

# **Отчет по лабораторной работе №12**

**Операционные системы**

Ардеев Никита Евгеньевич

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>7</b>
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>9</b>
<b>5</b>	<b>Выводы</b>	<b>14</b>
<b>6</b>	<b>Ответы на контрольные вопросы</b>	<b>15</b>

## Список иллюстраций

4.1	Выполнение программы 1 . . . . .	9
4.2	Написанная программа 1 . . . . .	9
4.3	Выполнение программы 2 . . . . .	10
4.4	Написанная программа 2 . . . . .	10
4.5	Выполнение программы 3 . . . . .	11
4.6	Написанная программа программа 3 . . . . .	11
4.7	Создание файла с программой 4 . . . . .	12
4.8	Написанная программа 4 . . . . .	13
4.9	Выполнение программы 4 . . . . .	13

## Список таблиц

# 1 Цель работы

Цель данной лабораторной работы - изучить основы программирования в оболочке ОС UNIX/Linux, научиться писать небольшие командные файлы.

## 2 Задание

1. Написать скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, в котором содержится его исходный код) в другую директорию backup в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор zip, bzip2 или tar. Способ использования команд архивации необходимо узнать, изучив справку.
2. Написать пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов.
3. Написать командный файл — аналог команды ls (без использования самой этой команды и команды dir). Требуется, чтобы он выдавал информацию о нужном каталоге и выводил информацию о возможностях доступа к файлам этого каталога.
4. Написать командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде аргумента командной строки

### 3 Теоретическое введение

Командный процессор (командная оболочка, интерпретатор команд shell) — это программа, позволяющая пользователю взаимодействовать с операционной системой компьютера. В операционных системах типа UNIX/Linux наиболее часто используются следующие реализации командных оболочек:

оболочка Борна (Bourne shell или sh) — стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций;

C-оболочка (или csh) — надстройка на оболочке Борна, использующая C-подобный синтаксис команд с возможностью сохранения истории выполнения команд;

оболочка Корна (или ksh) — напоминает оболочку C, но операторы управления программой совместимы с операторами оболочки Борна;

BASH — сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек C и Корна (разработка компании Free Software Foundation).

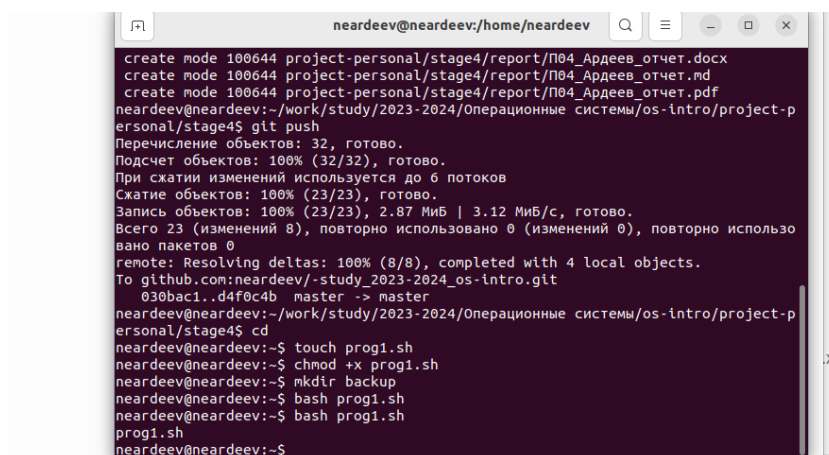
POSIX (Portable Operating System Interface for Computer Environments) — набор стандартов описания интерфейсов взаимодействия операционной системы и прикладных программ. Стандарты POSIX разработаны комитетом IEEE (Institute of Electrical and Electronics Engineers) для обеспечения совместимости различных UNIX/Linux-подобных операционных систем и переносимости прикладных программ на уровне исходного кода. POSIX-совместимые оболочки разработаны на базе оболочки Корна. Рассмотрим основные элементы программирования в оболочке bash. В других оболочках большинство команд будет совпадать с

описанными ниже.



## 4 Выполнение лабораторной работы

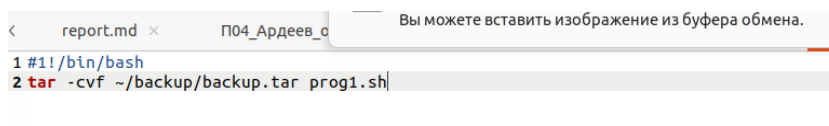
Создаю файл `prog1.sh` в котором буду писать программу с расширением `sh` (shell) с помощью утилиты `touch`, далее делаю его исполняемым с помощью `chmod +x`, открываю файл в любом текстовом редакторе, пишу в нем код, и после того как я написал программу в файле, я могу его запустить `bash` (рис. fig. 4.1).



```
neardeev@neardeev:/home/neardeev$ create mode 100644 project-personal/stage4/report/П04_Ардеев_отчет.docx
neardeev@neardeev:/home/neardeev$ create mode 100644 project-personal/stage4/report/П04_Ардеев_отчет.md
neardeev@neardeev:/home/neardeev$ create mode 100644 project-personal/stage4/report/П04_Ардеев_отчет.pdf
neardeev@neardeev:~/work/study/2023-2024/Операционные системы/os-intro/project-personal/stage4$ git push
Перечисление объектов: 32, готово.
Подсчет объектов: 100% (32/32), готово.
При сжатии изменений используется до 6 потоков
Сжатие объектов: 100% (23/23), готово.
Запись объектов: 100% (23/23), 2.87 МБ | 3.12 МБ/с, готово.
Всего 23 (изменений 8), повторно использовано 0 (изменений 0), повторно использовано пакетов 0
remote: Resolving deltas: 100% (8/8), completed with 4 local objects.
To github.com:neardeev/-study_2023-2024_os-intro.git
030bac1..d4f0c4b master -> master
neardeev@neardeev:~/work/study/2023-2024/Операционные системы/os-intro/project-personal/stage4$ cd
neardeev@neardeev:~$ touch prog1.sh
neardeev@neardeev:~$ chmod +x prog1.sh
neardeev@neardeev:~$ mkdir backup
neardeev@neardeev:~$ bash prog1.sh
neardeev@neardeev:~$ bash prog1.sh
prog1.sh
neardeev@neardeev:~$
```

Рис. 4.1: Выполнение программы 1

скрипт, который при запуске будет делать резервную копию самого себя (то есть файла, в котором содержится его исходный код) в другую директорию `backup` в вашем домашнем каталоге. При этом файл должен архивироваться одним из архиваторов на выбор `zip`, `bzip2` или `tar` (рис. fig. 4.2).

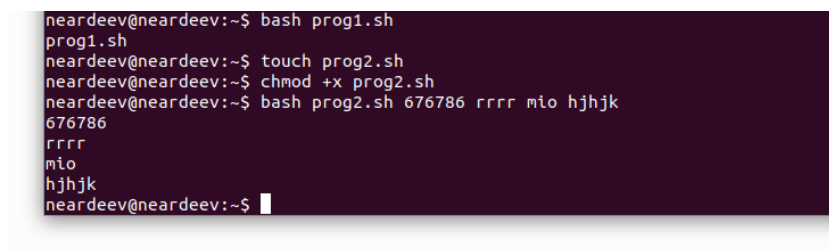


```
report.md x П04_Ардеев_о Вы можете вставить изображение из буфера обмена.
1 #!/bin/bash
2 tar -cvf ~/backup/backup.tar prog1.sh
```

Рис. 4.2: Написанная программа 1

```
#!/bin/bash
tar -cvf ~/backup/backup.tar prog1.sh
```

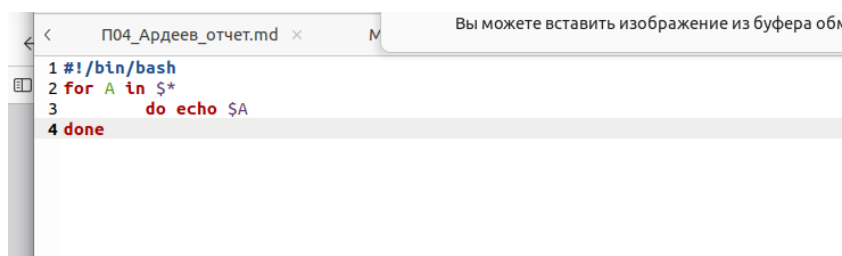
Создаю файл prog2.sh в котором буду пистать программу с расширением sh (shell) с помощью утилиты touch, далее делаю его исполняемым с помощью chmod +x, открываю файл в любом текстовом редакторе, пишу в нем код, и после того как я написал программу в файле, я могу его запустить bash (рис. fig. 4.3).



```
neardeev@neardeev:~$ bash prog1.sh
prog1.sh
neardeev@neardeev:~$ touch prog2.sh
neardeev@neardeev:~$ chmod +x prog2.sh
neardeev@neardeev:~$ bash prog2.sh 676786 rrrr mio hjhjk
676786
rrrr
mio
hjhjk
neardeev@neardeev:~$
```

Рис. 4.3: Выполнение программы 2

Пример командного файла, обрабатывающего любое произвольное число аргументов командной строки, в том числе превышающее десять. Например, скрипт может последовательно распечатывать значения всех переданных аргументов (рис. fig. 4.4).



```
1 #!/bin/bash
2 for A in $*
3     do echo $A
4 done
```

Рис. 4.4: Написання программа 2

```
#!/bin/bash
for A in $*
do echo $A
done
```

Создаю файл prog3.sh в котором буду писать программу с расширением sh (shell) с помощью утилиты touch, далее делаю его исполняемым с помощью chmod +x, открываю файл в любом текстовом редакторе, пишу в нем код, и после того как я написал программу в файле, я могу его запустить bash (рис. fig. 4.5).

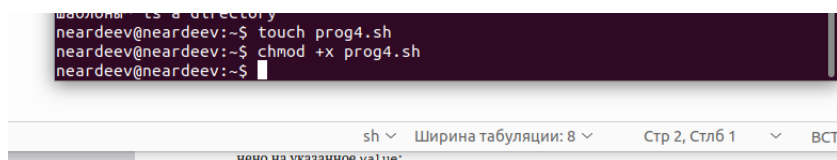


Рис. 4.5: Выполнение программы 3

Командный файл — аналог команды ls (без использования самой этой команды и команды dir). Требуется, чтобы он выдавал информацию о нужном каталоге и выводил информацию о возможностях доступа к файлам этого каталога. (рис. fig. 4.6).

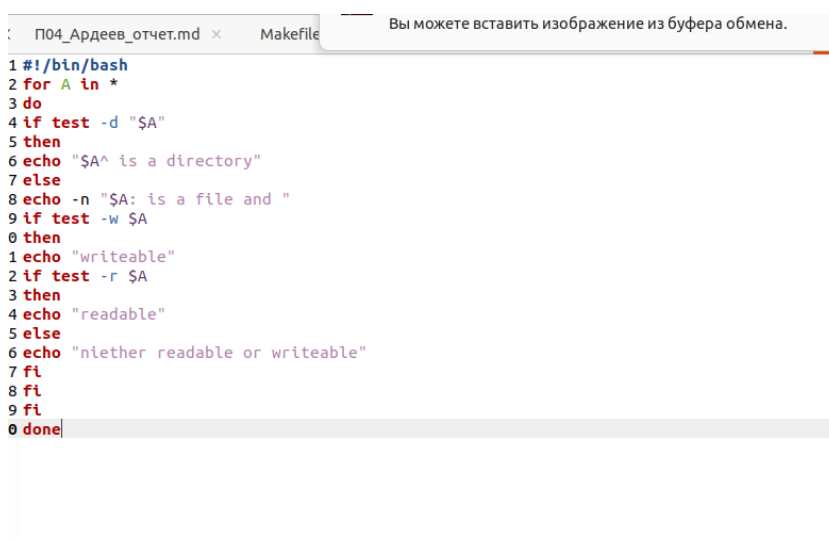


Рис. 4.6: Написанная программа программа 3

```
#!/bin/bash
for A in *
do
    if test -d "$A"
    then
```

```

        echo "$A^ is a directory"
else
    echo -n "$A: is a file and "
    if test -w $A
    then
        echo writeable
        if test -r $A
        then
            echo "readable"
        else
            echo "neither readable or writeable"
        fi
    fi
fi
done

```

Создаю файл prog4.sh в котором буду писать программу с расширением sh (shell) с помощью утилиты touch, далее делаю его исполняемым с помощью chmod +x, открываю файл в любом текстовом редакторе, пишу в нем код (рис. fig. 4.7).

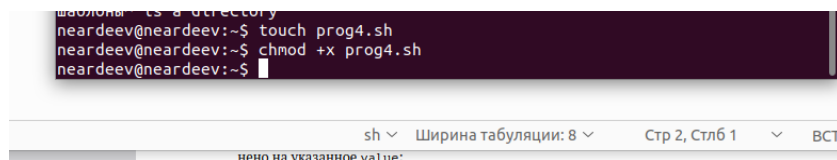


Рис. 4.7: Создание файла с программой 4

Командный файл, который получает в качестве аргумента командной строки формат файла (.txt, .doc, .jpg, .pdf и т.д.) и вычисляет количество таких файлов в указанной директории. Путь к директории также передаётся в виде аргумента командной строки (рис. fig. 4.8).

```
1 #!/bin/bash
2 format=""
3 directory=""
4 echo "Напишите формат файла"
5 read format
6 echo "Напишите директорию"
7 read directory
8 find "${directory}" -name "*,${format}" -type f | wc -l
```

Рис. 4.8: Написанная программа 4

И после того как я написал программу в файле, я могу его запустить bash (рис. fig. 4.9).

```
neardeev@neardeev:~$ touch prog4.sh
neardeev@neardeev:~$ chmod +x prog4.sh
neardeev@neardeev:~$ bash prog4.sh
Напишите формат файла
sh
Напишите директорию
/
wc: неверный ключ - «1»
По команде «wc --help» можно получить дополнительную информацию.
find: '/proc/tty/driver': Отказано в доступе
find: '/proc/1/task/1/fd': Отказано в доступе
find: '/proc/1/task/1/fdinfo': Отказано в доступе
find: '/proc/1/task/1/ns': Отказано в доступе
```

Рис. 4.9: Выполнение программы 4

```
#!/bin/bash
format=""
directory=""
echo "Напишите формат файла"
read format
echo "Напишите директорию"
read directory
find "${directory}" -name "*,${format}" -type f | wc -l
ls
```

## **5 Выводы**

При выполнении данной лабораторной работы я изучил основы программирования в оболочке ОС UNIX/Linux, научился писать небольшие командные файлы.

## 6 Ответы на контрольные вопросы

1. Объясните понятие командной оболочки. Приведите примеры командных оболочек.
2. Чем они отличаются?

Командные процессоры или оболочки - это программы, позволяющие пользователю взаимодействовать с компьютером. Их можно рассматривать как настоящие интерпретируемые языки, которые воспринимают команды пользователя и обрабатывают их. Поэтому командные процессоры также называют интерпретаторами команд. На языках оболочек можно писать программы и выполнять их подобно любым другим программам. UNIX обладает большим количеством оболочек. Наиболее популярными являются следующие четыре оболочки: –оболочка Борна (Bourne) - первоначальная командная оболочка UNIX: базовый, но полный набор функций; –С-оболочка - добавка университета Беркли к коллекции оболочек: она надстраивается над оболочкой Борна, используя С-подобный синтаксис команд, и сохраняет историю выполненных команд; –оболочка Корна - напоминает оболочку С, но операторы управления программой совместимы с операторами оболочки Борна; –BASH - сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек С и Корна (разработка компании Free Software Foundation).

3. Что такое POSIX? POSIX (Portable Operating System Interface for Computer Environments)- интерфейс переносимой операционной системы для компьютерных сред. Представляет собой набор стандартов, подготовленных

институтом инженеров по электронике и радиотехники (IEEE), который определяет различные аспекты построения операционной системы. POSIX включает такие темы, как программный интерфейс, безопасность, работа с сетями и графический интерфейс. POSIX-совместимые оболочки являются будущим поколением оболочек UNIX и других ОС. Windows NT рекламируется как система, удовлетворяющая POSIX-стандартам. POSIX-совместимые оболочки разработаны на базе оболочки Корна; фонд бесплатного программного обеспечения (Free Software Foundation) работает над тем, чтобы и оболочку BASH сделать POSIX-совместимой.

4. Как определяются переменные и массивы в языке программирования bash?
- Командный процессор bash обеспечивает возможность использования переменных типа строка символов. Имена переменных могут быть выбраны пользователем. Пользователь имеет возможность присвоить переменной значение некоторой строки символов. Например, команда `mark=/usr/andy/bin` присваивает значение строки символов `/usr/andy/bin` переменной `mark` типа строка символов. Значение, присвоенное некоторой переменной, может быть впоследствии использовано. Для этого в соответствующем месте командной строки должно быть употреблено имя этой переменной, которому предшествует метасимвол. Например команда `{имя переменной}` например, использование команд `b=/tmp/andy-ls -l myfile > blsls/tmp/andy - ls, ls - l >bls` приведет к подстановке в командную строку значения переменной `bls`. Если переменной `bls` не было предварительно присвоено никакого значения, то ее значением является символ пробел. Оболочка bash позволяет создание массивов. Для создания массива используется команда `set` с флагом `-A`. За флагом следует имя переменной, а затем список значений, разделенных пробелом. Например, `set -A states Delaware Michigan "New Jersey"` Далее можно сделать добавление в массив, например, `states[49]=Alaska`. Индексация массивов начинается с нулевого элемента.



5. Каково назначение операторов `let` и `read`? Команда `let` является показателем того, что последующие аргументы представляют собой выражение, подлежащее вычислению. Простейшее выражение - это единичный терм (*term*), обычно целочисленный. Целые числа можно записывать как последовательность цифр или в любом базовом формате. Этот формат — `radix#number`, где *radix* (основание системы счисления) - любое число не более 26. Для большинства команд основания систем счисления это - 2 (двоичная), 8 (восьмеричная) и 16 (шестнадцатеричная). Простейшими математическими выражениями являются сложение (+), вычитание (-), умножение (\*), целочисленное деление (/) и целочисленный остаток (%). Команда `let` берет два операнда и присваивает их переменной.
6. Какие арифметические операции можно применять в языке программирования `bash`?

Оператор Синтаксис Результат  
`!`*exp* Если *exp* равно 0, возвращает 1; иначе 0  
`!= exp1 != exp2` Если *exp1* не равно *exp2*, возвращает 1; иначе 0  
`% exp1 % exp2` Возвращает остаток от деления *exp1* на *exp2*  
`%= var=%exp` Присваивает остаток от деления *var* на *exp* переменной *var*  
`& exp1 & exp2` Возвращает побитовое AND выражений *exp1* и *exp2*  
`&& exp1 && exp2` Если и *exp1* и *exp2* не равны нулю, возвращает 1; иначе 0  
`&= var &= exp` Присваивает *var* побитовое AND переменных *var* и выражения *exp*  
`* exp1 * exp2` Умножает *exp1* на *exp2*  
`= var = exp` Умножает *exp* на значение *var* и присваивает результат переменной *var*  
`+ exp1 + exp2` Складывает *exp1* и *exp2*  
`+= var += exp` Складывает *exp* со значением *var* и результат присваивает *var*  
`- -exp` Операция отрицания *exp* (называется унарный минус)  
`- exp1 - exp2` Вычитает *exp2* из *exp1*  
`-- var -- exp` Вычитает *exp* из значения *var* и присваивает результат *var*  
`/ exp / exp2` Делит *exp1* на *exp2*  
`/= var /= exp` Делит *var* на *exp* и присваивает результат *var*  
`< exp1 < exp2`

Если *exp1* меньше, чем *exp2*, возвращает 1, иначе возвращает 0  
`<< exp1 << exp2` Сдвигает *exp1* влево на *exp2* бит  
`<= var <= exp` Побитовый сдвиг влево значения *var* на *exp*  
`<= exp1 <= exp2` Если *exp1* меньше, или равно *exp2*, возвращает 1; иначе

возвращает 0 = var = expr Присваивает значение expr переменной var == expr1==expr2 Если expr1 равно expr2. Возвращает 1; иначе возвращает 0 > expr1 > expr2 1 если expr1 больше, чем expr2; иначе 0 >= expr1 >= expr2 1 если expr1 больше, или равно expr2; иначе 0 » expr » expr2 Сдвигает expr1 вправо на expr2 бит »= var »=expr Побитовый сдвиг вправо значения var на expr ^ expr1 ^ expr2 Исключающее OR

выражений expr1 и expr2 ^= var ^= expr Присваивает var побитовое исключающее OR var и expr | expr1 | expr2 Побитовое OR выражений expr1 и expr2 |= var |= expr Присваивает var «исключающее OR» переменной var и выражения expr || expr1 || expr2 1 если или expr1 или expr2 являются ненулевыми значениями; иначе 0 ~ ~expr Побитовое дополнение до expr.

7. Что означает операция (( ))? Условия оболочки bash.

8. Какие стандартные имена переменных Вам известны? Имя переменной (идентификатор) — это строка символов, которая отличает эту переменную от других объектов программы (идентифицирует переменную в программе). При задании имен переменным нужно соблюдать следующие правила: § первым символом имени должна быть буква. Остальные символы — буквы и цифры (прописные и строчные буквы различаются). Можно использовать символ «\_»; § в имени нельзя использовать символ «.»; § число символов в имени не должно превышать 255; § имя переменной не должно совпадать с зарезервированными (служебными) словами языка. Var1, PATH, trash, mon, day, PS1, PS2 Другие стандартные переменные: -HOME — имя домашнего каталога пользователя. Если команда cd вводится без аргументов, то происходит переход в каталог, указанный в этой переменной. -IFS — последовательность символов, являющихся разделителями в командной строке. Это символы пробел, табуляция и перевод строки(new line). -MAIL — командный процессор каждый раз перед выводом на экран промптера проверяет содержимое файла, имя которого указано в этой переменной, и если содержимое этого файла изменилось с момента последнего ввода из него, то перед тем как вывести на терминал промптер, командный процессор

выводит на терминал сообщение You have mail (у Вас есть почта). –TERM — тип используемого терминала. –LOGNAME — содержит регистрационное имя пользователя, которое устанавливается автоматически при входе в систему. В командном процессоре Си имеется еще несколько стандартных переменных. Значение всех переменных можно просмотреть с помощью команды set.

9. Что такое метасимволы? Такие символы, как ' < > \* ? | " & являются метасимволами и имеют для командного процессора специальный смысл.
10. Как экранировать метасимволы? Снятие специального смысла с метасимвола называется экранированием метасимвола. Экранирование может быть осуществлено с помощью предшествующего метасимволу символа, который, в свою очередь, является метасимволом. Для экранирования группы метасимволов, ее нужно заключить в одинарные кавычки. Строка, заключенная в двойные кавычки, экранирует все метасимволы, кроме \$, ' , , ". Например, –echo выведет на экран символ, –echo ab'|'cd выдаст строку ab|cd.
11. Как создавать и запускать командные файлы? Последовательность команд может быть помещена в текстовый файл. Такой файл называется командным. Далее этот файл можно выполнить по команде bash командный\_файл [аргументы] Чтобы не вводить каждый раз последовательности символов bash, необходимо изменить код защиты этого командного файла, обеспечив доступ к этому файлу по выполнению. Это может быть сделано с помощью команды chmod +x имя\_файла Теперь можно вызывать свой командный файл на выполнение просто, вводя его имя с терминала так, как будто он является выполняемой программой. Командный процессор распознает, что в Вашем файле на самом деле хранится не выполняемая программа, а программа, написанная на языке программирования оболочки, и осуществит ее интерпретацию.
12. Как определяются функции в языке программирования bash? Группу ко-

манд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключенных в фигурные скобки. Удалить функцию можно с помощью команды `unset` с флагом `-f`. Команда `typeset` имеет четыре опции для работы с функциями: `-f` — перечисляет определенные на текущий момент функции; `--ft` — при последующем вызове функции иницирует ее трассировку; `--fx` — экспортирует все перечисленные функции в любые дочерние программы оболочек; `--`

`fu` — обозначает указанные функции как автоматически загружаемые. Автоматически загружаемые функции хранятся в командных файлах, а при их вызове оболочка просматривает переменную `FPATH`, отыскивая файл с одноименными именами функций, загружает его и вызывает эти функции.

13. Каким образом можно выяснить, является файл каталогом или обычным файлом? `ls -l` Если есть `d`, то является файл каталогом
14. Каково назначение команд `set`, `typeset` и `unset`? Используется команда `set` с флагом `-A`. За флагом следует имя переменной, а затем список значений, разделенных пробелом. Например, `set -A states Delaware Michigan "New Jersey"` Далее можно сделать добавление в массив, например, `states[49]=Alaska` . Индексация массивов начинается с нулевого элемента. В командном процессоре Си имеется еще несколько стандартных переменных. Значение всех переменных можно просмотреть с помощью команды `set`. Наиболее распространенным является сокращение, избавляющееся от слова `let` в программах оболочек. Если объявить переменные целыми значениями, любое присвоение автоматически трактуется как арифметическое. Используйте `typeset -i` для объявления и присвоения переменной, и при последующем использовании она становится целой. Или можете использовать ключевое слово `integer` (псевдоним для `typeset -i`) и объявлять переменные целыми. Таким образом, выражения типа `x=y+z` воспринимаются как арифметиче-

ские. Группу команд можно объединить в функцию. Для этого существует ключевое слово `function`, после которого следует имя функции и список команд, заключенных в фигурные скобки. Удалить функцию можно с помощью команды `unset` с флагом `-f`. Команда `typeset` имеет четыре опции для работы с функциями: `-f` — перечисляет определенные на текущий момент функции; `-ft` — при последующем вызове функции иницирует ее трассировку; `-fx` — экспортирует все перечисленные функции в любые дочерние программы оболочек; `-fu` — обозначает указанные функции как автоматически загружаемые. Автоматически загружаемые функции хранятся в командных файлах, а при их вызове оболочка просматривает переменную `FPATH`, отыскивая файл с одноименными именами функций, загружает его и вызывает эти функции. В переменные `top` и `day` будут считаны соответствующие значения, введенные с клавиатуры, а переменная `trash` нужна для того, чтобы отобрать всю избыточно введенную информацию и игнорировать ее. Изъять переменную из программы можно с помощью команды `unset`.

15. Как передаются параметры в командные файлы? Символ `$` является метасимволом командного процессора. Он используется, в частности, для ссылки на параметры, точнее, для получения их значений в командном файле. В командный файл можно передать до девяти параметров. При использовании где-либо в командном файле комбинации символов `$i`, где  $0 < i < 10$ , вместо нее будет осуществлена подстановка значения параметра с порядковым номером `i`, т.е. аргумента командного файла с порядковым номером `i`. Использование комбинации символов `$0` приводит к подстановке вместо нее имени данного командного файла. Примере: пусть к командному файлу `where` имеется доступ по выполнению и этот командный файл содержит следующий конвейер: `who | grep $1` Если Вы введете с терминала команду: `where andy`, то в случае, если пользователь, зарегистрированный в ОС UNIX под именем `andy`, в данный момент работает в ОС UNIX, на термини-

нал будет выведена строка, содержащая номер терминала, используемого указанным пользователем. Если же в данный момент этот пользователь не работает в ОС UNIX, то на терминал не будет выведено ничего. Команда `grep` производит контекстный поиск в тексте, поступающем со стандартного ввода, для нахождения в этом тексте строк, содержащих последовательности символов, переданные ей в качестве аргументов, и выводит результаты своей работы на стандартный вывод. В этом примере команда `grep` используется как фильтр, обеспечивающий ввод со стандартного ввода и вывод всех строк, содержащих последовательность символов `andy`, на стандартный вывод. В ходе интерпретации этого файла командным процессором вместо комбинации символов `$1` осуществляется подстановка значения первого и единственного параметра `andy`. Если предположить, что пользователь, зарегистрированный в ОС UNIX под именем `andy`, в данный момент работает в

ОС UNIX, то на терминале Вы увидите примерно следующее: `$ where andy andy`  
`ttyG Jan 14 09:12 $` Определим функцию, которая изменяет каталог и печатает список файлов: `$ function clist { > cd $1 > ls > }`. Теперь при вызове команды `clist` каталог будет изменен каталог и выведено его содержимое.

16. Назовите специальные переменные языка `bash` и их назначение.

- `$*` — отображается вся командная строка или параметры оболочки;
- `$?` — код завершения последней выполненной команды;
- `$$` — уникальный идентификатор процесса, в рамках которого выполняется командный процессор;
- `$!` — номер процесса, в рамках которого выполняется последняя вызванная на выполнение в командном режиме команда;
- `$-` — значение флагов командного процессора;
- `${#}` — возвращает целое число — количество слов, которые были результатом `$`;

`${#name}` — возвращает целое значение длины строки в переменной `name`;  
`${name[n]}` — обращение к `n`-ному элементу массива;  
`${name[*]}` — перечисляет все элементы массива, разделенные пробелом;  
`${name[@]}` — то же самое, но позволяет учитывать символы пробелы в самих переменных;  
`${name:-value}` — если значение переменной `name` не определено, то оно будет заменено на указанное `value`;  
`${name:value}` — проверяется факт существования переменной;  
`${name=value}` — если `name` не определено, то ему присваивается значение `value`;  
`${name?value}` — останавливает выполнение, если имя переменной не определено, и выводит `value`, как сообщение об ошибке;  
`{name=value}` — это выражение работает противоположно `{name=value}`. Если переменная определена, то подставляется `value`;  
`${name#pattern}` — представляет значение переменной `name` с удаленным самым коротким левым образцом (`pattern`);  
`${#name[*]}` и `${#name[@]}` — эти выражения возвращают количество элементов в массиве `name`.  
`$#` вместо нее будет осуществлена подстановка числа параметров, указанных в командной строке при вызове данного командного файла на выполнение.