Что касается специализированных символов, то на них стоит остановиться подробнее. Мы уже знаем как минимум, один такой символ - это символ "#". Какие ещё символы для BASH имеют особое значение?

- **Фигурные скобки.** При обнаружении фигурных скобок в строке BASH раскрывает их, подставляя находящиеся внутри значения в строку. Это позволяет конструировать новые строки по определённому шаблону. Например, команда "touch file{a,b,c}" создаст три файла с именами filea, fileb и files. А "echo file{4..7}" выведет на экран "file4 file5 file6 file7". Кроме диапазона внутри фигурных скобок можно указывать шаг, например: "echo file{2..20..2}"

- Тильда. Символ тильда (~) BASH заменяет на домашнюю директорию пользователя. Поэтому команда `echo ~`, запущенная пользователем jsmith выведет `/home/jsmith`. В случае, если требуется сослаться на домашнюю директорию другого пользователя, его имя указывается после символа "~", например: `ls ~root` выведет содержимое директории `/root` (при наличии соответствующих прав доступа).
- Обратные апострофы (`). Всё, что находится между обратными апострофами выполняется как команда и результат работы подставляется в исходную строку. Аналогом обратных апострофов является комбинация символов `$()`. Её использование делает текст сценария более наглядным, т.к. нет необходимости всматриваться: прямые апострофы или обратные.
- Маска имён файлов.
 - Символ "?" означает любой одиночный символ. Поэтому команда `echo ???` выведет на экран три знака вопроса. Но если в директории создать файл с именем zzz, то результатом будет `zzz`.
 - Символ "*" означает любое количество (включая ноль) любых символов. Команда `echo /etc/*.conf` покажет список всех файлов в директории /etc заканчивающихся на ".conf"
 - Квадратные скобки. Означают: «любой из символов, перечисленных внутри квадратных скобок» Пример: `echo /etc/[bnm]*s.conf`
 - Восклицательный знак (либо символ "^") стоящий первым внутри квадратных скобок. Означает: «любой символ, кроме перечисленных внутри квадратных скобок». Пример: `echo /etc/[!bnm]*.conf`
 - `[:digit:]` Означает любую цифру. Пример: `echo /etc/*[:digit:]*.conf`
 - И другие конструкции.



Полный перечень элементов, формирующих маску файлов можно найти здесь:

https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html

Что нужно сделать для того, чтобы специализированный символ, встреченный интерпретатором в строке, воспринимался как обычный алфавитно-цифровой символ и не обрабатывался специальным образом? Для этого его необходимо экранировать. Выражается это в том, что перед символом ставят обратный слеш (\). Например:


```
$ echo one # two
one
$ echo one \# two
one # two
```

Рис. 3.3. Экранирование спец-символа

3.1.1. Интерпретация кавычек

В примере нашего сценария, приведённого на рисунке 3.1. мы использовали команду `echo` с кавычками. Иногда можно встретить вариант, когда вместо кавычек используются апострофы, а иногда - где вообще никаких символов не ставят. Получается, что передаваемая команде `echo` строка может быть представлена в трёх вариантах: без граничных символов, с кавычками и апострофами. Разница заключается в том при использовании кавычек символ `#` а также символы маски файлов утрачивают своё особое значение. Однако, символы `$`, `\` и ``` продолжают интерпретироваться как специальные.

Внутри кавычек не должен встречаться символ кавычки (либо он должен быть экранирован), однако, может использоваться апостроф, например:

```
echo "Эта 'строка' не является ошибочной"
```

Рис. 3.4.

При использовании апострофов в качестве границ строки, символы `$`, `\` и ``` теряют свой специализированный смысл. В связи с этим, иногда кавычки называют «нестрогие кавычки», а апострофы - «строгие кавычки».



Внутри апострофов не может встречаться символ апострофа, даже экранированный. Если необходимо что-то выделить, это можно сделать с помощью кавычек.

Отдельно стоит сказать об упомянутом выше символе обратной кавычки. Рассмотрим такой пример:

```
[demo@localhost ~]$ echo "Today is date"
Today is date
```

Рис 3.5

Вывод, показанный на рис. 3.5. вполне ожидаемый, всё происходит в соответствии со стандартным поведением команды echo.

Теперь, если мы изменим команду, добавив в неё обратные апострофы, то получим:

```
[demo@localhost ~]$ echo "Today is `date`"  
Today is Wed Jul 23 17:08:18 EST 2017
```

Рис 3.6.

Что произошло? Команда date была выполнена интерпретатором, и результат её работы был подставлен в строку. Вместо обратных апострофов можно также применять конструкцию “\$()” Она более наглядно выглядит, т.к. при чтении сценария не требуется всматриваться, прямые ли там апострофы или обратные:

```
[demo@localhost ~]$ echo "Today is $(date)"
```

Рис. 3.7.

Это довольно мощный инструмент, позволяющий конструировать различные новые команды. Предположим, вам требуется создать директорию, имя которой совпадает с текущей датой в формате ГГГГ-ММ-ДД. Готовой команды, которая это может сделать, нет. Однако, у вас есть команда mkdir, создающая директории, и команда date, позволяющая получить дату в нужном формате. Объединив одно с другим, получим:

```
[demo@localhost ~]$ mkdir $(date "+%Y-%m-%d")
```

Рис. 3.8.

Примечание: Используя этот механизм, необходимо всегда оценивать возможность того, что результат работы находящейся внутри \$() команды будет содержать пробелы. Представьте, что команда date имеет следующий вид: date “+%d %m %Y” Тогда mkdir посчитает, что ему передали не один, а три параметра и создаст три директории: имя



одной будет содержать день, имя другой - месяц и имя третьей - год. Ведь таков синтаксис команды `mkdir`, согласно man-странице: если нам требуется создать несколько директорий, то нужно просто перечислить их через пробел. Для того, чтобы команда `mkdir` расценила содержащую пробелы строку как один аргумент, её надо заключить в кавычки. Т.е. `mkdir "$(date "+%d %m %Y")"`

Задание 3.1. Экранирование символов

Что выведет на экран команда:

```
$ echo \\\\\\\
```

- ☐ \
- ☐ \\
- ☐ \\\
- ☐ \\\\\\\
- ☐ ничего
- ☐ ошибку

Задание 3.2. Вывод специализированных символов

Вам надо вывести на экран следующую строку: Price: \$10

Напишите команду echo, которая позволит это сделать:

echo _____

есть ли альтернативные способы, как это сделать? _____

Задание 3.3. Подсчет количества процессов

Для выполнения задания вам понадобятся следующие команды:

ps - получение списка процессов (см man)
wc - подсчёт количества строк в списке
echo - вывод на экран

Напишите сценарий, который при вызове будет выводить: "Total: ZZ" (без кавычек) Где ZZ - общее количество работающих в данный момент процессов. Количество процессов можно получить с помощью команды ps, генерирующей их список, и команды wc, осуществляющей подсчёт (см. соответствующие руководства man)

Задание 3.4. Использование кавычек

Рассмотрите следующие строчки

а) `echo 'Он сказал: "Поехали!" '`

б) `echo "Let's go."`

Предположим, что они встречаются в некотором сценарии. Какое из этих утверждений будет верным?

1. Первая команда верная, а вторая - нет.
2. Вторая команда верная, а первая - нет.
3. Обе команды верные
4. Обе команды неверные.

Что произойдёт, если в строке "б" не будет кавычек?

Ответы к заданиям

Задание 3.1.

Что выведет на экран команда:

```
$ echo \\\\\\\
```

```
□ \\\
```

Задание 3.2.

Вам надо вывести на экран следующую строку: Price: \$10

Напишите команду echo, которая позволит это сделать:

```
echo "Price: \$10"
```

Задание 3.3.

Напишите сценарий, который при вызове будет выводить: "Total: ZZ" (без кавычек) Где ZZ - общее количество работающих в данный момент процессов.

```
#!/bin/bash

# Process counter

NP=$(ps -ef --no-headers | wc -l)

echo "Total: $NP"

exit 0
```

Задание 3.4.

Рассмотрите следующие строчки

а) echo 'Он сказал: "Поехали!" '

б) echo "Let's go."

Предположим, что они встречаются в некотором сценарии. Какое из этих утверждений будет верным?

3. Обе команды верные

Что произойдёт, если в строке "б" не будет кавычек?

Интерпретатор, обнаружив апостроф, будет считать, что им обозначается начало некой строки. И далее, будет искать парный (закрывающий апостроф) считая весь текст частью этой самой строки.

4. Организация диалога с пользователем: ввод данных

4.1. Ввод данных помощью команды read

Для ввода данных с клавиатуры используется команда “read”. Команда помещает считанное значение в переменную, передаваемую ей в качестве параметра. Например:

```
#!/bin/bash
echo -n "Enter Your Name: "
read NAME
echo "Hello, $NAME"
```

Рис.4.1. Использование команды read

Команда read имеет параметр «-p», который позволяет обойтись без дополнительной команды echo и вывести приглашение. Например:

```
[demo@localhost ~] $ read -p "Press Enter to continue..."
```

Рис. 4.2

Также стоит отметить параметр «-s» (silent), заставляющий read не отображать вводимые символы на экране и параметр «-n», после которого указывается количество считываемых символов. Например, можно считать один символ, при этом пользователю не придется нажимать Enter.

```
[demo@localhost ~] $ read -n 1 VAR
```

Рис. 4.3

Еще одним полезным параметром команды read является опция “-t”. Следом за ней указывается количество секунд, которое даётся пользователю для ввода данных, например, “read -t 5 KEY”. Если в течении 5 секунд не последует ввода данных, сценарий продолжит работу, а переменная KEY останется неинициализированной. При этом команда read проинициализирует переменную \$? значением, большим 128.

Используя перенаправление ввода, с помощью команды read легко прочитать первую строку любого файла:


```
read FIRSTLINE < /etc/passwd
```

Рис. 4.4

Еще одна возможность `read` - задать значение вводимой переменной по умолчанию. Если пользователь согласен с этим значением, ему остается просто нажать Enter. Значение задаётся с помощью опции `-i`, которая работает в паре с `-e`. Например:

```
[demo@localhost ~]$ read -p "How old are You? " -i "18" -e AGE  
How old are You? 18
```

Рис 4.5

Задание 4.1. Использование команды read

Для выполнения задания вам понадобятся следующие команды:

mpstat - вывод информации о нагрузке на процессор(ы)

Напишите сценарий, запрашивающий у пользователя два значения: интервал (\$I) и число повторов (\$C), и запускающий команду, выводящую данные о нагрузке на процессор \$C раз с интервалом \$I

Ответы к заданиям

Задание 4.1.

```
#!/bin/bash  
  
read -p "Interval: " I  
read -p "Count: " C  
  
mpstat $I $C  
  
exit 0
```

5. Условное выполнение

Довольно часто выполнение каких-то действий в сценарии зависит от определённых условий. Такими условиями может быть всё, что угодно: в памяти присутствует некий процесс, объем свободного дискового пространства снизился до критического уровня, запущена служба, открыт сетевой порт и т.д.

5.1. Понятие «истина» и «ложь» при анализе выражений; оператор if/then

Для этих целей BASH оснащён специальным оператором if/then, который иногда называют управляющей конструкцией. Для того, чтобы запомнить, как он работает, можно читать его, как “если/то”. Т.е. если выполняется некое условие, то исполнить определённые оператор(ы).

Что означает «условие выполняется»? Если условие - некое выражение, то оценивается двоичный результат, способный принимать значение «истина» либо «ложь». Здесь нужно отметить один важный момент: в других высокоуровневых языках программирования истинному утверждению соответствует число «1», а ложному «0». BASH ведёт себя иначе. Истинности соответствует число «0» а ложности «1». Это связано с тем, что результат выражения на самом деле не преобразуется к логическому типу, а является кодом возврата программы. В Linux за успешный код завершения программы принято считать число ноль, а за ошибочный, соответственно, единицу.

В общем виде конструкцию применения условного оператора можно представить следующим образом: вначале идёт ключевое слово if, за которым следует выражение, требующее проверки. Далее располагается ключевое слово then, свидетельствующее о начале блока команд, которые будут выполнены, если условие истинно. Если проверка входящих параметров располагается на одной строчке с then, после неё необходимо добавить точку с запятой «;». Конец блока команд должен быть завершён ключевым словом fi. Например:

```
if ping -c2 testhost.example.com &> /dev/null
then
  echo "Host testhost.example.com is online"
  scp /data/somefile.tar.gz testhost.example.com:/backup
fi
```

Рис. 5.1. Проверка доступности сетевого узла

Команда `ping` на рис. 5.1. вернет значение 0 (истина) если узел доступен и 1 (не истина), если нет. Соответственно, команды, находящиеся внутри блока `then-else` выполнятся только в случае истинного значения выражения.

Результат выполнения команды можно инвертировать. Делается это с помощью символа `!`:

```
if ! grep '^invisible' /etc/passwd ; then
    echo "user invisible not found"
    useradd invisible
fi
```

Рис. 5.2. Проверка существования строки в файле



Обратите внимание, что изображенный на рис. 5.2 пример - учебный. Проверять существование пользователя в системе через файл `/etc/passwd` - некорректно. Для этих целей используется команда `getent`. Например:

```
getent passwd jsmith
```

Довольно частые задачи - проверить атрибуты (либо просто существование) файла или сравнить две переменных, решаются с помощью специальной команды `test`. Выглядит это так:

```
if test -e '/etc/resolv.conf' ; then
    echo "File /etc/resolv.conf present in your system"
fi
```

Рис. 5.3. Проверка существования файла

Опция `-e` (`exist`) позволяет проверить существование файла в файловой системе.

У команды `test` есть не совсем обычный псевдоним - это `[`. Команда `[` эквивалентна по своим параметрам команде `test` за исключением того, что последним параметром у неё всегда должна быть закрывающаяся квадратная скобка `]`. Пример, приведенный на рис. 5.3. может выглядеть так:

```
if [ -e '/etc/resolv.conf' ] ; then
```

```
echo "File /etc/resolv.conf present in your system"
fi
```

Рис. 5.4. Использование команды "[" вместо test

Начиная с версии BASH 2.02 (проверить номер версии можно с помощью команды "bash --version") наряду с одинарными квадратными скобками можно использовать двойные. Смысл двойных скобок тот же, но они имеют ряд отличий, речь о которых пойдёт далее. Таким образом, приведенный на рис. 5.4. пример, может выглядеть так:

```
if [[ -e '/etc/resolv.conf' ]] ; then
    echo "File /etc/resolv.conf present in your system"
fi
```

Рис. 5.5 Использование "[" (синтаксис, специфичный для BASH)

Кроме этого, с оператором if можно использовать конструкцию ((...)) позволяющую выполнять арифметические операции. Внутри двойных скобок можно использовать операторы >, <, >=, <=, =, != для сравнения чисел.

```
A=$(ps -e --no-headers | wc -l)
B=100
NP=$(nproc)

if ((A>B*NP)) ; then
    echo "Too many processes"
fi
```

Рис. 5.6 Пример с проверкой значения

Какие еще проверки можно осуществлять внутри сценария?

Таблица 5.1. Сравнения чисел

Оператор	Значение
arg1 -eq arg2	Равно. Пример: if [[\$COUNT -eq 0]] ; then echo "COUNT=0 now..." fi
arg1 -ge arg2	Больше или равно (great or equal)
arg1 -ne arg2	Не равно
arg1 -gt arg2	Больше

arg1 -lt arg2	Меньше
arg1 -le arg2	Меньше или равно

Таблица 5.2. Сравнение строк

Оператор	Значение
строка1 = строка2	Строки равны (идентичны) Слева и справа от знака = должны быть пробелы. Можно использовать "=="
строка1 != строка2	Строки не равны
строка1 > строка2	Истинно, если строка1 в текущей локализации при лексикографическом упорядочении предшествует строке2 (при этом "а" считается "меньше" чем "А". Пример: STR1="aa" ; STR2="ab" if [[STR1 < STR2]] ; then echo "STR1 less then STR2" ; fi if [STR1 \< STR2] ; then echo "STR1 less then STR2" ; fi
строка1 < строка2	Истинно, если строка1 в текущей локализации при лексикографическом упорядочении идёт после строки2
-n строка	Строка ненулевой длины
-z строка	Строка нулевой длины

Таблица 5.3. Проверка файлов

Оператор	Значение
-e файл	Файл существует (exists). Ранее использовался оператор "-a", объявленный устаревшим. Пример: if [[-e /etc/resolv.conf]] ; then echo "File resolv.conf exist" fi
-f файл	Файл существует и является регулярным файлом
-d файл	Файл существует и является директорией
-s файл	Размер файла ненулевой
-r файл	Файл доступен для чтения
-w файл	Файл доступен для записи
-x файл	Файл доступен для исполнения
-N файл	Файл модифицировался с момента последнего чтения
-O файл	Файл существует и его владелец - текущий пользователь (т.е. тот, от лица кого делается проверка)
-G файл	Файл существует и его идентификатор группы соответствует идентификатору группы текущего пользователя

файл1 -nt файл2	Файл1 более новый (в соответствии с датой модификации), чем файл2. Пример if [[/store/data_x -nt /store/data_y]] ; then echo "File file_x much fresher" fi
файл1 -ot файл2	Файл1 более старый



Часто встречающаяся ошибка - это отсутствующий пробел при проверке условий, например: `if [bar="$foo"]`

Как мы уже знаем, "[" — это команда. Как и в случае любой другой команды, BASH предполагает, что после команды следует пробел, затем первый аргумент, затем снова пробел, и т.д. Поэтому нельзя писать всё подряд без пробелов! Правильно вот так:

`if [bar = "$foo"]`
bar, =, "\$foo" и] являются аргументами команды [, поэтому между каждой парой аргументов обязательно должен присутствовать пробел, чтобы шелл мог определить, где какой аргумент начинается и заканчивается.

5.2. Оператор if/then/else

Рассмотрим чуть более сложный вариант предыдущего оператора. Читается эта конструкция, как: «если...то...иначе». Т.е. если выполняется условие, то выполнить блок операторов 1, иначе блок операторов 2:

```
if <условие> ; then
    блок операторов 1
    ...
    блок операторов 1
else
    блок операторов 2
    ...
    блок операторов 2
fi
```

Рис. 5.8 Конструкция if/then/else

Пример:


```

if [[ $(nproc) -ge 2 ]] ; then
    echo "CPU Condition is good"
else
    echo "There are not enough CPUs in this system"
fi

```

Рис. 5.9. Оценка количества процессоров (пример if/then/else)

На практике бывает необходимо проверить не одно, а несколько условий. Для этих целей можно использовать вложенные друг в друга операторы if. Скажем, нам необходимо проверить, есть ли в системе пользователь с именем jsmith и если нет, то сделать дополнительную проверку, есть ли пользователь sbrown. Сделать это можно, к примеру, так:

```

if getent passwd jsmith > /dev/null ; then
    echo "User jsmith exist"
else
    if getent passwd sbrown > /dev/null ; then
        echo "User sbrown exist"
    else
        echo "Users does not exist"
    fi
fi

```

Рис. 5.10 Вложенные операторы if

Но можно поступить иначе. Для этого предусмотрена специальная секция elif (else-if), например:

```

if getent passwd jsmith > /dev/null ; then
    echo "User jsmith exist"
elif getent passwd sbrown > /dev/null ; then
    echo "User sbrown exist"
else
    echo "Users does not exist"
fi

```

Рис 5.11. Вариант с использованием elif

5.3. Оператор множественно выбора case

Для случаев, когда вариантов проверки более двух, предусмотрен специальный оператор case, который позволяет избежать множественных повторов if/then. Пример оператора case:

```
case "$1" in
1) echo "Sending SIGHUP signal"
   kill -SIGHUP $2
   ;;
2) echo "Sending SIGINT signal"
   kill -SIGINT $2
   ;;
3) echo "Sending SIGQUIT signal"
   kill -SIGQUIT $2
   ;;
9) echo "Sending SIGKILL signal"
   kill -SIGKILL $2
   ;;
*) echo "Signal number $1 is not processed"
   ;;
esac
```

Рис. 5.12 Пример использования case

Если значение допускает альтернативу (предположим, опция, выводящая справку, может выглядеть как "-h" или как "--help", то это также можно учесть в операторе case:

```
case "$1" in
1) echo "Sending SIGHUP signal"
   kill -SIGHUP $2
   ;;
2) echo "Sending SIGINT signal"
   kill -SIGINT $2
   ;;
3) echo "Sending SIGQUIT signal"
   kill -SIGQUIT $2
   ;;
9) echo "Sending SIGKILL signal"
   kill -SIGKILL $2
   ;;
```

```
-h | --help | -?)  
  echo "Usage: $0 Signal_Number PID"  
  exit 0  
;;  
) echo "Signal number $1 is not processed"  
;;  
esac
```

Рис 5.13 Пример использования case

Задание 5.1 Использование оператора ":"

Что произойдёт при попытке выполнить следующую строку:

```
:::;:::
```

- а) Отобразится ошибка "command not found"
- б) На экране отобразится строка «: : : :»
- в) Ничего. Но значение переменной \$? будет установлено в 0
- г) Ничего. Но значение переменной \$? будет установлено в 1

Задание 5.2. Оценка количества процессов

Для выполнения задания вам понадобятся следующие команды

ps - получение списка работающих процессов

mail - отправка почтового сообщения из консоли

Напишите сценарий, который проверяет количество запущенных процессов в системе. Если их меньше 200 (это значение хранить в виде переменной, чтобы можно было его в дальнейшем поменять), то сценарий не должен ничего делать. Иначе он должен отправлять почтовое сообщение пользователю root (для этого используется команда mail).

Ответы к заданиям

Задание 5.1

Что произойдёт при попытке выполнить следующую строку:

```
:::::;
```

в) Ничего. Но значение переменной \$? будет установлено в 0

Двоеточие – это пустой оператор. А точка с запятой – разделитель. Т.е shell не будет осуществлять никаких видимых действий (в т.ч. вывод на экран) но проинициализирует переменную \$?

Задание 5.2.

Напишите сценарий, который проверяет количество запущенных процессов в системе. Если их меньше 200 (значение хранить в виде переменной, чтобы можно было его в дальнейшем поменять), то сценарий не должен ничего делать. Иначе он должен отправлять почтовое сообщение пользователю root (для этого используется команда mail).

```
#!/bin/bash

MAX_P=200                # Maximum process

CUR_P=$(ps -ef --no-header | wc -l) # Current process amount

if [ $CUR_P -ge $MAX_P ]; then
    mail -s "Warning. Too many processes in the system" root < /dev/null > /dev/null
fi
```

Для проверки работоспособности сценария можно использовать ту же самую команду mail. Запущенная без параметров она переключается в интерактивный режим и предоставляет доступ к локальному почтовому ящику.

6. Арифметические операции

6.1. Арифметические операции над целочисленными переменными

В оболочке реализована возможность выполнения арифметических операций над целыми числами. Делается это с помощью конструкций `$[]` или `$(())`, например:

```
[demo@localhost] $ echo ${2+2}
4
[demo@localhost] $ echo $((2+2))
4
[demo@localhost] $ COUNT=1; echo ${${COUNT+1}*2}
4
[demo@localhost] $ echo $(expr 2 + 2) # обратите внимания на пробелы по краям
знака +
4
[demo@localhost] $ echo `expr 2 + 2`
4
```

Рис. 6.1. Пример вычисления арифметических операций

Вариант `"$[]"` был объявлен устаревшим, однако продолжает работать уже много лет, т.к. необходима обратная совместимость.

Также существует еще один способ, позволяющий выполнять арифметические операции. Это команда `let`. Её использование позволяет обойтись без знака `$` при указании переменных. Например:

```
[demo@localhost] $ A=10
[demo@localhost] $ B=20
[demo@localhost] $ C=A+B
[demo@localhost] $ echo $C
A+B
[demo@localhost] $ let C=A+B
[demo@localhost] $ echo $C
30
```

Рис. 6.2. Использование `let`

Помимо упомянутого выше сложения в BASH предусмотрено выполнение следующих операций:

Таблица 6.1.

Операция	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Целочисленное деление
**	Возведение в степень
%	Остаток от деления
+	Унарный плюс
-	Унарный минус
id++	Пост-инкремент (увеличение на единицу)
++id	Пре-инкремент

Что касается пре- и постинкремента, то их применение можно рассмотреть на примере:

```
[demo@localhost ~]$ A=10
[demo@localhost ~]$ echo $((A++))
10
[demo@localhost ~]$ echo $A
11
[demo@localhost ~]$ B=10
[demo@localhost ~]$ echo $((++B))
11
[demo@localhost ~]$ echo $B
11
```

Рис. 6.3 Пре- постинкремент

В первом случае с переменной A происходит сначала чтение значения и уже после увеличения его на единицу. Поэтому echo в первом случае выводит 10.

6.1. Операции над числами с плавающей точкой.

Т.к. BASH не поддерживает выполнение операций с плавающей точкой, для их выполнения можно воспользоваться внешней программой. Наиболее часто в этой связи используется `bc` (basic calculator). Он обладает большим количеством возможностей, ставящих его в ряд с профессиональными инженерными калькуляторами, однако его детальное рассмотрение не входит в данный курс, поэтому ограничимся несколькими наглядными примерами:

```
[demo@localhost ~] $ echo "3.4+0.8" | bc
4.2
[demo@localhost ~] $ echo "3*6+1" | bc
19
[demo@localhost ~] $ echo "3*(6+1)" | bc
21
```

Рис. 6.4. Использование калькулятора `bc`



Руководство по использованию калькулятора `bc` можно найти здесь:
https://www.gnu.org/software/bc/manual/html_mono/bc.html

6.2. Порядок обработки арифметических операций.

Арифметические операции выполняются в соответствии со своим приоритетом. Так, приоритет операций умножения и деления выше, чем у операций сложения и вычитания. Если необходимо переопределить стандартное поведение интерпретатора, в выражении можно использовать круглые скобки:

```
echo $[ 1 + 2 * 3 ]
7
echo $[ (1 + 2) * 3 ]
9
```

Рис. 6.5. Переопределение приоритетов.

Таблица 6.2. Приоритеты некоторых операций.

6.3. Работа со случайными значениями.

Иногда бывает необходимо сгенерировать одно или несколько случайных значений. Для этого в BASH предусмотрена переменная \$RANDOM. Она возвращает значения в диапазоне от 0 до 32767. Помните, что это не случайные, а псевдослучайные значения, они не должны использоваться для генерации ключей шифрования.

Сгенерировать значение в заданном диапазоне (к примеру, от 1 до 100) можно так:

```
[demo@localhost ~] $ echo $RANDOM
29754
[demo@localhost ~] $ echo $(((RANDOM % 100) + 1))
78
[demo@localhost ~] $ seq 1 100 | sort -R | head -n1
14
```

Рис.6.6. Получение случайного значения с \$RANDOM и без

Задание 6.1 Подсчет значения

Написать сценарий, подсчитывающий и выводящий на экран количество секунд в сутках.

Задание 6.2. Операции над числами с плавающей точкой

Предположим, вам необходимо сохранить в переменной сумму двух чисел с плавающей точкой (пусть это будет 4.3 и 1.2). Как это сделать?

Ответы к заданиям

Задание 6.1

Написать сценарий, подсчитывающий и выводящий на экран количество секунд в сутках.

```
#!/bin/bash

SEC_IN_MIN=60
MIN_IN_HR=60
HR_IN_DAY=24

SEC_IN_DAY=$(( $SEC_IN_MIN * $MIN_IN_HR * $HR_IN_DAY ))

echo "There are $SEC_IN_DAY seconds in a day."

exit 0
```

Задание 6.2

Предположим, вам необходимо сохранить в переменной SUMM сумму двух чисел с плавающей точкой (пусть это будет 4.3 и 1.2). Как это сделать?

```
SUMM=$(echo "4.3+1.2" | bc)
```

7. Циклы

7.1. Понятие тела цикла, счетчика, анализ условия выхода, бесконечный цикл.

Цикл - управляющая конструкция, предназначенная для многократного выполнения набора инструкций, называемых телом цикла. Каждое единичное выполнение цикла называется итерацией.

Выражение, определяющее, будет ли выполняться очередная итерация или цикл завершится, называется условием выхода или условием окончания цикла.

Переменная, хранящая текущий номер итерации, называется счётчиком цикла (при этом, цикл не обязательно должен содержать счётчик, так же как счётчик не обязательно должен быть единственным). Исполнение цикла обычно включает первоначальную инициализацию переменных цикла, проверку условия выхода, исполнение тела цикла и обновление переменной цикла на каждой итерации. Существует несколько видов циклов:

- безусловный цикл - цикл, выход из которого не предусмотрен логикой программы. Такие циклы ещё называют бесконечными.
- Цикл с предусловием — цикл, который выполняется, пока истинно некоторое условие, указанное перед его началом. Это условие проверяется **до** выполнения тела цикла, поэтому тело может быть не выполнено ни разу (если условие с самого начала ложно). В большинстве процедурных языков программирования реализуется оператор **while**, отсюда его второе название — while-цикл.
- Цикл с постусловием — цикл, в котором условие проверяется **после** выполнения тела цикла. Отсюда следует, что тело **всегда выполняется** хотя бы один раз.

7.2. Цикл for

Обычно используется для перебора каких-либо значений. Общий синтаксис:

```
for переменная in список  
do
```

```
команды
done
```

Примеры:

```
[demo@localhost ~] $ for HOST in station1 station2 station3 ; do
> ping -c2 $HOST &> /dev/null && echo "Host $HOST is alive"
> done
Host station1 is alive
Host station3 is alive
```

Рис. 7.1. Последовательный перебор значений

Если требуются числовые значения, то начало и конец можно проинициализировать одним из следующих вариантов:

```
for C in {20..100}
do
  echo $C
done

for C in $(seq 20 100)
do
  echo $C
done
```

Рис. 7.2. Последовательный перебор значения от 20 до 100

```
[demo@localhost ~]$ for C in {2..8..2} ; do echo $C; done
2
4
6
8
```

Рис. 7.3. Перебор значений с заданным шагом

В качестве значений списка не обязательно должны быть цифры. Он может быть проинициализирован результатами работы какой-либо команды, например:

```

echo "Network interfaces:"
for IF in $(netstat -i | cut -d ' ' -f1 | grep "^en")
do
    ip addr show $IF
done

```

Рис. 7.4. Вывод настроек всех Ethernet-интерфейсов

```

for U in $(cut -d: -f1 /etc/passwd)
do
    mkdir -p "/export/$U"
    chown $U:$U /export/$U
done

```

Рис. 7.5 Создание директорий для всех локальных пользователей

В качестве списка могут также выступать имена файлов, над которыми необходимо осуществить какие-либо операции⁶. В связи с этим можно встретить примеры, типа такого:

```

for FILE in /data/*
do
    echo -n "$FILE : "
    stat --format=%z "$FILE"
done

```

Рис. 7.6. Вывод даты и времени изменения атрибутов файлов в директории /data

Обратите внимание на то, что переменная FILE указана в кавычках. Это необходимо для того, чтобы сценарий сохранял работоспособность, если в директории встречаются файлы, содержащие пробел в имени. Иначе, если в директории окажется файл, содержащий в имени пробелы, скажем, "file 03", то его имя будет расценено, как два самостоятельных параметра, переданных команде stat и в выводе появятся ошибки:

```
stat: cannot stat '/data/file': No such file or directory
```

⁶ К циклам есть смысл прибегать, если операций множество или они требуют проверки каких-либо условий. Для простых случаев можно обойтись параметром “-exec” команды find

stat: cannon stat '03': No such file or directory



Если речь об обработке файлов, имена которых не носят заранее известного характера, всегда нужно помнить о том, что среди них могут оказаться файлы, содержащие в имени пробелы или другие специализированные символы, скажем, "*".

Существует также вариант цикла `for`, хорошо знакомый тем, кто сталкивался с программированием на языке "C":

```
for ((i=0;i<=10;i++))
do
    uptime
    sleep 1
done
```

Рис. 7.7. пример цикла `for` в стиле "C"

В двойных скобках указывается три выражения: инициализация счётчика, условие окончания цикла и изменение переменной счётчика.

7.3. Цикл `while`

Цикл `while` выполняется до тех пор, пока условие будет истинным (т.е. код возврата равен 0). Он проверяет условие перед началом каждой итерации, т.е. является циклом с предпроверкой. Используется в тех случаях, когда количество итераций не известно заранее, например:

```
#!/bin/bash

while [ ! -f /tmp/stopfile.txt ]
do
    echo "File stopfile.txt not found" > /dev/stderr
    sleep 1
done

echo "$USER" >> /tmp/stopfile.txt
```

Рис 7.8. Цикл while

Цикл, изображённый на рис. 7.8. будет выполняться до тех пор, пока в директории /tmp не появится файл с именем stopfile.txt

Если условие, проверяемое циклом while никогда не станет истинным, то цикл будет выполняться бесконечно⁷:

```
while true
do
    echo "Press [CTRL+C] to stop.."
    sleep 1
done

while ;; do echo "Press [CTRL+C] to stop"; sleep 1; done
```

Рис. 7.9. Бесконечный цикл while

7.4. Цикл until

Оператор цикла until проверяет условие в начале каждой итерации, но в отличие от while итерация возможна только в том случае, если условие ложно. Иначе говоря, цикл выполняется до тех пор, пока условие не станет истинным. Пример:

```
#!/bin/bash

COUNT=0

until [[ $COUNT -gt 10 ]]
do
    echo $COUNT
    sleep 1
    ((COUNT++))
done
```

Рис. 7.10 Цикл выполняется, пока переменная не станет больше 10

⁷ Существует возможность остановить содержащий бесконечный цикл сценарий "извне". Для этого используется команда timeout. Например, timeout 300 /usr/local/bin/example.sh Если сценарий не завершит свою работу через 300 секунд, то его выполнение прекратит команда timeout.

7.5. Операторы break и continue

Оператор continue предназначен для прерывания текущей итерации цикла и перехода к следующей. Пример:

```
#!/bin/bash

COUNT=0

while [[ $COUNT -lt 10 ]]
do
    ((COUNT++))
    if [[ $COUNT -eq 5 ]] ; then
        continue
    fi
    echo $COUNT
    sleep 1
done
```

Рис. 7.11. Пример оператора continue

Цикл на рис. 7.11. отобразит последовательность цифр 1 2 3 4 6 7 8 9 10. Цифра 5 не появится, т.к. при выполнении условия, проверяемого оператором if, сработает continue и цикл перейдёт к следующей итерации

Оператор break позволяет завершить выполнение цикла, даже в том случае, если не наступило соответствующее выходу значение счётчика. Пример:

```
#!/bin/bash

while true
do
    if [ -z "$(ls -A /home/user/data)" ] ; then
        break
    fi
    echo "Directory is not empty"
    sleep 1
done
```

Рис. 7.12. Пример оператора break

Цикл на рис. 7.12. будет выполняться до тех пор, пока директория не окажется пустой. Тогда сработает условие, и команда break осуществит выход

7.6. Вложенные циклы

Цикл называется вложенным, если он размещается внутри другого цикла. На первом проходе, внешний цикл вызывает внутренний, который исполняется до своего завершения, после чего управление передается в тело внешнего цикла. На втором проходе внешний цикл опять вызывает внутренний. И так до тех пор, пока не завершится внешний цикл. Как внешний, так и внутренний циклы могут быть прерваны командой break.

Операторам break и continue может быть передан необязательный параметр (целое число). В обычных обстоятельствах действие оператора распространяется на текущий цикл. Если же оператор вызывается с параметром N, то его действие будет распространяться на внешний цикл, отстоящий от текущего на уровень N. (1 соответствует текущему уровню).

Пример вложенных циклов:

```
#!/bin/bash

for i in 1 2 3 4 5 6 7 8 9 10
do
    for((j=1;j<=i;j++))
    do
        echo -n "*"
    done          # окончание тела внутреннего цикла (j)
    echo ""
done              # окончание тела внешнего цикла (i)
```

Рис 7.13. Вложенные циклы while

Задание 7.1 Использование цикла for

Напишите сценарий, который используя цикл `for`, осуществляет подсчет всех файлов в директории `/etc/sysconfig/network-scripts`, имеющих права доступа `"rw-r--r--"`. Обязательно прочитайте примечание к ответу на это задание.

Ответы к заданиям

Задание 7.1

Напишите сценарий, который используя цикл `for`, осуществляет подсчет всех файлов в директории `/etc/sysconfig/network-scripts`, имеющих права доступа `"rw-r--r--"`.

```
#!/bin/bash

TOTAL=0

for LIST in /etc/sysconfig/network-scripts/* ; do
    FL=$(ls -l "$LIST")
    if echo "$FL" | grep -q -- '-rw-r--r--' ; then
        ((TOTAL++))
    fi
done

echo "Files: $TOTAL"
```

более компактный вариант:

```
#!/bin/bash

for LIST in $(ls -l /etc/sysconfig/network-scripts) ; do
    echo $LIST
done | grep -- '-rw-r--r--' | wc -l
```

Использование цикла `for` является условием данной задачи. В реальных обстоятельствах для решения этой задачи можно было бы вполне обойтись и без него (а также без `wc`):

```
#!/bin/bash

echo Total: $(ls -l /etc/sysconfig/network/scripts | grep -c '^-rw-r--r--')
```

Всегда проверяйте, существует ли более простой вариант решения поставленной задачи

Примечание: Если требуется найти файлы, удовлетворяющие определённым условиям, используйте утилиту `find`. Это готовый и очень мощный инструмент, позволяющий осуществлять поиск файлов по различным критериям. Формат команды “`find область_поиска критерии`” Критерии поиска:

- `-name` искать по имени файла, например: `find /etc -name "r*.conf"`
- `-iname` искать по имени файла без учета регистра
- `-user` искать файлы, принадлежащие пользователю с указанным именем
- `-uid` искать файлы, принадлежащие пользователю с определенным идентификатором
- `-group` искать файлы, принадлежащие группе
- `-size` искать файлы определенного размера (задаётся в байтах). Можно указывать буквы “к” и “М” для указания кило- и мегабайт. Перед размером может стоять знак плюс или минус, означающий «более чем» или «менее, чем»
- `-perm XYZ` - искать файлы с определенными правами доступа
- `-type <ТИП>` искать файлы определенного типа: `f`-регулярные файлы, `d`-директории, `l`-символьные ссылки, `b` - блочные устройства ит.д.

Таким образом, поставленная задача решается следующей командой:

```
find /etc/sysconfig/network-scripts -type f -perm 644 -maxdepth 1 | wc -l
```

8. Использование средств отладки, типичные ошибки

8.1 Отладка сценария

К сожалению, BASH обладает довольно скромными возможностями, помогающими отыскать проблему в неработающем сценарии. У него нет отдельного отладчика, более того - иногда выводимые им ошибки порой слабо помогают при отладке. Рассмотрим вот такой (содержащий ошибку) пример:

```
A=54
if [ $A -gt 50 ]
then
  echo $A
fi
```

Рис 8.1

При попытке его выполнить BASH выдаст ошибку:

```
-bash: [54: command not found
```

Рис. 8.2

Начинающего разработчика эта ошибка ставит в тупик, хотя смысл её довольно прост: отсутствующий пробел привёл к тому, что квадратная скобка и следующее за ней значение стали восприниматься, как единое целое. Другой пример:

```
for A in 1 2 3
do
  echo $A
# done
exit 0
```

Рис 8.3 Пропущен конец тела цикла (done)

Приведёт к ошибке

```
- bash: syntax error: unexpected end of file
```

Рис. 8.4 BASH не сообщает о том, что пропущено ключевое слово.

Как видно из рис 8.4. в ошибке говорится о возникшем в неподобающем месте (done-то отсутствует) конце сценария. Более того: если этот сценарий сохранить в файле, то будет указан номер строки, породившей ошибку, но это будет не тот номер, который нужен нам!

Поэтому, при написании кода необходимо стараться сделать его более наглядным и читаемым, чтобы упростить дальнейший анализ и отладку. Сравним два сценария:

```
#!/bin/bash
P_NUM=0; for F in /proc/[0-9]*/status ; do S_VOL=$(grep '^VmSwap' "$F" | awk '{print $2}')
if [ ! -z "$S_VOL" ]; then if [ $S_VOL -gt 0 ]; then P_PID=$(grep '^Pid' "$F" | awk '{print $2}');
echo "Process $P_PID use swap" ; ((P_NUM++)); fi; fi; done | sort -k 2 -g | column -t
if [ $P_NUM -eq 0 ]; then echo "No process with swapped data"; exit 1; fi;exit 0
```

Рис. 8.5

```
#!/bin/bash

# Сценарий проверяет, есть ли в системе процессы, данные которых
# располагаются в области swap и выводит их список.
#
# Написан 10-11-2018 Иваном Ивановым iivanov@example.com

P_NUM=0                                # Счетчик найденных процессов

for F in /proc/[0-9]*/status           # перебираем в цикле файлы статуса всех процессов
do
    S_VOL=$(grep '^VmSwap' "$F" | awk '{print $2}')
    if [ ! -z "$S_VOL" ]; then
        if [ $S_VOL -gt 0 ]; then
            P_PID=$(grep '^Pid' "$F" | awk '{print $2}') # Получаем идентификатор процесса
            echo "Process $P_PID use swap"
            ((P_NUM++))
        fi
    fi
done | sort -k 2 -g | column -t         # Сортируем список и форматируем вывод

if [ $P_NUM -eq 0 ]; then
    echo "No process with swapped data"
    exit 1
fi
```

```
exit 0
```

Рис. 8.6

Эти сценарии делают одно и то же. И несмотря на то, что первый сценарий гораздо компактнее, разобраться в его работе значительно сложнее, несмотря на то, что речь идёт о совсем простом примере, а реальный сценарий может оказаться гораздо объёмнее. Поэтому крайне желательно не пренебрегать комментариями, использованием переменных и структурированием кода, делающими чтение сценария более простым.

Другая (более сложная) проблема - сценарий запускается без сбоев, но не делает то, что нужно. Например, нам требуется сценарий, который удаляет в текущей директории все файлы, в именах которых содержатся пробелы. Пусть он выглядит так:

```
BADFILE=$(ls | grep ' ' )  
rm -f "$BADFILE"
```

Рис. 8.7

Несмотря на отсутствие ошибок при выполнении, работать этот сценарий не будет, т.к. взятая в кавычки переменная BADFILE с точки зрения интерпретатора выглядит как одна длинная строка, разделённая символами “\n”. Убедиться в этом можно, добавив вывод переменной \$BADFILE перед командой rm.

В данном случае поступить можно гораздо проще, например, так:

```
rm -f '*' '*'
```

Рис 8.8

В ситуациях, когда необходима отладка, могут помочь

- вставленные в критических точках сценария команды echo, позволяющие отследить состояние переменных
- Фильтр tee, позволяющий добавить вывод на экран, если команда работает с перенаправлением
- Команда exit, позволяющая запускать сценарий по частям
- Специальные опции самого BASH

8.2. Специальные опции BASH

Эти опции можно задавать при вызове обработчика, например:

```
[demo@localhost ~]$ bash -n ./test01.sh
```

Рис 8.9

Опции можно указывать в строке shebang:

```
#!/bin/bash -n  
# текст сценария  
...
```

И, наконец, третий вариант - опции можно указывать в тексте самого сценария:

```
#!/bin/bash  
  
оператор 1  
оператор 2  
set -n      # либо set -o noexec  
оператор 3  
оператор 4  
set +n      # либо set +o noexec  
оператор 5  
оператор 6
```

Опция “-n” позволяет проверить наличие синтаксических ошибок, не запуская сам сценарий.

Опция -v (--verbose) показывает каждую строку сценария при её обработке. Этот же эффект включается в тексте самого сценария установкой set -v или set -o verbose.

Опция -x выводит каждую команду в том виде, в котором она поступает обработчику, подставляя значения переменных, масок и проч. В сценарии она

включается через `set +x` или `set -o xtrace`. При вызове интерпретатора опции можно комбинировать, например: `"$ bash -xv script.sh"`

Также в сценарии можно вставить настройку `set -u` (или `set -o nounset`) и тогда, при обращении к любой необъявленной ранее переменной будет выводиться сообщение об ошибке `"unbound variable"`.

Еще один переключатель, о котором стоит знать: `set -f` (`set -o noglob`) Он отключает генерацию имён файлов с помощью масок. Например:

```
[ demo@localhost ~]$ cd /etc
[ demo@localhost etc]$ ls m*.conf
man_db.conf mke2fs.conf
[ demo@localhost ~]$ set -f
[ demo@localhost etc]$ ls m*.conf
ls: cannot access m*.conf: No such file or directory
```

Рис 8.10

К важным с точки зрения отладки переключателям стоит отнести `set -e` (`set -o errexit`). если он включен, то выполнение сценария завершится, как только команда вернёт значение «не истина» (т.е. выполнится с ошибкой).

Данный переключатель бывает полезно объединить с `"-x"` тогда будет наглядно видно, в каком именно месте сценария произошла ошибка. В тоже время этот переключатель следует применять с осторожностью. Тот же `grep` вернёт неуспех, если ничего не нашел в переданных ему данных и сценарий завершиться. Иногда лучше регулярно проверять код возврата, чтобы выводить человеческие сообщения об ошибках, чем просто завершать скрипт по любому поводу.

Вообще говоря, с помощью `set` можно многое изменять в поведении интерпретатора. Получить список всех параметров и их текущие значения можно с помощью команды `"set -o"`, а их предназначение – в `man`-странице `man bash (1)`

Задание 8.1. Иллюстрация работы режима отладки

Цель задания: Проиллюстрировать работу режима отладки

Создайте следующий сценарий с именем filestat.sh:

```
#!/bin/bash

DIR=/etc

for FILE in $(find $DIR -maxdepth 1 -type f -name "p*") ; do
    echo "File $FILE is $(stat --printf='%s' $FILE) bytes"
done
```

Запустите его сначала командой `/bin/bash filestat.sh` а затем командой `/bin/bash -x filestat.sh`

Обратите внимание на вывод отладочной информации. Попробуйте также запустить сценарий командой `/bin/bash -xv filestat.sh`

8.3. Типичные ошибки, допускаемые при разработке

Для начала - несколько рекомендаций, которые справедливы не только для BASH-сценариев:

- Не используйте для хранения самих сценариев, и их данных стандартные системные директории (типа, /usr/bin).
- Не переопределяйте переменные окружения (типа \$USER);
- Не используйте вывод сообщений на русском языке (во всяком случае, безальтернативно). Это может создать проблемы, когда понадобится воспользоваться консолью с кодировкой, не поддерживающей кириллицу или передать сценарий коллеге, не говорящему по-русски;
- Если сценарий выполняет какие-либо потенциально опасные для системы/данных действия, можно добавить параметр "--force" или "--dry-run". В то же время необходимо предусмотреть автоматическое выполнение сценария на случай, если потребуется запустить его через планировщик. Хороший пример: программа rm, у которой есть ключ "-i" заставляющий спрашивать подтверждения действия, а также ключ "-f" заставляющий ее продолжить пакетную работу, даже если возникли проблемы с удалением конкретного файла. Если бы интерактивность rm была бы не отключаемой, ее бы пришлось искать замену сразу после первой же необходимости обработать директорию с несколькими тысячами файлов.
- Не пишите ничего в log-файлы через перенаправление вывода. Используйте утилиту logger.

Далее, для того, чтобы понимать, как именно происходит выполнение команды, нужно представлять себе последовательность действий интерпретатора. Прежде чем приступить к выполнению, оболочка осуществляет вот такой набор подстановок:

- Развертывание фигурных скобок
- Подстановка тильды
- Подстановка позиционных параметров
- Подстановка переменных
- Вызов команд, заключенных внутри \$()
- Вычисление арифметических выражений
- Разбиение на слова. В качестве разделителя используется переменная \$IFS
Не разбиваются те части команды, которые находятся внутри кавычек
- Подстановка масок (шаблонов) в путях

Далее BASH проверяет наличие псевдонимов (alias) и осуществляет поиск команды в следующем порядке:

1. Среди встроенных в интерпретатор команд
2. Среди объявленных функций
3. В каталогах, определенных в переменной \$PATH (этого не происходит, если к команде указан путь)

Теперь разберём, какие проблемы могут возникнуть при вызове команд в тексте сценария:

8.3.1 Пробелы в именах файлов

Что касается самих сценариев, то довольно частой ошибкой является то, что разработчик забывает о том, что в именах файлов могут присутствовать пробелы. Из-за этого можно встретить вот такие примеры кода:

```
for F in $(ls *.txt); do
    some_command $F
...
done
```

Рис 8.17

Это не сработает, если в имени файла встретятся пробелы. Цикл пройдёт по каждому слову из названия, помещая их в переменную F. Причем, заключить всю команду ls в кавычки не получится (for F in “\$(ls *.txt)” ; do) т.к. вывод теперь рассматривается как одно слово, и вместо того, чтобы пройти по каждому из файлов в списке, цикл выполнится только один раз, при этом F примет значение, являющееся конкатенацией всех имён файлов через пробел. Использование команды ls здесь является излишним. Корректный вариант приведённого выше примера должен выглядеть так:

```
for F in *.txt; do
    some_command "$F"
...
done
```

Рис. 8.18 Обратите внимание на кавычки в “some_command”

Еще один вариант проблемной команды

```
cp $source $target
```

Рис. 8.19

Как только встретится содержащий пробелы файл, посыпятся ошибки типа: «cp: cannot stat ... : No such file or directory». Устранить эту проблему можно, добавив кавычки к именам переменных. Однако, это не панацея. Что если среди копируемых файлов встретится файл, имя которого начинается с дефиса? Команда cp посчитает, что вы передаёте ей ещё один параметр. Избежать такой ситуации можно, добавив команде cp двойной дефис в качестве параметра. Это является указанием на то, что следует прекратить поиск опций (см man (1) bash):

```
cp -- "$source" "$target"
```

Рис 8.20

8.3.2 Попытка читать из файла и писать в него же в одном конвейере.

В зависимости от того, как построен конвейер и реализованы входящие в него команды, файл может обнулиться или оказаться обрезанным до размера буфера, выделяемого системой для конвейера, в общем, результат будет не тем, который нужен. Например:

```
grep 'demo' file.txt | sort > file.txt
```

Рис 8.21

Один из вариантов обхода такой ситуации - создание временного файла:

```
grep 'demo' file.txt > tmpfile ; sort tmpfile > file.txt ; rm -f tmpfile
```

Рис 8.22

8.3.3 Вывод переменной без использования кавычек

```
echo $FILENAME
```

Рис. 8.23

Поскольку переменная \$FILENAME не заключена в кавычки, она будет не только разделена на слова, но и возможно содержащийся в ней шаблон будет преобразован в имена совпадающих с ним файлов.

```
VAR=*.zip
echo "$VAR"    # выведет *.zip
echo $VAR      # выведет список файлов, чьи имена заканчиваются на .zip
```

Рис 24

Задание 8.2 Поиск ошибок

Посмотрите на приведённый ниже сценарий. Содержит ли он какие-либо ошибки? Если да, то какие?

```
#!/bin/bahs

RES= $(PWD)

if [$FNAME = "process" ] then
  grep '--r' $RES/*.txt
fi
```

Ответы к заданиям

Задание 8.2

Посмотрите на приведённый ниже сценарий. Содержит ли он какие-либо ошибки? Если да, то какие?

```
#!/bin/bahs

RES= $(PWD)

if [$FNAME = "process" ] then
  grep '--r' $RES/*.txt
fi
```

1. Допущена опечатка при указании shebang (/bin/bahs)
2. При инициализации переменной RES лишний пробел справа от знака равенства
3. Команда pwd записана в верхнем регистре
4. При проверке условия пропущен пробел справа от знака "["
5. Отсутствуют кавычки при указании переменной FNAME: если она окажется пустой, условие перестанет работать
6. Пропущен символ ";" перед словом then
7. grep в таком варианте работать не будет, т.к. расценить '--r' как переданную опцию.
8. Переменная \$FNAME, скорее всего, никогда не примет такое значение, т.к. команда возвращает абсолютный путь, всегда начинающийся с знака "/"

9. Анализ командной строки

Традиционно, повлиять на работу программы можно, передав ей какие-либо параметры при вызове. Например, «заставить» команду `ls` показывать скрытые файлы, можно, передав ей в качестве параметра опцию `-a`. Точно также можно осуществить вызов сценария, передав ему в командной строке какие-либо параметры. Для этого существует несколько специальных переменных:

- `$0` - Переменная, в которой хранится имя самого сценария
- `$1` - Переменная, в которой хранится первый переданный сценарию параметр. Т.е. если сценарий называется `script.sh` и вызывается с параметрами `"/usr/local/bin/script.sh -N /dir1"` то переменная `$1` будет содержать `"-N"`
- `$2` - Переменная, в которой хранится второй переданный сценарию параметр. Т.е. в сценарии из предыдущего примера переменная `$2` будет содержать `"/dir1"`
- ...
- `${10}` - Переменная, в которой хранится десятый переданный сценарию параметр. Здесь и далее имена переменных используются с фигурными скобками (но вообще говоря, желательно избегать сценариев с таким большим количеством параметров)
- `$#` - Переменная, в которой хранится количество переданных сценарию параметров
- `$*` - Переменная, в которой хранится вся строка с переданными сценарию параметрами, как единое целое.
- `$@` - Переменная, в которой хранится вся строка с переданными сценарию параметрами, как набор отдельных строк.
- `$$` - Переменная, содержащая PID самого сценария.

Количество (а точнее - размер) передаваемых сценарию параметров неограничен. Его можно определить с помощью команды

```
getconf ARG_MAX
```

`ARG_MAX` - это максимально возможный размер буфера, обрабатываемый системным вызовом `exec()`. Кроме команды `getconf` получить информацию по

ограничениям в более развёрнутом виде можно с помощью команды `xargs: "xargs --show-limits"`

Как проверить, присутствует ли тот или иной параметр в списке? Простой способ сделать это, выглядит следующим образом:

```
#!/bin/bash

for ARG in "$@" ; do
    echo $ARG
done
```

Рис 9.1. Вывод списка параметров, переданных сценарию

9.1. Чтение параметров, переданных сценарию при вызове

В соответствии со стандартом POSIX, передаваемые программе опции должны быть позиционно-независимые. Т.е. можно написать `"ls -a -l"` а можно `"la -l -a"`. Это создаёт проблемы при написании сценария, т.к. заранее не известно, какой параметр будет в переменной \$1, а какой - в \$2. Более того, пользователь имеет право написать `"ls -la"`, что ещё усложняет задачу.

Для решения этой проблемы есть специальная команда - `getopts` (На самом деле команды две: `getopts`, являющаяся встроенной и вызываемая внешняя `getopt`, находящаяся в директории `/usr/bin` Встроенная команда внесена в стандарт POSIX, и не требует установки дополнительного пакета, т.к. является частью самого BASH. Внешняя, согласно man-странице обладает расширенными возможностями, к примеру, имеет механизм обработки опций с двумя дефисами).

Чтобы определить, был ли передан определенный параметр сценарию и не загромождать его лишними проверками, можно использовать следующую конструкцию:

```
#!/bin/bash

if [ -z "$*" ]
then
    echo "Error: no any parameters"
    exit 1
fi
```

```

while getopts "abc" OPT
do
    case $OPT in
        a) echo "Option $OPT detected" ; touch /tmp/pid01 ;;
        b) echo "Option $OPT detected" ; rm -f /tmp/pid01 ;;
        c) echo "Option $OPT detected" ;;                # Just inform about option
    esac
done

```

Рис. 9.2 Пример использования getopts

Тут мы запускаем цикл, который выполняет команду getopts. В свою очередь getopts получает список допустимых опций – “abc” (обратите внимание – допустимые опции указываются без знака тире), и при нахождении одной из них – передаёт её в переменную \$OPT. Далее уже case выполняет действия, связанные с переданной опцией.

9.2. Использование команды shift для сдвига позиционных параметров

Если сценарий предполагает работу со списком каких-либо файлов, совершенно необязательно разыскивать их имена среди позиционных переменных \$1, \$2 итд. Предположим, сценарий вызывается вот так:

```
$ ./process.sh file1.txt file2.mp3 file3.avi file4.zip file5.jpg file6.bak
```

Тогда переменная \$1 содержит “file1.txt”, переменная \$2 - “file2.mp3” и т.д. Зная количество переданных сценарию параметров (переменная \$#) можно обработать их все в цикле. Но есть способ, который позволяет эту процедуру упростить. Это команда shift. Она сдвигает весь список параметров влево, после чего переменная \$1 будет содержать “file2.mp3”, переменная \$2 - “file3.mp3” итд. Список при этом укорачивается на единицу. Переменная \$0 при этом не изменяется (А вот \$* и \$# - да, поэтому их можно использовать для проверки условия окончания цикла). Значение, которое было самым первым до сдвига - теряется.

Таким образом, если нам необходимо обработать каждый из переданных сценарию файлов, можно поступить следующим образом:

```
#!/bin/bash

COUNT=1

while [[ -n "$1" ]]
do
    echo "Параметр $COUNT имеет значение $1"
    ((COUNT++))
    shift
done
```

Рис 9.3. Использование команды shift

По умолчанию, shift осуществляет сдвиг на одну позицию. В качестве параметра можно передавать этой команде целочисленное значение, указывающее, на сколько позиций осуществить сдвиг. Например, "shift 3". Соответственно, команда "shift #" полностью обнулит строку параметров полностью.

Задание 9.1. Определение опций сценария

Написать сценарий, который допускает вызов с двумя опциями: -с или -d. С опцией "-с" он должен создавать директорию /tmp/datastore, а с опцией "-d" - удалять её. При отсутствии опций, при их некорректной комбинации, а также при получении нераспознаваемого параметра сценарий должен выводить соответствующие уведомления.

Ответы к заданиям

Задание 9.1.

Написать сценарий, который допускает вызов с двумя опциями: "-c" или "-d". С опцией "-c" он должен создавать директорию /tmp/datastore, а с опцией "-d" - удалять её. При отсутствии опций, при их некорректной комбинации а также при получении нераспознаваемого параметра сценарий должен выводить соответствующие уведомления.

```
#!/bin/bash

DR=/tmp/datastore

if [ $# -eq 0 ] ; then
    echo "usage: $0 -d or $0 -c"
    exit 1
fi

case "$1" in
-c | --create) mkdir $DR 2> /dev/null
;;
-d | --delete) rm -rf $DR
;;
*) echo "Incorrect options"
;;
esac
```

10. Работа со строками

10.1. Операторы для сравнения строк

Для сравнения строк используются выражения, приведенные ранее в таблице 5.2. Если требуется сравнить два строковых значения, сделать это можно, например, так:

```
if [ "$x" == "valid" ]; then
    echo "x имеет значение 'valid'"
fi
```

Рис. 12.1. Сравнение строк

Использование кавычек с переменной *x* необходимо, т.к. в случае, если переменная пуста, BASH интерпретирует конструкцию как «[== "valid"]» и произойдет ошибка. От этой ситуации спасет использование двойных квадратных скобок. При их применении кавычки будут необязательны. Обратите внимание на разницу в следующем примере:

```
[[ "$x" == z* ]]      # истина, если x начинается с символа z
[[ "$x" == "z*" ]]   # истина, если переменная x равна z*
```

Рис. 12.2. Сравнение строк с использованием [[]]

10.2. Обработка строк собственными средствами BASH

В BASH предусмотрен механизм манипуляции над ранее определенными переменными. Список конструкций, позволяющих манипулировать значениями переменных представлен в таблице:

Таблица 12.1.

Конструкция	Описание	Пример
<code>\${переменная^}</code>	Перевести первый символ строки в верхний регистр	<code>\$ TEST1="linux"</code> <code>\$ echo \${TEST1^}</code> Linux
<code>\${переменная^^}</code>	Перевести все символы в верхний регистр	<code>\$ echo \${TEST1^^}</code> LINUX
<code>\${переменная,}</code>	Перевести первый символ в нижний регистр	<code>\$ TEST1="LINUX"</code> <code>\$ echo \${TEST1,}</code>
<code>\${переменная,,}</code>	Перевести все символы в нижний регистр	<code>\$ echo \${TEST1,,}</code>

	регистр	linux
<code>\${переменная~}</code>	Заменить регистр первого символа строки на противоположный	<code>\$ echo \${TEST1~}</code> lINUX
<code>\${переменная~~}</code>	Заменить регистр всех символов строки на противоположный	<code>\$ echo \${TEST1~~}</code> linux
<code>\${переменная#шаблон}</code>	Удаляет из строки, хранящейся в переменной, минимальную подстроку, заданную шаблоном. Поиск ведётся с начала строки. При несовпадающем шаблоне изменений не происходит.	<code>\$FN=my.file.txt</code> <code>\$echo \${FN#my}</code> .file.txt <code>\$echo \${FN#*.}</code> file.txt
<code>\${переменная##шаблон}</code>	Удаляет из строки, хранящейся в переменной, максимальную подстроку, заданную шаблоном. Поиск ведётся с начала строки.	<code>\$FN=my.file.txt</code> <code>\$echo \${FN##my}</code> .file.txt <code>\$echo \${FN##*.}</code> txt

Обратите внимание на то, что само значение переменной не меняется.

Для того, чтобы значение переменной гарантированно оказалось в нужном регистре (верхнем либо нижнем), необязательно прибегать к манипуляциям со строкой, описанным в таблице 12.1. Для этого достаточно объявить переменную с помощью встроенной команды `declare`.

`declare -l STR1` обеспечит ситуацию, при которой в хранимой переменной строке все символы верхнего регистра будут автоматически преобразовываться в символы нижнего регистра. (Отключить это свойство можно через вызов `declare +l STR1`). Аналогично, можно объявить переменную через `declare -u`. Тогда все символы будут в верхнем регистре. Например:

```
[demo@localhost ~]$ declare -u STR2
[demo@localhost ~]$ STR2="Hello, World"
[demo@localhost ~]$ echo $STR2
HELLO, WORLD
[demo@localhost ~]$ declare +u STR2
[demo@localhost ~]$ STR2="Hello, World"
[demo@localhost ~]$ echo $STR2
Hello, World
```

Рис. 12.3

10.3. Команды tr и cut

Самый простой способ заменить что-то в потоке данных - это использовать утилиту tr (translate). В качестве параметра ей передаются два набора символов: «что менять» и «на что менять».

```
[demo@localhost ~]$ echo "Linux" | tr 'x' 'z'
linuz
[demo@localhost ~]$ echo "'inux" | tr '[zxc]' 'X'
linuX
[demo@localhost ~]$ echo "Linux" | tr '[a-z]' '[A-Z]'
LINUX
[demo@localhost ~]$ echo "Linux" | tr 'n' '\t'
Li      ux
```

Рис. 12.4. Использование tr для замены

Команда tr также может использоваться для удаления символов из текстового потока. Для этого предусмотрена опция “-d”:

```
[demo@localhost ~]$ echo "abcdef" | tr -d [bcd]
aef
```

Рис 12.5. Удаление символов

Довольно часто строка представляет собой целый набор каких-то данных, из которых необходимо выделить какую-то конкретную часть. Это можно осуществить с помощью команды cut (не путать с cat). Команда имеет два основных параметра: -d (разделитель) и -f (номер поля). Предположим, нам необходимо получить список локальных пользователей системы и расположение их домашних каталогов. Эта информация хранится в файле /etc/passwd:

```
root:x:0:0:root:/root:/bin/bash
...
user:x:1000:1000::/home/user:/bin/bash
```

Рис. 12.6. Данные о пользователях

Данные хранятся в виде полей, разделённых двоеточием. Для нашей задачи нам требуется первое и шестое поле. Отделить от прочей информации их можно так:

```
cut -d: -f1,6 /etc/passwd
```

или более наглядный вариант:

```
cut -d: -f1,6 --output-delimiter=' ' /etc/passwd | column -t
```

Рис. 12.7.

Если данные в файле не имеют разделителя, но имеют стандартную ширину, то можно воспользоваться опциями -b (байты) или -c (символы), например:

```
cut -c1-4 /etc/passwd
```

Рис.12.8. Отобразить первые 4 символа каждой строки файла /etc/passwd

Иногда необходимо, чтобы строки в потоке данных оказывались отсортированными. Для этого применяется команда `sort`. По-умолчанию она сортирует данные по алфавиту, при этом строки, содержащие цифры, располагаются выше, а буквы нижнего регистра выше букв верхнего. Некоторые полезные параметры команды `sort`:

- -u - при множественном повторе значений выводить только первое (`uniq`)
- -k N - сортировать по полю с номером N (разделителем полей считается пробел)
- -r - сортировать в обратном порядке
- -n - Числовая (`numeric`) сортировка.

Предположим, нам нужно получить список локальных пользователей системы, отсортированный в алфавитном порядке. Это можно сделать так:

```
cut -d:1 -f1 /etc/passwd | sort
```

Рис. 12.9. Отсортированный список локальных пользователей

Задание 10.1 Получение уникальных значений

Цель задания: научиться отбирать из потока уникальные строки

Для выполнения задания вам потребуются следующие команды:

ps - получение списка процессов системы
sort - сортировка данных

Каждый процесс, работающий в системе, имеет владельца (пользователя, с правами которого он выполняется. Напишите сценарий, позволяющий получить отсортированный по алфавиту список всех пользователей, от лица которых в данный момент выполняются какие-либо процессы, например:

```
avahi  
named  
postfix  
...
```

Ответ к заданию 10.1

В арсенале системного администратора присутствует команда `ps`, которая позволяет вывести список всех процессов с их атрибутами: PID'ом, приоритетом, владельцем и т.д. Нас интересует только владелец. Соответственно, команда будет выглядеть так:

```
ps -e -o user
```

Рассмотрите её вывод. Для каждого процесса выводится его владелец. Список получился разрозненным, наша задача заключается в том, чтобы его отсортировать и удалить повторы.

Кроме того, команда `ps` выводит заголовок. От него нужно избавиться, иначе строка заголовка также окажется в результатах. Для отключения заголовка у команды `ps` есть собственное средство - опция `"--no-headers"`. В итоге получаем следующий вид сценария:

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

```
ps -e -o user --no-headers | sort -u
```

```
exit 0
```