

Глава 10. Написание сценариев Bash.

1. Сценарии оболочки.

Сценарии оболочки

- Сценарий – текстовый файл с командами и синтаксическими конструкциями
- Команды выполняются последовательно
- Могут запускаться как команды или как аргумент при запуске оболочки

Сценарий оболочки представляет собой текстовый файл, который состоит из системных и встроенных команд и синтаксических конструкций встроенного в оболочку языка программирования.

Сценарии предназначены для автоматизации выполнения рутинных задач.

Оболочка последовательно интерпретирует и выполняет команды, заданные в сценарии.

Примечание: Эти же команды могут быть выполнены простым последовательным вызовом их в командной строке оболочки.

Для файлов сценариев оболочки Bash принято использовать суффикс `.sh`.

Сценарии оболочки могут быть исполнены двумя различными путями:

1. Имя файла сценария можно указать в качестве аргумента командной строки при запуске оболочки, на языке которой написан данный скрипт. В этом случае файл сценария должен быть доступен для чтения.

2. Сценарий можно запустить так же, как и обычные системные команды. Файл сценария в этом случае должен быть доступен для чтения и исполнения.

Пример:

Запуск сценария путем явного вызова оболочки и указания имени сценария в качестве аргумента:

Глава 10. Написание сценариев Bash.

```
$ bash myscr1.sh  
Privet!
```

Сценарии оболочки

- Для отладки полезны опции `-v` или `-x`
- Конструкция `#!` – указывает оболочку, которая исполняет команды (shebang)
- Команда `source` или `.` – inline подстановка

При отладке сценариев полезны опции `-v` и `-x` `bash`.

1. Опция `-v` переводит оболочку в режим подробного информирования о работе. В этом режиме отображаются команды сценария перед их интерпретацией.

2. Опция `-x` отображает результаты интерпретации команд.

Для автоматического запуска требуемой оболочки в случае, если сценарий запускается как команда, в первой строке сценария должна находиться строка с полным именем оболочки:

Пример:

```
$ cat myscr1.sh
#!/bin/bash
echo 'Privet!'
```

***Примечание:** Строка `#!/bin/bash` указывает с помощью какой оболочки должен быть интерпретирован и выполнен сценарий. Эта директива должна находиться в файле сценария в первой строке. Запуск сценария при неявном вызове оболочки возможен, если на файл сценария имеются разрешения для чтения и исполнения:*

```
$ chmod a+rx myscr1.sh
$ ./myscr1.sh
Privet!
```

***Примечание:** Обратите внимание на то, что нахождение исполняемого файла в текущем каталоге не означает возможности его запуска без указания пути к нему.*

При необходимости выполнения сценария в контексте вызывающего сценария

Глава 10. Написание сценариев Bash.

(оболочки) необходимо использовать команду точка (.), которая называется *inline* подстановкой.

Пример:

```
$ . .bashrc
```

Примечание: Если просто вызвать сценарий из другого сценария, то вызываемый сценарий будет исполнен в собственной оболочке. То есть переменные, значения которых были установлены в вызываемом сценарии, не будут известны в вызывающем сценарии. *Inline* подстановка часто используется для считывания в теле сценария переменных, заданных в другом файле.

Пример:

```
$ cat myscr2rc
VAR1='Snova Privet!'
```

```
$ cat myscr2.sh
#!/bin/bash
. myscr2rc
echo $VAR1
```

```
$ ./myscr2.sh
Snova Privet!
```

Примечание: В этом примере в коде сценария `myscr2.sh` была выполнена *inline* подстановка содержимого файла `myscr2rc`, в котором была установлена переменная `VAR1`.

2. Задание.

1. Напишите сценарий `scr1.sh` так, чтобы в нем определялась переменная `V1` и выводилось ее значение.
2. Запустите сценарий, указывая оболочку явно.
3. Измените скрипт для неявного вызова оболочки.
4. Перепишите сценарий `scr1.sh` таким образом, чтобы из него вызывался сценарий `scr2.sh`, который и печатал бы значение переменной `V1`.
5. Перепишите сценарий `scr1.sh` таким образом, чтобы значение переменной `V1` считывалось бы из файла `scr1rc`.

3. Использование переменных оболочки.

Использование переменных оболочки

- Переменные – строки
- Можно выполнять простейшие арифметические действия
- Имена чувствительны к регистру
- Имя должно начинаться с буквы или символа _

Переменные оболочки являются строковыми переменными.

Память, необходимая для размещения значений переменных, выделяется оболочкой динамически по мере надобности.

Если переменные хранят строки, которые представляют собой целые числа, то допускается выполнение простейших арифметических операций:

Пример:

```
$ V1=10
$ echo $(( V1+1 ))
11
```

Имя переменной должно использовать только символы английского алфавита, цифры и символ подчеркивания.

Имя переменной должно начинаться либо с буквы, либо с символа подчеркивания.

Имена переменных чувствительны к регистру.

Для исключения неверной интерпретации оболочкой имя переменной можно поместить в фигурные скобки.

Пример:

```
$ f1=str
$ echo $f1
str
$ echo ${f1}
```

Глава 10. Написание сценариев Bash.

```
$ echo ${f1}1  
str1
```

Примечание: Здесь переменной `f1` присвоено значение `str`. При необходимости добавить к строке `str` символ `1` это нельзя сделать без экранирования имени переменной фигурными скобками, так как иначе оболочка пытается вывести значение переменной `f11` вместо того чтобы добавить символ `1` к значению переменной `f1`.

Использование переменных оболочки

- `${}`
- `${имя:-значение}`
- `${имя:=значение}`
- `${имя:+значение}`

Имеется возможность использовать “значение по умолчанию” для переменной. Это значит следующее: если переменная определена, то используется ее значение. В противном случае используется “значение по умолчанию”. Синтаксис его следующий: `${переменная:-значение}`.

Пример:

```
$ V1=abcdef
$ echo ${V1:-09876}
abcdef

$ echo ${V2:-09876}
09876
$ echo $V1
abcdef

$ echo $V2
```

***Примечание:** В этом примере переменная V1 определена, а V2 - нет. Поэтому для переменной V1 конструкция `${V1:-09876}` дает значение переменной, а для V2 `${V2:-09876}` - возвращает “значение по умолчанию” 09876. При этом значения переменных V1 и V2 не изменяются.*

Если необходимо назначить не имеющей значения переменной “значение по умолчанию”, следует использовать конструкцию `${переменная:=значение}`.

Пример:

```
$ echo ${V2:=12345}
```


Глава 10. Написание сценариев Bash.

```
12345
$ echo $V2
12345
```

Примечание: Из этого примера видно, что так как переменная V2 не была определена, то вместо ее было использовано значение 12345, которое при этом было назначено этой переменной.

1. При необходимости использовать вместо определенной переменной заданную строку используют конструкцию `${переменная:+значение}`.

Пример:

```
$ echo ${V2:+'nu i dela...'}
nu i dela...
```

Примечание: В этом примере вместо значения непустой переменной была использована строка - "значение по умолчанию".

4. Массивы

Массивы

- `declare`
 - `-A`
 - `-a`
- `unset`

Для создания простого массива используется команда `declare -a` (не обязательна)

Ассоциативный массив создается командой `declare -A` (обязательна)

Пример: Создадим простой массив и поработаем с его элементами:

```
$ declare -a array
$ array[0]=abc
$ array[1]=123
$ echo ${array[0]} — первый элемент массива
abc
$ echo ${array[1]} — второй элемент массива
123
$ echo ${array[*]} — все элементы массива
abc 123
$ echo ${#array[*]} — количество элементов массива
2
$ array=(a b c) — другой способ определить или переопределить массив
$ echo ${#array[*]}
3
$ echo ${array[@]} — другой способ получить все элементы массива
a b c
$ unset array — уничтожение массива
$ echo ${array[*]}
Ничего не получаем
```

Пример: с ассоциативным массивом

```
$ declare -A ages — объявление
```

Глава 10. Написание сценариев Bash.

```
$ ages[John]=25
$ ages[Marry]=23
$ echo ${ages[*]}
23 25
$ echo ${!ages[*]} — получение ключей массива
Marry John
$ echo ${#ages[*]}
2
$ echo ${ages[John]}
25
```

Для работы с массивами как правило приходится использовать циклы.

5. Интерактивная установка значений переменных.

Интерактивная установка значений переменных

- Команда `read` – считывает значения переменных из стандартного потока ввода

Для задания значения переменной пользователем предназначена команда `read`, которая читает вводимые данные из стандартного потока ввода.

Пример:

```
$ echo -n 'Введите строку: '; read var; echo $var
Введите строку:Добрый день!
Добрый день!
```

Примечание В этом примере с клавиатуры должно быть введено значение переменной `var`. Опция `-n` команды `echo` использована здесь для отключения перевода строки после вывода строки приглашения.

Пример: Можно одновременно ввести значения для нескольких переменных сразу:

```
read var1 var2 var3
```

Если команде `read` передается большее количество значений, чем число указанных в команде `read` переменных, то все лишние значения в виде целой строки присваиваются последней переменной в списке.

Если задано больше переменных, чем введено значений, то переменным, для которых не хватило значений, присваивается пустая строка.

При отсутствии переменной как аргумента при выполнении команды `read` считанное значение записывается в переменную `Reply`.

6. Позиционные параметры.

Позиционные параметры

- Позиционные параметры – аргументы скрипта
- \$0 – имя команды;
- \$1 до \$9 – аргументы с 1 по 9
- \${10}... – аргументы с 10 и далее
- Команда `shift` – сдвигает позиционные параметры

Позиционные параметры позволяют передавать сценариям оболочки аргументы, указанные в командной строке.

В Bash можно использовать десять позиционных параметров: \$0, \$1, ... \$9. Они содержат:

1. \$0 – имя команды;
2. параметры от \$1 до \$9 - значения девяти аргументов командной строки от первого до, соответственно, девятого.

Пример:

```
$ cat param.sh
#!/bin/bash
echo $1
echo $2

$ ./param.sh first second
first
second
```

***Примечание:** В сценарии `param.sh` содержатся две команды `echo`, выводящие содержимое первого и второго позиционных параметров, которое соответствует первому и второму аргументу командной строки.*

Позиционные параметры

- `$*` - строка из значений всех аргументов командной строки, разделенных символом разделителем по умолчанию, заданному в переменной `IFS`;
- `$@` - строка из значений всех аргументов командной строки, разделенных пробелами;
- `$#` - количество аргументов командной строки;
- `$?` - код возврата предыдущей команды;
- `$$` - PID оболочки.

Кроме позиционных параметров в Bash используются следующие специальные параметры:

1. `$*` - содержит строку, составленную из значений всех аргументов командной строки, разделенных символом разделителем по умолчанию, заданному в переменной `IFS`;
2. `$@` - содержит строку, составленную из значений всех аргументов командной строки, разделенных пробелами;
3. `$#` - количество аргументов командной строки;
4. `$?` - код возврата предыдущей команды;
5. `$$` - PID оболочки.

Пример:

```
$ cat comline.sh
#!/bin/bash
echo $*

$ ./comline.sh myarg
./comline.sh myarg
```

Примечание: В этом примере сценарий выводит командную строку целиком.

При необходимости получить значение аргументов (опций) командной строки, содержащей более девяти таких аргументов, необходимо использовать команду `shift`.

При использовании команды `shift`, вызванной без аргументов, позиционные параметры переназначаются со смещением на одну позицию вправо

Глава 10. Написание сценариев Bash.

***Примечание:** Таким образом, \$1 получит значение, которое ранее имел \$2, параметр \$2 – значение \$3 и так далее.*

Пример:

```
$ cat param.sh
#!/bin/bash
echo $1
echo $2
echo And now all parameters are shifted.
shift
echo $1
echo $2
```

```
$ ./param.sh first second third
first
second
And now all parameters are shifted.
second
third
```

***Примечание:** В этом сценарии сначала выводятся аргументы командной строки без сдвига. То есть, выводятся первый и второй аргумент командной строки, а после использования команды shift – второй и третий, хотя использованы все те же позиционные параметры \$1 и \$2.*

Для сдвига более чем на одну позицию вправо необходимо указать величину сдвига, используя аргумент команды shift.

Пример: shift 4 сдвинет параметры на четыре позиции вправо.

Вернуться после сдвига параметра к их предыдущим значениям нельзя.

***Примечание:** Поэтому при необходимости запомнить более девяти аргументов командной строки необходимо сохранить в переменных оболочки первые параметры, значения которых будут потеряны при сдвиге.*

При необходимости назначить новые значения позиционным параметрам можно воспользоваться командой set, которая позволяет сконструировать командную строку заново, назначая новые значения сразу всем позиционным параметрам.

Пример:

```
$ cat param.sh
#!/bin/bash
echo $1
echo $2
set 1 2
echo $1
echo $2

$ ./param.sh first second
first
second
1
```

Примечание: Здесь продемонстрировано, что изменение всех аргументов командой `set` приводит к изменению позиционных параметров.

При необходимости сохранить значение одного или нескольких позиционных параметров следует использовать команду `set`, указав в требуемых позициях позиционные параметры, не подлежащие изменению.

Пример:

```
set $1 $2 newarg $4
```

Примечание: В этом случае командная строка текущей оболочки содержит только четыре аргумента. Команда `set` изменяет все их сразу, а для сохранения значений первому, второму и четвертому аргументам просто присваиваются их же старые значения. Третий аргумент получает здесь новое значение - `newarg`.

7. Задание.

1. С помощью какой команды оболочки можно получить значение переменной, имя которой содержится в другой переменной?
2. Каким образом экспортировать переменную, чье имя содержится в другой переменной?
3. Куда помещаются экспортированные переменные?
4. Поместите в переменную ENV все имена переменных, находящихся в `environ`.
5. Где можно увидеть переменные окружения, установленные для процесса сервера SMTP в вашей системе?
6. Напишите простой сценарий оболочки, считывающий значения трех переменных из командной строки и выводящий их значения в стандартный поток вывода. Проверьте его работу, вводя два, три и четыре значения.
7. На работу команды `read` оказывает влияние переменная окружения `IFS`. Она устанавливает символ - разделитель для ввода. По умолчанию - пробел. Как проверить работу этой переменной, не повливав на работу интерактивной оболочки?
8. Напишите сценарий, выводящий имя команды, количество аргументов и PID процесса оболочки.
9. Проверьте работу сценария, используя явный вызов оболочки `bash`.
10. Исследуйте работу опции `-v` `bash`.
11. Измените сценарий так, чтобы он выводил все аргументы командной строки.
12. Измените сценарий, используя команду `shift`. Проверьте, сдвигаются ли значения позиционных параметров.
13. С помощью команды `set` установите в сценарии новые значения позиционных параметров.

8. Оператор test.

Оператор test

- Проверка существования и атрибутов файлов
- Сравнение файлов
- Проверка установки опций оболочки
- Сравнение строк
- Сравнение целых чисел
- Конструкция [] – синоним test

Встроенная команда `test` позволяет проверять выполнение условий, задаваемых в качестве аргументов.

Виды проверок, выполняемых командой `test`, можно разделить на следующие категории:

1. проверка существования и атрибутов файлов
2. сравнение файлов
3. проверка установки опций оболочки
4. сравнение строк
5. сравнение целых чисел.

В случае если тест выполнен успешно, команда `test` возвращает нулевой код возврата. В противном случае – ненулевой.

Пример: используя опцию `-e` команды `test` можно проверить существует ли файл:

```
$ test -e /etc/passwd
$ echo $?
0
$ test -e not_existent_file
$ echo $?
1
```

***Примечание:** В первом случае команда `test` вернула нулевой код возврата, поскольку файл, указанный в качестве аргумента, существует. Во втором случае был указан несуществующий файл и команда вернула код ошибки.*

Глава 10. Написание сценариев Bash.

Конструкция `test` может использоваться в другой форме, в которой команда `test` заменяется квадратными скобками с условием внутри скобок `[условие]`.

Пример: та же задача, что была поставлена в предыдущем примере, может быть решена таким образом:

```
$ [ -e /etc/passwd ]
$ echo $?
0
```

Примечание: Команда `[-e /etc/passwd]` эквивалентна команде `test -e /etc/passwd`.

Наиболее часто используемые опции команды `test`, связанные с проверкой файлов:

1. `-e` - файл существует;
2. `-f` - файл является обычным файлом (plain file);
3. `-d` - файл является каталогом;
4. `-h` или `-L` - файл является символической ссылкой;
5. `-r` - файл доступен для чтения;
6. `-w` - файл доступен для записи;
7. `-x` - файл доступен для исполнения;
8. `-s` - файл не пуст;
9. `-N` - файл был модифицирован с момента, когда он был последний раз открыт для чтения.

Формат вызова `test`, который используется при сравнении файлов следующий:

1. `[file1 -nt file2]` - возвращает истину если первый файл имеет более позднюю дату модификации. Если второй файл не существует, то команда возвращает истину.
2. `[file1 -ot file2]` - возвращает истину если первый файл имеет более раннюю дату модификации. Если первый файл не существует, то команда возвращает истину.
3. `[file1 -ef file2]` - возвращает истину если между файлами установлена жесткая связь.

Опция `-o` позволяет проверять установку опций оболочки:

Пример:

```
$ set -o noclobber
$ [ -o noclobber ]
$ echo $?
0
```

Глава 10. Написание сценариев Bash.

```
$ set +o noclobber
$ [ -o noclobber ]
$ echo $?
1
```

Для сравнения строк применяется следующий формат вызова `test` :

1. `[str1 = str2]` - проверка на совпадение строк;
2. `[str1 != str2]` - проверка на несовпадение строк;
3. `[str1 < str2]` - истина если при сортировке строка `str1` окажется раньше, чем `str2`;
4. `[str1 > str2]` - истина если при сортировке строка `str1` окажется позже, чем `str2`;
5. `[-z str]` - истина если длина строки нулевая;
6. `[-n str]` - истина если длина строки ненулевая.

Сравнение целых чисел производится с помощью следующих опций команды `test` :

1. `-eq` - равенство;
2. `-ne` - неравенство;
3. `-lt` - меньше;
4. `-le` - меньше или равно;
5. `-gt` - больше;
6. `-ge` - больше или равно.

Пример:

```
$ [ 1 -lt 2 ]
$ echo $?
0
```

```
$ [ 1 -eq 2 ]
$ echo $?
1
```

9. Условное исполнение команд.

Условное исполнение команд

- `&&` - успешный запуск предыдущей команды
- `||` - неудачный запуск предыдущей команды
- `if условие`
 `then`
 команды
 `fi`

Операторы условного исполнения команд `&&` и `||` позволяют исполнять команды в зависимости от успешности выполнения предыдущих команд.

Если оператор `&&` установлен между двумя командами, то вторая из них будет исполнена только в случае успеха предыдущей.

Пример:

```
$ ls /tmp &> /dev/null && cd /tmp
$ pwd
/tmp
```

***Примечание:** В этом примере первая команда (`$ ls /tmp &> /dev/null`) используется лишь для проверки существования каталога, переход в который осуществляет вторая команда (`cd /tmp`) при условии успешного исполнения первой.*

Пример: Более изящный вариант выполнения этой же задачи заключается в использовании команды `test` для проверки существования целевого каталога:

```
$ [ -d /tmp ] && cd /tmp
$ pwd
/tmp
```

***Примечание:** Команда `test` в этом примере проверяет существование каталога. В случае существования этого каталога осуществляется переход в него.*

Если между командами установлен оператор `||`, то вторая команда будет выполнена в случае неудачного завершения работы первой команды.

Глава 10. Написание сценариев Bash.

Пример: требуется перейти в каталог d1. В случае его отсутствия этот каталог необходимо создать и перейти в него. Эту задачу можно решить следующим образом:

```
$ [ -d d1 ] || mkdir d1 ; cd d1
$ pwd
/tmp/d1
```

Примечание: Команда `test` проверила наличие каталога. Если он отсутствует, то этот каталог создается. Далее осуществляется переход в заданный каталог.

Оболочка предоставляет также специальный оператор условного выполнения `if`

Пример: предположим, что требуется обеспечить выход из некоторого сценария с ошибкой если в командной строке установлено отличное от единицы число аргументов командной строки. При этом команда должна сообщать об ошибке и выдавать подсказку о правильном варианте ее использования. Ниже приведен текст программы:

```
#!/bin/bash

if [ $# -ne 1 ]
then
    cat <<- ERR
        Не хватает аргументов.
        Использование:
        if.sh file
        Аргумент file должен быть обычным файлом.
    ERR
    exit 1
fi

ls -l $1
```

Примечание: Команда `if` исполняет блок команд после команды `then` только в случае получения нулевого кода возврата команды, указанной в качестве ее аргумента. В данном случае аргумент команду `if` - это команда `test`, которая в данном случае проверяет неравенство количества аргументов, переданных сценарию, единице. Если количество аргументов не равно единице, то выполняется блок операторов после `then` до команды `fi`, заканчивающей `if`. Конструкция `cat <<- ERR` - это "here document". Она позволяет указывать целый блок данных непосредственно в теле скрипта, передавая его в данном случае в стандартный поток ввода команде `cat` для вывода на экран. Блок данных ограничен строкой `ERR`. В этом примере используется оператор `<<-` вместо `<<` для игнорирования табуляций перед блоком данных.

Ниже приведены результаты испытания программы:

```
$ ./if.sh
Не хватает аргументов.
Использование:
if.sh file
Аргумент file должен быть обычным файлом.
```

Глава 10. Написание сценариев Bash.

```
$ ./if.sh if.sh
-rwxr--r--    1 aberes    aberes          172 Окт  8 19:34 if.sh
```

Команда `if` допускает использование команды `elif` для выполнения дополнительной проверки.

Пример:

```
#!/bin/bash

if [ $# -ne 1 ]
then
    cat <<- ERR
        Не хватает аргументов.
        Использование:
        if.sh file
        Аргумент file должен быть обычным файлом.
    ERR
    exit 1
elif [ ! -f $1 ]
then
    echo -n 'Тип файла '
    file $1
    exit 1
fi

ls -l $1
```

Примечание: Если в качестве аргумента задан не обычный файл, то выводится тип этого файла.

```
$ ./if.sh .
Тип файла .: directory
```

В команде `if` можно также использовать `else` для указания блока команд, которые должны исполняться в случае, если предыдущие проверки не закончились успехом.

Пример: программу можно модифицировать следующим образом:

```
#!/bin/bash

if [ $# -ne 1 ]
then
    cat <<- ERR
        Не хватает аргументов.
        Использование:
        if.sh file
        Аргумент file должен быть обычным файлом.
    ERR
```

Глава 10. Написание сценариев Bash.

```
        exit 1
elif [ ! -f $1 ]
then
        echo -n 'Тип файла '
        file $1
        exit 1
else
        ls -l $1
fi
```


10. Оператор case.

Оператор case

```
case слово in
  шаблон1 )
    команды
    ;;
  шаблон2 )
    команды
    ;;
  *)
    команды
    ;;
esac
```

Оператор `case` позволяет проверить значение, содержащееся в переменной, на совпадение с заданными шаблонами.

Метасимволы шаблонов сравнения для `case` такие же, как метасимволы шаблонов для имен файлов.

Синтаксис команды `case` в общем виде таков:

```
case слово in
  шаблон1 )
    команды
    ;;
  шаблон2 )
    команды
    ;;
esac
```

Сравнение слова с шаблонами производится последовательно. Как только найдется совпадение выполняются соответствующие команды и дальнейших проверок не выполняется.

Пример: Допустим, что в текущем каталоге располагаются символические ссылки на сценарии запуска служб GNU/Linux. Требуется написать сценарий, который будет запускать службы, передавая сценарию запуска службы аргумент `start`, если в качестве аргумента будет использована символическая ссылка, начинающаяся с букв `S` или `s`. Если же ссылка

Глава 10. Написание сценариев Bash.

будет начинаться с букв K или k, то служба должны останавливаться и, следовательно, их сценариям должен передаваться аргумент stop.

Ниже приведен текст сценария:

```
$ cat case.sh
#!/bin/bash

[ $# -ne 1 ] && exit 1;

FIRST=`echo $1 | cut -c1`

case $FIRST in
    [Ss] )
        echo "Стартую $1"
        ./$1 start
        ;;
    K|k )
        echo "Останавливаю $1"
        ./$1 stop
        ;;
    * )
        echo "Статус $1"
        ./$1 status
        ;;
esac
```

Примечание: В этом сценарии в начале проверяется количество аргументов. Если оно не равно 1, то осуществляется выход с ошибкой. Далее в переменную FIRST помещается первая буква аргумента командной строки (предполагается, что это имя символической ссылки).

Команда case проверяет соответствие содержащейся в переменной FIRST буквы шаблону [Ss]. Этот шаблон обозначает множество, аналогично с обычными файловыми шаблонами. То есть, этому шаблону удовлетворяют либо S, либо s.

Если буква, содержащаяся в FIRST удовлетворяет этому шаблону, то выводится сообщение о запуске службы и она запускается.

Для остановки службы (в качестве примера) используется иной шаблон, имеющий в данном случае тот же смысл: либо K, либо k. Вертикальная черта обозначает "или". Однако, ее можно применять даже для целых строк или шаблонов.

Если содержимое переменной FIRST не совпадает и с этим шаблоном, то выводится информация о текущем статусе службы.

Ниже приведен пример работы этого сценария:

```
# ls *postfix
K11postfix postfix S89postfix

# ./case.sh S89postfix
```

Глава 10. Написание сценариев Bash.

```
Стартую S89postfix
* Starting postfix... [ ok ]

# ./case.sh postfix
Статус postfix
* status: started

# ./case.sh K11postfix
Останавливаю K11postfix
* Stopping postfix... [ ok ]
```

11. Задание.

1. Как с помощью `test` проверить является файл специальным файлом символического устройства?
2. Предполагается, что имеется переменная `STR1`, которой, вероятно, присвоено значение `str1`. Однако доподлинно это не известно и переменная может быть не инициализирована вообще. Как, используя `test`, проверить содержит ли она значение `str1`?
3. Проверьте установлен ли в текущей оболочке запрет на использование символа конца файла `C^D` для выхода из сеанса.
4. С помощью `test` проверьте имеется ли жесткая связь между `gzip` и `gunzip`.
5. Напишите сценарий, проверяющий имя текущего каталога и выводящий сообщение об ошибке, если оно короче пяти символов.
6. Требуется проверить относится ли файл, заданный в качестве аргумента, каталогом или же обычным файлом. Если верно последнее, то сценарий должен выводить имя файла и его размер. В случае, если размер файла превышает 1Кб, то размер должен выводиться в килобайтах. Если размер превышает мегабайт - в мегабайтах.
7. Изучите любой сценарий запуска в `/etc/init.d`. Как в этом сценарии используется команда `case` ?
8. Напишите сценарий, анализирующий список пользователей, находящихся в настоящий момент в системе. В скрипте список пользователей должен быть преобразован в одну строку. В случае, если имеется хотя-бы один сеанс `root` должно выдаваться предупреждающее сообщение. Анализ должен быть осуществлен с помощью команды `case`.

12. Циклы.

Циклы

- Цикл `for` для работы со списком
`for переменная in список`
`do`
 команда
`done`
- Цикл `for` со счетчиком
`for ((i=1; i<=N ; i++))`
`do`
 команда
`done`

Для программирования в оболочке доступны три вида циклов:

1. `for` - этот оператор позволяет создавать циклы типа перебора значений;
2. `while` - цикл, выполняющийся до тех пор, пока истинно некоторое условие;
3. `until` - цикл, выполняющийся до тех пор, пока некоторое условие ложно.

Пример: Допустим, что имеется некоторый набор значений, которые должны быть последовательно присвоены некоторой переменной, требующейся для произведения каких-либо операций. В этом случае удобно использовать команду `for`.

Пусть, например, требуется в цикле вывести права доступа к нескольким каталогам:

```
#!/bin/bash

for DIR in /etc /tmp /var
do
    echo -n "Права доступа к $DIR "
    ls -ld $DIR | cut -c2-11
done
```

Примечание: В этом сценарии переменная `DIR` последовательно принимает три значения: `/etc`, `/tmp` и `/var`. Это достигается с помощью команды `for`, в которой список значений указан после `in`. Тело цикла начинается после команды `do` и ограничивается `done`.

Ниже показаны результаты работы сценария:

Глава 10. Написание сценариев Bash.

```
$ ./for.sh
Права доступа к /etc  rwxr-xr-x
Права доступа к /tmp  rwxrwxrwt
Права доступа к /var  rwxr-xr-x
```

Если в команде `for` не указан список после директивы `in`, то переменная цикла будет последовательно принимать значения, соответствующие аргументам командной строки.

Пример: Изменим предыдущий сценарий так, чтобы имена каталогов можно было бы задавать в качестве аргументов в командной строке:

```
#!/bin/bash

[ $# -lt 1 ] && exit 1

for DIR
do
    if [ -d $DIR ]; then
        echo -n "Права доступа к $DIR "
        ls -ld $DIR | cut -c2-11
    elif [ ! -e $DIR ]; then
        echo "$DIR не существует"
    fi
done
```

***Примечание:** В этом сценарии переменная `DIR` последовательно принимает значения аргументов командной строки, так как в команде `for` не задан список значений. Ниже приведен пример работы такого сценария:*

```
$ ./for.sh /usr /rsu /root
Права доступа к /usr  rwxr-xr-x
/rsu не существует
Права доступа к /root rwx-----
```

Циклы

- Цикл `while` работает с предпроверкой условия

```
counter=0
while [[ $counter -le N ]]
do
    echo $counter
    (( counter++ ))
done
```

- Цикл `until` работает аналогично `while`

Операторы `while` и `until` удобно использовать для организации итерационных (со счетчиком) циклов.

Пример: для вывода в цикле списка значений от 1 до 10 можно написать такой сценарий:

```
#!/bin/bash

i=1
while [ $i -le 10 ]; do
    echo $i
    i=$((i+1))
done
```

Этот скрипт последовательно выводит значения:

```
$ ./while.sh
1
2
3
4
5
6
7
8
9
10
```

Глава 10. Написание сценариев Bash.

Цикл `while` работает до тех пор, пока команда, указанная в качестве ее аргумента, возвращает успешный код завершения.

Цикл `until` работает пока команда - аргумент заканчивается неудачей.

Пример: последовательно удалим из полного доменного имени узла (FQDN) подстроки до первой точки. Сначала должно быть выведено полное имя узла, затем домен, затем родительский домен, и так далее, пока строка не станет пустой.

```
#!/bin/bash

HN=`hostname`

until [ -z $HN ]; do
    echo $HN
    HN=`echo -n $HN | tr '.' '\n' | sed '1d' | tr '\n' '.'`
done
```

Примечание: В сценарии используется переменная `HN`, которой в начале присваивается значение - доменное имя узла. Затем из него в цикле удаляется подстрока от начала строки до первой встретившейся точки. Для этого в строке - имени узла точки сначала заменяются на переводы строки, затем удаляется первая строка, и в заключение переводы строки снова заменяются на точки. Цикл `until` работает в этом примере до тех пор, пока содержимое переменной `HN` не станет пустой строкой.

Результаты работы сценария:

```
$ hostname
nechto.zamislovatoe.tmn.ru

$ ./until.sh
nechto.zamislovatoe.tmn.ru
zamislovatoe.tmn.ru
tmn.ru
ru
```


13. Задание.

1. Напишите сценарий, позволяющий в цикле создавать в текущем каталоге символические ссылки на файлы, заданные как его аргументы. Причем для каждого файла должно запрашиваться подтверждение на создание ссылки. В этом сценарии должна использоваться команда `for`.
2. Измените предыдущий сценарий так, чтобы для организации цикла использовались бы `while` или `until`.
3. Напишите сценарий, который помещает идентификаторы пользователей в ассоциативный массив. Затем этот массив используется для того, чтобы напечатать идентификатор пользователя после запроса имени пользователя. Запросы имени должны поступать до тех пор пока пользователь явно не укажет, что желает завершить работу программы. Для получения сведений о пользователях удобно использовать команду `getent`.

14. Функции.

Функции

```
function имя()
{
    команды
}
```

Функции, представляют собой именованные блоки кода, написанные на языке оболочки.

Синтаксис описания функции:

```
function имя()
{
    команды
}
```

Код функции описывают в начале сценария до первого вызова функции.

Для вызова функции достаточно просто указать ее имя.

Пример: Для демонстрации использования функций оболочки модифицируем программу `if.sh` из параграфа об условном выполнении команд. В этой программе можно, например, вынести часть кода, связанную с выводом сообщения об ошибке, в функцию.

```
$ cat if.sh
#!/bin/bash

function err_msg()
{
    cat <<- ERR
        Не хватает аргументов.
        Использование:
        if.sh file
```

Глава 10. Написание сценариев Bash.

```

        Аргумент file должен быть обычным файлом.
    ERR
}

if [ $# -ne 1 ]
then
    err_msg
    exit 1
elif [ ! -f $1 ]
then
    echo -n 'Тип файла '
    file $1
    exit 1
fi

ls -l $1
```

Пример работы сценария `if.sh`:

```
$ ./if.sh
Не хватает аргументов.
Использование:
if.sh file
Аргумент file должен быть обычным файлом.
$ ./if.sh /etc/passwd
-rw-r--r--    1 root    root          2033 Авг 19 23:24
/etc/passwd
```

Для передачи аргументов в функцию их указывают после имени функции, разделяя пробелами.

В теле функции обращение к переданным аргументам производится с помощью позиционных параметров.

Пример: модификация `if.sh`, демонстрирующая передачу аргументов в функцию.

```
$ cat if.sh
#!/bin/bash

function err_msg()
{
    echo "Ошибка: $1"
    cat <<- ERR
        Не хватает аргументов.
        Использование:
        if.sh file
        Аргумент file должен быть обычным файлом.
```

Глава 10. Написание сценариев Bash.

```
        ERR
    }

    if [ $# -ne 1 ]
    then
        err_msg '001'
        exit 1
    elif [ ! -f $1 ]
    then
        echo -n 'Тип файла '
        file $1
        exit 1
    fi

    ls -l $1
```

Примечание: Здесь в функцию передается строка - код ошибки. В теле функции этот код считывается из позиционного параметра.

Пример работы программы:

```
$ ./if.sh
Ошибка: 001
Не хватает аргументов.
Использование:
if.sh file
Аргумент file должен быть обычным файлом.
```

Функции, описанные в отдельных файлах, считываются с помощью inline подстановки.

Пример: Перенесем описание функции `err_msg` в файл `myfunctions`. Ниже приводятся содержимое файла `myfunctions` и измененный текст сценария `if.sh`:

```
$ cat myfunctions

function err_msg()
{
    echo "Ошибка: $1"
    cat <<- ERR
        Не хватает аргументов.
        Использование:
        if.sh file
        Аргумент file должен быть обычным файлом.
    ERR
}

$ cat if.sh
```

Глава 10. Написание сценариев Bash.

```
#!/bin/bash

. ./myfunctions

if [ $# -ne 1 ]
then
    err_msg '001'
    exit 1
elif [ ! -f $1 ]
then
    echo -n 'Тип файла '
    file $1
    exit 1
fi

ls -l $1
```

***Примечание:** В сценарии `if.sh inline` считывается содержимое файла `myfunctions` и функция `err_msg`, описанная в нем становится доступной для использования в сценарии.*

15. Задание.

6. Введите непосредственно в командной строке код функции, выводящей страницу `man` для темы, указанной в качестве аргумента функции. В теле функции перед вызовом `man` используйте команду `env` для временной установки переменной окружения `LANG` в значение `ru_RU.UTF-8`. Испытайте работу функции в командной строке Bash.

7. Измените сценарий `case.sh`, продемонстрированный в параграфе о команде `case`, так, чтобы команды, выполняющиеся при совпадении переменной `FIRST`, были реализованы в функциях.

Wells_20210115.txt

salary_20210115.txt

tab_staff_20210115.txt

18. Глава 10 Задание п.2

1. Напишите сценарий `scr1.sh` так, чтобы в нем определялась переменная `V1` и выводилось ее значение.

Ответ:

```
$ mkdir scripts
$ cd scripts/
$ cat > scr1.sh << END
> V1=test
> echo \"$V1
> END
$ cat scr1.sh
V1=test
echo $V1
```

2. Запустите сценарий, указывая оболочку явно.

Ответ:

```
$ bash scr1.sh
test
```

3. Измените скрипт для неявного вызова оболочки.

Ответ:

```
$ vi scr1.sh
$ cat scr1.sh
#!/bin/bash
V1=test
echo $V1
$ chmod a+x scr1.sh ← Нужно сценарий сделать исполняемым
$ ./scr1.sh ← Текущий каталог не находится в переменной PATH
test
```

4. Перепишите сценарий `scr1.sh` таким образом, чтобы из него вызывался сценарий `scr2.sh`, который и печатал бы значение переменной `V1`.

Ответ:

```
$ cat scr1.sh
```

```
#!/bin/bash
V1=test
. scr2.sh
$ cat scr2.sh
echo $V1
$ ./scr1.sh
test
```

Обратите внимание что `scr2.sh` не обязательно должен быть исполняемым и в нем нет магического числа (`#!/bin/bash`).

5. Перепишите сценарий `scr1.sh` таким образом, чтобы значение переменной `V1` считывалось бы из файла `scr1rc`.

Ответ:

```
$ cat scr1.sh
#!/bin/bash
source scr1rc
. scr2.sh
$ cat scr1rc
V1=testrc
$ ./scr1.sh
testrc
```

Обратите внимание что вместо команды «`.`» была использована команда `source`. Обе команды эквивалентны. Команда `source` делает код более читабельным.

19. Глава 10 Задание п.7

1. С помощью какой команды оболочки можно получить значение переменной, имя которой содержится в другой переменной?

Ответ:

Для решения нужно почитать `man bash`, раздел `Parameter Expansion`. Цитата из `man`:

```
If the first character of parameter is an exclamation point (!),
and
parameter is not a nameref, it introduces a level of indirection.
Bash
uses the value formed by expanding the rest of parameter as the new
pa-
rameter; this is then expanded and that value is used in the rest
```


of

the expansion, rather than the expansion of the original parameter.

This is known as indirect expansion. The value is subject to tilde expansion,

parameter expansion, command substitution, and arithmetic expansion.

If parameter is a nameref, this expands to the name of the

parameter referenced by parameter instead of performing the complete

indirect expansion. The exceptions to this are the expansions of

`${!prefix*}` and `${!name[@]}` described below. The exclamation point

must immediately follow the left brace in order to introduce indirect

tion.

```
$ V1=123
```

```
$ V2=V1
```

```
$ echo ${!V2}
```

```
123
```

2. Каким образом экспортировать переменную, чье имя содержится в другой переменной?

Ответ:

```
$ export $V2
```

```
$ env | grep V1
```

```
V1=123
```

3. Куда помещаются экспортированные переменные?

Ответ:

Специальную область памяти с названием окружение (environ), которая связана с процессом.

4. Поместите в переменную ENV все имена переменных, находящихся в environ.

Ответ:

```
$ ENV=$(env | awk -F= '{print $1}')
```

5. Где можно увидеть переменные окружения, установленные для процесса сервера SMTP в вашей системе?

Ответ: SMTP сервер прослушивает порт 25, поэтому для начала выясним номер процесс для этого порта:

Глава 11. Ответы к заданиями

```
$ sudo netstat -tanp | grep ':25.*0.0.0.0.*LISTEN'
tcp        0      0 127.0.0.1:25          0.0.0.0:*          LISTEN
355/master
```

Нас интересует последняя колонка и в ней только PID процесса:

```
$ sudo netstat -tanp | grep ':25 *0.0.0.0:\ *.*LISTEN' | awk
'{print $NF}' | awk -F/ '{print $1}'
355
```

Теперь мы можем получить переменные окружения с которыми был запущен этот процесс:

```
sudo cat /proc/$(sudo netstat -tanp | grep ':25 *0.0.0.0:\
 *.*LISTEN' | awk '{print $NF}' | awk -F/ '{print $1}')/environ |
tr '\0' '\n'
```

Переменные находятся в файле `/proc/PID_процесса/environ`

Команда `tr '\0' '\n'` заменяет символ Nul на перевод строки, чтобы было удобней читать полученный результат.

6. Напишите простой сценарий оболочки, считывающий значения трех переменных из командной строки и выводящий их значения в стандартный поток вывода. Проверьте его работу, вводя два, три и четыре значения.

Ответ:

```
$ cat vars.sh
#!/bin/bash
echo -n "Enter 3 values: "
read V1 V2 V3
echo V1: $V1
echo V2: $V2
echo V3: $V3
$ ./vars.sh
Enter 3 values: 1 2 3
V1: 1
V2: 2
V3: 3
$ ./vars.sh
Enter 3 values: 1 2
V1: 1
V2: 2
V3:
$ ./vars.sh
```

```
Enter 3 values: 1 2 3 4
```

```
v1: 1
```

```
v2: 2
```

```
v3: 3 4
```

7. На работу команды `read` оказывает влияние переменная окружения `IFS`. Она устанавливает символ - разделитель для ввода. По умолчанию - пробел. Как проверить работу этой переменной, не повлияв на работу интерактивной оболочки?

Ответ:

```
$ IFS=: read V1 V2
```

```
1:2
```

```
$ echo $V1 $V2
```

```
1 2
```

8. Напишите сценарий, выводящий имя команды, количество аргументов и PID процесса оболочки.

Ответ:

```
$ cat args.sh
```

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

```
echo "Command name: $0"
```

```
echo "Number of args: $#"
```

```
echo "PID: $$"
```

```
$ ./args.sh a b c
```

```
Command name: ./args.sh
```

```
Number of args: 3
```

```
PID: 635
```

9. Проверьте работу сценария, используя явный вызов оболочки `bash`.

Ответ:

```
$ bash ./args.sh a b c
```

```
Command name: ./args.sh
```

```
Number of args: 3
```

```
PID: 636
```

10. Исследуйте работу опции `-v` `bash`.

Ответ:

```
$ bash -v ./args.sh a b c
```

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

```
echo "Command name: $0"
```

Глава 11. Ответы к заданиями

```
Command name: ./args.sh
echo "Number of args: $#"
```

Number of args: 3

```
echo "PID: $$"
```

PID: 637

11.Измените сценарий так, чтобы он выводил все аргументы командной строки.

Ответ:

```
$ echo 'echo "All args: $@"' >> args.sh
$ ./args.sh a b c
Command name: ./args.sh
Number of args: 3
PID: 638
All args: a b c
```

12.Измените сценарий, использовав команду shift. Проверьте, сдвигаются ли значения позиционных параметров.

Ответ:

```
$ cat args.sh
#!/bin/bash
echo "Command name: $0"
echo "Number of args: $#"
```

echo "PID: \$\$"

```
echo "All args 0 shift: $@"
shift
echo "All args 1 shift: $*"
shift
echo "All args 2 shift: $*"

$ ./args.sh a b c
Command name: ./args.sh
Number of args: 3
PID: 646
All args 0 shift: a b c
All args 1 shift: b c
All args 2 shift: c
```

13.С помощью команды `set` установите в сценарии новые значения позиционных параметров.

Ответ:

```
$ cat args.sh
#!/bin/bash
echo "Command name: $0"
echo "Number of args: $# "
echo "PID: $$"
echo 'asfter: set $1 BBB CCC DDD'
set $1 BBB CCC DDD
echo "All args 0 shift: $@"
shift
echo "All args 1 shift: $*"
shift
echo "All args 2 shift: $*"

$ ./args.sh a b c
Command name: ./args.sh
Number of args: 3
PID: 650
asfter: set $1 BBB CCC DDD
All args 0 shift: a BBB CCC DDD
All args 1 shift: BBB CCC DDD
All args 2 shift: CCC DDD
```

20. Глава 10 Задание п.11

1. Как с помощью `test` проверить является файл специальным файлом символического устройства?

Ответ:

```
$ test -c /dev/tty1
$ echo $?
0
```

2. Предполагается, что имеется переменная `STR1`, которой, вероятно, присвоено значение `str1`. Однако доподлинно это не известно и переменная может быть не

инициализирована вообще. Как, используя `test`, проверить содержит ли она значение `str1`?

Ответ: Обратите внимание на кавычки в команде они необходимы иначе может быть ошибка:

```
$ test "$STR1" == "str1"
$ echo $?
1
$ STR1=str1
$ test "$STR1" == "str1"
$ echo $?
0
$ unset STR1
$ test $STR1 == "str1"
-bash: test: ==: ожидается использование унарного оператора
```

3. Проверьте установлен ли в текущей оболочке запрет на использование символа конца файла `C^D` для выхода из сеанса.

Ответ:

```
$ test -o noclobber
$ echo $?
1
```

4. С помощью `test` проверьте имеется ли жесткая связь между `gzip` и `gunzip`.

Ответ:

```
$ test $(which gzip) -ef $(which gunzip)
$ echo $?
1
$ test $(which perlbug) -ef $(which perlthanks)
$ echo $?
0
```

5. Напишите сценарий, проверяющий имя текущего каталога и выводящий сообщение об ошибке, если оно короче пяти символов.

Ответ:

```
$ cat checkdir.sh
#!/bin/bash
if [ "$(pwd | awk -F/ '{printf length($NF)}')" -le 5 ]
then
```

```
echo "Error length shorter then 5" > /dev/stderr
exit 1
fi
$ ./checkdir.sh
$ echo $?
0
$ cd /etc
$ /home/user/scripts/checkdir.sh
Error length shorter then 5
$ echo $?
1
```

6. Требуется проверить относится ли файл, заданный в качестве аргумента, каталогом или же обычным файлом. Если верно последнее, то сценарий должен выводить имя файла и его размер. В случае, если размер файла превышает 1Кб, то размер должен выводиться в килобайтах. Если размер превышает мегабайт - в мегабайтах.

Ответ:

```
$ cat testfile.sh
#!/bin/bash
if [ $# -ne 1 ]
then
    echo "Usage: $0 {1arg}"
    exit 1
fi
if [ ! -e $1 ]
then
    echo "Can not stat file: $1"
    exit 2
fi
if [ -f $1 ]
then
    ls -lh $1 | sed -r 's/^(.+ ){4}(.+) (.+ ){3}(.+)/\2 \4/'
elif [ -d $1 ]
then
    echo $1 is directory
fi
```

Глава 11. Ответы к заданиями

```
$ ./testfile.sh /etc
/etc is directory
$ ./testfile.sh src1.sh
Can not stat file: src1.sh
$ ./testfile.sh scr1.sh
36 scr1.sh
$ ./testfile.sh /bin/vim
2,3M /bin/vim
$ ./testfile.sh
Usage: ./testfile.sh {larg}
$ ./testfile.sh qwert asdfg
Usage: ./testfile.sh {larg}
```

7. Изучите любой сценарий запуска в `/etc/init.d`. Как в этом сценарии используется команда `case` ?

Ответ:

Анализируется первый аргумент команды и если он равен `start`, то служба запускается, `stop` — останавливается, `restart` — перезапускается и т.д.

8. Напишите сценарий, анализирующий список пользователей, находящихся в настоящий момент в системе. В скрипте список пользователей должен быть преобразован в одну строку. В случае, если имеется хотя-бы один сеанс `root` должно выдаваться предупреждающее сообщение. Анализ должен быть осуществлен с помощью команды `case` .

Ответ:

```
$ cat checkroot.sh
#!/bin/bash
case $(who | awk '{printf $1}') in
*root*)
    echo "root session opened!!!"
    ;;
*)
    echo "All OK"
    ;;
esac
$ ./checkroot.sh
All OK
$ who
```



```
user      pts/1          2021-01-14 07:59 (192.168.1.198)
$ ./checkroot.sh
root session opened!!!
$ who
user      pts/1          2021-01-14 07:59 (192.168.1.198)
root     pts/2          2021-01-14 13:32 (192.168.1.198)
```

21. Глава 10 Задание п.13

1. Напишите сценарий, позволяющий в цикле создавать в текущем каталоге символические ссылки на файлы, заданные как его аргументы. Причем для каждого файла должно запрашиваться подтверждение на создание ссылки. В этом сценарии должна использоваться команда `for`.

Ответ:

```
$ cat link.sh
#!/bin/bash
for FILE
do
    echo -n "Create link for file ${FILE} in current
directory(y/n): "
    read ANSW
    [ "${ANSW}" == "y" -o "${ANSW}" == "Y" ] && ln -s ${FILE} .
done
$ ./link.sh /etc/ ~/.bashrc
Create link for file /etc/ in current directory(y/n): n
Create link for file /home/user/.bashrc in current
directory(y/n): y
$ ls -l etc .bashrc
ls: невозможно получить доступ к etc: Нет такого файла или
каталога
lrwxrwxrwx. 1 user user 18 янв 14 15:13 .bashrc ->
/home/user/.bashrc
```

2. Измените предыдущий сценарий так, чтобы для организации цикла использовались бы `while` или `until`.

Ответ:

```
$ cat link2.sh
#!/bin/bash
```

```
while [ $# -ne 0 ]
do
echo -n "Create link for file $1 in current directory(y/n): "
read ANSW
[ "${ANSW}" == "y" -o "${ANSW}" == "Y" ] && ln -s $1 .
shift
done
$ ./link2.sh /dev /tmp
Create link for file /dev in current directory(y/n): y
Create link for file /tmp in current directory(y/n): y
$ ls -l dev tmp
lrwxrwxrwx. 1 user user 4 янв 14 15:19 dev -> /dev
lrwxrwxrwx. 1 user user 4 янв 14 15:19 tmp -> /tmp
```

3. Напишите сценарий, который помещает идентификаторы пользователей в ассоциативный массив. Затем этот массив используется для того, чтобы напечатать идентификатор пользователя после запроса имени пользователя. Запросы имени должны поступать до тех пор пока пользователь явно не укажет, что желает завершить работу программы. Для получения сведений о пользователях удобно использовать команду `getent`.

Ответ:

```
$ cat assocarr.sh
#!/bin/bash
declare -A uids
# Для создания массива используем цикл for
for name in $(getent passwd | awk -F: '{print $1}')
do
    uids[$name]=$ (getent passwd $name | awk -F: '{print $3}')
done

# Обратите внимание как создан бесконечный цикл.
# Команда .. это ничего не делать
while [ .. ]
do
    echo -n 'Enter username: '
    read user

    # Никогда не надейтесь что пользователи будут что-то правильно
    вводить
```

```
if [ -z "${uids[${user:-noUser}]}" ]
then
echo 'Invalid user name'
else
echo "Uid of user $user is ${uids[$user]}"
fi
echo -n 'Do you want try another user?(Y/n)'; read answ
answ=$(echo ${answ:=y} | cut -c1 | tr N n)
if [ "${answ}" == n ]
then
echo "Goodby"
exit 0
fi
done
#echo ${uids[*]}
#echo ${!uids[*]}

$ ./assocarr.sh
Enter username:
Invalid user name
Do you want try another user?(Y/n)
Enter username: root
Uid of user root is 0
Do you want try another user?(Y/n)Y
Enter username: user
Uid of user user is 1000
Do you want try another user?(Y/n)foo
Enter username: qwerty
Invalid user name
Do you want try another user?(Y/n)n
Goodby
```

22. Глава 10 Задание п.15

1. Введите непосредственно в командной строке код функции, выводящей страницу `man` для темы, указанной в качестве аргумента функции. В теле функции перед вызовом `man` используйте команду `env` для временной установки переменной окружения `LANG` в

значение `ru_RU.UTF-8`. Испытайте работу функции в командной строке Bash.

Ответ:

```
$ function manru () {  
> env LANG=ru_RU.UTF-8 man $@  
> }
```

или если в одну строку

```
function manru () { env LANG=ru_RU.UTF-8 man $@; }
```

```
$ LANG=en_US.UTF-8
```

```
$ man -P head man
```

```
MAN(1)                                Manual pager utils
```

MAN(1)

NAME

man - an interface to the system reference manuals

SYNOPSIS

man [man options] [[section] page ...] ...

man -k [apropos options] regexp ...

man -K [man options] [section] term ...

man -f [whatis options] page ...

```
$ manru -P head man
```

```
MAN(1)                                Утилиты просмотра справочных страниц
```

MAN(1)

НАЗВАНИЕ

man - доступ к системным справочным страницам

СИНТАКСИС

man [параметры man] [[раздел] страница ...] ...

man -k [параметры apropos] регвыр ...

man -K [параметры man] [раздел] термин ...

man -f [whatis параметры] страница ...

2. Измените сценарий `case.sh`, продемонстрированный в параграфе о команде `case`, так, чтобы команды, выполняющиеся при совпадении переменной `FIRST`, были реализованы в функциях.

Глава 11. Ответы к заданиям

Ответ:

```
$ cat case.sh
```

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

```
[ $# -ne 1 ] && exit 1;
```

```
FIRST=`echo $1 | cut -c1`
```

```
function start() {
```

```
    echo "Стартую $1"
```

```
    ./$1 start
```

```
}
```

```
function stop() {
```

```
    echo "Останавливаю $1"
```

```
    ./$1 stop
```

```
}
```

```
function status() {
```

```
    echo "Статус $1"
```

```
    ./$1 status
```

```
}
```

```
case $FIRST in
```

```
    [Ss] )
```

```
        start $1
```

```
;;
```

```
    K|k )
```

```
        stop $1
```

```
;;
```

```
    * )
```

```
        status $1
```

```
;;
```

```
esac
```

```
$ bash case.sh scr1.sh
```

```
Стартую scr1.sh
```

```
testrc
```

