Учебный курс

Разработка скриптов Bash для администраторов РЕД ОС

Автор: Груздев П.Ю.

г.Екатеринбург
2026

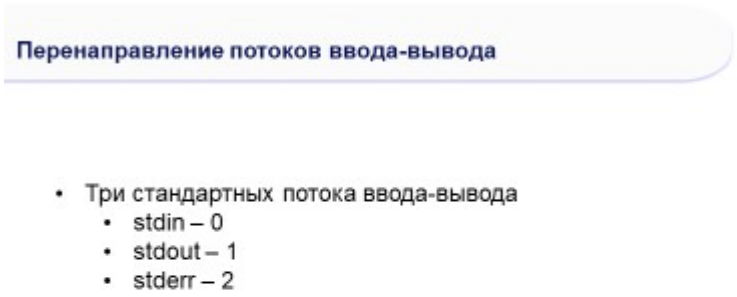
Оглавление

Глава 1. Текстовые файлы и потоки....4
1. Перенаправление потоков ввода-вывода....4
2. Конвейеры и фильтры....12
3. Команда echo....14
4. Получение даты – date....16
5. Задание....17
6. Просмотр файлов с помощью more и less....18
7. Объединение файлов с помощью cat....20
8. Команды head и tail....22
9. Задание....24
10. Вырезание текста с помощью cut....25
11. Потоковый редактор sed....27
12. Потоковый редактор awk....33
13. Задание....40
14. Сравнение файлов и каталогов....41
15. Сортировка строк....44
16. Вывод неповторяющихся строк....46
17. Объединение строк двух файлов по общему полю....47
18. Задание....48
19. Подсчет количества и нумерация строк....49
20. Замена символов в строках с помощью команды tr....51
21. Задание....54
22. Команда xargs....55
Глава 2. Регулярные выражения....56
1. Классификация регулярных выражений....56
2. Поиск текста с помощью grep....59
3. Использование обратных ссылок....63
4. Задание....65
5. Использование регулярных выражений с sed....66
6. Регулярные выражения в awk....68
7. Задание....70
Глава 3. Написание сценариев Bash....71
1. Сценарии оболочки....71
2. Задание....75
3. Использование переменных оболочки....76
4. Массивы....80
5. Интерактивная установка значений переменных....82
6. Позиционные параметры....83
7. Задание....87
8. Оператор test....88
9. Условное исполнение команд....91
10. Оператор case....95
11. Задание....98
12. Циклы....99
13. Задание....103
14. Функции....104
15. Задание....108
Ответы к заданиями....109

16. Глава 1 задание п.5.....109
17. Глава 1 Задание п.9.....110
18. Глава 1 Задание п.13.....111
19. Глава 1 Задание п.18.....112
20. Глава 1 Задание п.21.....116
21. Глава 2 Задание п.4.....117
22. Глава 2 Задание п.7.....119
23. Глава 3 Задание п.2.....121
24. Глава 3 Задание п.7.....122
25. Глава 3 Задание п.11.....128
26. Глава 3 Задание п.13.....131
27. Глава 3 Задание п.15.....134

Глава 1. Текстовые файлы и потоки.

1. Перенаправление потоков ввода-вывода.



Если процесс требует чтения или записи в файл, то этот файл должен быть сначала открыт, т.е. создан поток (stream) данных.

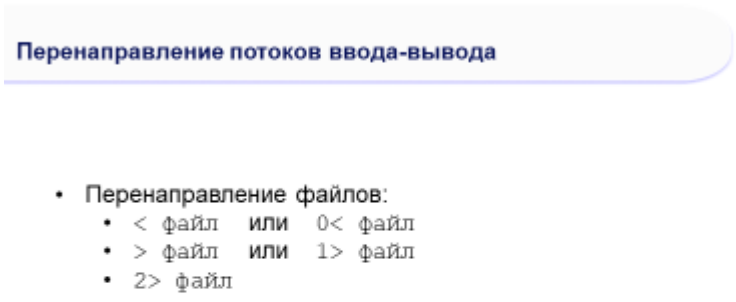
Процедура открытия файла создает структуру в ядре, называемую файловым дескриптором.

Все файловые дескрипторы пронумерованы.

С любым пользовательским процессом при его создании связываются три файловых дескриптора (потока):

1. стандартный поток ввода (`stdin`), файловый дескриптор которого – `0`;
2. стандартный поток вывода (`stdout`), файловый дескриптор – `1`;
3. стандартный поток вывода ошибок (`stderr`), файловый дескриптор – `2`.

Примечание: Поток ввода открыт на чтение, а потоки вывода и ошибок – на запись. Обычно по умолчанию стандартный поток ввода связан с клавиатурой, а стандартные потоки вывода и ошибок связаны с дисплеем.



Оболочка Bash позволяет перенаправить стандартные потоки в файлы, для чего используются следующие операторы:

1. `< имя_файла` или `0< имя_файла` – перенаправление стандартного потока ввода;
2. `> имя_файла` или `1> имя_файла` – перенаправление стандартного потока вывода;
3. `2> имя_файла` – перенаправление стандартного потока вывода ошибок

Если файла, в который производится запись не существует, то такой файл создается

Пример: с помощью такой команды можно отправить электронное письмо с текстом, содержащемся в файле `letter`:

```
$ mail -s 'Pismo' user1 < letter
```

Примечание: Эта команда направит электронное письмо пользователю `user1`. *Тема письма (Subject), указана после опции* `-s`, *а текст письма передан через стандартный поток ввода из файла* `letter`.

Пример: Следующая команда использует перенаправление стандартного потока вывода в файл `ls.txt`:

```
$ ls > ls.txt
```

Примечание: В файле `ls.txt` *в результате работы такой команды окажется список файлов в текущем каталоге, выведенный командой* `ls` *в стандартный поток вывода.*

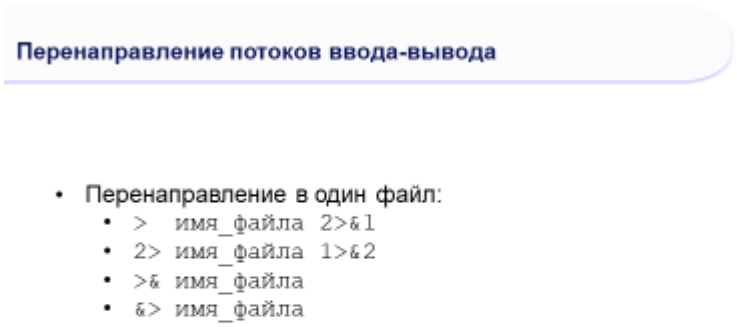
Пример: Аналогично в файл можно перенаправить ошибки:

```
$ ls -ld /etc /ctc 2> ls.err
drwxr-xr-x   87 root     root         6064 Окт 15 18:57 /etc
$ cat ls.err
ls: /ctc: No such file or directory
```

Примечание: В этом примере поток вывода ошибок был перенаправлен в файл `ls.err`*. В силу того, что в качестве аргумента команды* `ls -ld` *был задан несуществующий каталог* `/ctc`*, то команда вывела соответствующее сообщение об ошибке, которое и было перенаправлено в файл* `ls.err`*. Содержимое файла было выведено с помощью команды* `cat`*.*

Пример: Поток вывода перенаправлен в файл `ls.txt` одновременно с перенаправлением потока ошибок в файл `ls.err`.

```
$ ls -ld /etc /ctc > ls.txt 2> ls.err
```



Чтобы перенаправить оба потока вывода в один и тот же файл следует использовать оператор сцепления потоков `&`

Оболочка bash позволяет следующие эквивалентные формы записи перенаправления потоков вывода в один и тот же файл:

- `> имя_файла 2>&1`
- `2> имя_файла 1>&2`
- `>& имя_файла`
- `&> имя_файла`

Примеры:

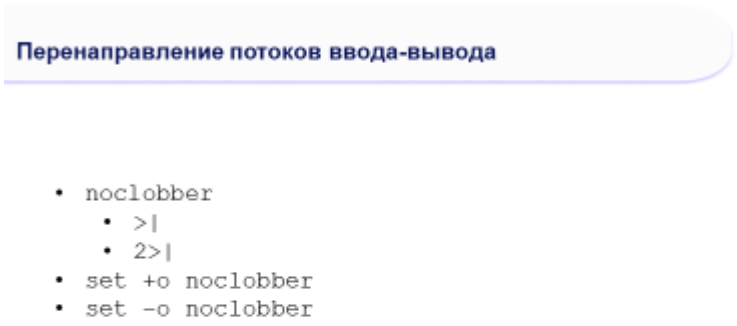
```
$ ls -ld /etc /ctc > ls.txt 2>&1
$ ls -ld /etc /ctc 2> ls.txt >&2
$ ls -ld /etc /ctc &> ls.txt
```

Операции перенаправления потоков вывода и вывода ошибок в файл стирают его содержимое, записывая новое содержимое взамен старого.

Операцию перенаправления можно использовать для стирания содержимого файлов и создания новых пустых файлов.

Пример: для стирания содержимого файла `ls.txt` можно использовать команду

```
$  > ls.txt
```



Оболочка Bash позволяет исключить стирание содержимого файлов при перенаправлении в них потоков вывода или ошибок с помощью установки флага `noclobber` командой `set -o`.

Пример:

```
$ set -o noclobber
```

Значение всех флагов оболочки можно с помощью команды `set -o`.

Пример:

```
$ set -o
allexport       off
braceexpand     on
emacs           on
errexit         off
hashall         on
histexpand      on
history         on
ignoreeof       off
interactive-comments    on
keyword         off
monitor         on
noclobber       on
noexec          off
noglob          off
nolog           off
notify          on
```

```
nounset         off
onecmd          off
physical        off
posix           off
privileged      off
verbose         off
vi              off
xtrace          off
```

Примечание: Листинг, выведенный командой `set -o`*, демонстрирует, что после выполнения пользователем команды* `set -o noclobber`*, данная опция была активизирована (состояние* `on`*). Установка этой опции оболочки предотвращает перезапись существующих файлов с помощью операций вывода.*

Пример: Попробуем очистить существующий файл `dir.txt`.

```
$ > dir.txt
bash: dir.txt: cannot overwrite existing file
```

Примечание: Так как установлена опция `noclobber`*, то оболочка не позволила переписать (в данном случае стереть) существующий файл.*

Для перезаписи содержимого файла при установленной опции `noclobber` можно воспользоваться операторами:

- `>|` - для перенаправления потока вывода с гарантированной перезаписью файла;
- `2>|` - для перенаправления потока вывода ошибок с гарантированной перезаписью файла.

Пример:

```
$ ls -l >| dir.txt
```

Примечание: Эта команда запишет подробную информацию о содержимом текущего каталога в существующий файл `dir.txt`*, не обращая внимания на то, что файл уже существует и установлена опция оболочки* `noclobber`*.*

Отключить опцию `noclobber` (как и другие опции по аналогии) можно с помощью команды: `set +o noclobber`

Если необходимо добавить в существующий файл информацию из потоков вывода, то можно использовать следующие операторы:

- `>>` - для перенаправления потока вывода на добавление к файлу;
- `2>>` - для перенаправления потока вывода ошибок на добавление к файлу

Если файла не существует, то создается новый файл.

Пример: следующая команда добавит в файл `dir.txt` имя текущего каталога:

```
$ pwd >> dir.txt
```

Некоторые команды, работающие с текстовыми файлами, позволяют вместо имени файла указать стандартный поток ввода. Для чего используется символ `-`

Пример: команда

```
vi -
```

позволяет считать редактируемый текст из стандартного потока ввода вместо открытия файла.

Для завершения потока ввода, производимого с клавиатуры, следует набрать сочетание клавиш `Ctrl-D`, передающее в поток ввода символ конца файла.

Пример:

```
$ view -
Privet
<Ctrl-D>
```

Примечание: В этом примере команда `view` *позволила считать содержимое потока ввода с клавиатуры, окончание которого было обозначено с помощью сочетания* `Ctrl-D`.

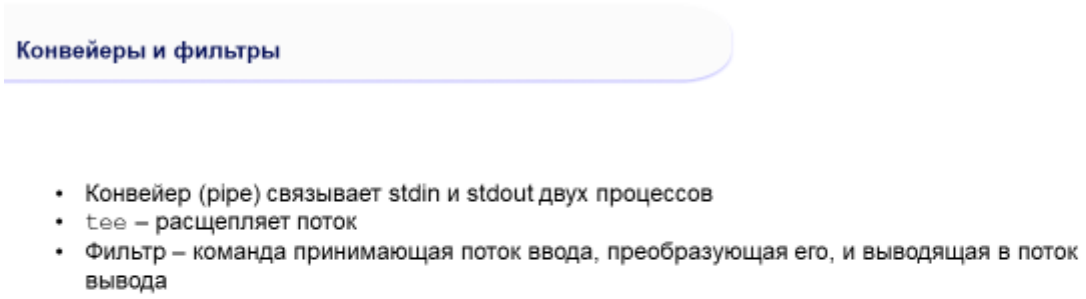
Конструкция *here document* (документ здесь) позволяет вместо символа конца файла (EOF), который не может быть использован внутри файла, воспользоваться любым другим удобным символом.

Пример: следующая конструкция обеспечивает посылку письма пользователю `user1`, где тело письма передается через поток ввода команды `mail` с помощью here document:

```
$ mail -s HereDoc user1 << .
This is Here Document here!
.
```

Примечание: Обратите внимание, что в предыдущем примере вместо использования символа окончания потока здесь используется символ точки. Этот символ был задан в качестве ограничителя потока ввода в командной строке << . . Это обозначает, что текст, вводимый через стандартный поток ввода, должен быть ограничен символом точка. Символ – ограничитель должен находиться в строке, перед которой чтение стандартного потока ввода прекращается. Символ – ограничитель должен быть единственным символом в строке (не считая символа перевода строки).

2. Конвейеры и фильтры.



Конвейер (pipe) позволяет направить стандартный поток вывода (stdout) одного процесса в стандартный поток ввода (stdin) другого процесса.

Для организации конвейера необходимо поставить символ конвейера – вертикальную черту | между командами, потоки которых необходимо объединить.

Пример:

```
$ last | view -
```

Примечание: Выводимый командой `last` *список входивших в сеанс пользователей передан через конвейер команде просмотра* `view` *(один из вариантов вызова* `vi`*). Знак тире после команды* `view` *сообщает, что данные должны быть введены из стандартного потока ввода, в который передает данные команда* `last` *через свой поток вывода.*

Команда `tee` позволяет организовать вывод информации, полученной из потока ввода, в файлы, указанные как аргументы, и в стандартный поток вывода одновременно.

Пример: команда `ps` в примере ниже выведет список процессов в файл `ps.txt` и в стандартный поток вывода.

```
$ ps | tee ps.txt
  PID TTY          TIME CMD
 1720 pts/0    00:00:00 bash
 2438 pts/0    00:00:00 ps
 2439 pts/0    00:00:00 tee
$ cat ps.txt
  PID TTY          TIME CMD
```

```
 1720 pts/0    00:00:00 bash
 2438 pts/0    00:00:00 ps
 2439 pts/0    00:00:00 tee
```

Примечание: Содержимое файла `ps.txt` *совершенно идентично выводу команды* `ps`*. Обратите внимание на присутствие в списке процессов команды* `tee`*.*

Примечание: Команда `tee` *наиболее часто используется для отладки работы сложных конвейеров, состоящих из множества команд. Эту команду удобно устанавливать в месте конвейера, которое вызывает подозрения. Так как в файле, указанном в качестве аргумента* `tee`*, будет находиться та же информация, что была передана в данном месте конвейера, то по ней легко можно будет определить наличие и суть ошибок.*

Фильтр – это не интерактивная команда, способная принимать поток данных через стандартный поток ввода, обрабатывать его, и выводить обработанные данные в стандартный поток вывода.

Пример: команда `wc -l` – фильтр, подсчитывающий количество строк в потоке ввода и передающий результат подсчета в поток стандартного вывода. Конвейер, показанный ниже, подсчитывает число процессов, работающих в системе от имени пользователя `user1`.

```
$ ps -u user1 | wc -l
      8
```

Команды, находящиеся в конвейере должны удовлетворять следующим требованиям:

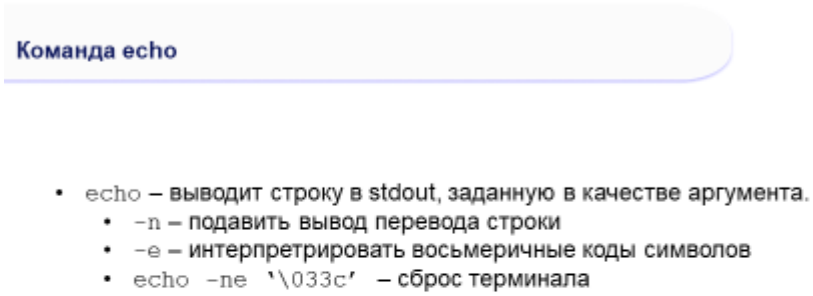
1. Первая команда конвейера должна уметь выводить информацию в стандартный поток вывода.
2. Команды, находящиеся внутри конвейера, должны быть фильтрами.
3. Последняя команда в конвейере должна уметь читать стандартный поток ввода.

Пример: для отправки суперпользователю письма с отсортированным списком пользователей, вошедших в сеанс, можно использовать такой конвейер:

```
$ who | sort | mail -s 'Logged users' root
```

Примечание: В этом примере команды `who` *и* `mail` *не являются фильтрами, но поскольку команда* `who` *осуществляет вывод в стандартный поток вывода, а* `mail` *читает поток ввода, то их можно использовать, соответственно, в начале и конце конвейера. Команда* `sort`*, осуществляющая сортировку, является фильтром, то есть она читает из стандартного потока ввода и выводит отсортированный текст в стандартный поток вывода, поэтому она вполне может находиться где-либо в середине конвейера.*

3. Команда echo.



Команда `echo` выводит на экран текстовую строку, заданную в качестве ее аргумента или значение переменной, перед которой должен стоять символ `$`

Пример:

```
$ echo "It's cool"
It's cool
$echo $PS1
/u:/w$
```

Примечание: В первом примере команда `echo` *просто выводит строку, заданную ей в качестве аргумента. Во втором примере выводится значение переменной окружения* `PS1`*, для чего перед именем переменной поставлен символ извлечения значения переменной* `$`*.*

Команда `echo` автоматически вставляет после выводимой строки символ перевода строки. Для отмены этого можно использовать опцию `-n`.

Опция `-e`, позволяет команде `echo` интерпретировать восьмеричные числа как символы ASCII.

Пример:

```
$ echo -e '\101\t\102'
A       B
```

Примечание: В этой команде выводятся буквы `A` *и* `B`*, заданные их восьмеричными кодами ASCII, которые разделены символом табуляции.*

Пример: Та же команда, но без опции `-e` выведет следующее:

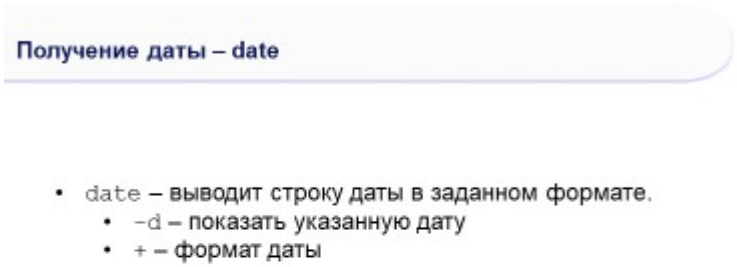
```
$ echo '\101\t\102'
\101\t\102
```

Часто при ошибочном выводе двоичного файла на терминал у последнего сбиваются настройки. Один из способов сброса настроек состоит в посылке на терминал ESC последовательности

Пример:

```
$ echo -ne '\033c'
```

4. Получение даты – date



Команда `date` позволяет выводить дату/время (текущую или указанную) в определенном формате, а также задавать системные дату/время

Примеры:

```
$ date
Пн 06 мая 2024 09:04:28 +05

$ date -d yesterday
Вс 05 мая 2024 09:04:50 +05

$ date +"%Y-%m-%d"
2024-05-06
```

Примечание: Для задания системных даты/времени используется опция `-s`

5. Задание.

1. Получите список всех процессов в системе и перенаправьте вывод в файл `ps.txt`.

2. Выполните команду поиска всех обычных файлов в каталоге `/usr` так, чтобы найденные имена файлов были записаны в файл `SysComm` в домашнем каталоге, а поток ошибок был записан в нуль-устройство `/dev/null`.

3. Выполните ту же команду, но так, чтобы потоки вывода и ошибок были записаны в файл `SysComm`.

4. Допишите в конец файла `SysComm` информацию о текущей дате и времени, выводимую командой `date`.

5. Выведите текущую дату в формате: YYYY-MM-DD hh:mm:ss

6. Установите опцию `noclobber` и сотрите содержимое файла `ps.txt` с помощью перенаправления вывода. Снимите опцию `noclobber` .

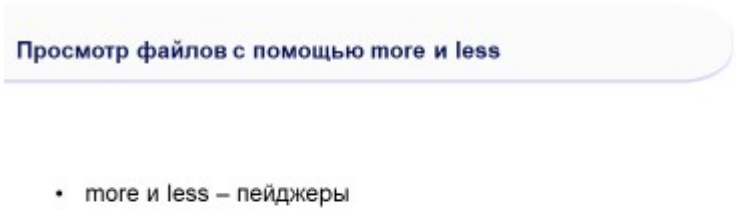
7. Проведите поиск файлов символьных ссылок в каталоге `/usr/share/doc` так, чтобы их список был выведен в отсортированном виде с помощью фильтра `sort`.

8. Выведите сортированный список пользователей, вошедших в сеанс, в системе в файл `seans.txt` и на экран одновременно.

9. Определите из `man ascii` восьмеричный код системного звукового сигнала и выведите его с помощью `echo`.

10. Опция `-n` команды `echo` подавляет добавление перевода строки. С помощью `man` определите, как это можно сделать иначе.

6. Просмотр файлов с помощью more и less.



Команды `more` и `less` позволяют постранично просматривать текстовые файлы.

Пример: приведенная ниже команда позволяет просмотреть файл `/etc/passwd`.

```
$ less /etc/passwd
```

Обе эти команды позволяют читать текст из стандартного потока ввода, причем при чтении из него аргументы – имена файлов задавать не следует.

Обобщающее название этих команд – пейджеры (pager).

Примечание: Первый из них считается стандартным и присутствует во множестве разновидностей UNIX систем. Второй из них - `less` *, разработан позже, и не смотря на свое название (*`more` *– больше,* `less` *– меньше), обладает гораздо большими возможностями, чем* `more`*.*

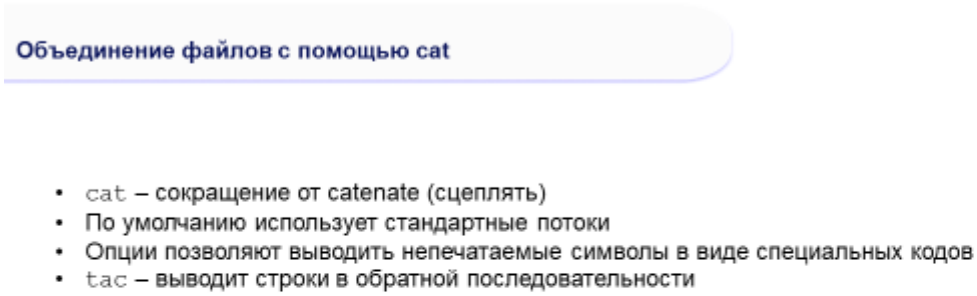
Пейджер `less` используется в GNU/Linux по умолчанию.

Примечание: Именно он работает, когда пользователь получает помощь в системе `man`*, так как страница помощи, найденная* `man`*, передается для отображения пэйджеру* `less`*. Одно из наиболее удобных свойств* `less` *заключается в том, что в отличие от* `more` *, для промотки экрана в нем можно пользоваться обычными клавишами управления курсором вверх и вниз, а также клавишами* `PgUp` *и* `PgDn`*. Многие из команд управления промотки* `more` *и* `vi` *поддерживаются в* `less`*.*

Наиболее часто используемых команд `more` и `less`:

Действие	more	less
Выход из пэйджера.	q	q
Получение встроенной помощи по командам.	h	h
Прокрутка страницы вперед.	Space	PgDn
Прокрутка страницы назад.	Ctrl-B	PgUp
Прокрутка на строку вперед.	Return	Ctrl-N
Прокрутка на строку назад.		Ctrl-P
Переход в конец файла.		G
Переход в начало файла.		g
Поиск по шаблону (регулярному выражению).	/шаблон	/шаблон

7. Объединение файлов с помощью cat.



Команда `cat` выводит в стандартный поток вывода содержимое файла, заданное в качестве аргумента.

Если задано несколько файлов, то их содержимое выводится последовательно.

Примечание: Команда последовательно объединяет в выводимом потоке содержимое этих файлов (название `cat` *– от слова* `catenate` *– объединять).*

Пример: следующая команда выведет в стандартный поток вывода содержимое файлов `/etc/issue` и `/etc/issue.net`, содержащих приветствие, выводимое на экран перед входом в сеанс, соответственно, на локальном и удаленном компьютерах.

```
$ cat /etc/issue{,.net}
```

Примечание: в этом примере была использована способность Bash, называемая перебором. Фактически, введенная команда эквивалентна следующей:

```
$ cat /etc/issue /etc/issue.net
```

Если аргумент – имя файла не задан, то используется чтение из стандартного потока ввода.

Пример: следующая команда прочитает текст с клавиатуры и запишет его в файл `f1`.

```
$ cat > f1
Привет!
<Ctrl-D>
```

```
$ cat f1
Привет!
```

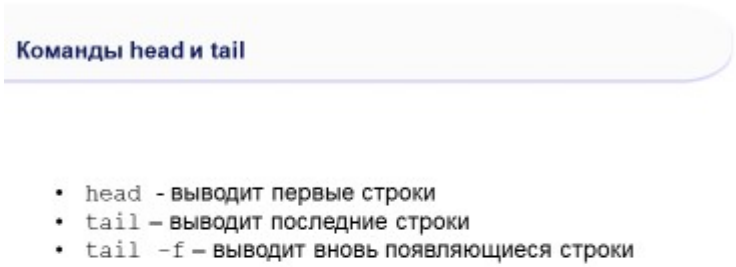
Примечание: В первой команде этого примера в файл `f1` *с помощью перенаправления стандартного потока вывода записан текст, который был принят командой* `cat` *из стандартного потока ввода. Ввод с клавиатуры был завершен с помощью нажатия сочетания клавиш* `Ctrl-D`*. Далее, с помощью команды* `cat` *было выведено содержимое файла* `f1`*.*

Команда `cat` может объединять и двоичные файлы, если они указаны как аргументы.

Не следует только выводить содержимое двоичных файлов на терминал, так как это может нарушить его настройки и потребует его инициализации.

У команды `cat` имеется “зеркальный” двойник – команда `tac`, позволяющая выводить строки файла, заданного в качестве аргумента, в обратном порядке.

8. Команды head и tail.



Команда `head` предоставляет возможность вывести заданное количество первых строк файла или потока, если файл не указан в качестве аргумента.

По умолчанию выводится десять первых строк.

Требуемое количество строк может быть указано в качестве опции

Пример:

```
$ head -3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

Примечание: В этом примере выведены первые три строки файла `/etc/passwd`.

Для вывода последних строк из файла используется команда `tail`.

Требуемое количество срок может быть задано через опцию, а по умолчанию выводится десять строк.

Пример: для вывода десяти последних в списке процессов, выводимых командой `ps -e`, можно использовать `tail` :

```
$ ps -e | tail
 3308 ?        00:00:00 mapping-daemon
 3310 ?        00:00:00 clock-applet
 3312 ?        00:00:00 notification-ar
 3314 ?        00:00:00 mixer_applet2
```

```
 3321 ?        00:00:03 wnck-applet
 3325 ?        00:01:59 soffice.bin
 3344 ?        00:00:00 getstyle-gnome
 3362 ?        00:00:47 xpdf
 3653 pts/1    00:00:00 ps
 3654 pts/1    00:00:00 tail
```

Примечание: Команда `tail` *предоставляет удобную опцию* `-f`*, которая позволяет выводить последние строки заданного файла динамически отображая новые строки, появляющиеся в конце файла. Это бывает полезно, например, при мониторинге файлов журналов.*

Опция `-n` команды `tail` позволяет указать не только сколько последних строк показывать, но и с какой строки начать выводить файл

Пример: выведем из потока две последние строки и поток начиная со второй строки

```
$ echo -e 'one\ntwo\nfree\nfour' | tail -n 2
free
four
$ echo -e 'one\ntwo\nfree\nfour' | tail -2
free
four
$ echo -e 'one\ntwo\nfree\nfour' | tail -n +2
two
free
four
```

Примечание: Первые две команды эквивалентны.

Опция `-f` позволяет не выходить из `tail`, а выводить вновь появившиеся строки из файла. Это очень удобно, когда вы смотрите журналы при отладке.

9. Задание.

1. В чем разница между командами `more /etc/profile` и `more < /etc/profile` ? Заметно ли отличия в командах при их выполнении?

2. Команда `man` обладает опцией для явного указания пэйджера, в котором будет отображаться найденная страница помощи. Опробуйте ее действие.

3. Как можно запустить `man` так же, как и в предыдущем задании, но используя переменную окружения `PAGER` ?

4. Команда `less` позволяет открыть несколько файлов, указав их в качестве аргументов.

5. Команда `cat` обладает опцией, устанавливающей нумерацию выводимых строк. Определите, какая она.

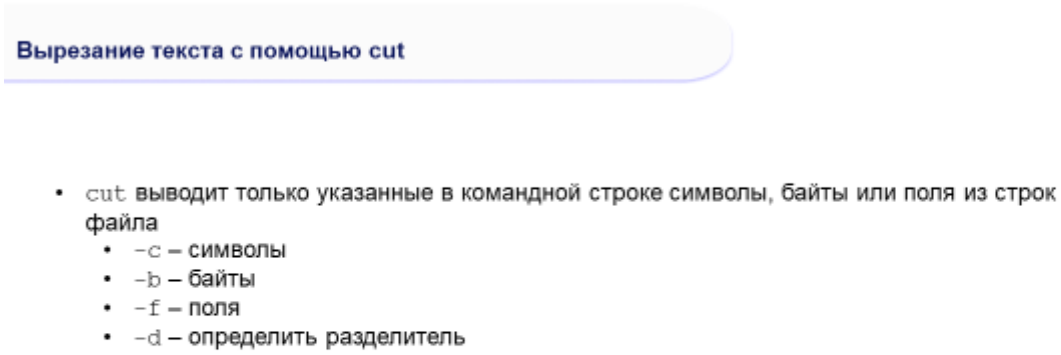
6. Найдите опцию `cat` , позволяющую этой команде удалять последовательно повторяющиеся переводы строки, оставляя лишь один перевод строки.

7. Выведите содержимое файла `/etc/passwd` в обратном порядке следования строк.

8. Для чего предназначена опция `-b` команды `tac` ?

9. Получите имена трех наиболее объемных файлов в каталоге `/usr/bin`.

10. Вырезание текста с помощью cut.



Команда `cut` выводит только указанные в командной строке символы, байты или поля из строк файла, направляя выводимые данные в стандартный поток вывода.

Если файл для чтения не задан, то осуществляется ввод из `stdin`.

Для указания требуемых символов строки для вывода необходимо использовать опцию `-c`, после которой необходимо указать номера символов через запятую или тире:

Пример:

```
$ ls -ld
drwx------   51 user1   user1       3752 Окт 22 21:04 .
$ ls -ld | cut -c1-10,35-43
drwx------    3752
```

Примечание: В этом примере продемонстрировано, как из информации, выводимой командой `ls -ld` *были извлечены требуемые диапазоны символов так, что в результате осталась информация только о правах доступа к каталогу и о его размере.*

Можно указать номера требуемых байтов в строке для вывода, используя опцию `-b`.

Если строка разделена табуляцией на поля, то с помощью команды `cut -f` можно вывести требуемые поля строк.

Пример: если требуется вывести из файла `/etc/hosts`, содержащего соответствия IP адресов именам хостов, только заданные в нем IP адреса без имен хостов, можно выполнить следующую команду:

```
$ cut -f1 /etc/hosts
```

```
127.0.0.1
#212.193.90.39
212.193.90.25
```

Иной разделитель полей можно указать после опции `-d`.

Пример: Получим список пользователей системы и их UID. Это можно сделать путем вывода первого и третьего полей файла `/etc/passwd`. Разделитель полей в этом файле – двоеточие.

```
$ cut -f1,3 -d: /etc/passwd
```

11. Потоковый редактор sed.

Потоковые редакторы представляют собой фильтры, не интерактивно (“на лету”) редактирующие проходящий через них поток текста.

Все изменения, вносимые в текст потоковыми редакторами, производятся в потоке и не затрагивают содержимое файла, откуда этот поток текста считан.

Некоторые потоковые редакторы обладают встроенными скриптовыми языками программирования, позволяющими писать сложные сценарии обработки потока текста.

Потоковый редактор `sed` позволяет либо прочитать входной поток для обработки из стандартного потока ввода, либо осуществить чтение и обработку текстового файла, заданного в качестве аргумента.

Обработанный текст передается в стандартный поток вывода.

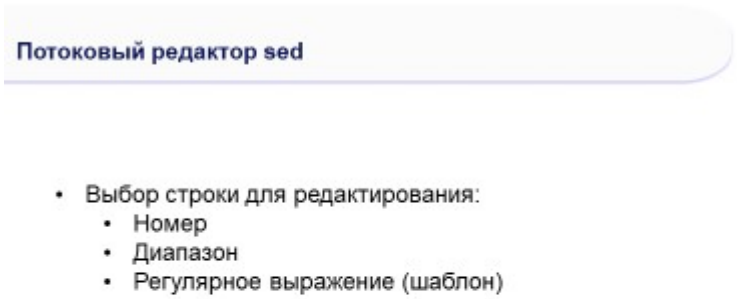
Если потоковому редактору не указан никакой сценарий обработки текста, то поток выводится в `stdout` без изменений.

Редактор `sed` позволяет удобно работать с целыми строками текста.

Пример: для получения списка процессов без заголовка можно с помощью `sed` удалить первую строку из потока. Для этого необходимо использовать команду `d` языка программирования `sed`.

```
$ ps | sed 1d
 1757 pts/0    00:00:00 bash
 1870 pts/0    00:00:00 ps
 1871 pts/0    00:00:00 sed
```

Примечание: Команда `1d` *удалила в потоке первую строку.*



Перед командами `sed` нужно указывать строки, с которыми производятся какие-либо действия.

Адреса можно указывать следующим образом:

1. Номером строки
2. Диапазоном строк с указанием первого и последнего номеров строк в диапазоне через запятую

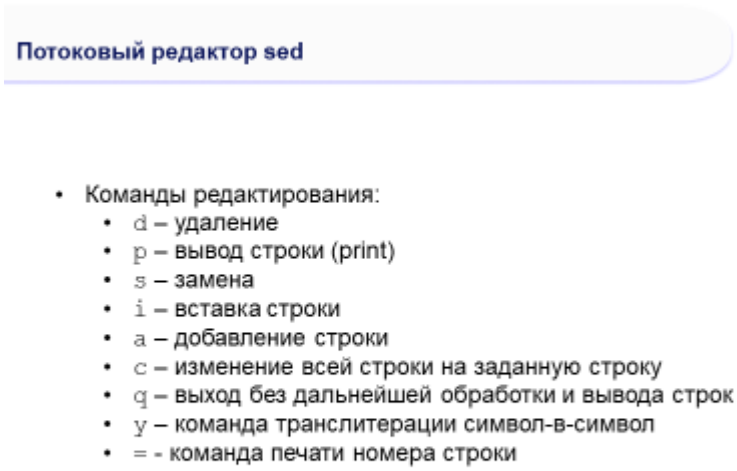
Пример: команда `sed '2,4d'` удалит в потоке строки со второй по четвертую включительно.

3. Можно указать диапазон до последней строки. Последняя строка обозначается символом доллара `$`
4. С помощью регулярного выражения или, в простейшем случае, подстроки, которая должна находиться в адресуемых строках.

Пример: Ниже приведена команда, которая выведет список процессов, не связанных с какими-либо терминалами (их имена содержат подстроку `tty`), но не псевдотерминалами (например, `pts/1`). Это достигается с помощью удаления из потока всех строк, в которых имеется подстрока `tty`. Искомая подстрока (регулярное выражение), используемая для адресации удаляемых в потоке строк, должна быть задана в косых чертах.

```
$ ps -A | sed '/tty/d'
```

Примечание: В результате выполения этой команды будет отображен список процессов, либо не связанных с терминалами вообще, либо связанных с псевдотерминалами.



Наиболее часто используются следующие команды `sed`:

1. `d` – удаление
2. `p` – вывод строки (print)
3. `s` – замена искомой подстроки (регулярного выражения) на заданную подстроку
4. `i` – вставка строки
5. `a` – добавление строки
6. `c` – изменение всей строки на заданную строку
7. `q` – выход без дальнейшей обработки и вывода строк
8. `y` – команда транслитерации символ-в-символ
9. `=` - команда печати номера строки

Команда `p` выводит указанные в диапазоне строки в дополнении ко всем строкам текстового потока

Для подавления вывода всех строк текстового потока можно использовать опцию `-n` утилиты `sed`

Пример: чтобы увидеть все строки учетных записей (из файла `/etc/passwd`), начиная с первой и заканчивая строкой, в которой встречается строка `daemon` выполним команду:

```
$ sed /daemon/q /etc/passwd
root:x:0:0:System Administrator:/root:/bin/bash
bin:x:1:1:bin:/:/dev/null
daemon:x:2:2:daemon:/:/dev/null
```

Примечание: Из примера видно, что для достижения поставленной цели была использована команда выхода q. *Как только* sed *встретил строку* daemon, *работа* sed *была завершена. Обратите внимание на то, что в этом примере файл для чтения задан в качестве аргумента.*

Пример: демонстрация того, как sed применяется для замены подстрок. Для замены в потоке, полученном при чтении файла /etc/passwd, всех вхождений подстроки bash на zsh выполняем команду

```
$ sed s/bash/zsh/ /etc/passwd
```

Примечание: Здесь после команды s *в косых чертах указана подстрока для поиска –* bash. *Далее указана подстрока замены –* zsh.

Пример: Ниже приведен более сложный пример замены подстрок в потоке. Требуется вывести содержание файла /etc/group (файл, содержащий информацию о группах пользователей и членстве в них) с учетом следующих условий:

- должны быть выведены только группы, членом которых является пользователь user1;
- все разделители полей – знаки двоеточия должны быть заменены на символ подчеркивания.

```
$ sed -n /user1/s/:/_/gp /etc/group
adm_x_4_root,adm,daemon,user1
wheel_x_10_root,user1
ftpadmin_x_51_user1
ldap_x_55_user1
users_x_100_user1
webmaster_x_103_user1
user1_x_500_
```

Примечание: В этом примере перед командой s *используется адресация с помощью подстроки* user1. *То есть, во входном потоке выбираются только строки, содержащие подстроку* user1. *Команда* s *заменяет все вхождения символа двоеточия в каждой строке на подчеркивание. После косой черты установлен модификатор* g. *Именно он заставляет команду* s *заменить все вхождения искомой подстроки (двоеточия) на подстроку – заменитель (подчеркивание). Если бы он отсутствовал, то в каждой адресуемой строке был бы заменен только первый символ двоеточия. После модификатора* g *следует команда* p. *Эта команда печатает адресуемые строки, то есть те, в которых здесь произведена замена. Использование команды* p *вынуждает добавить опцию* -n *команды* sed, *Без этой опции* sed *без изменений выводит в stdout строки, которые не обрабатывались. Таким образом, если бы не было опции* -n, *отменяющей вывод необработанных строк, были бы выведены все строки потока, а те, которые были обработаны, были бы выведены дважды.*

Пример: демонстрация того, как можно выделить требуемые строки в потоке, вставив перед ними заданную строку (например, строку – разделитель). Для получения списка всех последних входов в систему пользователя user1 так, чтобы после каждой строки, где

упоминается этот пользователь, выводилась строка ##################### выполняем команду

```
$ last | sed '/user1/a \
#####################'
```

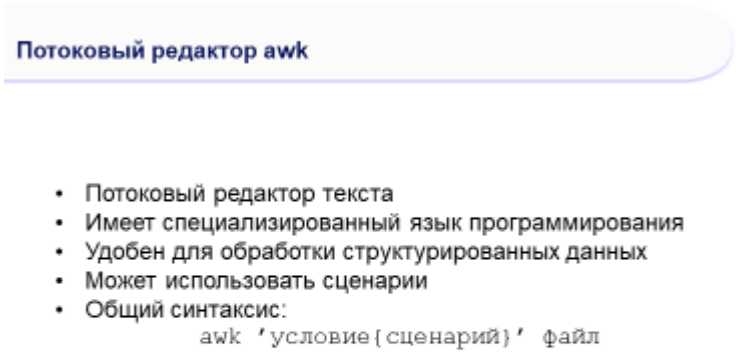
Примечание: Такой не совсем обычный формат команды связан с тем, что в командной строке оболочки введена команда, которая обычно используется в скриптах `sed`*. В скрипте в первой строке этой команды находилась бы конструкция* `/user1/a`*, которая позволила бы адресоваться ко всем строкам, в которых есть строка* `user1`*. Команда* `a` *заставляет* `sed` *добавить после каждой адресуемой строки то, что указано на следующей строке после этой команды.*

Пример: Последний пример использования `sed` показывает содержимое файла сценария `sed`, реализующий ту же задачу.

```
$ cat sed.scr
/user1/a \
#################
$ last | sed -f sed.scr
```

Примечание: Содержимое файла `sed.scr` *выведено командой* `cat`*. Последняя строка примера показывает, что для указания* `sed` *имени файла скрипта надо указывать опцию* `-f`*.*

12. Потоковый редактор awk.

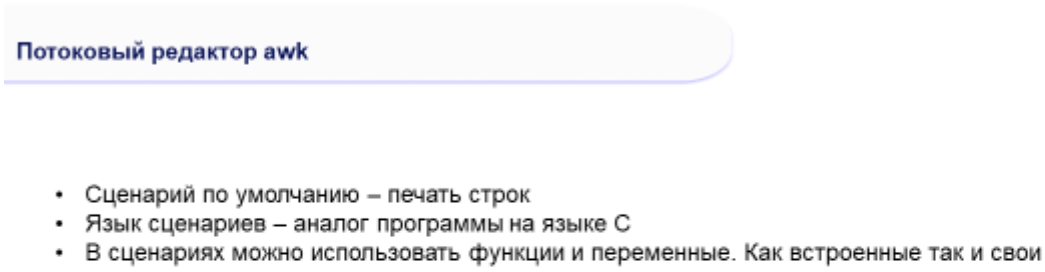


Утилита `awk`, GNU версия которой называется `gawk`, предназначена для потокового редактирования текста и обладает специализированным языком программирования, исключительно удобным для обработки структурированных данных.

Основная задача `awk` состоит в поиске и обработке строк в потоке.

Программа для `awk` в общем виде выглядит таким образом:

```
условие{ команды }
условие{ команды }
...
условие{ команды }
```



Команды могут находиться в файле сценария, имя которого должно быть указано после опции `-f`, либо могут задаваться в качестве аргумента команды `awk`.

Если условие для команды не указано, то обработке подвергаются все строки.

Если условие указано, а команда – нет, то в `stdout` выводятся только строки, удовлетворяющие условию.

Пример: следующая команда выведет из файла `/etc/passwd` все строки, содержащие подстроку `bash`.

```
$ awk '/bash/' /etc/passwd
root:x:0:0:System Administrator:/root:/bin/bash
user1:x:500:500::/home/user1:/bin/bash
user1:x:502:502::/home/user1:/bin/bash
figus:x:503:503::/home/figus:/bin/bash
```

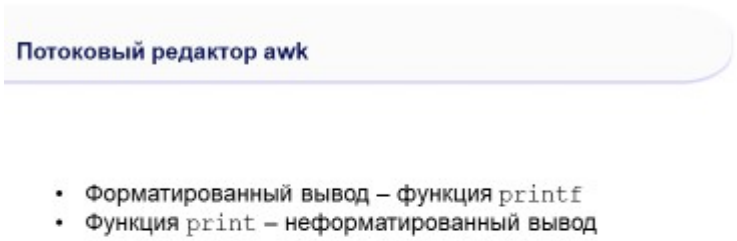
Примечание: В этом случае не была задана команда, но был задан шаблон – строка `bash`.

Утилита `awk` позволяет обращаться к полям строк, разделенных с помощью пробелов или табуляций, следующим образом:

1. `$0` – вся строка
2. `$1` – первое поле
3. `$2` – второе и так далее

Утилита `awk` мощный инструмент анализа и манипуляции данными. В рамках данного курса нет возможности изучить все возможности этой программы. Если вы хотите более подробно изучить `awk` почитайте соответствующую литературу, например: https://www.gnu.org/software/gawk/manual/gawk.pdf или `sed` и `awk`: https://github.com/manish-old/ebooks-2/blob/master/O%27Reilly%20-%20sed%20_%20awk%202nd%20Edition.pdf

Если разделитель между полями отличается от пробела или табуляции, то этот символ-разделитель можно указать после опции `-F`



Вывод полей и другой информации обеспечивает команда `print`.

Пример: выведем только имя пользователей, использующих `bash`, и их UID. Для этого необходимо вывести первое и третье поле этого файла.

```
$ awk -F: '/bash/{print $1,$3}' /etc/passwd
root 0
admin 500
user1 502
user2 503
```

В качестве условия отбора строк можно использовать операции сравнения.

Пример: Для вывода только тех пользователей, чьи UID больше 500 можно отказаться от шаблона и профильтровать строки по условию `$3>500`.

```
$ awk -F: '$3>500{print $1,$3}' /etc/passwd
user1 502
user2 503
```

Пример: Для вывода только тех пользователей, чьи UID больше 500 с использованием форматированного вывода функцией `printf`.

```
$ awk -F: '$3>500{printf "%s:%d\n",$1,$3}' /etc/passwd
user1:502
```

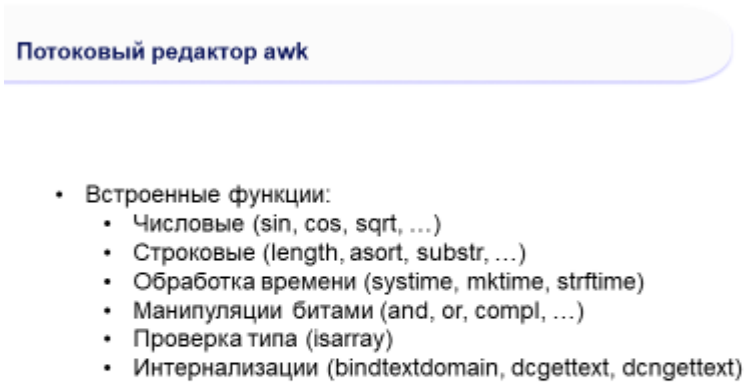
```
user2:503
```

Язык `awk` обладает собственными заранее определенными переменными и позволяет пользователю определять свои переменные:

1. `NR` – номер строки в потоке.
2. `NF` – номер поле в потоке.
3. `FS` – разделитель полей.
4. `FILENAME` – имя файла

Пример: перед именами пользователей выведем номера строк, считанных из файла `/etc/passwd`.

```
$ awk -F: '$3>500{print NR,$1,$3}' /etc/passwd
48 user1 502
49 figus 503
```



В `awk` имеется большое количество встроенных функций:

1. Числовые (`sin`, `cos`, `sqrt`, …)
2. Строковые (`length`, `asort`, `substr`, …)
3. Обработка времени (`systime`, `mktime`, `strftime`)
4. Манипуляции битами (`and`, `or`, `compl`, …)
5. Проверка типа (`isarray`)
6. Интернализации (`bindtextdomain`, `dcgettext`, `dcngettext`)

Пример: получим список только тех файлов, длина имен которых больше 15 символов.

```
$ ls | awk 'length($0)>15'
lpi05102003.tar.bz2
Plan_Sem_PM_1.rtf
Plan_Sem_PM_1.sxw
Plan_Sem_PM_2.rtf
Plan_Sem_PM_2.sxw
```

Примечание: В данном случае фильтруются только те строки, длина которых больше 15 символов.

13. Задание.

1. Получите столбец из имен пользователей, находящийся в данный момент в сеансе.

2. Получите столбец, составленный из символов строк – имен файлов в текущем каталоге таким образом, чтобы были выведены символы с пятого до последнего.

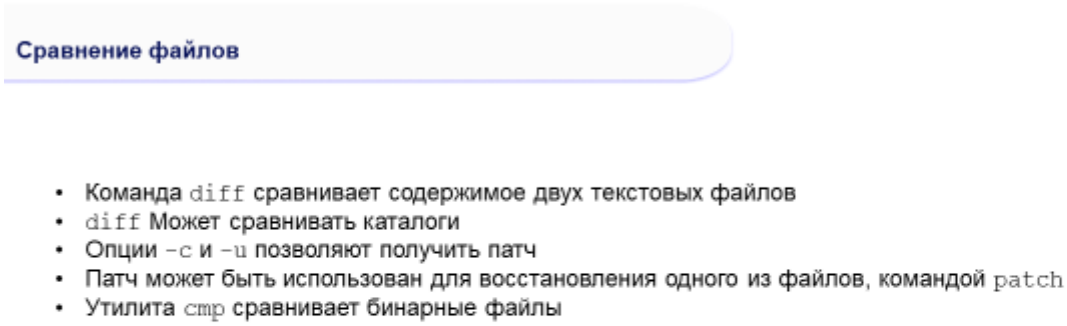
3. С помощью `sed` получите список только тех процессов, которые связаны с каким-либо терминалом (фильтр по строке `tty`).

4. В полученном списке процессов с помощью `sed` замените все строки `user` на `provider`.

5. Используя `awk` из списка процессов, полученного командой `sed`, удалите второй столбец.

6. С помощью `awk` пронумеруйте полученный в результате предыдущих действий список.

14. Сравнение файлов и каталогов.



Команда `diff` принимает в качестве аргумента два имени файлов или каталогов и проводит их сравнение.

Пример:

```
$ ps > ps1.txt
$ ps > ps2.txt
$ diff ps1.txt ps2.txt
3c3
<  3084 pts/0    00:00:00 ps
---
>  3088 pts/0    00:00:00 ps
```

Примечание: В этом примере список текущих процессов был выведен в два различных файла. Так как вызов команды `ps` *производился дважды, то PID процессов у этих команд различался. Это и подтверждает вывод команды* `diff`*. Оба получившихся файла содержат по три строки. Последние, третьи, строки отличаются (*`3c3`*).*

Опция -q отменяет вывод отличающихся строк и печатает только сообщение о том, имеются ли отличия.

Пример:

```
$ diff -q ps1.txt ps2.txt
Файлы ps1.txt и ps2.txt различаются
```

Опция `-i` применяется в случае, если необходимо игнорировать регистр в сравниваемых файлах.

Опции `-u` и `-c` выводят информацию, соответственно в обобщенном (unified) и контекстном (context) форматах:

Пример:

```
$ diff -u ps1.txt ps2.txt
--- ps1.txt     2003-11-20 23:18:16 +0500
+++ ps2.txt     2003-11-20 23:18:21 +0500
@@ -1,3 +1,3 @@
   PID TTY          TIME CMD
  2164 pts/0    00:00:00 bash
- 3084 pts/0    00:00:00 ps
+ 3088 pts/0    00:00:00 ps
$ diff -c ps1.txt ps2.txt
*** ps1.txt     2003-11-20 23:18:16 +0500
--- ps2.txt     2003-11-20 23:18:21 +0500
***************
*** 1,3 ****
    PID TTY          TIME CMD
   2164 pts/0    00:00:00 bash
!  3084 pts/0    00:00:00 ps
--- 1,3 ----
    PID TTY          TIME CMD
   2164 pts/0    00:00:00 bash
!  3088 pts/0    00:00:00 ps
```

Особенностью unified и context форматов является вывод информации об именах сравниваемых файлов.

Вывод различий в форматах unified и context может быть потом использован для реконструкции одного из сравниваемых файлов по другому с помощью известных отличий этих файлов.

Для восстановления используется утилита `patch`, позволяющая обновлять содержимое файлов с помощью так называемых “заплаток” (patches).

Примечание: Файлы – “заплатки”, создаются с помощью утилиты `diff`*, а обновление файлов с помощью этих “заплаток” производится утилитой* `patch`*.*

Для по байтного сравнения бинарных файлов удобно применять команду `cmp`.

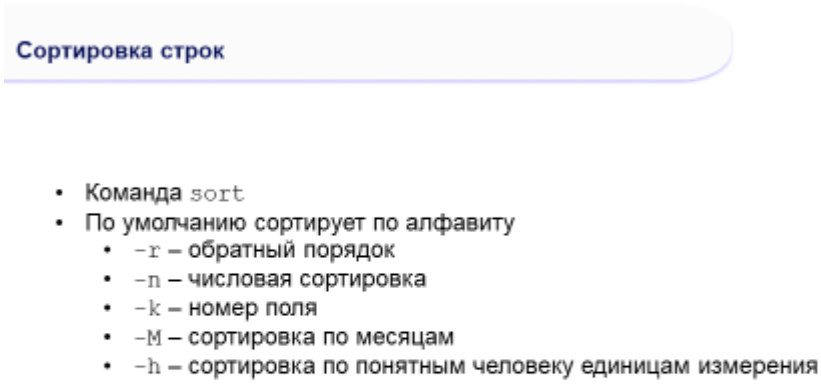
Пример:

```
$ dd if=/dev/urandom of=r1.bin count=10
10+0 входных записей
10+0 выходных записей
$ dd if=/dev/urandom of=r2.bin count=10
```

```
10+0 входных записей
10+0 выходных записей
$ cmp f1.bin f2.bin
$ cmp r1.bin r2.bin
r1.bin r2.bin различаются: байт 1, строка 1
```

Примечание: В этих примерах были созданы четыре бинарные файла: `f1.bin` *и* `f2.bin`*, заполненные нулями, и* `r1.bin` *и* `r2.bin`*, заполненные случайными числами. Размер всех файлов одинаковый. Сравнив эти файлы с помощью* `cmp`*, замечаем, что файлы* `f1.bin` *и* `f2.bin` *не отличаются. В то же время,* `r1.bin` *и* `r2.bin` *- различны.*

15. Сортировка строк.



Команда `sort` предназначена для сортировки строк файлов, указанных как аргументы команды. В случае если файлы не указаны, команда читает данные из стандартного потока ввода (stdin).

По умолчанию команда `sort` сортирует строки в алфавитном порядке по возрастанию.

При необходимости сортировки строк в порядке убывания используется опция `-r` команды `sort`.

Пример: приведенная ниже команда выводит список имен подкаталогов корневого каталога в обратном алфавитном порядке.

```
$ ls / | sort -r
var
usr
tmp
swap
sbin
root
proc
opt
mnt
lib
home
etc
dev
boot
```

bin

Команда sort способна также сортировать текстовый поток не только по целым строкам, но и по отдельным полям строк.

Разделителем полей по умолчанию считается пробел. Если используется иной разделитель полей, его следует указать после опции -t.

Для указания номера поля для сортировки используется опция -k.

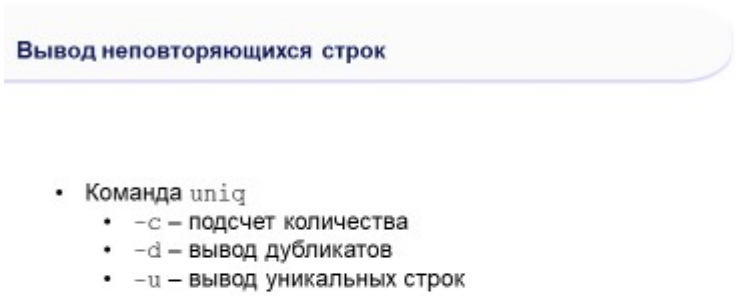
Опция -n позволяет задать команде sort не алфавитный, а числовой порядок сортировки.

Пример: Следующая команда показывает, как отсортировать первые десять строк файла /etc/passwd, содержащего учетные записи пользователей, в порядке возрастания их UID.

```
$ head /etc/passwd | sort -t: -k3 -n
root:x:0:0:System Administrator:/root:/bin/bash
bin:x:1:1:bin:/:/dev/null
daemon:x:2:2:daemon:/:/dev/null
adm:x:3:4:adm:/var/adm:/dev/null
lp:x:4:7:lp:/var/spool/lpd:/dev/null
sync:x:5:0:sync:/:/bin/sync
shutdown:x:6:0:shutdown:/:/sbin/shutdown
halt:x:7:0:halt:/:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/dev/null
news:x:9:13:news:/var/spool/news:/dev/null
```

Примечание: В этом примере опция разделителем полей является двоеточие, что и установлено опцией -t. Сортировка была выполнена по третьему полю (опция -k3), в котором содержатся UID пользователей. Сортировка была проведена в числовом порядке упорядочения, что было установлено опцией -n.

16. Вывод неповторяющихся строк.



Фильтр `uniq` позволяет удалять повторения строки из потока и чаще всего используется после фильтра `sort`, так как входной поток должен быть заранее отсортирован.

Пример: Следующая команда выведет строки файлов `f1` и `f2` так, что если какая-либо строка имеется в обоих файлах, то в поток вывода попадет только одна ее копия:

```
cat f1 f2 | sort | uniq
```

Опция `-c` команды `uniq` позволяет подсчитать количество вхождений каждой строки во входном потоке.

Опция `-d` позволяет вывести только дублирующихся строк, что позволяет, например, узнать какие строки имеются одновременно в разных файлах,

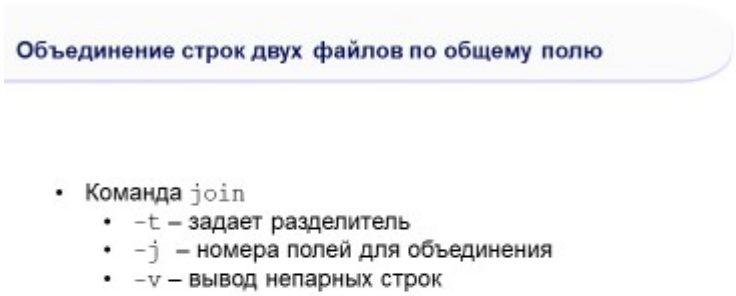
Опция `-u` выводит только уникальные (недублированные) строки.

Для игнорирования регистра при сравнении строк можно установить опцию `-i`.

Если строки входного потока разделены на поля, то можно пропустить заранее заданное количество полей до определения уникальности строки. Для пропуска первых полей, разделенных пробелами, необходимо указать их количество после опции `-f`.

Пример: команда `uniq -f6` пропустит шесть первых полей входного потока при определении уникальности строк.

17. Объединение строк двух файлов по общему полю.



Команда `join файл1 файл2` построчно объединяет содержимое заранее отсортированных файлов по общему полю. Выходная информация направляется в стандартный поток вывода (stdout).

По умолчанию предполагается, что поля отделяются друг от друга пробелами или табуляцией.

Если не указано иное, то объединение производится по первым полям строк файлов.

Опция `-t` позволяет указать символ – разделитель полей, отличный от пробела или табуляции.

Если необходимо произвести объединение строк файлов не по первым полям строк, то номера этих полей необходимо указать после опций `j1` – для первого файла и `j2` – для второго файла. Содержимое файлов должно быть отсортировано по тем полям строк, по которым производится объединение сток.

Для того, чтобы вывести вместе с объединенными строками непарные строки, которые обычно не выводятся, необходимо использовать опцию `-a`. После этой опции необходимо указать номер файла, из которого будут выведены непарные строки.

Опция `-v` позволяет вместо объединенных строк вывести только непарные строки того файла, номер которого указан после этой опции.

18. Задание.

1. Введите команду, сравнивающую содержимое вашего домашнего каталога с домашним каталогом суперпользователя.

2. Определите, с помощью какой опции `diff` можно рекурсивно сравнить эти каталоги.

3. Получите файл `patch.txt` с отличиями файлов `ps1.txt` и `ps2.txt` в контекстном формате. Удалите файл `ps2.txt` и восстановите его содержимое с помощью файлов `ps1.txt` и `patch.txt`.

4. Проверьте отличаются ли последние два килобайта файлов `r1.bin` и `r2.bin`

5. Получите список групп пользователей в системе, отсортированный по GID в обратном числовом порядке.

6. С помощью утилит `find`, `head` и `sort` получите список из десяти файлов в домашнем каталоге, занимающих наибольшее дисковое пространство.

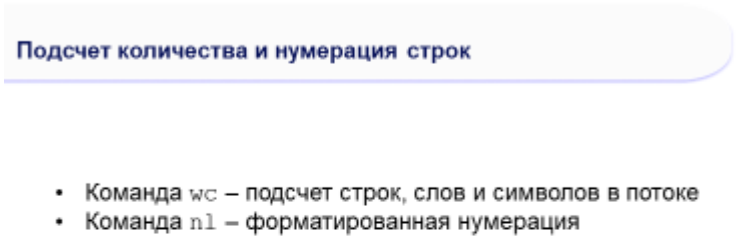
7. Выведите список тех файлов текущего каталога, для которых можно найти в этом каталоге файлы, первые три символа в именах которых совпадают не менее, чем у двух файлов.

8. Имеется два каталога без подкаталогов. Как с помощью `join` получить список файлов, которые имеются в обоих каталогах?

9. Как сделать то же самое для файлов, которые имеются только в одном из каталогов и отсутствуют в другом?

10. Как получить список файлов кроме тех, которые имеются в обоих каталогах?

19. Подсчет количества и нумерация строк.



Команда `wc` позволяет подсчитывать количество символов, слов и строк в файле,указанном в качестве аргумента. Если файл не указан, то команда `wc` читает стандартный поток ввода.

Пример:

```
$ wc /etc/hosts
      3       8      92 /etc/hosts
```

Примечание: В этом примере подсчитано количество строк, слов и символов в файле `/etc/hosts`.

При необходимости можно установить вывод только числа:

1. строк – с использованием опции `-l`
2. слов – при установленной опции `-w`
3. символов – при использовании опции `-c`

Пример: для определения количества пользователей, работающих в настоящее время в сеансе, можно использовать команду:

```
$ who | wc -l
      1
```

Команда `nl` позволяет пронумеровать строки в тексте, считанном из файла или из

потока ввода.

Команда `cat -b` также нумерует строки, однако возможности команды `nl` намного шире.

Опции `nl` позволяют устанавливать формат нумерации срок, особым образом нумеровать пустые строки, устанавливать шаг нумерации и тому подобное.

20. Замена символов в строках с помощью команды tr.

- Фильтр `tr` – изменяет, удаляет или удаляет повторы символов в потоке
- Использует концепцию набора символов [:...:]

Команда `tr` читает поток ввода и позволяет:

1. Заменять символы в потоке. Для этого необходимо задать два набора символов. Символы из первого набора будут заменены соответствующими по порядку символами из второго набора.
2. При использовании опции `-d` удалять в потоке символы, указанные в наборе.
3. Исключать повторения символов в потоке. Символы, повторы которых должны быть исключены, задаются в наборе. Для использования этого режима работы `tr` следует использовать опцию `-s`.
4. Заменять символы в потоке с последующим устранением их повторов. При этом задаются два набора символов и устанавливается опция `-s`. На первом шаге команда `tr` заменяет символы из первого набора на соответствующие символы из второго, а затем устраняет повторения символов, заданных во втором наборе.
5. Удалять символы из потока с последующим устранением повторов если установлены опции `-d` и `-s`. При этом следует использовать два набора символов: первый – для указания удаляемых символов, и второй, указывающий на символы, повторы которых должны быть устранены.

Пример: замена символов

```
$ echo tarelka | tr a-z A-Z
TARELKA
```

Примечание: Здесь были заданы два набора символов – все буквы английского алфавита в нижнем регистре, которые заменяются на буквы в верхнем регистре.

Пример: Для удаления символов перевода строки во входном тексте можно использовать команду, показанную ниже:

```
$ ls / | tr -d '\n'
binbootdevetchomelibmntoptprocrootsbinswaptmpusrvar
```

Примечание: Здесь в качестве входного потока был использован вывод команды `ls /`*, из которого были удалены все переносы строки.*

Пример: Для демонстрации работы опции `-s` подходит следующая команда:

```
$ echo root | tr -s o
rot
```

Примечание: В этом примере были устранены повторения символа o.

Опция `-c` команды `tr` позволяет инвертировать смысл задаваемого множества символов, то есть удалить при использовании `-d` все, кроме символов, указанных в наборе.

Пример: команда `tr -dc 0-9` удалит во входном потоке все, кроме цифр.

В случае, если в первом наборе встречаются повторяющиеся символы, то символу из первого набора будет сопоставлен тот символ из второго набора, который встретился последним.

С опцией `-t` команда `tr` обрезает длину первого набора по длине второго, для того, чтобы количество символов в них равнялось.

Можно также указать класс символов из набора предопределенных символов:

Класс	***Символы***
`[:alnum:]`	Символы алфавита в любом регистре и цифры.
`[:alpha:]`	Символы алфавита в любом регистре.
`[:blank:]`	Пустое множество.
`[:cntrl:]`	Управляющие символы.
`[:digit:]`	Десятичные цифры.
`[:graph:]`	Все символы, которые могут быть напечатаны кроме пробела.
`[:lower:]`	Алфавитные символы в нижнем регистре.
`[:print:]`	Все символы, которые могут быть напечатаны.
`[:punct:]`	Все символы пунктуации.

[:space:]	Горизонтальное или вертикальное пустое (пробел, табуляция).
[:upper:]	Алфавитные символы в верхнем регистре.
[:xdigit:]	Шестнадцатеричные цифры.

Пример:

```
$echo "str" | tr [:lower:] [:upper:]
STR
```

Примечание: Здесь символы в нижнем регистре заменены символами в верхнем регистре.

21. Задание.

1. Определите, используя файл `/etc/passwd` , содержащий данные об учетных записях пользователей, сколько пользователей зарегистрировано в системе.

2. Определите для скольких пользователей оболочкой по умолчанию является `bash`.

3. Сколько имеется пользователей, `UID` которых больше 100?

4. Выведите пронумерованный список файлов в текущем каталоге.

5. Сделайте то же, но с шагом нумерации равным двум.

6. Выведите последние три строки файла `/etc/passwd` , заменив разделители – двоеточия на разделители – вертикальные черты.

7. Подсчитайте с помощью команды `wc` сколько в получившемся потоке имеется вертикальных черт.

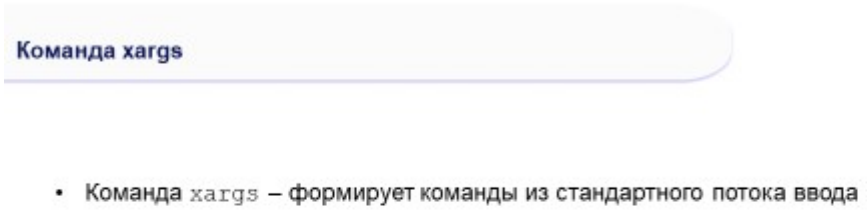
8. Сколько символов – десятичных цифр содержится в файле `/etc/sysctl.conf` ?

9. Получите список всех PID процессов, связанных с терминалами, так, чтобы весь список был выведен в одну строку, а после каждого PID процесса был указан соответствующий ему терминал. Подсказка: отфильтруйте вывод с помощью `awk` из списка всех процессов в системе.

10. Получите пронумерованный список всех пользователей (имена получите из `/etc/passwd`) в системе так, чтобы он выводился в виде одной строки. Для номера должно отводиться три позиции (разряда).

11. Измените предыдущую команду, добавив в нее `sed` и `tr`, так, чтобы перед каждым номером строки и после самих строк учетных записей были вставлены строки «+++++».

22. Команда xargs.



Команда `xargs` использует данные, передаваемые ей из стандартного потока ввода, в качестве аргументов для конструирования команды, по умолчанию – `echo`.

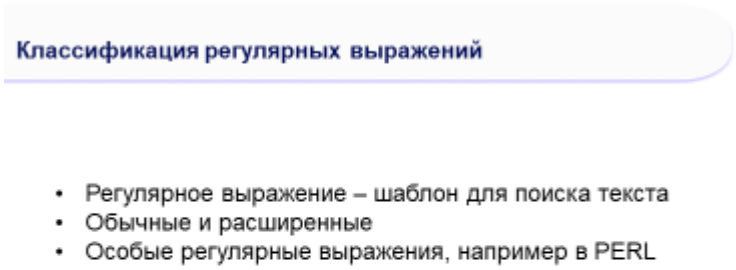
Пример: часто команда `xargs` используется с командой `find`. Обе приведенные ниже команды делают одно и то же – ищут и удаляют core-файлы, остающиеся в системе после программных сбоев:

```
$ find /usr -type f -name "core.*" -exec rm -f {} \;
$ find /usr -type f -name "core.*" | xargs rm -f
```

Примечание: В первом случае обработка найденных файлов производится командой `find`*, которая вызывает команду* `rm`*. Во втором случае имена найденных файлов отправляются через конвейер команде* `rm`*. В силу особенностей обработки потоков и запуска процессов в Linux второй вариант предпочтительнее с точки зрения быстродействия.*

Глава 2. Регулярные выражения.

1. Классификация регулярных выражений.



Регулярные выражения – это специальные шаблоны (pattern), используемые для поиска строк в текстовых файлах и потоках.

Все регулярные выражения строятся из обычных алфавитно-цифровых символов и так называемых метасимволов.

Метасимволы позволяют задавать один или более обычных алфавитно-цифровых символов.

Регулярные выражения подразделяются на:

обычные регулярные выражения;

регулярные выражения с расширенным синтаксисом.

Не все текстовые утилиты, работающие с регулярными выражениями, поддерживают расширенный синтаксис регулярных выражений.

Классификация регулярных выражений

- Метасимволы – шаблоны и квантификаторы

Шаблоны обычные	Квантификаторы обычные
. – любой символ	* - любое количество
^ - начало строки	? – ноль или один раз
$ - конец строки	
[] - диапазон	

Шаблоны расширенные	Квантификаторы расширенные
\< – начало слова	+ - один и более раз
\> – конец слова	{n} – n раз
() – группа символов	{n,m} – от n до m раз
	{n,} – не менее n раз

Метасимволы, составляющие регулярные выражения представлены двумя классами:

1. Шаблоны – специальные символы, заменяющие один или более обычных символов;

2. Квантификаторы – указатели количества вхождений символа или набора символов, находящихся в регулярном выражении непосредственно перед ними.

Примечание: То есть шаблоны указывают что должно находиться в данном месте искомой строки, а квантификаторы – сколько раз оно должно там встречаться.

Примечание: Помимо перечисленных выше регулярных выражений имеется несколько более расширенных их наборов. Например, в языке Perl используется большее количество регулярных выражений.

Примечание: Множества символов, задаваемые в квадратных скобках, имеют следующий смысл: в данном месте должен находиться любой один из символов, входящих в множество.

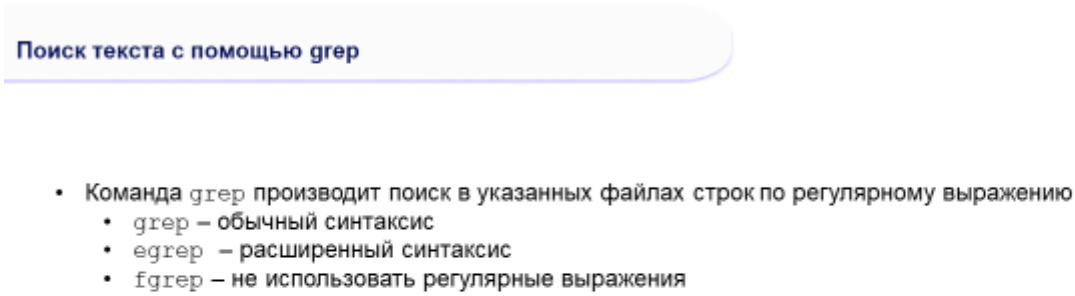
Пример: множество `[0-3]` включает в себя все числа от нуля до трех, следовательно регулярному выражению `^[0-3]$` удовлетворяют все строки, в которых находится единственный символ – число от нуля до трех (`^` - начало строки, `$` - конец строки).

Помимо множеств, заданных с помощью перечисления символов в квадратных скобках, существуют заранее определенные множества (классы) символов, приведенные в таблице ниже.

Класс	Значение
`[:alnum:]`	Множество алфавитно-цифровых символов.
`[:alpha:]`	Алфавитные символы.
`[:blank:]`	Пустые символы – пробел и табуляция.

Класс	Значение
[:cntrl:]	Символы с восьмеричными кодами от 000 до 037 и 177.
[:digit:]	Десятеричные цифры.
[:graph:]	Изображаемые символы: алфавитно-цифровые и символы пунктуации.
[:lower:]	Буквы в нижнем регистре.
[:print:]	Печатаемые символы: алфавитно-цифровые, пробел и символы пунктуации.
[:space:]	Табуляция, вертикальная табуляция, пробел, перевод страницы и строки.
[:upper:]	Буквы в верхнем регистре.
[:xdigit:]	Шестнадцатеричные символы.

2. Поиск текста с помощью grep.



Команда `grep` производит поиск в указанных файлах строк по регулярному выражению.

Команда выводит в случае удачного поиска, имя файла и строку, удовлетворяющую заданному регулярному выражению.

Если входные файлы не заданы, то команда производит ввод из стандартного потока ввода.

При этом используются три разновидности команды `grep` (ниже идет речь о GNU версии команды `grep`):

1. `grep` – интерпретирует регулярные выражения с обычным (basic) синтаксисом;
2. `grep -F` или `fgrep` – не интерпретирует регулярные выражения, воспринимая их как обычные текстовые строки;
3. `grep -E` или `egrep` – позволяет работать с расширенным синтаксисом регулярных выражений.

Примечание: Команды `grep`, `egrep` *и* `fgrep`, *разработанные в рамках GNU, реализованы в виде жестких связей. То есть, все эти три команды являются одним файлом. В других разновидностях UNIX систем это не так. К особенностям работы GNU версии этих команд следует отнести способность автоматического переключения* `egrep` *в режим работы* `grep` *в тех случаях, когда использование расширенного синтаксиса не возможно и должен быть использован (в силу особенностей реализации библиотек поиска) обычный синтаксис. Приведенные ниже примеры не всегда будут работать так же в других операционных системах, например, в Solaris и FreeBSD.*

Пример: Для поиска всех пользователей системы, использующих оболочку Bash

достаточно выполнить команду `grep` с обычным регулярным выражением.

```
$ grep 'bash$' /etc/passwd
root:x:0:0:System Administrator:/root:/bin/bash
user1:x:502:502::/home/user1:/bin/bash
figus:x:503:503::/home/figus:/bin/bash
user1:x:504:504::/home/user1:/bin/bash
```

Примечание: Регулярное выражение взято в кавычки для предотвращения интерпретации символа доллара оболочкой. Доллар здесь обозначает конец строки.

Пример: Для нахождения во всех файлов в каталоге `/etc/sysconfig/` строк, содержащих IP адреса в стандарте Ipv4 можно использовать команду `egrep` с расширенным синтаксисом регулярных выражений:

```
$ egrep '[0-9]{1,3}(\.[0-9]{1,3}){3}' /etc/sysconfig/*
```

Примечание: В этой команде обеспечивается поиск последовательности из одной, двух или трех любых десятичных цифр, которая должна быть троекратно продолжена такой же последовательностью, каждая из которых отделена от предыдущей последовательности точкой.

Командой `fgrep` удобно пользоваться для поиска строк, содержащих символы, имеющие особое значение, как регулярные выражения.

Пример: можно получить список всех процессов, не связанных с какими-либо терминалами:

```
$ ps -A | fgrep '?'
```

Примечание: Команда `fgrep` *воспринимает знак вопроса просто как обычный символ, который должен быть найден в строке.*

Опция `-i` позволяет производить поиск без учета регистра.

Пример: ниже демонстрируется, как с помощью поиска с игнорированием регистра найти все процессы в системе, где в командной строке есть подстроки `X` или `x` :

```
$ ps -A | grep -i x
  1546 ?        00:00:01 xfs
  1599 ?        00:00:19 X
  1752 ?        00:00:00 xvt
```

Если требуется подсчитать число вхождений регулярного выражения, то для этого используется опция `-c`

Пример:

```
$ egrep -c '^.{1,3}:' /etc/passwd
7
```

Примечание: В приведенном выше примере подсчитано сколько имеется пользователей, длины имен которых не превышают три символа.

Опция `-v` команды `grep` инвертирует алгоритм поиска: команда начинает искать строки, не удовлетворяющие регулярному выражению.

Пример: для того, чтобы найти список всех групп из файла `/etc/group` , в которые не входит пользователь `user1` выполним команду

```
$ grep -v user1 /etc/group
```

Поиск слов выполняется командой `egrep` с использованием метасимволов регулярных выражений, соответственно, `\<` и `\>` или с помощью опции `-w`.

Пример: для поиска всех строк в файле `/etc/sysctl.conf` , в которых встречается одиночный слово `Enable` , но не `Enables` , можно использовать команду:

```
$ egrep '\<Enable\>' /etc/sysctl.conf
# Enable the magic-sysrq key
# Enable tcp_syncookies
```

Пример:

```
$ grep -w Enable /etc/sysctl.conf
# Enable the magic-sysrq key
# Enable tcp_syncookies
```

При необходимости можно также установить режим поиска по целой строке, для чего используется опция `-x` .

Пример: Приведенная ниже команда позволит отыскать в каталоге `/etc` все файлы, которые содержат строки, состоящие из одного любого символа:

```
$ grep -x '.' /etc/*
```

Использование опции `-n` позволяет установить режим вывода номеров строк, в которых найдены искомые строки.

Пример: Для получения строки файла `/etc/hosts` , содержащие подстроки `local` , и номера этих строк выполним команду

```
$ grep -n local /etc/hosts
1:127.0.0.1             localhost.localdomain localhost
```

Примечание: Здесь в первом поле вывода перед двоеточием указан номер строки, в которой найдена искомая подстрока.

Опция `-r` заставляет команду `grep` работать рекурсивно, обрабатывая файлы в подкаталогах.

Пример: в каталоге /usr/share/doc/HOWTO/ требуется найти все файлы, содержащие строку `trainer` :

```
$ grep -r trainer /usr/share/doc/HOWTO/
```

3. Использование обратных ссылок.

- Обратная ссылка (back reference) – позволяет обратиться к найденной подстроке

Существует специальная конструкция, позволяющая в регулярном выражении обращаться к уже найденной с помощью символов группировки подстроке.

Она называется “обратная ссылка” (*back reference*) и записывается так: `\1` – что значит повтор строки, удовлетворившей найденному регулярному выражению, которое было указано, группировкой символов, в регулярном выражении до вхождения символа обратной ссылки.

Обратные ссылки очень полезны при поиске и замене.

Пример: получим список всех пользователей системы, у которых имя пользователя (первое поле `/etc/passwd`) совпадает с именем исполняемого файла оболочки (последнее поле `/etc/passwd`). Такую задачу можно решить, используя обратную ссылку:

```
$ egrep '^(.+):.*\1$' /etc/passwd
sync:x:5:0:sync:/:/bin/sync
shutdown:x:6:0:shutdown:/:/sbin/shutdown
halt:x:7:0:halt:/:/sbin/halt
```

Примечание: В этом регулярном выражении шаблоном для имени пользователя является `^(.+):` *, то есть строка из ненулевого количества любых символов. При этом специально использован символ группирования, так как с помощью него запоминается строка символов, соответствующая имени пользователя. Далее эта же найденная строка с помощью обратной ссылки* `\1` *сравнивается с именем файла оболочки.*

Примечание: Вообще говоря, приведенный выше пример, не будет работать в других операционных системах, если в них не используются GNU версия `grep`*. Это связано с концептуальной невозможностью использования обратных ссылок в программах поиска подстрок,*

реализующих расширенный синтаксис регулярных выражений. GNU версия `egrep` *(он же* `grep`*) обеспечивает возможность поиска строк с использованием обратных ссылок.*

Пример: Ниже приведена реализация предыдущего примера, которая не использует `egrep`:

```
$ grep '^\(..*\):.*\1$' /etc/passwd
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
```

В регулярном выражении может быть использовано более одного символа обратной ссылки.

Если используется более одной обратной ссылки, то должно быть использовано несколько символов группировки, а номера у обратных ссылок должны соответствовать порядковым номерам вхождений символов группировки в регулярном выражении.

Пример: получим список всех IP адресов и имен хостов из файла `/etc/hosts` (связывает IP адрес и имя компьютера в сети) только для тех хостов, чьи IP адреса начинаются либо со `127`, либо со `192`, а также у этих хостов должны быть указаны полностью определенное доменное имя (Fully Qualified Domain Name – FQDN) и короткое имя (или наоборот – порядок не важен).

```
$ egrep '^(192|127)\..*([[:alnum:]]+)\..*\2$' /etc/hosts
127.0.0.1               localhost.localdomain localhost
```

Примечание: В данном примере первый символ группирования предназначен для организации перечисления (инфиксного оператора `или`*), а второй – для организации поиска по обратной ссылке. Так как искомые имена компьютеров могут быть определены по второму символу группирования, то используется оператор обратной ссылки* `\2`*.*

4. Задание.

1. Составьте регулярное выражение для пустой строки.

2. Как выявить в текстовом файле все строки, содержащие символ с восьмеричным кодом `007`?

3. Найдите опцию, позволяющую `grep` выводить имена файлов перед найденными строками.

4. С помощью `grep` получите список всех пользователей системы, с `10<GID<100`.

5. Отфильтруйте все сообщения, находящиеся в файле `/var/log/messages` (или `/var/log/syslog`) так, чтобы были выведены сообщения о событиях, происшедших вчера в диапазоне с `8.30` утра до `12.30`.

6. Найдите в каталоге `/etc/rc.d` и его подкаталогах все файлы, содержащие слово `fsck`. В выводе должны присутствовать имена найденных файлов.

7. С помощью команд `ps`, `grep`, `sort`, `uniq` подсчитайте сколько процессов в настоящий момент связано с каждым виртуальным терминалом (`tty`) в вашей системе.

8. Получите список всех учетных записей пользователей вашей системы, у которых имя домашнего каталога (последнего каталога в пути) совпадает с именем пользователя.

9. Проверьте, имеются ли в файле `/var/log/messages` (или `/var/log/syslog`) сообщения, у которых во времени совпадают часы и минуты (например, `09:09`).

5. Использование регулярных выражений с sed.

- Используются для нахождения строк, подлежащих обработке
- Указываются в косых чертах
- Поддерживает обычный синтаксис
- Не поддерживает обратных ссылок
- Можно использовать символ &
- Опция `-r` – включает расширенный синтаксис и обратные ссылки

Регулярные выражения позволяют находить строки, которые должны подвергнуться обработке утилитой `sed`.

Регулярное выражение указывается в косых чертах.

Опция `-e` команды `sed` используется для задания нескольких директив (опреаторов) `sed` в одной командной строке.

Пример: для вставки строки – разделителя после учетных записей, использующих в качестве оболочки `bash`, с предварительным удалением строк учетных записей пользователей, использующих оболочки, имена которых не заканчиваются на `sh`, можно использовать команду:

```
$ sed -e '/[^s][^h]$/d' -e '/bash/a \
@@@@@@@@@@@@@@@@@@@@' /etc/passwd
root:x:0:0:System Administrator:/root:/bin/bash
@@@@@@@@@@@@@@@@@@@@
bamboo:x:501:100::/home/bamboo:/bin/csh
user1:x:502:502::/home/user1:/bin/bash
@@@@@@@@@@@@@@@@@@@@
figus:x:503:503::/home/figus:/bin/bash
@@@@@@@@@@@@@@@@@@@@
user1:x:504:504::/home/user1:/bin/bash
```

Примечание: Первая директива удаляет все строки, не заканчивающиеся на строку `sh`*, а вторая добавляет разделительную строку только после строк, заканчивающихся на* `bash`*.*

Пример: удобно использовать `sed` для замены строк в потоке, выделяя нужные строки с помощью регулярных выражений. Для удаления из потока, считанного из файла `/etc/group` все вхождения подстроки `daemon` в конце строк на `pokemon` выполним команду

```
$ sed 's/daemon$/pokemon/' /etc/group
```

По умолчанию утилита `sed` работает с базовым набором регулярных выражений.

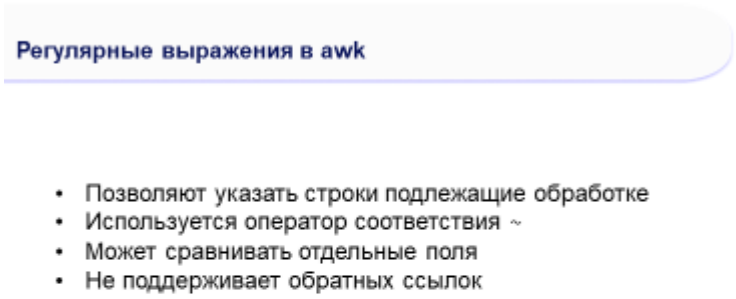
Для поддержки расширенных регулярных выражений нужно использовать опцию `-r` (`-E`).

При замене подстрок имеется возможность вставлять в подстроку – замену исходную подстроку, удовлетворившую шаблону. Для обращения к подстроке, удовлетворившей найденному регулярному выражению, в `sed` используется оператор `&` .

Пример: создадим сценарий для копирования всех всех файлов текущего каталога с суффиксом `.txt` так, чтобы к имени файлов была добавлена строка `.bak` и выполним его:

```
$ ls *.txt | sed 's/.*txt$/cp -v & &.bak/' | sh
«bamboo.txt» -> «bamboo.txt.bak»
«div_4.txt» -> «div_4.txt.bak»
«div.txt» -> «div.txt.bak»
```

6. Регулярные выражения в awk.



В `awk` регулярные выражения могут использоваться аналогично тому, как они используются в `sed` – для указания строк, которые должны подлежать обработке.

Пример: следующая команда выведет список только тех процессов в системе, которые были запущены в промежутке с 15:20 до 15:29 :

```
$ ps -ef | awk '/[[:blank:]]15:2[0-9][[:blank:]]/{print $5,$8}'
15:20 /usr/lib/openoffice/program/soffice.bin
15:29 [kjournald]
```

Примечание: Эта команда выводит два столбца из потока, генерируемого командой `ps -ef` *: столбец с информацией о времени запуска процесса и столбец с именем команды, запустившей процесс.*

Утилита `awk` позволяет проверять на соответствие регулярному выражению не только целые строки, но и отдельные поля строк.

Для сопоставления строки или поля регулярному выражению можно использовать оператор ~ .

Пример: Ниже приведен пример команды, которая выполняет ту же задачу, что и предыдущая:

```
$ ps -ef | awk '$5~/15:2[0-9]/{print $5,$8}'
15:20 /usr/lib/openoffice/program/soffice.bin
```

```
15:29 [kjournald]
```

Примечание: Синтаксис этой команды намного яснее предыдущей: здесь проверяется совпадение только пятого поля строки с регулярным выражением. Это достигается с помощью оператора ~ , который требует удовлетворения данного поля регулярному выражению.

Для получения списка строк, не удовлетворяющих регулярному выражению, используется оператор `!~` .

Пример: команда, приведенная ниже, выдаст список только тех процессов, которые НЕ были запущены в промежуток с 15:20 до 15:29 :

```
$ ps -ef | awk '$5!~/15:2[0-9]/{print $5,$8}'
```

Команда `awk` (в версии GNU) позволяет работать с расширенным синтаксисом регулярных выражений.

Пример: получим список всех процессов, в командной строке которых есть `bash` или `csh` :

```
$ ps -A | awk '$4~/(ba|c)sh/'
 1749 pts/0    00:00:00 bash
 7372 pts/0    00:00:00 csh
```

Утилита `awk` не поддерживает обратных ссылок, а символ группирования здесь используется лишь вместе с оператором перечисления `или` (инфикс `|`).

Примечание: Однако, практически это и не требуется, так как в `awk` имеется возможность представления строки в виде набора полей, которые легко можно сравнивать между собой.

7. Задание.

1. С помощью `sed` получите список только тех процессов, которые связаны с виртуальными терминалами с первого по четвертый (`tty1 - tty4` или `pts/1 - pts/4` – в зависимости от системы).

2. Произведите замену строки `user` в потоке, считанном из файла `/etc/passwd`, на строку `extrauser` .

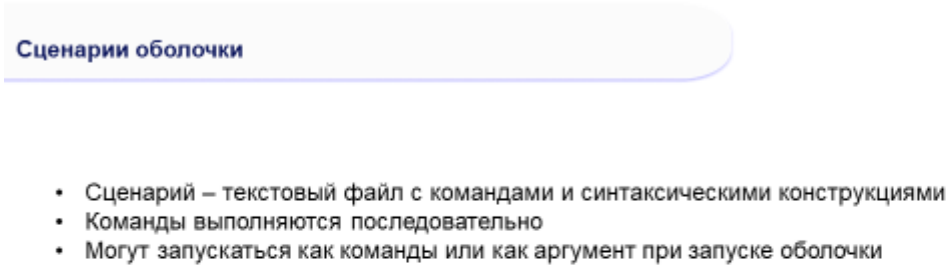
3. С помощью `awk` получите список учетных записей пользователей, для которых в качестве оболочки установлены `/bin/nologin`, `/bin/false`, `/dev/null` или же вообще в седьмом поле `/etc/passwd` для них ничего не указано.

4. Как с помощью `awk` проверить не вставлены ли в текст символы звукового сигнала (beep)?

5. С помощью `sed` и `ls` сделайте копии всех файлов с суффиксом `txt` в каталог `~/backups` так чтобы имена конечных файлов выглядели так: `file_YYYYMMDD.txt`. Где `YYYYMMDD` это текущая дата. Предварительно проверьте наличие и при его отсутствии создайте каталог ~/`backups`.

Глава 3. Написание сценариев Bash.

1. Сценарии 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`.

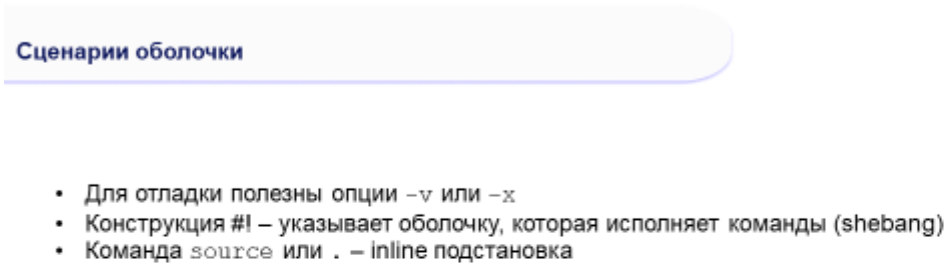
Сценарии оболочки могут быть исполнены двумя различными путями:

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:

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



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

(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.

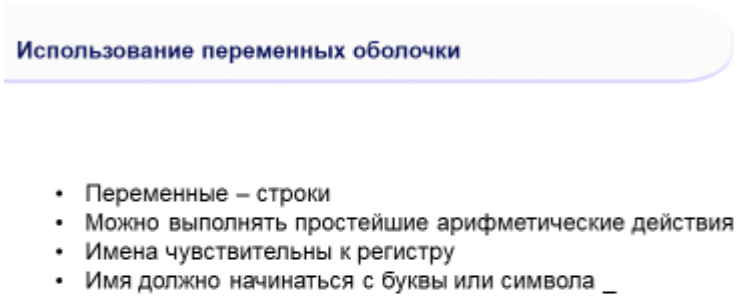
2. Запустите сценарий, указывая оболочку явно.

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.



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 $f11
```

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

<u>Примечание:</u> 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`*.*

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}
```

```
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

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
```

```
$ 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.

- 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:

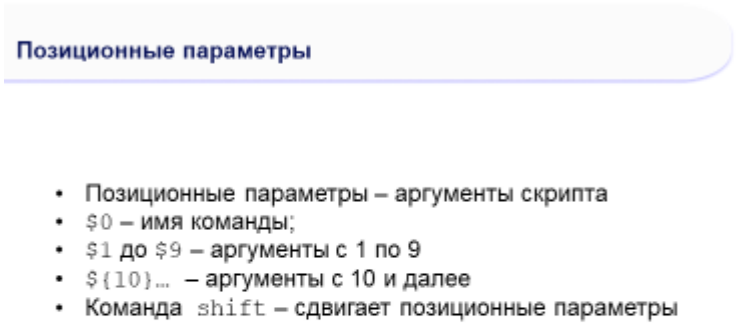
```
read var1 var2 var3
```

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 позволяют передавать сценариям оболочки аргументы, указанные в командной строке.

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

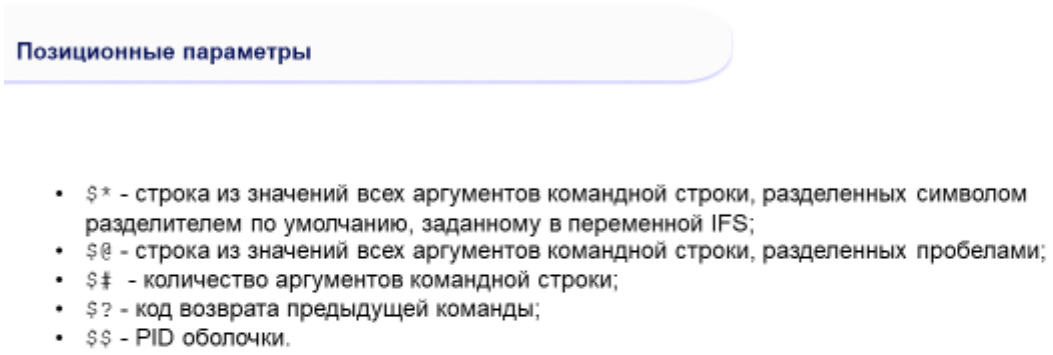
1. `$0` – имя команды;
2. параметры от `$1` до `$9` - значения девяти аргументов командной строки от первого до, соответственно, девятого.

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.*



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

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
```

```
2
```

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 повлияв 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`.

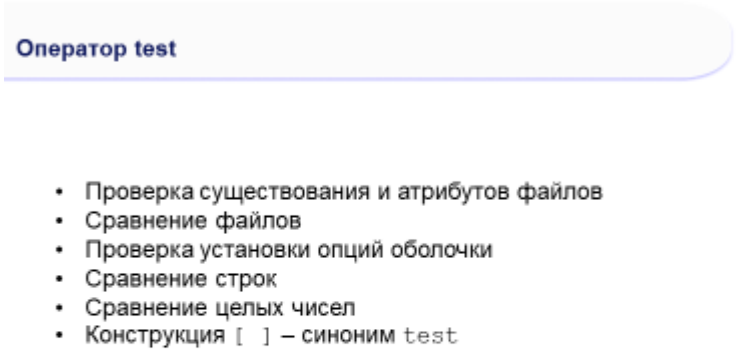
10. Исследуйте работу опции `-v bash`.

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

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

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

8. Оператор 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.

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
```

```
$ 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.

- `&&` - 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.

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.
```

```
$ ./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
```

```
        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

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
```

```
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 символьного 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`.

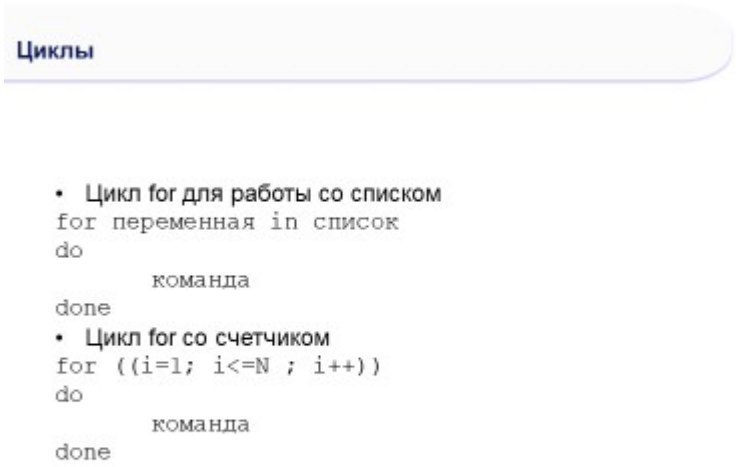
5. Напишите сценарий, проверяющий имя текущего каталога и выводящий сообщение об ошибке, если оно короче пяти символов.

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.



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:

```
$ ./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:

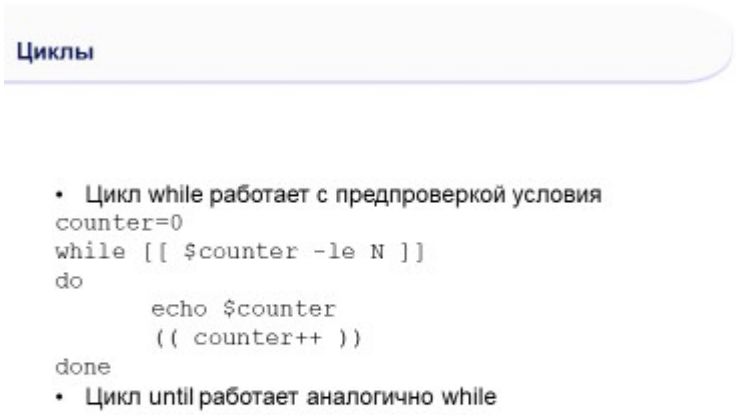
```
#!/bin/bash

[ $# -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------
```



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
```

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, 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
```

```
                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.
```

```
        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
```

```
#!/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.

Ответы к заданиями

16. Глава 1 задание п.5

1. Получите список всех процессов в системе и перенаправьте вывод в файл ps.txt.

Ответ:

```
$ ps -e > ps.txt
```

2. Выполните команду поиска всех обычных файлов в каталоге `/usr` так, чтобы найденные имена файлов были записаны в файл `SysComm` в домашнем каталоге, а поток ошибок был записан в нуль-устройство `/dev/null`.

Ответ:

```
$ find /usr -type f > SysComm 2> /dev/null
```

3. Выполните ту же команду, но так, чтобы потоки вывода и ошибок были записаны в файл `SysComm`.

Ответ:

```
$ find /usr -type f >& SysComm
```

4. Допишите в конец файла `SysComm` информацию о текущей дате и времени, выводимую командой `date`.

Ответ:

```
$ date >> SysComm
```

5. Выведите текущую дату в формате: YYYY-MM-DD hh:mm:ss

Ответ:

```
$ date +"%Y-%m-%d %T"
```

6. Установите опцию `noclobber` и сотрите содержимое файла `ps.txt` с помощью перенаправления вывода. Снимите опцию `noclobber` .

Ответ:

```
$ set -o noclobber
$ >| ps.txt
$ set +o noclobber
```

7. Проведите поиск файлов символьных ссылок в каталоге `/usr/share/doc` так, чтобы их список был выведен в отсортированном виде с помощью фильтра `sort`.

Ответ:

```
$ find /usr/share/doc -type l | sort
```

8. Выведите сортированный список пользователей, вошедших в сеанс, в системе в файл seans.txt и на экран одновременно.

Ответ:

```
$ who | sort | tee seans.txt
```

9. Определите из `man ascii` восьмеричный код системного звукового сигнала и выведите его с помощью `echo`.

Ответ:

```
$ echo -e '\007'
```

10.Опция `-n` команды `echo` подавляет добавление перевода строки. С помощью `man` определите, как это можно сделать иначе.

Ответ: из `man echo:` `\c  produce no further output`

```
$ echo -e abc\\c
```

17. Глава 1 Задание п.9

1. В чем разница между командами `more /etc/profile` и `more < /etc/profile`? Заметно ли отличия в командах при их выполнении?

Ответ: См. на последнюю строку. В случае с `more /etc/profile` показывает процент считанного файла, а при перенаправлении (<) из файла процент не показывает. Во втором случае (<) more показывает данные полученные не из файла, размер которого известен, а из стандартного потока ввода и more «не знает» когда он закончится.

2. Команда man обладает опцией для явного указания пэйджера, в котором будет отображаться найденная страница помощи. Опробуйте ее действие.

Ответ:

```
$ man man
```

```
$ man -P more man
```

3. Как можно запустить man так же, как и в предыдущем задании, но используя переменную окружения `PAGER` ?

Ответ:

```
$ PAGER=cat
```

```
$ export PAGER
```

```
$ man man
```

или еще проще

```
$ PAGER=cat man man
```

4. Команда `less` позволяет открыть несколько файлов, указав их в качестве аргументов.

Ответ:

```
$ less f1 f2
```

переключиться на следующий файл командой `:n`

Это можно узнать из встроенной справки (команда `h`).

5. Команда `cat` обладает опцией, устанавливающей нумерацию выводимых строк. Определите, какая она.

Ответ:

```
$ cat -n
```

6. Найдите опцию `cat` , позволяющую этой команде удалять последовательно повторяющиеся переводы строки, оставляя лишь один перевод строки.

Ответ:

```
$ cat -s
```

7. Выведите содержимое файла `/etc/passwd` в обратном порядке следования строк.

Ответ:

```
$ tac /etc/passwd
```

8. Для чего предназначена опция `-b` команды `tac` ?

Ответ: присоединяет разделитель к началу строки

9. Получите имена трех наиболее объемных файлов в каталоге `/usr/bin`.

Ответ:

```
$ ls -Ss /usr/bin
total 47940
5232 ld.gold
3104 dwp
2284 vim
1524 systemd-analyze
 984 ld.bfd
 944 bash
...
```

опция `-S` сортирует вывод по размеру. Опция `-s` показывает размер файлов. Общая сводка по размерам всех файлов нам не нужна, поэтому нужно выводить только со 2 строки (команда tail -n +2) а, затем, получить только 3 первых строки:

```
$ ls -Ss /usr/bin | tail -n +2 | head -3
```

или другой вариант

```
$ ls -Ss /usr/bin | head -4 | tail -3
```

18. Глава 1 Задание п.13

1. Получите столбец из имен пользователей, находящийся в данный момент в сеансе.

Ответ:

```
$ who | cut -f1 -d" "
```

2. Получите столбец, составленный из символов строк – имен файлов в текущем каталоге

таким образом, чтобы были выведены символы с пятого до последнего.

Ответ:

```
$ ls | cut -c5-
```

3. С помощью `sed` получите список только тех процессов, которые связаны с каким-либо терминалом .

Ответ:

```
$ ps -ef | sed "/ [?] /d"
```

необходимы пробелы перед и после ? знака

4. В полученном списке процессов с помощью `sed` замените все строки `user` на `provider`.

Ответ:

```
$ ps -ef | sed "/ [?] /d"| sed 's/user/provider/'
```

5. Используя `awk` из списка процессов, полученного командой `sed`, удалите второй столбец.

Ответ:

```
$ ps -ef | sed "/ [?] /d"| sed 's/user/provider/' | awk '{$2=""; print $0}'
```

`{$2=""; print $0}` - метод простой но не совсем правильный. Фактически вторая колонка не удаляется, но обнуляется.

Такая команда `awk '{for(col=1;col<NF;col++) {if (col!=2) printf $col" "}; print $NF }'` действительно удаляет вторую колонку.

6. С помощью `awk` пронумеруйте полученный в результате предыдущих действий список.

Ответ:

```
… | awk '$0=NR" "$0'
```

19. Глава 1 Задание п.18

1. Введите команду, сравнивающую содержимое вашего домашнего каталога с домашним каталогом суперпользователя.

Ответ:

```
# diff ~root ~user
```

2. Определите, с помощью какой опции `diff` можно рекурсивно сравнить эти каталоги.

Ответ:

```
# diff -r ~root ~user
```

3. Получите файл `patch.txt` с отличиями файлов `ps1.txt` и `ps2.txt` в контекстном формате. Удалите файл `ps2.txt` и восстановите его содержимое с помощью файлов `ps1.txt` и `patch.txt`.

Ответ:

```
$ ps > ps1.txt
$ ps > ps2.txt
$ diff -c ps1.txt ps2.txt > patch.txt
$ rm ps2.txt
$ patch -o ps2.txt ps1.txt patch.txt
```

4. Проверьте отличаются ли **последние два килобайта!!!** файлов `r1.bin` и `r2.bin`.

Ответ:

Получим 2 файла со случайными данными и разного размера

```
$ dd if=/dev/urandom of=r1.bin bs=1024 count=1000
$ dd if=/dev/urandom of=r2.bin bs=1024 count=2000
```

Чтобы сравнивать два последних килобайта нужно определить размер файла т. к. в `cmp` можно указать сколько пропустить от начала (`man cmp`).

Утилита `stat` показывает разные характеристики файлов в т.ч. и размер:

```
$ stat -c %s r1.bin
```

получив размер мы можем указать сколько пропустить от начала каждого файла. Итоговый ответ:

```
$ cmp r1.bin r2.bin $(($(stat -c %s r1.bin)-2048)) $(($(stat -c %s r2.bin)-2048))
```

5. Получите список групп пользователей в системе, отсортированный по GID в обратном числовом порядке.

Ответ:

```
$ cat /etc/passwd | sort -t: -nk3 -r
```

6. С помощью утилит `find`, `head` и `sort` получите список из десяти файлов в домашнем каталоге, занимающих наибольшее дисковое пространство.

Ответ:

```
$ find ~ -type f -exec ls -sh {} \; | sort -hr | head
```

7. Выведите список тех файлов текущего каталога, для которых можно найти в этом каталоге файлы, первые три символа в именах которых совпадают не менее, чем у двух

файлов.

Решение:

Для начала получим список из первых 3 букв имен файлов:

```
$ ls | cut -c1-3
Wel
Wel
bam
bam
div
div
emp
emp
fir
msd
…
```

Из этого списка нужны только повторяющиеся строки. Дополнительно выведем количество повторений и отсортируем по количеству повторений:

```
$ ls | cut -c1-3 | sort | uniq -dc | sort -nr
      6 tex
      5 tab
      2 sam
      2 sal
      2 emp
      2 div
      2 bam
      2 Wel
```

Мы увидели что повторения есть, но пока не видим имен файлов.

Чтобы получить имена мы сконструируем команды для получения этих имен:

```
$ ls | cut -c1-3 | sort | uniq -dc | sort -nr | awk '{print "echo "$0"*"}'
echo       6 tex*
echo       5 tab*
echo       2 sam*
echo       2 sal*
echo       2 emp*
```

```
    echo       2 div*
    echo       2 bam*
    echo       2 Wel*
```

Теперь остается только передать команды на выполнение оболочке. Итоговый ответ:

```
    $ ls | cut -c1-3 | sort | uniq -dc | sort -nr | awk '{print
"echo "$0"*"}' | bash
    6 text.txt text1 text1.txt text2 text_h.txt text_t.txt
    5 tab_class.txt tab_dept tab_dept.dat tab_dept.txt
tab_staff.txt
    2 sample sample.txt
    2 salary.txt salary_2.txt
    2 empty1 empty2
    2 div.txt div_4.txt
    2 bamboo bamboo.txt
    2 Wells Wells.txt
```

Важное замечание. Иногда необходимо «налету» сконструировать скрипт и тут же его выполнить.

Другой ответ с использованием цикла. Циклы часто бывают удобнее, но не в этом случае:

```
    $ { for f in $(ls | cut -c1-3| sort | uniq -d); do echo -n $
(ls ${f}*| wc -l )" "; echo ${f}*; done } | sort -nr
    6 text.txt text1 text1.txt text2 text_h.txt text_t.txt
    5 tab_class.txt tab_dept tab_dept.dat tab_dept.txt
tab_staff.txt
    2 sample sample.txt
    2 salary.txt salary_2.txt
    2 empty1 empty2
    2 div.txt div_4.txt
    2 bamboo bamboo.txt
    2 Wells Wells.txt
```

8. Имеется два каталога без подкаталогов. Как с помощью `join` получить список файлов, которые имеются в обоих каталогах?

Ответ:

```
# ls ~root | sort > /tmp/root.ls
# ls ~admin | sort > /tmp/admin.ls
```

```
# join  /tmp/root.ls /tmp/admin.ls
```

9. Как сделать то же самое для файлов, которые имеются только в одном из каталогов и отсутствуют в другом?

Ответ:

```
# join  -v1 /tmp/root.ls /tmp/admin.ls
# join  -v2 /tmp/root.ls /tmp/admin.ls
```

10.Как получить список файлов кроме тех, которые имеются в обоих каталогах?

Ответ:

```
# join  -v1 v2  /tmp/root.ls /tmp/admin.ls
```

20. Глава 1 Задание п.21

1. Определите, используя файл `/etc/passwd` , содержащий данные об учетных записях пользователей, сколько пользователей зарегистрировано в системе.

Ответ:

```
$ cat /etc/passwd | wc -l
```

2. Определите для скольких пользователей оболочкой по умолчанию является bash.

Ответ:

```
$ awk -F':' '$7~/bash$/' /etc/passwd | wc -l
```

3. Сколько имеется пользователей, UID которых больше 100?

Ответ:

```
$ awk -F':' '$3>100' /etc/passwd | wc -l
```

4. Выведите пронумерованный список файлов в текущем каталоге.

Ответ:

```
$ ls | nl
```

5. Сделайте то же, но с шагом нумерации равным двум.

Ответ:

```
$ ls | nl -i 2
```

6. Выведите последние три строки файла `/etc/passwd` , заменив разделители – двоеточия на разделители – вертикальные черты.

Ответ:

```
$ tail -n 3 /etc/passwd | tr  ':' '|'
```

7. Подсчитайте с помощью команды `wc` сколько в получившемся потоке имеется вертикальных черт.

Ответ:

```
$ tail -n 3 /etc/passwd | tr ':' '|' | tr -dc '|' | wc -m
```

8. Сколько символов – десятичных цифр содержится в файле `/etc/sysctl.conf` ?

Ответ:

```
$ cat /etc/sysctl.conf | tr -dc 0-9 | wc -m
```

9. Получите список всех PID процессов, связанных с терминалами, так, чтобы весь список был выведен в одну строку, а после каждого PID процесса был указан соответствующий ему терминал. Подсказка: выделите нужные поля с помощью awk из списка всех процессов в системе.

Ответ:

```
$ ps -eo pid,tty | awk '$2!="?" && $2!="TT" {printf $0"\t"}'
```

10.Получите пронумерованный список всех пользователей (имена получите из `/etc/passwd`) в системе так, чтобы он выводился в виде одной строки. Для номера должно отводиться три позиции (разряда).

Ответ:

```
$ cut -f1 -d: /etc/passwd | nl -w3 | awk '{printf $0"\t"}'
```

11.Измените предыдущую команду, добавив в нее `sed` и `tr`, так, чтобы перед каждым номером строки были вставлены строки «+++++» и после самих строк учетных записей были вставлены строки «--------».

Ответ:

```
$ cut -f1 -d: /etc/passwd | nl -w3 | awk '{printf $0"\t"}' |
tr '\t' '\n' | sed -e '/^..[0-9]$/i++++++++++++' -e
'/^[a-zA-Z]/a------------'
```

21. Глава 2 Задание п.4

1. Составьте регулярное выражение для пустой строки.

Ответ: `^$`

2. Как выявить в текстовом файле все строки, содержащие символ с восьмеричным кодом 007?

Ответ:

```
$ grep $(echo -ne '\007') <file>
```

3. Найдите опцию, позволяющую `grep` выводить имена файлов перед найденными строками.

Ответ:

```
$ grep --help | grep filename
```

Опция **`-H, --with-filename`** `print the file name for each match`

4. С помощью `grep` получите список всех пользователей системы, с 10<GID<100.

Ответ: В регулярных выражениях нельзя сравнивать числа как числа. Для регулярных выражений числа это тоже строки. Кроме того в данном примере нужно правильно определить позицию где находится GID в файле `/etc/passwd`.

```
$ grep '^..*:.*:[0-9]*:[1-9][0-9]:' /etc/passwd
```

5. Отфильтруйте все сообщения, находящиеся в файле/var/log/messages так, чтобы были выведены сообщения о событиях, происшедших вчера в диапазоне с 8.30 утра до 12.30.

Ответ:

Перед выполнением задания следует ознакомится со структурой записей в журнале:

```
$ sudo tail -1 /var/log/messages
```

```
Jan 14 07:59:25 lin00 systemd: Started Session c1 of user user.
```

Обратите внимание как задается время и на какой оно позиции находится.

Здесь так же как и с числами: в регулярных выражениях нет возможности анализировать время напрямую. Чтобы решить эту задачу, надо для начала определить интервалы времени, которые нас интересуют:

8:30-8:59

9:00-9:59

10:00-10:59

11:00-11:59

12:00-12:29

Т.о. у нас есть 2 интервала по полчаса и 3 по часу. Интервалы 10 и 11 часов мы можем определить как одно рег. выражение:

```
1[01]:[0-5][0-9]
```

Теперь этот интервал в 2 часа можем объединить с интервалом 9 часов:

```
(09|1[01]):[0-5][0-9]
```

Остались 2 интервала:

`08:[3-5][0-9]` и `12:[0-2][0-9]`

Теперь все это объединяем в общее регулярное выражение:

```
(08:[3-5][0-9]|(09|1[01]):[0-5][0-9]|12:[0-2][0-9]):
```

Теперь нужно получить вчерашнюю дату

```
$ LANG=C date -d '1 day ago'
```

Мы получим что-то подобное: `Wed Jan 13 08:09:06 UTC 2021`

`LANG=C` нужно, чтобы получить нужный язык. Из всего вывода нам нужны только месяц и день. см. `date --help`

```
$ LANG=C date -d '1 day ago' +'%b %_d'
```

```
Jan 13
```

Наконец команда будет следующая:

```
$ sudo egrep "^$(LANG=C date -d '1 day ago' +'%b %_d') (08:[3-
5][0-9]|(09|1[01]):[0-5][0-9]|12:[0-2][0-9]):" /var/log/messages
```

6. Найдите в каталоге `/etc/rc.d` и его подкаталогах все файлы, содержащие слово `fsck` . В выводе должны присутствовать имена найденных файлов.

Ответ:

```
$ fgrep -r fsck /etc/rc.d/
```

7. С помощью команд `ps`, `grep`, `sort`, `uniq` подсчитайте сколько процессов в настоящий момент связано с каждым виртуальным терминалом (tty) в вашей системе.

Ответ:

```
$ ps -eo tty | fgrep -v '?' | fgrep -v TT | sort | uniq -c
```

8. Получите список всех учетных записей пользователей вашей системы, у которых имя домашнего каталога (последнего каталога в пути) совпадает с именем пользователя.

Ответ: Посмотрим на структуру текста в passwd

```
$ getent passwd user
```

`user``:x:1000:1000::/home/`**`user`**`:/bin/bash`

Нам нужно искать совпадения в первом и шестом полях, поэтому:

```
$ getent passwd | egrep '(.+):.*:[0-9]+:[0-9]+:.*:.*/\1:'
```

или

```
$ getent passwd | grep '\(..*\):.*:[0-9][0-9]*:[0-9][0-9]*:.*:.*/\
1:'
```

9. Проверьте, имеются ли в файле `/var/log/messages` (файл системного журнала) сообщения, у которых во времени совпадают часы и минуты (например, `09:09`).

Ответ:

sudo egrep '^... .. ([0-9][0-9]):\1:' /var/log/messages

22. Глава 2 Задание п.7

1. С помощью `sed` получите список только тех процессов, которые связаны с виртуальными терминалами с первого по четвертый (`tty1 – tty4` или `pts/1 – pts/4` – в зависимости от системы).

Ответ:

```
$ ps -ef | sed -n '/ pts\/[1-4] /p'
```

2. Произведите замену строки `user` в потоке, считанном из файла `/etc/passwd`, на строку `extrauser` .

Ответ:

```
$ cat /etc/passwd | sed 's/user/extrauser/g'
```

3. С помощью `awk` получите список учетных записей пользователей, для которых в качестве оболочки установлены `/bin/nologin`, `/bin/false`, `/dev/null` или же вообще в седьмом поле `/etc/passwd` для них ничего не указано.

Ответ:

```
$ cat /etc/passwd | awk -F:
'$7~/(bin\/(nologin|false)|\/dev\/null|^$)/'
```

4. Как с помощью `awk` проверить не вставлены ли в текст символы звукового сигнала (`beep`)?

Ответ:

```
$ echo -e 'String whith beep\007\nString whithout beep' | awk
'$0~/\a/'
```

5. С помощью `sed` и `ls` сделайте копии всех файлов с суффиксом `txt` в каталог `~/backups` так чтобы имена конечных файлов выглядели так: `file_YYYYMMDD.txt`. Где `YYYYMMDD` это текущая дата. Предварительно создайте каталог `backups`.

Ответ:

Получим сначала дату в нужном формате. Читаем вывод команды `date --help`

```
$ date +%Y%m%d
```

20210115

Далее создадим каталог `~/backups`:

```
$ test -d ~/backups || mkdir ~/backups
```

Теперь получим список файлов и передадим его команде `sed`

```
$ ls *.txt  | sed "s#\(.*\)\(.txt$\)#cp '\0' ~/backups/'\1_$
(date +%Y%m%d)\2'#"
```

Обращаем внимание на одинарные кавычки тут они обязательны т. к. имена файлов могут иметь пробелы или другие спец символы.

Еще одна особенность команда `s` в `sed` использует другой символ разделитель - #. Это сделано для удобства, чтобы не экранировать символ /.

Так же обратите внимание на то, как сформировано имя нового файла из обратных ссылок и командной подстановки.

Символ ~ нельзя помещать в одинарную кавычку.

Остается только выполнить команды и проверить результат

```
$ ls *.txt  | sed "s#\(.*\)\(.txt$\)#cp '\0' ~/backups/'\1_$
(date +%Y%m%d)\2'#" | bash
```

```
$ ls ~/backups/
bamboo_20210115.txt    sample_20210115.txt
```

```
text_20210115.txt
    div_20210115.txt       second_20210115.txt
text_h_20210115.txt
    div_4_20210115.txt     sort_m_20210115.txt
text_t_20210115.txt
    first_20210115.txt     split_20210115.txt
three_20210115.txt
    msdos_20210115.txt     tab_class_20210115.txt
time_20210115.txt
    personal_20210115.txt  tab_dept_20210115.txt
Wells_20210115.txt
    salary_20210115.txt    tab_staff_20210115.txt
```

23. Глава 3 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.

24. Глава 3 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 ex-
    pansion,  parameter expansion, command substitution, and arithmetic ex-
    pansion.  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 indirec-
    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 связанна 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:

```
$ 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"
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
```

25. Глава 3 Задание п.11

1. Как с помощью `test` проверить является файл специальным файлом символьного 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
$ ./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 {1arg}
$ ./testfile.sh qwert asdfg
Usage: ./testfile.sh {1arg}
```

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)
```

26. Глава 3 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
```

27. Глава 3 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.

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
```