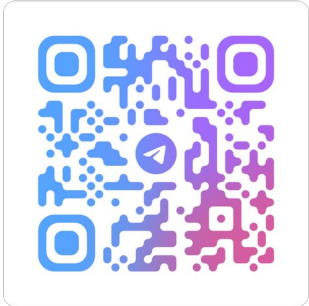
САМОУЧИТЕЛЬ BASH

Версия от 2025-06-26

Связаться с автором по делу: https://t.me/+RBlffCL4R243MjNi



ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ

БЛАГОДАРНОСТИ

Особую благодарность выражаю **Роману Шубину**, автору телеграм канала

Bash Days | Linux | DevOps (https://t.me/bashdays)

И

Дмитрию Малинину, автору телеграм канала

Useful Tools | Linux | GitOps | DevOps (https://t.me/gitgate)

Также благодарю телеграм-слушателей курса за помощь в создании и тестировании практической части. (в алфавитном порядке):

_@_, Albert L., Aleksandr, Alex, Alexander, Alexander Dankov, artem, Artem, Artyom Kucherov, Axet Nord, bondage,
Deleted Account, Dmitriy Filatov, Dmitry Balashov, E A,
Egor Pravoy, Evgeniy Kornilov, FeeLinS, Grewi Saratov, Iam, Igor, @IgorKovtunov, Kirill Bykovskiy, Konstantin, Konstantin Redkin,
Loki, Max Zotov (Sf1nk5), Michael Ivanov, Muhammed Hilal,
naakinn .naa, Nikolay, PFox, RedTimeAlex, Rlol (Linux enjoyer),
Roman Britva, Shaon, Shutov_Nikolay, Stanislav Balkusov,
Stanislav Starodub, Tatler, Vitali K, Vladimir Kazakov,
Vladimir Klimov, Volodya, Александр, Александр_w, Александр Малов,
Александр П, Алексей Ушаков, Андрей, Артём, Валентин,
Виктор Вангели, Виталий Р, Владимир, Владимир Храмов,
Геннадий Шевченко, Георгий Куликов, Денис Березин, Дмитрий, Евгений Веревкин, Евгений Политико, Иван Т., Игорь, Илья,
Мальцев Денис, Мистер Выдрингтон, Михаил Байбуз, Николай Атрашков,
Николай М., Николай Орда, Николай Попов, Олег Николаенко, Сергей,
Старый□□, Шурик, юн, Юрий, □□А.

ВОПРОСЫ И ОТВЕТЫ

Кому пригодится этот курс bash?

Если Вы школьник-энтузиаст, студент, начинающий admin или dev-ops. Добро пожаловать.

С какого возраста можно изучать bash?

Я бы сказал, что лет с 12. Раньше начинать проблематично из-за отсутствия минимальных знаний по информатике, математике, английскому. Но если Вам 7 и вы очень этого хотите - я не против. Пробуйте разобраться. Откуда я знаю, может Вы гений.

Кому не пригодится этот курс?

- Если Вы в ближайший год не собираетесь использовать bash - проходить курс практически бесполезно. Через год останется малюсенькая часть знаний. Практически все придется изучать заново, потому что bash - оболочка/язык с довольно высоким уровнем вхождения.

- Если Вы знаете bash и решили потешить свое самолюбие - не стоит тратить времени. Используйте свои знания по-назначению - зарабатывайте деньги.

У меня пока нет собственного компьютера с linux, что делать?

- На Windows 10 и выше можно установить систему WSL 2 и выбрать дистрибутив (рекомендую Debian или Ubuntu, чем новее, тем лучше).

- На старых компьютерах с Windows 7,8 проще всего установить VitualBox , и в нем настроить виртуальную машину Debian или Ubuntu. Вполне хватит 1Гб памяти и 10-30 Гб диска. Графическую оболочку ставить не нужно. Она Вам не пригодится. Работать будем только в терминале.

- На смартфоне/планшете android можно установить приложение termux (доступно в Google Play) - это эмулятор среды linux. Изучать можно, но рекомендую использовать первые два способа.

У меня не получается настроить WSL/VirtualBox/termux Поможете настроить?

Нет, не помогу. Это как-раз порог вхождения. (Степень вашего желания пройти курс) Просите друзей, знакомых, изучайте тему в поисковиках...

Сколько нужно времени на изучение курса?

Если изучать две темы (главы) в неделю - потребуется около трех месяцев. Это реально. Вначале быстрее, потому что задания проще.

А если я буду использовать ChatGPT для ответов?

Да без проблем - хотите обучить ИИ bash'у - дерзайте. Хотите научится сами - придется заставлять СЕБЯ. От себя замечу - на момент 2025 года ИИ очень часто выдает ответы с очень грубыми ошибками. И Вы, не имея опыта не сможете их найти.

На что нужно обратить особое внимание при написании скриптов?

Задачу, практически всегда, можно решить различными способами, и используя различные средства. Для решения выбирайте те средства, с которыми лучше знакомы. При решении задач следует придерживаться следующих принципов в порядке убывания значимости:

1. Понятность и читаемость решения задачи. (комментарии + понятные переменные).

2. Оптимальность решения задачи по скорости выполнения.

Особое внимание уделите первому пункту. Чем понятнее программа, тем быстрее Вы ее отладите. Обязательно комментируйте код, потому что иногда приходится адаптировать код при изменении условий через несколько лет.

Если код на bash выполняется слишком медленно, возможно задачу нужно решать другими средствами (AWK, Perl, Python, C, GO или RUST).

Я уже немного знаю bash можно ли в решении задач использовать то, что я изучил самостоятельно?

Курс разработан таким образом, что задания можно решить только изученными средствами. Не нужно спешить.

В теории иногда встречается фраза "в старых версиях bash и в других оболочках это может не работать." Что это значит?

Курс основан на версии bash 5.0 и выше. Узнать Вашу версию можно с помощью команды **echo $BASH_VERSION**. Другие оболочки это **sh, dash, tsch, csh, fish** и многие другие. Хотя базовые операторы в разных оболочках похожи - работать они могут по-разному. А некоторые операторы могут вообще отсутствовать. Проверить оболочку можно командой **echo $SHELL**

Можно ли не выполнять практические задания, там же и так все понятно?

В любом случае Вы потратите время на собственную практику. Либо сейчас, во время обучения, либо во время реальной работы, при отладке скриптов, когда результат нужен срочно, времени нет, а скрипт не хочет работать. Да и при выполнении заданий знания закрепляются лучше. И если Вам все понятно - значит задания выполните очень быстро, и все заработает сразу.

Я не знаю, как начать выполнять задание N...

1. Еще раз перечитайте теорию. Помогает.

2. Если вариантов решения нет вообще - мельком взгляните на ответы, и постарайтесь самостоятельно выполнить задание.

3. Разберите построчно чужие ответы, но это не лучший вариант.

В задании требуется написать решение на bash, но я знаю, что проще использовать специализированную программу (sed, awk, bc, sort, grep...)

Да, Вы правы. Когда будете писать скрипты для реальных задач - так и сделаете, но курс bash предназначен для обучения. При выполнении задач курса придерживайтесь условий заданий. Возможно однажды Вам придется решить задачу, а программы, к которой Вы привыкли не окажется на сервере по причинам безопасности (закрытый контур) или скудости ресурсов (контейнер или роутер). Чем меньше внешних средств требуется для решения задачи - тем выше квалификация программиста.

Я заглянул в конец курса. Там такие большие программы. Bash очень сложный?

Если человек умеет читать, разницы нет одно предложение, абзац или книга. Большинство этого кода написано людьми, которые сами учили bash по этому курсу. Их имена приведены в разделе благодарностей.

Будет нелегко. Но если смогли заставить себя продолжить, даже когда хотелось все бросить, если Вы смогли освоить bash, - значит у Вас есть сила воли, значит сможете освоить и другие направления IT. Соберите волю в кулак и вперед!

ЗАМЕЧАНИЯ И ПРИНЯТЫЕ ОБОЗНАЧЕНИЯ:

Операторы bash и внешние программы, которые можно выполнить в оболочке будут выделены, например **bash** , ***ls*** , ***printf*** .

По правилам русского языка знаки препинания должны стоять сразу после слова, но при программировании знаки пунктуации являются частью языка программирования, поэтому знаки препинания после операторов будут указаны через пробел, или не будет указаны вообще. Например: **bash** , **ls** , **printf** .

Определения, комментарии, замечания будут выглядеть так:

 # Это комментарий, определение или замечание.

 # Это очень важный комментарий, определение или замечание.

Код скрипта может быть указан так: **echo "Hello world!"**

или так:

```
#!/bin/bash
echo "Hello world!"
```

Иногда, в конце скрипта или после конкретного оператора, результат вывода на экран может быть приведен в виде комментария, при этом если вывод на экран производится с разделителем 'перевод строки' он может быть приведен в виде комментария с разделителем 'пробел' для компактности:

```
#!/bin/bash
echo 123 #123
for i in {0..9};do
  echo $i
done
#0 1 2 3 4 5 6 7 8 9
```

Оглавление

1Редакторы, первый скрипт.

При изучении bash Вам потребуется набирать тексты скриптов (программ bash) Для этого Вам нужно изучить какой-либо консольный (терминальный) текстовый редактор.

Из простых могу посоветовать **nano** (уже установлен практически в каждой системе). Для начинающих рекомендую пользоваться именно им.

Второй по простоте и функциональности - **mcedit**. Он входит в комплект поставки файлового менеджера mc. Его нужно устанавливать дополнительно.

Третий редактор **vim** (**vi**) Это очень функциональный редактор для профессионалов. **vi** обычно уже установлен в системе, **vim** (расширенная версия **vi**) нужно устанавливать дополнительно.

vim очень сложен в освоении, но потом позволяет быстро работать. Сравнить можно с автомобилем. Сначала нужно купить, научиться, получить права, но потом передвижение становится очень быстрым.

Если Вы не профессиональный программист - пользуйтесь **nano/mcedit**. Если вы ITшник - найдите время изучить **vim**. Потом окупится сторицей. Если решились - начните с **vimtutor** , этого этого будет достаточно для прохождения курса.

 # Хотя Вы можете копировать и вставлять тексты скриптов - рекомендую набирать их руками. Двигательную память никто не отменял. Запомните лучше и на более долгий срок.

Итак. Набираем первый скрипт. Открываем терминал, набираем
nano firstscr.sh <Enter>

```
#!/bin/bash
clear
echo Hello world!
```

Сохраняем (CTRL-x)

Запустить программу можно командой **bash firstscr.sh**
после запуска увидим на экране результат выполнения:

1. Экран очистится. (команда **clear**)
2. На экран будет выведена строка **Hello world!**

Что происходит при выполнении команды **bash firstscr.sh**

1. Запускается **bash**
2. Построчно выполняет скрипт **firstscr.sh**

Данный способ запуска называется **явным запуском.**

Потому что мы в командной строке конкретно указали программу, исполняющую скрипт - bash .

Если при запуске возникает ошибка - значит либо Вы что-то не правильно написали, либо у Вас в системе отсутствует bash. Устраните ошибку и выполните запуск снова.

Поздравляю. Вы - начинающий программист bash.

Симолы "**#!**" называюся шебанг. И они указывают, какую программу использовать для обработки данного скрипта. (в данном случае **/bin/bash**) **при неявном запуске**.

Иногда можно увидеть в скриптах **#!/usr/bin/env bash**
Запуск через env позволяет более гибко настроить среду исполнения, но в большинстве случаев это не требуется. Обычно я использую **#!/bin/bash** .

Часть строки начинающиеся с символа "**#**" это комментарии и они не выполняются.

Обычно в скриптах один оператор/команда в строке, но иногда несколько команд можно разместить в одной строке разделив их символом ";"

Злоупотреблять объединением не стоит, но когда это логично - то можно.

Задание 1.

1. Набрать и выполнить скрипт явным запуском.

```
#!/bin/bash
clear
# comment
echo Hello world! #also comment
```

2. Скрипт п1. Закомментировать команду clear. Выполнить скрипт.

3. Скрипт п1. Поменять местами 2,3 строки. Выполнить скрипт.

4. Скрипт п1. Заменить #!/bin/bash на #!/bin/awk . Выполнить скрипт.

5. Сделать вывод о выполнении строки с шебанг при явном запуске скрипта.

6. Скрипт п1. Объединить 2,3 строки в одну . Проверить правильность выполнения.

7. Ответить, что делает команда echo.

8. Найти комментарии.

2Структура каталогов

Я не знаю Ваш уровень, поэтому, поэтому придется рассмотреть структуру каталогов и файлов. Если Вам все это знакомо - значит с выполнением заданий не будет проблем.

Игнорировать задания не рекомендую. Иногда простые задания сложно выполнить.

Файловая система linux начинается с корневого каталога, который обозначается /

Все остальные каталоги присоединены к этому, как ветки дерева к стволу.

В корневом каталоге содержатся каталоги, каждый из которых выполняем определенную функцию. Рассмотрим некоторые из них.:

/bin – каталог для служебных исполняемых (бинарных или двоичных) файлов

/dev – каталог для устройств.

/etc – здесь хранятся основные файлы настроек программного обеспечения.

/home - каталог для хранения файлов и настроек пользователей системы, кроме администратора

/media – в этот каталог монтируются (присоединяются) файловые системы всяких флэшек, фотоаппатратов, плееров и CD/DVD. Обычно сейчас это происходит автоматически.

/mnt – каталог для монтирования других файловых систем

/opt – для дополнительных программ пользователя (1с и телеграм ставятся именно туда)

/var – каталог для хранения изменяемых данных. Там находятся программы, базы данных, сайты и логи и архивы.

/tmp – для временных файлов. Каталог очищается при каждой перезагрузке. Не стоит там хранить важные файлы.

/usr – в этом каталоге основные программы, для работы пользователя в системе.

Каталоги размещаются в файловой системе. И что самое главное, каталоги могут быть размещены на разных файловых системах и даже разных физических дисках. А иногда и на разных компьютерах. Два соседних каталога могут физически находится на разных материках!!!

Есть еще другие каталоги, но на начальном уровне мы их не будем рассматривать. На ближайшее время Ваши «родные каталоги» /home и /tmp.

Крайне не рекомендую менять что-либо в каталогах, о которых ничего не знаешь Имея права root можно обрушить систему, просто удалив «ненужный файл или каталог.»

В каталогах могут быть подкаталоги. Например /home/username . Каталоги могут содержать другие подкаталоги и файлы. Файлы - могут содержать информацию.

Адрес файла может быть абсолютным (тогда он содержит весь путь от корня и начинается с символа / например /bin/bash) или относительным. тогда он читается от текущего каталога. например data/log.txt говорит о том, что файл log.txt находится в подкаталоге data текущего каталога.

Отсутствие любых символов / в имени файла или каталога обозначает, что они находятся в текущем каталоге. Например data , my_file.txt .

Кроме этого, иногда используют специальные символы для работы с относительными каталогами.

./ - текущий каталог. Например: ./scr.sh

../ - вышестоящий каталог

 ../data - обозначает , что каталог data находится на одном уровне с текущим (../.. выше текущего на 2 уровня)

~/ - домашний каталог текущего пользователя.
Его местоположение прописано в файле /etc/passwd для каждого пользователя в системе. Для "обычных" пользователей /home/<user> - где <user> - имя текущего пользователя в системе.
Для root /root. Для пользователей, от имени которых запускаются сервисы (демоны, службы), может быть указан любой каталог.

С обозначением ~/ нужно быть аккуратнее. Оно может не корректно работать, если скрипты будут запускаться автоматически. (cron, systemd, init.d)

Имена могут содержать расширение (тип файла) например file.txt, arch.tar.gz, book.fb2

И теперь самое главное:

1. В отличие от windows имена файлов и каталогов регистрозависимые. Т.е LOG Log log lOg - разные имена.

2. Linux допускает имена с пробелами и русские, но использование таких имен считается дурным тоном в IT.

 # Создавая скрипты Вы всегда должны помнить, что пользователь может задать имя файла/каталога с пробелом или русскими буквами.

3. Расширения файлов нужны для облегчения понимания о содержимом файла. Но не гарантируют, что файл имеет именно указанное содержимое.

4. Имена файлов и каталогов могут содержать различные спецсимволы, типа !#@, но на начальном этапе рекомендую использовать только латинские буквы (желательно маленькие), цифры, знак подчеркивания и точку.

5. Имена файлов(каталогов) не должны, без необходимости, начинаться с точки. Так обозначаются "скрытые файлы".

6. Существуют символы, недопустимые в именах файлов(каталогов). Причем для разных файловых систем запрещенные символы различны.

Задание 2

Условие: Текущий каталог /home/tagd

1. Привести пример абсолютного имени файла myfile.txt , находящегося в данном каталоге.

2. Привести три различных примера относительного пути к указанному файлу.

3. Привести относительный путь к каталогу /var/log .

3Работа с каталогами.

Для изменения текущего каталога есть команда **cd** (change dir)

cd /home/username - перейти в указанный каталог (указан абсолютный путь).

cd work - перейти в каталог **work**, находящийся в текущем каталоге (относительный путь)

cd .. - переход в вышестоящий каталог.

cd ../../work -сначала подняться на два уровня, затем перейти в каталог **work**.

cd - (минус) переход в предыдущий каталог.

Если выполнить **cd** без параметров - попадете в домашний каталог. Очень удобно.

А если Вы вдруг "заблудились, гуляя по каталогам" Есть специальная команда:
pwd (print work dir), чтобы узнать, в каком каталоге Вы сейчас находитесь.

Если вы запросите переход в каталог, которого не существует - получите сообщение об ошибке.

Команда для создания каталогов.
mkdir (make directory) например:

mkdir work - создаст каталог work в текущем подкаталоге.

Если Вы укажите абсолютный путь, то команда сможет создать только один каталог, являющийся последним элементом указанного пути. т.е

mkdir /tmp/work/first не создаст каталог **first**, если нет каталога **work** в папке **/tmp .**

Для того, чтобы предыдущая команда не выдавала ошибку, есть специальный ключ. **-p**

mkdir -p /tmp/work/first в этом случае будут созданы все родительские (parent) каталоги, которые будут необходимы.

Для удаления ПУСТОГО каталога есть команда **rmdir** .

Задание 3

1. Перейти в каталог /tmp

2. Перейти в домашний каталог.

3. Убедиться, что вы находитесь именно в домашнем каталоге.

4. Не меняя текущего каталога создать каталог work в /tmp

5. Создать каталог work текущем каталоге и перейти в него.

6. Сразу перейти в /tmp/work . Убедится, что перешли правильно.

7. Вернуться в предыдущий каталог одной командой.

8. Попрактиковаться в переходах между двумя каталогами work различными способами.

9. Создать одной командой каталог /tmp/test/day/primary.

10. удалить каталог test из каталога /tmp .

4Работа с файлами

Файлы похожи на каталоги, с тем исключением, что каталоги содержат файлы и подкаталоги, а файлы содержат информацию.

Пути к файлом ничем не отличаются от пути к каталогу. Например:
/var/log/auth.log
/home/username/123

Создать файл можно несколькими способами:

1. **> filename**
2. **:> filename** (команда "дятел")
3. **:>> filename** (команда "долбодятел")
4. **touch filename**
5. **nano filename** (набрать текст и сохранить)

nano здесь для примера. Может быть любой редактор.

В чем разница между этими командами?

Команда №1 будет работать не на всех системах, поэтому пользоваться ей не рекомендую.

Команды ведут себя по-разному, в том случае, если **filename** уже существует и содержит информацию.

Команда 2(1) удалит всю информацию, если файл уже существует.

3-4 сохранят информацию.

5 команда не создаст файл, если его специально не сохранить.

При создании bash - скриптов (программ) рекомендую пользоваться **:>>** , touch , или создавать файл с помощью редактора.

Если при переборе истории команд, Вы, нечаянно, повторно выполните команду создания файла, при использовании команды **>** или **:>** можно стереть работу нескольких часов. Проверено на личном опыте.

Для копирования используется программа cp (сокращение от copy) для перемещения/переименования команда mv (сокращение от move)

cp SOURCE DEST

Программа скопирует **SOURCE** в **DEST**
Копирование со сменой имени и/или пути.

Если **DEST** - это каталог, то, копия файла с тем же именем будет размещена в каталоге **DEST**.

Если указать новое имя файла - файл будет скопирован с новым именем.

Например:
cp config config.bak
cp config config.1
cp config /var/backups/prog/config.20250114

Именно этими командами нужно начинать правку любых конфигурационных файлов.

mv SOURCE DEST
Аналогична команде **cp** , за исключением того, что исходный файл удаляется.

Для удаления файлов есть команда **rm**

rm /tmp/file1.txt
rm /home/username/123.txt

ПРЕДУПРЕЖДЕНИЕ. Эта команда может быть очень опасна при использовании с ключами.

Некоторые предлагают выполнить начинающим пользователям команду типа. **НЕ ВЫПОЛНЯТЬ!!**

rm -fr /*
или
rm -rf /*

При выполнении этой команды с правами администратора все данные на вашем компьютере будут удалены. **НЕ ВЫПОЛНЯТЬ.**

Сейчас в команду встроили некоторую защиту, но все-равно пользоваться ей нужно аккуратно.

Как же отличить по имени файл от каталога? Ответ прост НИКАК!

В nix-системах все есть файл. И каталог и устройство и файл и линк.

Для просмотра списка файлов и каталогов есть команда ls. Поведением команды можно управлять используя ключи

ls - просто краткий список файлов и подкаталогов (иногда цветной)
ls -l /tmp длинный список темпового каталога.
ls -la ~/ длинный список, включая скрытые файлы и каталоги (начинаются с точки)
ls /dir/filename /dir/filename1 /dir/filename2 вывод информации о нескольких файлах. Файлы или каталоги перечисляются через пробел.

*# ПРИМЕЧАНИЕ: если в качестве аргумента указан каталог, то команда **ls** отобразит содержимое этого каталога. Чтобы узнать свойства самого каталога нужно добавить ключ **-d**.*
ls -ld ~/

Скрытые файлы и каталоги обычно хранят информацию о настройках и при обычной работе они только мешают, поэтому их и скрывают из списка. И ДА, ЕСЛИ ОНИ МЕШАЮТ ПРИ РАБОТЕ, ЭТО НЕ ЗНАЧИТ, ЧТО ИХ МОЖНО УДАЛИТЬ.

Вот и подошли к ответу на вопрос, как отличить файл от каталога.
если использовать ls -l то в первой колонке первый символ обозначает:

- обычный файл
b блочный (block)
c символьный (character)
d -каталог (directory)
l - символическая ссылка (link)
p - канал (pipe)
s - сокет (socket)

Задание 4

Условие: Выполнять задание, находясь в домашнем каталоге.

1. Создать файл test.txt в каталоге /tmp командой **:>** ,просмотреть содержимое /tmp и убедиться, что файл существует.

2. Удалить только что созданный файл командой **rm** , убедится, что файл был удален.

3. повторяя п1-2 испробовать все остальные способы создания файлов. (:>> , touch , nano)

4. Сделать вывод об "удобстве" создания файлов разными способами.

5. Создать файл test.txt в каталоге /tmp с помощью редактора. Добавить любое содержимое.

6. Повторяя п5, при необходимости, испробовать все остальные способы повторного создания файлов. (:>> , touch , :>)

7. Подтвердить или опровергнуть выводы о сохранении содержимого файла. (содержимое проверять при открытии текстовым редактором)

8. Создать файл hidden.cfg в каталоге /tmp

9. Переименовать hidden.cfg в .hidden.cfg

10. Проверить наличие .hidden.cfg

11. Скопировать файл .hidden.cfg в .hidden.cfg.1

12. Удалить файлы из п.11 одной командой

13. Проверить домашний каталог на наличие файлов test.txt и .hidden.cfg . Удалить, при наличии.

Лирическое отступление. history

При выполнении последнего задания приходилось много однотипных команд набирать "руками".

Эту задачу можно облегчить используя команду **history**.
Обычно все действия пользователя в терминале записываются в специальный файл-журнал. И потом могут быть повторно использованы или прочитаны.

Перемещение по журналу производится стрелками Вверх(Up)/Вниз(Down)
Просмотр всего журнала - команда **history.** Текущую команду можно отредактировать и снова использовать.

Кроме этого можно использовать несколько "горячих клавиш" для упрощения набора:

Ctrl-r позволит осуществлять быстрый поиск по истории при наборе текста.
Alt-. - добавление к набранной команде последнего параметра из предыдущей команды.
Ctrl-l (буква L) - быстро очистит экран терминала и сохранит Ваши секреты (но помните, что набранные команды сохраняются в журнале history)

Надеюсь теперь вы будете тратить меньше времени на работу в консоли.

5Маски, подстановки, расширения.

Маски, подстановки, расширения используются для работы с большим числом файлов (как-бы группировка).

Маска заменяет один или несколько символов в имени файла/каталога

Например:

* - заменяет любое количество (в том числе ноль) любых символов

? - заменяет один любой символ. (??? - три любых)

Подстановки.

Подстановка - та же маска, только более конкретная:

[123] - заменяет один любой символ из списка (в данном случае 1,2 или 3)

[a-g] - заменяет один любой символ из указанного диапазона (от a до g)

[^123] - Отрицание. Заменяет один любой символ кроме тех, что в списке.

[^a-g] - Заменяет один любой символ кроме тех, что в диапазоне от a до g

Еще существуют **расширенные подстановки**. Для того, чтобы они работали нужно, чтобы была включена опция extglob

shopt -s extglob

*(patt1|patt2|patt3) - Соответствует любому числу вхождений указанного шаблона(ов). Т.е может быть или patt1 или patt2 или patt3.

?(pattern-list) - Соответствует 0 или 1 вхождению любого указанного шаблона.

+(pattern-list) - Соответствует не менее, чем одному вхождению указанных шаблонов.

@(pattern-list) - Соответствует только одному вхождению указанных шаблонов.

!(pattern-list) - Соответствует всему, за исключением любого из указанных шаблонов.

Они нужны в особо тяжелых случаях с файлами и строковыми переменными.

Кроме этого существуют **расширения**.

{abc,qwe,err} -список значений

{1..3} - диапазон значений

{01..120..10} -диапазон значений с шагом 10, все значения одинаковой длины. (001 011 ... 111)

{A..Z..3} -диапазон значений с шагом 3

Отличие маски(подстановки) от расширения в том, что маска говорит, что файл МОЖЕТ иметь такой формат. Расширение указывает, что файл (в общем случае строка) ДОЛЖЕН иметь такой формат. Мало того, при расширении создается полный список строк с диапазоном значений, указанном в фигурных скобках.

Пример:

Допустим в текущем каталоге есть два файла: **file1.txt, file2.txt**

ls file?.txt # результат **file1.txt file2.txt**

ls file[1-3].txt # результат **file1.txt file2.txt**

ls file{1..3}.txt # **file1.txt file2.txt** и ошибка нет файла '**file3.txt**'

Задание 5

Условие: Задание выполнять, находясь в домашнем каталоге.

1. В каталоге /tmp создать подкаталоги bash days chat с помощью расширения.

2. В каждом созданном каталоге создать файлы (различными способами, но с помощью расширений.) file01.txt2txt file02.txt2txt file04.txt2txt . Сделать вывод.

3. Проверить получение списка созданных файлов с помощью различных масок. (Привести 3 примера)

4. Удалить созданные файлы с помощью масок на файлы. В команде rm ключи не задействовать. Проверить, что все файлы удалены.

5. Привести аналог команды touch /tmp/file{1..3}.txt2txt одной строкой, без использования расширений.

6Права доступа

Это очень сложная тема. Для того, чтобы ее освоить нужно три умения:

1. Считать до семи.

2. Уметь складывать и вычитать цифры 4 2 1

3. Запомнить три буквы rwx и еще четыре буквы ugo=a

В linux и других nix-системах есть три основных права:

- Read (чтение) обозначается буквой r или цифрой 4
- Write (запись) обозначается буквой w или цифрой 2
- eXecution (выполнение) обозначается буквой x или цифрой 1

 # Почему право выполнения Обозначается x, а не e? Потому что в английском буква x произносится как ЭКС.

Все права определяются суммой этих прав.

Например: rw=6 (4+2)

rx=5 (4+1)

отсутствие всех прав =0.

Что дают эти права?

Право r - разрешает только прочитать содержимое. (Для каталога читать список файлов и их свойств (права, размер...))

Право w - разрешает записать (создать, изменить, удалить). (Для каталога удалять и добавлять/создавать файлы).

Право x - уникальное. Потому что его использование отличается для файлов и каталогов.

Для файлов x - возможность неявного исполнения (в windows аналог exe, cmd или bat)

Для каталогов - возможность сделать каталог активным (войти в него).

 # Обычно, для каталогов права rx используются вместе.

Для того, чтобы сделать назначение прав более гибким ввели еще четыре понятия:

1. Пользователь (User) обозначается буквой u или первая цифра (как-бы сотни). Пользователя иногда называют владельцем.

2. Группа (Group) обозначается буквой g или вторая цифра. (как-бы десятки)

3. Остальные (Other) обозначается буквой o или третья цифра. (единицы)

4. Все (All) обозначается буквой a и означает, что права пользователя группы и остальных будут одинаковыми. a = ugo.

 # Оговорки (как-бы сотни, как-бы десятки сделаны из-за того, что формально, права являются числами в восьмеричной системе.

Как правило, права либо одинаковые для пользователя, группы, всех, либо уменьшаются от пользователя к группе и далее для всех.

Например 750 обозначает:

Для пользователя права 7 (4+2+1 или rwx)

Для группы 5 (4+1 или rx иногда пишут r-x)

для всех остальных 0 (---)

Вы уже видели права доступа в выводе команды ls -l. Они стоят в первой колонке

В третьей колонке стоит пользователь (владелец файла/каталога)

В четвертой колонке - группа, владеющая файлом/каталогом)

-rwxr-xr-x 1 tagd tagd 569 июн 23 12:35 1.sh

Для изменения прав используется команда **chmod** (change mode).

С помощью этой команды вы можете изменить права файла, **владельцем которого являетесь.**

Менять права любого (даже чужого) файла может только администратор (root)

например:

```
:>>~/test.txt
chmod 750 ~/test.txt
#или
mkdir -p ~/work/tagd
chmod 775 ~/work/tagd
```

Кроме числового формата установки прав - есть текстовый. Например:

```
chmod u=rwx,g=rx,o=  ~/test.txt
chmod u=rwx,g=rwx,o=rx  ~/work/tagd
```

Кроме установки есть формат изменения прав.

```
chmod u+w,g-x,o+r ~/test.txt
```

если команда выполнится успешно, то для файла ~/test.txt владелец (user) получит право на запись, группа будет лишена права исполнения, всем будет разрешено читать файл.

```
chmod a+rx ~/work/tagd
```

если команда выполнится успешно, то для каталога будут добавлены права чтения списка файлов и их свойств для всех.

Категорию для всех (a) , иногда, можно опустить. Но работает не на всех системах. В скриптах использовать не желательно. Например:

```
chmod +rx ~/work/tagd
```

Текстовый вариант установки прав гибче и проще, потому что в числовом виде права можно только установить. А в текстовом еще и изменить:

1 Не важно, какие права до этого файлы или каталоги имели. Им будут добавлены или убраны указанные права.

2. Текстовый вариант позволяет задать/изменить права только одной категории прав, не затрагивая других.

Например:

```
chmod u+x ~/test.txt # добавит выполнение только владельцу.
```

Сделать такое с числовым форматом тоже можно, но для этого нужно узнать, какие права были сначала. Ведь права для группы и остальных должны остаться прежними.

Например допустим:

```
ls -l ~/test.txt # выдает результат rw-rw--r--
```

Мысленно преобразуем в число 664. Мысленно добавляем право исполнение для владельца 764, и выполняем команду

```
chmod 764 ~/test.txt
```

А если начальные права были 444, то потребуется выполнить команду

```
chmod 544 ~/test.txt
```

Т.е цифры устанавливаемых прав в числовом виде зависят от изначально заданных прав. Надеюсь ход мысли ясен.

И осталось рассмотреть одно уникальное право.

X оно задает возможность задать право Сделать каталог активным для каталогов, но игнорируется для файлов.

Обычно применяется при рекурсивном (для самого каталога и всех вложенных файлов и каталогов).Например:

```
chmod -R +x ~/work/tagd
```

не только задаст право x для каталога tagd и вложенных каталогов, но и сделает все файлы внутри исполняемыми.

Чтобы ограничить задание прав только каталогами используют X. Это единственна большая буква при задании прав.

```
chmod -R a+X ~/work/tagd
# или
chmod -R a+rX ~/work/tagd
```

ключ -R вначале отвечает за рекурсию, и к правам отношения не имеет.

 # ВНИМАНИЕ Рекурсивное задание прав, без должного внимания и особой аккуратности может сделать Вашу систему неработоспособной. Я лично однажды убил систему из-за пренебрежения этими правилами.

Кроме основных прав есть еще специальные (setuid, setgid, sticky bit). В пределах данного курса вы должны знать то, что они существуют, но вам пока не нужны.

Для изменения владельца и группы файла служит команда **chown** (change owner)

chown [OPTION]... [OWNER][:[GROUP]] FILE...

Ее использование не вызывает сложностей.

Обратите внимание:

Если на каталог заданы права wx, но нет права r. Вы можете изменять файл, но только нужно точно знать его имя, потому что посмотреть список файлов каталога вы не сможете. (Аналог - тёмная комната)

Если у каталога установлены права w, то Вы можете удалить файл внутри этого каталога, даже если файл защищен от записи (нет права w на файл).

Если на файл стоит право x, но нет права r, то выполнить его вы тоже не сможете, потому что нельзя прочитать содержимое.

Задание 6

1. Преобразовать в текстовый вид следующие права для файлов 777, 666, 550, 444, 330

2. Расписать те же права рекурсивно для каталогов, но не для файлов.

3. Выполнить команду ls -la на любых каталогах и расшифровать права, двух файлов и двух каталогов.

4. Написать 6 примеров команд для установки различный прав в числовом виде

5. Написать еще 6 примеров для установки и изменения различный прав в текстовом виде, выполнить, проверяя результат командой ls -l

6. Пояснить, к какой категории (ugo) относятся права выполнения и записи в rxrwr и rrwxw

7. Проверить возможность редактирования и сохранения файла, владельцем которого Вы являетесь, с правами rrrw.

8. Определить, как будут выставлены права в случае конфликта (u-x, u+x) (a=rwx,u-r)(g=rwx,a-r) Сделать вывод.

7Программы Команды Алиасы

Хотя по русски правильно писать псевдоним, я буду писать алиас русскими буквами. Простите меня.

В NIX-философии - каждая программа должна выполнять одну функцию, но делать это хорошо.

Вы уже сталкивались с использованием программ. Да, **touch , ls , chmod , mkdir , rmdir , rm** - это не команды - это программы.

Для того, чтобы отличить команду от программы нужно воспользоваться командой **type**. А еще лучше, **type -a command|program|alias**

Обычно формат запуска любой программы имеет следующий вид:

program [OPTION]... [FILE]... Все что указывается в квадратных скобках В ДАННОМ СЛУЧАЕ необязательные параметры.

[FILE] - обозначает один или несколько файлов. [OPTION] - один или несколько ключей, которые могут менять поведение программы.

Вы уже видели, что команду **ls** запустить разными способами:

```
ls -l -a
ls -la
ls -la ~/work /tmp /dev
```

Практически то же самое относится и другим программам, которые мы уже использовали. Конечно, есть и исключения. например, формат запуска программы сжатия 7zip имеет следующий вид:

7z <command> [<switches>...] <archive_name> [<file_names>...] [<@listfiles>...]

Обратите внимание command и archive_name указаны без квадратных скобок, следовательно являются обязательными параметрами.

Местоположение некоторых параметров должно быть строго задано, некоторые параметры можно менять местами, объединять или разделять.

Обычно, если ключ не требует дополнительных параметров, его можно объединить с другими ключами.

Например в команде **7z** command должен быть первым параметром, а файлы должны идти после имени архива.

В команде **ls** ключи можно менять местами и группировать. Вот пример абсолютно одинаковых команд:ls -la file

```
ls -l -a file
ls -al file
ls ~/ -la # для особых извращенцев
```

Тут же возникает вопрос, а много у команды может быть ключей?

Да, бывают команды очень мощные/сложные.

А как же тогда запомнить все ключи?

А все ключи запоминать и не нужно. Нужно помнить только что какая программа что делает. А как ее запускать и какие ключи использовать можно посмотреть в специальном справочнике (manual). И для этого, как вы уже догадались есть специальная программа - **man**

В вызывается просто:

man program Здесь program - имя программы. Самый тривиальный вариант - **man man** - справка о том, как пользоваться справкой.

*# Нужно взять за правило: Когда встречаетесь с незнакомой командой, или видите незнакомый ключ - вы должны сначала посмотреть **man** этой команды.*

Для начала: перемещаться по man'у можно стрелками, пробелом, клавишами PgUp. PgDown. Для выхода - кнопка q (Но обязательно в английской раскладке)

Для быстрого/частого/лёгкого использования команд с большим числом параметров или ключей можно использовать команду **alias**. Здесь проще привести пример, для понимания.

```
alias cdw='cd /home/username/work'
```

Если выполнить эту команду (обратите внимание на одинарные кавычки), то в следующий раз, чтобы попасть в рабочий каталог, достаточно будет набрать **cdw** вместо длинной-предлинной команды.

Данный алиас прослужит до выхода из bash. Чтобы сделать алиас постоянным, его нужно прописать в файл ~/.bashrc . Работать начнет при следующем запуске bash.

По-умолчанию в скриптах алиасы не работают. И это правильно. Но при интерактивной работе в оболочке они экономят много времени.

Просмотреть значение конкретного алиаса можно командой **alias <aliasname>** . Получить полный список можно просто выполнив **alias** для отмены конкретного алиаса выполните

```
alias <aliasname>=
# или
unalias  aliasname1  aliasname2 ...
```

Аналогом программы **man** для встроенных команд bash является команда **help**. Например:

```
help cd
help alias
```

Межпрограммное взаимодействие.

Существует несколько способов передачи информации между программами. Вот некоторые, наиболее часто используемые в bash:

1. Стандартные потоки stdin, stdout, stderr.
2. Файлы.
3. Код возврата errorlevel.
4. Переменные.
5. Сигналы.

Неявный запуск.

1.Набираете файл **scr1.sh** с текстом:

```
#!/bin/bash
# здесь любые адекватные команды
```

2. **chmod +x ./scr1.sh** # установка прав

3. **./scr1.sh** # запуск

Путь к скрипту (хотя бы локальный) указывать обязательно, иначе bash начнет искать скрипт в папках, указанных в переменной PATH, а если не найдет, то и не выполнит.

Этот способ запуска скрипта неявный. Поскольку интерпретатор записан внутри самого скрипта, и система выбирает интерпретатор, указанный после шебанг. (#!)

 # Неявный способ требует наличие права исполнения.

Задание 7

1. Изучить **man** следующих команд: **touch** , **ls** , **chmod** , **mkdir** , **rmdir** , **rm** .

2. Изучить **help** команды **type** .

3. Написать одним предложением, что кроме создания файла может делать **touch** .

4. **ls** - записать назначение ключей t,S,X,r,h. Проверить на папке /usr/bin .

5. **chmod** объяснить ключи -v -R

6. **mkdir** объяснить ключи -m -v

7. **rm** записать действия ключей -r -f -i -v (не выполнять)

8. Протестировать команды **type** , **type -a** на различных командах/программах, включая **printf** .

9. Самостоятельно изучить команду **which** , сравнить с командой **type** . Сделать вывод для чего можно применить команду **which** .

10. Придумать (вспомнить, найти в поисковике) какой-нибудь нужный алиас.

 # Благодаря пользователю telegram @K3nny2k , выяснилось, что на многих RHEL дистрибутивах which сам по себе алиас, который выводит "дополнительную" информацию, и вывод может отличаться от других дистрибутивов.

8Стандартные потоки. Часть 1.

Стандартные протоки.

stdin (полный путь /dev/stdin) входной поток. По умолчанию клавиатура. Из этого потока можно только читать данные. Дескриптор (номер) этого потока 0.

stdout(/dev/stdout) выходной поток. По-умолчанию терминал (ну, экран монитора) можно только писать данные. Через этот поток программа передаёт основные данные. Дескриптор этого потока 1.

stderr(/dev/stderr) выходной поток. По-умолчанию терминал. Через этот поток программа обычно передаёт данные об ошибках. Но принципиально он ничем не отличается от stdout. Дескриптор этого потока 2.

Потоки stdout и stderr разделили для того, чтобы сообщения об ошибках и основные данные не перемешивались и их можно было бы разделить.

Приведу аналогию. Телевизор. Пульт - stdin, Фильмы - Основные данные stdout. Реклама - ошибка stderr.

Было бы круто в жизни разделить эти два потока. Фильмы перенаправить на основной экран, рекламу на какой-нибудь маленький телевизор, желательно выключенный.

А в nix-системах это сделали. И несмотря на то, что на экран они выводятся одновременно, мы всегда их можем разделить.

 # При написании скриптов сообщения об ошибках и отладочную информацию желательно выводить на stderr.

Кроме стандартных (0-2) в скриптах можно использовать дескрипторы 3-9. Теоретически можно использовать и бóльшие номера, но не желательно, поскольку они могут быть задействованы оболочкой bash.

Перенаправление потоков.

В перенаправлении потоков нет ничего сложного. Мало того, Вы его уже использовали.

Команды "Дятел" **:> filename** и "долбодятел" **:>> filename**, по сути являются перенаправлением stdout команды : (двоеточие) в файл filename.

Да-да "двоеточие" - это команда bash, аналог команды **true**. Она ничего не выводит и завершается без ошибок.) . Не верите? Проверьте сами **help** :

Далее prg - это любая команда или программа, которая выводит данные в stdout и/или stderr и/или/возможно читает данные с stdin.

prg1 > file_or_dev

Перенаправит stdout prg1 в файл или на устройство. Если файл существовал - он вначале очистится. Для этой команды номер дескриптора можно опустить.

prg1 >> file_or_dev

Перенаправит stdout prg1 в файл или на устройство. Если файл существовал - информация будет добавлена в конец файла.

prg1 < file_or_dev

Аналогична предыдущим, за исключением того, что осуществляется чтение из file_or_dev и перенаправление на stdin prg1

prg1 | prg2 | prg3 ...

Передаст stdout prg1 на stdin prg2 и т.д. (stderr, любой из команд/программ, по прежнему будет выведен на экран).

На английском это называется pipe (труба) На русском - дело труба имеет другой смысл, поэтому по русски термин называется конвейер.

Конвейеры применяются для последовательной обработки (фильтрации) данных одной или несколькими программами.

prg1 |& prg2

объединить stdout и stderr первой программы и передать на stdin второй.

prg 2> file_or_dev

перенаправить stderr prg в файл или на устройство. Эта конструкция часто применяется когда нужно "подавить" сообщения от ошибках.

prg1 2> /dev/null | prg2

Результатом этой команды будут перенаправления:

stderr prg1 перенаправляется на **/dev/null**

stdout prg1 перенаправляется на **stdin** prg2

Устройство /dev/null - "черная дыра". В **/**dev/null можно записать любое число данных, но при попытке чтения - система "скажет", что все данные уже считаны, и закроет устройство.

Ну, и остался один вопрос, как перенаправить один поток в другой:

2>&1 данная команда перенаправит stderr в stdout.

 # Обратите внимание на & . Если его не указать, то stderr будет перенаправлен в файл с именем 1.

Есть еще один сложный вопрос, множественные перенаправления в одной команде. Приведу исключительно для полноты картины.

Когда в одной строке используется несколько перенаправлений они начинают применяться по-арабски (справа налево):

prg1 >file_or_dev 2>&1 или сокращенная форма **prg1 &>file_or_dev**

 можно расписать так:

1. **(2>&1)** stderr перенаправляется в stdout.

2. **(>file_or_dev)** stdout (а он уже "содержит" stderr) перенаправляется в файл или на устройство.

Если перенаправить в файл - получим полный лог работы программы, если перенаправить в null программа "молча" выполнит свою работу.

prg1 2>&1 >file_or_dev | prg2

1. **(>file_or_dev)** stdout prg1 перенаправляется в file_or_dev

2. (**2>&1**) stderr prg1 перенаправляется в "новый" stdout

3. **(| prg2)** "новый" stdout перенаправляется на stdin prg2

результатом этой команды будет то, что stdout prg1 будет выведен в file_or_dev , а stderr prg1 будет передан prg2 для дальнейшей обработки.

Иногда, крайне редко, может возникнуть ситуация, когда нужно по конвейеру передать не stdout, а stderr. А stdout, по-прежнему выдавать на экран. Это можно сделать следующим образом:

prg1 3>&1 1>&2 2>&3 | prg2

Комбинация **3>&1 1>&2 2>&3** просто поменяет местами потоки stdout и stderr, используя поток 3 в качестве промежуточного В этом случае ошибка **prg1** будет передана **prg2**, А основные данные **prg1** будут выведены на экран по потоку stderr.

Кроме вышесказанного к перенаправлениям можно отнести команду **tee**.

tee filename или **tee -a filename**

Эта команда передает данные, полученные с stdin на stdout (но еще записывает их в файл filename. С ключом -a информация в файл будет добавляться)

Эта команда может быть использована для контроля, что происходит внутри конвейера, состоящего из нескольких частей или сохранения промежуточных данных для дальнейшей обработки. Или просто для визуального контроля на экране данных записываемых в файл. Например:

echo 123 |tee filename.txt

одновременно выдаст данные на экран и запишет в файл.

Многие начинающие пользователи начинают сходить с ума, когда программа выдает данные на экран, но не получается перенаправить их в файл. (например mplayer).

 # Запомните. если программа выдает данные на экран, но их не получается перенаправить в файл, значит данные передаются на stderr.

Задание 8

Условие: Всё исполняемое оформляем в виде скрипта(ов).

1. Чем stderr отличается от stdout?

2. Чем stdin отличается от stderr?

3. Написать команду перенаправления stderr в файл /tmp/myerr.log .

4. Почему при выполнении команды **ls ~/ 2>> /tmp/myerr.log**
Создается пустой файл?

5. Измените параметры команды п4, чтобы заполнить myerr.log информацией.

6. Напишите команду **ls** с одновременным перенаправлением > соответствующих потоков в файлы stdout.txt и stderr.txt. (Нужно придумать такую команду, чтобы оба файла заполнялись одновременно.)

7. Выполните команду п6 2 раза подряд. Просмотрите файлы stdout.txt и stderr.txt с помощью редактора.

8. В команде п 6 замените > на >> и снова выполните 2 раза подряд.

9. Не открывая файлов ответить, сколько одинаковых записей содержат файлы stdout.txt stderr.txt. проверить догадку и объяснить.

10. В чем разница между командами:

prg1 2>&1 >file_or_dev | prg2

prg1 >file_or_dev 2>&1 | prg2

9Программы cat, head, tail, less, more

cat [OPTION]... [FILE]

(concatenate) Это не кот. Объединение. В обычном режиме выводит указанные файлы на stdout.

В конвейере, без параметров просто передаст данные с stdin на stdout, только в одну колонку и без всяких цветов. В конвейере с параметрами stdin нужно заменить на - (минус). Пример:

ls -la |cat - file1.txt

сначала будет выведен результат команды **ls**, а затем содержимое файла file1.txt

ls -la |cat file1.txt -

сначала будет выведено содержимое файла file1.txt, а затем результат команды **ls**.

head [OPTION]... [FILE]

Голова - аналог **cat**, но позволяет ограничить вывод информации первыми строками (или байтами) файла (без параметров 10 первых строк.)

tail [OPTION]... [FILE]

Хвост - аналог **head**, но позволяет ограничить вывод информации последними строками (или байтами) файла (без параметров 10 последних строк.)

Ну и команда **less**. С этой командой Вы уже знакомы. Именно в ней открывается **man**. Кроме обычного перемещения в ней работает поиск. Для поиска вниз нужно нажать / и задать строку поиска. Enter. Переход к совпадению ниже - буква **n**, совпадению выше **b** . Для поиска вверх нажать ? и задать строку поиска. Enter. для к совпадению выше буква **n**, для перехода к совпадению ниже буква **b.**

Задание 9

1. Изучить **man** команд **head** , **tail** , **less** , **cat** , **more**.

2. Создать файл /tmp/20.txt

3. С помощью редактора пронумеровать строки от 1 до 20, сохранить.

4. **ls -la /usr/bin |less** опробовать поиск цифр в обоих направлениях.

Условие Оформить все следующие команды задания в виде одного скрипта. Считать, что файл /tmp/20.txt существует и правильно заполнен.

5. С помощью программы **head** вывести первые 5 строк файла.

6. С помощью программы **tail** вывести последние 5 строк файла.

7. Комбинируя обе программы с помощью конвейера вывести на экран строки с номерами 7-11.

8. Добиться того же результата поменяв **head** и **tail** местами.

9. С помощью команды **ls -l** (плюс другие ключи) и **tail** вывести на экран 5 самых больших файлов из папки /usr/bin

10. Изменить команду, для вывода 5 самых маленьких файлов из той же папки.

11. Объяснить, что делает команда **head -n-5**

12. Расписать, что делает команда **ls -l /usr/bin |cat -n -**

13. Сравнить команды **less** и **more**. Сделать вывод.

10 Коды завершения программ. (errorlevel)

Для того, чтобы знать, успешно выполнилась программа или нет, ввели понятие код завершения программы. Обычно это целое число в диапазоне 0-255.

0 - программа завершилась нормально.

Не ноль - программа завершилась с ошибкой.

Почему так сделали? Потому что вариант правильного завершения всего один. А вариантов неправильного завершения много.

Номер кода ошибки говорит, в чем проблема.

Если применительно к людям, то 0- абсолютно здоров, Остальные коды указывают, чем болен. Например 1 - ОРВИ, 2-ГРИПП, 3-КОВИД...

Для анализа статуса завершения существуют специальные операторы.

&& - успех (истина)

|| ошибка. (ложь)

Например: **prg1 && prg2 || prg3**

В случае успешного выполнения prg1, также выполниться prg2

В случае ошибки в prg1 сразу будет выполнена prg 3

 # В этой конструкции есть подводный камень. Если при выполнении prg2 произойдет ошибка, prg3 также будет выполнена.

Будьте внимательны логика данной конструкции применима только к bash В других оболочках логика работы может отличаться.

Если prg1 представляет собой конвейер - код ошибки (по-умолчанию) определяется последней командой конвейера.

Например

ls -l filename && echo Успех || echo Ошибка

сообщение будет разным, в зависимости от того, существует ли файл или нет.

Эти операторы не знают, какой номер ошибки. Они просто проверяют 0 или НЕ 0.

Код ошибки записывается в специальную переменную $?

Операторы не обязательно применять парой, можно и по отдельности.

Например:

cd "/full/path/name" || exit

данная команда прервет выполнение программы, если переход с каталог не произошел (команда **cd** завершилась с ошибкой) При сменах каталога в скриптах такую проверку делать нужно, чтобы быть уверенным, что все работает, так как должно.

Обратите внимание код ошибки и текстовое сообщение об ошибке, выдаваемое на stderr -это не одно и то же. Хотя конечно, они связаны.

Коды ошибок у всех программ свои. Один и тот же код возврата может говорить о разных ошибках у разных программ.

Код успешного завершения у всех единый. 0

Для отладки есть специальные команды, выдающие гарантированный код возврата.

```
true # код всегда 0
: # (двоеточие) - аналог (замена) команды true
false # код всегда не 0.
```

Эти команды ничего не принимают и не выдают в стандартные потоки, не используют параметры, только возвращают errorlevel.

Задание 10

Условие: Задания 2-10 оформить в виде скрипта. Каждая команда должна выдавать результат работы в виде текста "Успех" и "Ошибка" на stderr и одновременно в файл /tmp/err.log

1. Самостоятельно изучить команды **wc** , **diff** , **cmp** , **b2sum**, **md5sum** , **sha1sum** , **sha256sum** .

2. Привести основные отличие **cmp** от **diff**

3. Привести примеры использования **wc** для подсчета строк, байт, слов, максимальной длины строки.

4. В /tmp/kurs оптимально создать три каталога (1, 2, 3) с правами 700. Копировать несколько файлов из /bin с помощью масок. Каталоги 1, 2 одинаковые, 3 немного отличный, но не пустой.

5. Подсчитать количество файлов в каждом с помощью **wc**.

6. Сравнить каталог 1 с каталогами 2 и 3 помощью **diff**.

7. В каталоге 2 изменить права на файлы на 644 с помощью текстового способа.

8. Сравнить каталоги 1 и 2 с помощью **diff**. Сделать вывод.

9. Создать (или копировать) в каталоги 1 и 2 текстовый файл с одинаковым именем, но разным содержимым. Сравнить каталоги. Сделать вывод.

10. Сравнить два текстовых файла из п.9 с помощью **b2sum** , **md5sum** , **sha1sum** , **sha256sum** . (хэш сравнивать визуально) В чем основное отличие **b2sum** от остальных программ вычисления контрольных сумм?

11. Объяснить errorlevel команды **ls -la /no_file* | tail -3**

11 Константы, кавычки, экранирование, escape-последовательности.

Поскольку Вы уже используете скрипты - сегодня будет "практическое" изучение темы. Во время изучения Вы будете пробовать выполнять скрипты, для лучшего понимания.

Константы.

Константы в bash это наборы символов которые не являются комментарием, не являются внешней программой или ключевым словом bash и не изменяются в процессе использования.

В команде **echo Hello world!** текст Hello world! как раз и является константой.

При работе с файлами и каталогами вы тоже работали с константами, потому что названия файлов и каталогов, в том виде, каком Вы их задавали - это константы.

В bash всего два типа констант текстовые и числовые (и при этом только целые).

Если с текстом как-бы все понятно, то числа могут быть и текстовой и числовой константой. Зависит от контекста.

Выполните скрипт.

```
#/bin/bash
echo 3+5 #текстовая константа
#3+5
echo $((3+5)) # числовые константы
#8
```

В данном примере в первой строке 3+5 - текстовая константа. Одна! состоящая из трех букв (буква 3 буква "+" буква 5), во втором примере - 3 и 5 числовые константы. Результат будет разный.

Кавычки.

Кавычки бывают двух типов одиночная ' и двойная ".

Одиночные кавычки - "жесткие" (как папа).

Двойные - "мягкие" (как мама). Рассмотрим разницу между ними.

Выполните скрипт:

```
#/bin/bash
echo $((3+5)) #8
echo "$((3+5))" #8
echo '$((3+5))' #$((3+5))
```

Одиночные кавычки настолько "жесткие", что как сказано, так и сделано.

Двойные кавычки интерпретируют, записанные в них операторы и переменные.

Но есть еще одна очень важная функция кавычек. Они позволяют правильно трактовать пробелы в начале и в конце вывода.

Видоизмените программу, добавив перед "$" 5 пробелов и выполните:

```
#/bin/bash
echo      $((3+5))
#8

echo "     $((3+5))"
#     8

echo '     $((3+5))'
#     $((3+5))
```

Кроме этого, есть еще одна причина использовать кавычки. Константы с пробелами или спецсимволами.

Вы уже видели, что некоторые команды (а на самом деле многие) могут работать с несколькими файлами. И файлы разделяются пробелами.

 # Если файл содержит пробел или несколько в своем имени - он будет интерпретирован как два или более файлов. Для правильной интерпретации таких имен файлов их необходимо заключать в кавычки.

Использовать можно любые кавычки, но со временем Вы научитесь правильно выбирать кавычки для той или иной ситуации.

Чтобы избежать возможных проблем имена файлов нужно заключать в кавычки.

Использовать имена файлов с пробелами у ITшников считается признаком недостатка образования, скажем так. Если нужно разделить слова - используйте подчеркивание.

Однажды я столкнулся с ситуацией, когда SAMBA-сервер (обычный файловый сервер для win файлов) Правильно выдавал имена файлов для

linux -клиентов и не правильно для win (какие-то буквоцифры вроде хэша). Я долго не мог понять в чем дело. Оказалось пользователь на linux в конце имени файла ставил пробел. И windows начинала сходить с ума от такой дерзости. Не все операционные системы одинаково относятся к пробелам и разным символам.

У кавычек есть еще одна функция. Они позволяют передавать символ перевода строки в константе:

```
#!/bin/bash
echo ' mama
  myla
      ramu'
```

Проверьте сами, двойные кавычки работают так же?

Если в константе встречается одна кавычка - вся константа должна быть заключена в кавычки другого типа.

Например:

```
#!/bin/bash
echo '123"123'
echo "123'123"
```

Экранирование.

В bash есть специальный символ \, он говорит, что следующий за ним символ - это просто символ, а ни какая-нибудь трактовка в стиле bash.

Так происходит с любым символом, кроме перевода строки.

Экранирование в конце строки, сообщает bash, что команда продолжается и на следующей строке. Символ \ как-бы отменяет перевод строки. Это улучшает читаемость текста, если команда очень длинная. (Кроме этого допускается разбивать длинные команды по символу | или & . В этом случае экранировать перевод строки не нужно.)

Символ экранирования \ не должен быть внутри одинарных кавычек, иначе превратится в обычную букву \ .

Экранирование распространяется на ОДИН символ. Если несколько спецсимволов идут подряд - экранировать нужно каждый.

```
#!/bin/bash
echo 123\"123 #Экранирование кавычки
echo 123\'123 #Экранирование кавычки
echo длинная длинная \
 очень длинная \
 команда
```

```
echo 123 123 |
     wc -w # Отступ позволяет легче заметить, что это
           # продолжение.

echo 123 123 \|wc -w #Экранирование символа конвейера
ls -l &&
    echo OK #строки две, а команда одна
ls -l ||
    echo ERR # аналогично
```

Размещение одной команды конвейера в строке резко улучшает читаемость программы.

Escape(эскейп)-последовательности (управляющие последовательности)

Есть еще специальные символы, которые иногда очень нужны. Они начинаются со знака \, но в данном контексте это не знак экранирования. Вот они

\a Сигнал (звонок).

\b backspace (возврат на одну позицию влево без затирания символа)

\e символ escape

\E символ escape

\f Перевод страницы (form feed)

\n Новая строка.

\r Возврат каретки.(на начало строки)

\t Горизонтальная табуляция.

\v Вертикальная табуляция.

\\ Обратная дробная черта (\)

\nnn символ с восьмеричным кодом (1-3 восьмеричных цифры)

\xHH шестнадцатеричным кодом(1-2 шестнадцатеричных цифры)

\uHHHH Символ Unicode (ISO/IEC 10646) с шестнадцатеричным кодом HHHH (1-4 шестнадцатеричных цифры)

\UHHHHHHHH Символ Unicode (ISO/IEC 10646) с шестнадцатеричным кодом HHHHHHHH (1-8 шестнадцатеричных цифр)

\cx код символа control-x (x - латинская буква) \cj=перевод строки \ci=горизонтальная табуляция...

Всем, кто решил "позвенеть" \a сообщаю, что данная последовательность раньше выводила сигнал на спикер, которого в современных компьютерах обычно нет. А если и есть, то звука все-равно может не быть.

Чтобы использовать управляющий символ внутри константы нужно вставить его в специальную конструкцию **$'ESC_CHAR'**

```
#!/bin/bash
echo 1234567890$'\n'ABC
echo 2345678901$'\t'ABC
echo 3456789012$'\b'ABC
echo 4567890123$'\r'ABC
echo 5678901234$'\e'ABC
echo 6789012345$'\u2665'ABC
echo 7890123456$'\U1F4BB'ABC

#Есть еще способ:
# ключ -e обязателен
# два слэша
echo -e мама\\nмыла\\nраму
# или кавычки
echo -e "мама\nмыла\nраму"
```

Еще один способ будет рассмотрим при изучении оператора **printf .**

Задание 11

Условие: файлы создаем в каталоге /tmp Пункты 2-6 выполняем скриптом.

1. Самостоятельно изучить программы **dirname basename hostname**

2. Попробовать создать несколько файлов с символами в именах " ' * # \> : & ( ) [ ] { } $ "пробел" (по одному символу на файл).

    2.1. В случае ошибки попробовать экранирование символа.

    2.2. Дополнительно попробовать использовать кавычки.

3. Просмотреть список созданных файлов, затем удалить все созданные файлы из п.2

4. Три самых длинных команды из задания 10 оформить переносами строки с сохранением работоспособности.

5. Привести 5 примеров вывода на экран констант с эскейп-последовательностями.

6. То же, что п.5, но реализовать через echo -e

12 Переменные.

Переменная - именованное место в памяти для хранения изменяемых данных.

Имя переменной может состоять из букв, цифр, знака подчеркивания. Имя не должно начинаться с цифры.

Как и в файлах регистр важен. Как правило, в **bash**, переменные набираются большими буквами, для того, чтобы их легче было отличать от ключевых слов и программ.

В bash переменные бывают двух типов: строковые и целые.

Максимальное число 9223372036854775807, минимальное - 9223372036854775808. (64 бита со знаком).

 # Будьте осторожны. В отличие от других языков программирования bash не обрабатывает переполнения. Если к максимальному числу прибавить 1, число станет отрицательным!!! Ошибки при этом не возникает. Работать с числами вблизи максимальных значений нужно очень аккуратно.

По традиции, целочисленные переменные, используемые в циклах принято начинать с букв I,J,K,L,M,N

Вещественных (с десятичной точкой) не предусмотрено. Но работа с вещественными числами возможна посредством сторонних программ (**awk** , **bc** , **dc** , и других).

Переменных типа data/time тоже нет. Аналогично вещественным, работа с ними возможна с помощью оператора **printf** и других программ (**date** , **gawk**).

Важно найти баланс между длиной имени переменной и ее смыслом.

Например: CUR_DIR вполне понятно, и удобнее, чем MY_CURRENT_DIRECTORY и в тоже время понятнее, чем C или CD.

Когда несколько переменных объединяются (конкатенация) в одной команде для лучшей читаемости переменную можно заключить в фигурные скобки:

FULL_FILE_NAME=${CUR_DIR}/${CUR_FILE}.${EXT}

Основным оператором описания переменных является **declare**

Небольшую справку по операторам **bash** можно получить командой **help**. Например **help declare**.

Кроме оператора **declare** есть еще несколько операторов описания переменных (**local**, **export**, **readonly**) но их функционал полностью перекрывается оператором **declare**.

*# Для описания переменных рекомендую пользоваться оператором **declare** . Если решите "избавиться" от функции - не придется менять **local** на **declare** .*

Для того, чтобы использовать значение, сохраненное в переменной, перед именем нужно поставить знак $. Видимо потому, что информация стоит дорого.

Значения переменных, описанные как целые в основных арифметических операциях могут использоваться без знака $ и без (()) (в старых версиях bash и в других оболочках это может не работать.)

Например:

```
#!/bin/bash
declare -i I J
I=J+2
```

Переменная J используется без знака $.

Для присвоения значения переменной используется знак =

 # ВАЖНО: при присвоении вокруг знака = не должно быть пробелов.

Примеры:

```
#!/bin/bash
declare -i I=7 J K=1 # описаны целые переменные I, J, K .
                     # переменным I и K присвоены значения.

declare -a A # переменная A описана как индексированный
             # (классический) массив.

declare -A BUSINESS86 # переменная BUSINESS86 описана как
             #массив с текстовыми индексами  (Ассоциативный)
             # В старых версиях bash и других оболочках
             # может не работать.

declare -l LOWREG # значению переменной ПРИ ПРИСВОЕНИИ
                  # будет установлен нижний регистр.

declare -u UPREG # значению переменной ПРИ ПРИСВОЕНИИ будет
                 # установлен верхний регистр.
```

```
declare -r READONLY # Значение переменной READONLY нельзя
                    # будет изменить/удалить/переопределить.
                    # Фактически именованная константа.

declare -x EXPORT # переменная EXPORT будет видна
                  # дочерним процессам.

declare -t VAR # используется при отладке программ
               # (как-правило больших)

declare -n LINK # Указатель(ссылка) на переменную, имя
              # которой присвоено переменной LINK.
```

для того, чтобы "сбросить" уже установленные атрибуты нужно заменить "-" на "+".

```
declare +x EXPORT # после этого объявления переменная
    #EXPORT перестанет экспортироваться в дочерние процессы.
```

Еще раз повторю. "-" - установить атрибут "+" сбросить. Добро пожаловать в логику bash.

Определение -r отменить не получится. Используйте аккуратно.

Область видимости переменных:

Все переменные **bash** по-умолчанию глобальные. Т.е. переменные, объявленные в основной программе будут видны И МОГУТ БЫТЬ ИЗМЕНЕНЫ внутри функций. Часто это приводит к появлению трудно обнаруживаемых ошибок.

Чтобы переменная стала локальной (используемой только внутри функции) ее нужно описать оператором **declare** или **local** внутри функции.

Операторы **declare** или **local** работают одинаково, но **local** может применяться только внутри функций.

Значение переменной может передаваться только от родительского процесса к дочернему (как гены).

Для просмотра описания переменных и значений используется оператор **declare -p VAR1 VAR2**

Его нужно использовать при отладке, для контроля значения и флагов переменных, а также массивов и функций.

Оператор **declare** без параметров выведет на экран значение всех описанных переменных и функций (осторожно их очень много.)

Для сброса переменной (очищения памяти, занимаемой переменной) используется команда **unset .** Например: **unset VAR1 VAR2**

*# Примечание: Повторюсь. Сбросить переменную, описанную как **readonly** не получится. Не получится и переопределить ее и как локальную. Значение переменной будет постоянным, вплоть до окончания работы данной копии **bash** или **subshell** в котором переменная была описана. При описании переменной **readonly** + **export** ее нельзя будет переопределить/изменить и во всех дочерних процессах и подсистемах.*

 # Примечание: Использование значения переменной, в строке, вместо команды будет интерпретировано, как команда Например:

```
#!/bin/bash
VAR="ls -la"
$VAR # будет выведено содержимое текущего каталогам.

eval $VAR # то же самое, даже хуже
```

Оператор **eval**, очень опасный. Использование его может привести к взлому системы. Используйте его, если знаете что делаете. А лучше - не используйте. Именно использование этого оператора позволяет реализовать инъекции.

Задание 12

1. Самостоятельное изучение команд **du** (особенно ключи b s h d) **df**

2. Написать скрипт, в котором описаны переменные: целые I,J,K; обычные массивы A7, B13 ;переменные для экспорта PA, VASYA .

3. Задать произвольные начальные значения всем переменным.

4. Снять опцию экспорта для переменной VASYA

5. Описать переменные I3 -как целую, B - как текстовую верхнего регистра. Переменным значения не задавать)

6. Присвоить переменной B значение "yes", Переменной I значение "no", переменной PA значение "Сегодня в школу не пойдем!"

7. С помощью команды echo распечатать значения всех описанных выше переменных. В виде: echo "VAR=$VAR"

8. Вывести на экран значения и описание всех описанных выше переменных с помощью declare. Сделать вывод о значениях переменных и удобстве применения команд для контроля значений переменных.

9. Значения каких переменных могут быть использованы без знака $ и в каком случае? Привести пример на переменных описанных выше.

10. Можно ли переменной, описанной как массив присвоить значение "1". Например: A7="1"

11. Написать второй скрипт. В скрипте задать переменные HOME_DIR, FILE_NAME, FILE_EXT. (значения на Ваше усмотрение)

12. Все возможные ошибки выводить на экран и в лог-файл с полным именем и суффиксом .err

13. Результат каждой операции подтверждать с помощью списка каталога.

14. Создать файл HOME_DIR / FILE_NAME . FILE_EXT

15. Создать резервную копию файла с полным именем и суффиксом ".1" на конце.

16. Удалить исходный файл.

17. Восстановить исходный файл с помощью команды переименования.

13 Системные переменные.

Оператор **declare** позволяет вывести на экран переменные и функции, которые заданы системой. Они могут быть полезны при написании скриптов. Состав этих переменных может отличаться от системы к системе, но все же есть общепринятые. Вот некоторые из них:

BASH=/usr/bin/bash - указывает, где в системе находится bash. Используется при неявном запуске скриптов через **env**: #!/usr/bin/env bash

BASH_SUBSHELL - уровень подсистемы исполнения (subshell).

BASH_VERSION='5.1.4(1)-release' версия bash. (На старых версиях bash некоторые нововведения могу не работать. Если у Вас на машине скрипт работает, а на другой машине - нет - сравните версии.

COLUMNS=104 - число столбцов терминала. (важный параметр особенно при взаимодействии с пользователем и написании игр)

LINES=17 - число строк терминала (важный параметр особенно при Выводе на экран и написании игр). Эти переменные могут инициализироваться после выполнения оператора **clear**.

HOME=/home/username - домашний каталог пользователя

HOSTNAME=compname - название этого компьютерах

HOSTTYPE=x86_64 - тип процессора и разрядность операционной системы.

IFS=$' \t\n' - разделители полей bash.

PWD=/dev/shm - текущий каталог. Кстати /dev/shm - это диск в памяти(ramdisk). Есть не на всех системах. Обычно равен половине памяти системы. Может использоваться как очень быстрый, но не очень большой аналог /tmp. Он, очищается при перезагрузке.

OLDPWD=/home/username/Downloads - предыдущий каталог (именно с помощью этой переменной команда "**cd -**" определяет, куда нужно перейти.

OSTYPE=linux-gnu Тут все должно быть понятно.

PATH=/usr/local/bin:/usr/bin:/bin - одна из самых важных переменных в системе. Она задает каталоги, в которых система будет искать запускаемую программу, если не указан полный путь.

Именно из-за переменной PATH могут неправильно работать скрипты, проверенные и отлаженные под пользователем, и запускаемые из cron (планировщик заданий). Если предполагается запуск скрипта через cron - в самом начале скрипта задайте переменную PATH. Обратите внимание, пути разделяются символом двоеточие. Если в системе есть несколько программ, - будет запущена программа из каталога, который ближе к знаку =

PPID - parent process id - номер, родительского процесса.

SECONDS - Переменная показывает время (в секундах) выполнения скрипта. Или время, прошедшее после ее обнуления.

SHELL=/bin/bash

SHLVL=2 - уровень вложенности оболочки bash.

TERM=xterm-256color - Название текущего терминала. Важная переменная если нужно анализировать клавиатурные коды функциональных клавиш.

UID=1000 - user id - код пользователя в системе. (если 0 - то root) постоянный для пользователя

EUID=1000 - эффективный user id(если 0 - то root) (может меняться)

LOGNAME=username имя пользователя в системе

USER=username то же имя пользователя в системе.

PREFIX="/data/data/com.termux/files/usr" Важная переменная termux (эмулятор терминала на системе android), которая позволяет правильно работать с каталогами в терминале.

Переменные LOGNAME и USER дублируют друг-друга. Это делается для совместимости. Дело в том, что на разных ОС переменные назвали по-разному. И для того, чтобы при переносе с одной системы на другую все работало одинаково, - используют переменные с одинаковыми значениями.

Задание 13

Условие: Использовать только изученные средства.

1. . Самостоятельное изучение программ **sort** (особенно ключи n k h u), **uniq**, **tar**(особенно ключи c v f j J z x)

Создать скрипт, в котором:

2. Задать рабочий каталог WORK_DIR следующим образом:

- Проверить наличие каталога /dev/shm

- Если каталог существует, WORK_DIR=/dev/shm

- Если не существует, WORK_DIR=/tmp

3. Задать каталог архивов "backup", как подкаталог рабочего каталога.

4. файлы архива создавать архивном каталоге.

5. Лог должен быть персональным, для каждого пользователя в системе, который будет выполнять скрипт. Лог хранить в рабочем каталоге В лог выводить (как минимум) Название операции, размер полученного архива (в байтах) и размер исходного каталога, а также время операции (Использовать системную переменную SECONDS). В лог ошибок - ошибки выполнения программ.

6. Создать архив каталога /usr/bin (или /etc/ на выбор) в архивном каталоге .

7. Распаковать архив в подкаталог с именем пользователя в рабочем каталоге.

8. После распаковки сравнить исходный каталог и каталог с распакованными файлами.

9. После сравнения и проверки Архив и каталог с разархивированными файлами удалить, используя опцию защиты корневого каталога от удаления.

10. Повторить пп 4-9 для разных фильтров архивирования(bzip2, xz,gzip). (3 раза)

11. Сделать вывод о степени сжатия и времени выполнения.

14 Перенаправление потоков. Часть 2.

Первая часть Глава #8

Оператор exec.

У оператора **exec** две функции:

1. **exec commad** - билет в один конец. Перед выполнением команды, bash фактически завершается и управление передаётся команде. Команду можно выполнить с очисткой переменных окружения. И все... Обратно управление bash не передается. Может использоваться для применения изменений в конфигурационных файлах bash. Например, вы добавили алиас в ~/.bashrc . Для того, чтобы его использовать нужно выйти-войти, или запустить новую копию **bash**, или самый лучший и простой способ - **exec bash** .

2. Перенаправление. Это штука крайне нужная и очень понятная. Служит для связывания файла или устройства с дескриптором. В других языках программирования есть аналог команда open

Примеры:

```
exec 3<fileneme.txt # файловый дескиптор 3 будет использован
                    #для чтения fileneme.txt

exec 4>filename.txt # файловый дескиптор 4 будет использован
                    # для перезаписи fileneme.txt

exec 5>>filename.txt # файловый дескиптор 5 будет
                   # использован для добавления fileneme.txt

exec 6<>filename.txt # файловый дескиптор 6 будет
               # использован для чтения и записи fileneme.txt
exec 7>&-
exec 8<&-        # файловые дескипторы 7, 8 будут закрыты,
                 # хотя обычно в этом нет необходимости
```

Приведу несколько полезных примеров:

exec 2>/dev/null - все сообщения об ошибках "помножить на ноль" На код ошибки (типа errorlevel) это не влияет. Т.е. При использовании этой команды сообщения об ошибках всех дальнейших команд на экран выводиться не будут.

exec 1>filename.txt все сообщения, кроме ошибок отправить в файл. (рекомендую использовать эту команду только а скриптах. При использовании в интерактивном режиме полностью потеряете вывод на экран)

exec 1>filename.txt 2>&1 все сообщения, включая ошибки отправить в файл.

кстати, есть сокращенная форма этой команды:

exec &>filename.txt

Эти конструкции НУЖНО использовать когда скрипт запускается из **cron**. Альтернативой является запуск скрипта с помощью программы **nohup**.

exec 2> >(tee stderr.log) - Все сообщения на stderr будут перенаправлены программе **tee**, которая запишет их в файл stderr.log и выведет на экран.

 # Обратите внимание на пробел между знаками > > и отсутствие пробела между >(

Файл будет содержать вывод всех соощений об ошибках скрипта, но будет перезаписываться при каждом новом запуске скрипта. Чтобы файл не перезаписывался, как обычно, нужно использовать **tee -a** .

Конструкция **>(command)** называется подстановкой процессов. Запуск команды происходит асинхронно в **subshell** Т.е в отдельном процессе, со своими переменными, параллельно с другими командами скрипта.

Если программа command будет производить вывод на экран, никто не сможет гарантировать, что очередность вывода будет соблюдена (сначала скрипт, потом command или наоборот). Вывод даже может продолжаться после окончания работы основного скрипта.

Аналогично вывод команды может быть перенаправлен на stdin **<(command)** . Обращаю внимание между **>(** или **<(** пробелов быть не должно. Пример: **cat <(ls ~/)**

 # Команда exec перенаправляет потоки по-умолчанию для всех команд скрипта.

Команда **exec** не затронет передачу данных от команды к команде посредством конвейера и перенаправления, специально указанные в команде.

И еще один вид перенаправления. Содержимое переменной VAR передается на **stdin** command .

command <<< $VAR

Задание 14

После выполнения пунктов 2-10 выводить сообщение о номере пункта в лог-файл. Скрипт выполнить дважды.

создать скрип.

Переменные LOG=/tmp/log и ERR=/tmp/err LIST=/tmp/list должны быть описаны.

1. Проверить наличие файлов /tmp/log и /tmp/err.

 - В случае наличия файлов - вывести сообщение на экран о наличии файлов.

 - При отсутствии - создать файлы.

2. Установить права доступа файлы 600 текстовым способом.

3. Перенаправить вывод основных данных скрипта в файл /tmp/log , вывод ошибок скрипта в файл /tmp/err

4. Получите развернутый список файлов временного каталога с подробной информацией о правах доступа, владельцах и группах, запишите его в файл LIST

5. Создать во временном каталоге 3 пустых файла с именами u1, u2, u3.

6. Допишите информацию (только о новых файлах), в файл LIST, используя самую оптимальную маску.

7. Убрать право чтения на файлы u1 и u2 для всех, кроме владельца, текстовым способом.

8. Допишите информацию об изменениях в файл LIST, используя самую оптимальную маску.

9. Попробуйте создать файл /usr/bin/u4. Результат операции запишите в LOG.

10. Удалите файлы u1-u3

15 Группировка команд и subshell (подоболочка).

Обычно команды выполняются по одной в строке, или в конвейере (последовательно), но иногда результаты работы нескольких команд нужно объединить. Что при этом происходит?

Несколько команд выполняются как единая команда.

Допустим нужно объединить результат (вывод) работы трех команд prg1 prg2 prg3

При группировке,

- данные, поступающие на stdin группы, передаются на stdin первой программы (prg1)
- stdout всех программ "суммируется" в один stdout группы (prg1;prg1;prg3),
- stderr всех программ "суммируется" в один stderr группы (prg1 ;prg1 ;prg3).
- errorlevel группы будет равен коду возврата последней ВЫПОЛНЕННОЙ команды (prg3 в данном случае).

Для группировки команд есть два способа:

1 {...} Например:

```
{ prg1
  prg2
  prg3
 }
```

или однострочная запись: **{ prg1; prg2; prg3;}**

 # Обратите внимание, при однострочной записи в конце стоит ; и после начальной скобки пробел. Это важно, иначе будет ошибка.

2 (...) Например:

```
(prg1
  prg2
  prg3)
```

или однострочная запись: **(prg1; prg2; prg3)**

В плане группировки и перенаправления стандартных потоков, оба способа работают одинаково. Но есть отличия.

При {...} программы работают в той же оболочке. Все переменные видны внутри группировки. Все изменения переменных также сохранятся после окончания группировки.

При (...) создаётся отдельная подоболочка (subshell) со своими переменными. Изменения переменных внутри этой подоболочки теряются при окончании группировки.

Это также значит, что если Вы внутри группировки сделаете переход в другой каталог, то по окончанию группировки текущий каталог окажется тем же, что был до группировки.

Кроме этого, на создание подоболочки тратятся бóльшие ресурсы, поэтому (...) будет выполняться дольше, чем {...}. особенно это заметно в цикле.

Есть еще одно отличие. Результат вывода второй группировки можно использовать так же, как и результат вывода переменной. $(...) например.

```
A=$(ls ; cat ~/.bashrc);echo $A
#или еще проще
echo $(ls ; cat ~/.bashrc)
```

Где могут применяться группировка? Самый простой и понятный вариант:

prg1 && { prg2; prg3; prg4;}

В этом случае программы 2,3,4 выполнятся при успешном выполнении prg1. Или не выполняться все вместе, при завершении prg1 с ошибкой.

Задание 15

1. Исправить ошибки в скрипте без изменения вида скрипта. Скрипт должен выдавать на экран текущий временный каталог и активный каталог.

```
#!/bin/bash
declare FILE=test.txt
declare SHM=/dev/shm
declare TMP=/tmp
declare CURTMP

cd
ls "$SHM" && {cd $SHM; touch "$FILE" ,CURTMP=$SHM) || {cd
$TMP; touch "$FILE" , CURTMP=$TMP}
echo "$CURTMP"
pwd
```

2. Привести скрипт к виду одна команда в строке.

3. Изменить скрипт, используя группировку (...). Результат выполнения должен быть аналогичен п.1

4. С помощью ключа добиться вывода двух каталогов в одной строке через пробел.

5. Изменить скрипт п.3 чтобы на экран не выводилось ничего кроме требуемых каталогов.

6. Конструкции типа **prg1 && prg2 || echo "ERR"** и **prg1 || prg2 && echo "OK"** имеют неприятную "особенность". Назвать особенность.

7. С помощью группировки вместо prg2 сделать конструкции п6 логичными (чтобы печать сообщения на экран зависела только от результата prg1).

8. Протестировать пп6,7, заменив prg1,prg2 программами true и false в различных сочетаниях (4 шт). Доказав, что "особенность" п.6 можно исправить.

16 Параметры и специальные переменные.

Параметры.

Вы уже использовали параметры. Например:

cat -n -s firstscr.sh firstscr.sh.1

Здесь **cat** - команда, -n -s firstscr.sh firstscr.sh.1 - параметры.

Параметры бывают именованные и позиционные. именованные - можно поменять местами и результат не изменится, потому что их действие зависит только от имени. При перестановке позиционных параметров результат будет другой.

В данном примере на экран выводятся файлы firstscr.sh firstscr.sh.1 , у которых нумеруются(-n) строки, и несколько пустых строк подряд, будет заменяться одной (-s)

Вывод: -n -s - именованные параметры, имена файлов - позиционные.

Любой bash-скрипт может принимать параметры. И эти параметры ВСЕГДА ПОЗИЦИОННЫЕ.

Если Вы захотите сделать некоторые из них именованными - придется писать обработку. Но это не сложно.

Для параметров выделены специальные нумерованные переменные.

$0 - полное название выполняемого скрипта.

$1..$9 -позиционные параметры 1-первый 2 -второй... 9 девятый

Если параметров больше, чем 9 Придется обрабатывать частями. Алгоритм следующий:

Сначала обрабатываем несколько параметров, допустим 3, потом выполняем команду **shift** (сдвиг), которая сдвигает параметры. Проще показать на примере:

скрипт был выполнен со следующими параметрами:

./param.sh 1 2 3 4 5 6 7 8 9 10 11 12

```
#!/bin/bash
# всего параметров 12
# обработать можем только 9
# обрабатываем 3
#(допустим выводим на экран)
echo $1 # 1
echo $2 # 2
echo $3 # 3
shift 3
# первые три значения параметра выбрасываются, остальные
# значения сдвигаются влево.
echo $1 # 4
echo $2 # 5
echo $3 # 6
echo $4 # 7
echo $5 # 8
echo $6 # 9
echo $7 # 10
echo $8 # 11
echo $9 # 12
```

 # Если есть хоть один параметр, то первый параметр всегда заполнен.

В новых версиях bash для указания параметров с номерами больше 9 можно использовать переменные ${10}, ${11}... Фигурные скобки здесь обязательны, иначе результат будет аналогичен: ${1}0, ${1}1

Число параметров всегда известно. Для этого есть специальная переменная $# Причем заметьте, переменная $# показывает число ОСТАВШИХСЯ параметров. т.е. после применения команды **shift** значение $# меняется.

Если использовать **shift** без параметров - сдвиг произойдет на 1. Этот прием применяется при обработке параметров в цикле.

Есть еще две переменных которые отвечают за параметры $@ и $* Они содержат полный список параметров. Переменные очень похожи по функционалу, но отличия есть. Они касаются использования параметров в кавычках и с пробелами Чаще используется $@

Команда **set**.

Команда выполняет множество функций. Сейчас мы рассмотрим одну из них - установку параметров.

set -- prm1 prm2 prm3... prmN

после выполнения данной команды

$1=prm1;$2=prm2; $3=prm3; ${N}=prmN

Команду очень удобно применять при тестировании скриптов с параметрами. Не нужно каждый раз вызывать программу с параметрами. Можно Тестируемые параметры прописать прямо внутри скрипта. Главное после отладки помнить, что тестовые параметры нужно убрать.

Будьте осторожны. Параметры, с которыми был запущен скрипт, будут удалены. Если Вам эти параметры нужны - сначала обработайте их.

Символы **--** означают, что именованные параметры самой команды закончились. проще привести пример:

ls -- -l

ls: невозможно получить доступ к '-l': Нет такого файла или каталога Т.е. после символов **--** '-l' воспринимается не как именованный параметр, а как имя файла.

Команда **set** может "разбирать" параметры не только по пробелам. Можно задать и другие разделители. Например:

```
KEEPIFS=$IFS
IFS=":"
set -- $PATH
IFS=$KEEPIFS
```

Разберет переменную $PATH по отдельным каталогам.

Будьте осторожны. Переменная IFS используется многими командами bash. Перед ее модификацией желательно сохранить ее начальное значение в другую переменную, а после использования - восстановить значение по-умолчанию.

Специальные переменные.

Помимо специальных переменных $0,$1-$9, ${10}, ${11}... $@, $* и $#

Есть еще несколько:

$? код завершения (ошибки), errorlevel

$! PID последнего запущенного нами фонового процесса

$$ PID текущего процесса

$- (минус) Текущие опции bash

$ _ (подчеркивание) последний параметр предыдущей команды.

Задание 16

Условие: Всё задание выполняем одним скриптом. Текст, если нужен, добавляем в виде комментариев.

1. Инициализировать переменную DT результатом команды получения даты

   $(date -d"20240915 21:12:37")

2. Переменную DT разобрать на составные части командой set.

3. Объявить и инициализировать полученными значениями переменные: (WEEK_DAY DAY MON YEAR как целые) TIME T_ZONE как строковые.

4. Распечатать полученные переменные в виде VAR=var_value

5. Установить для переменной TIME режим экспорта.

6. Вывести на экран информацию об объявлениях и значениях всех вышеуказанных переменных.

7. В домашнем каталоге создать файл вида имякомпьютера_год_месяц_день.log

8. Для проверки вывести результат в виде ls -l имя_файла, удалить файл в случае успеха.

9. Не меняя программы пп1-6 сделать так, чтобы WEEK_DAY печаталась большими буквами, а TIME T_ZONE маленькими.

10. Вывести на экран. WEEK_DAY T_ZONE в виде VAR=var_value.

11. Вывести на экран имя скрипта.

12. Программно (из самого скрипта) вывести на экран содержимое скрипта с нумерацией не пустых строк.

13. Разобрать переменную TIME на часы, минуты секунды.

14. В комментарии предложить способ, как обойтись без переменной DT с сохранением функциональности скрипта.

15. При запуске перенаправить результат выполнения скрипта в файл lesson.log .

17 Вычисления в bash.

Математические операции

Для выполнения арифметических операций применяется оператор **let**, расширение ((...)) или просто описание переменных в целочисленный тип оператором **declare -i**. (некоторые операции, описанные ниже, при описании в целочисленный тип оператором **declare -i**, могут не работать.

Решение, что использовать при математических операциях - остается за Вами. Я рекомендую ((...)) .

Внутри математической подсистемы ((...)) допускается использовать значения числовых переменных без знака $

 # Числовые константы, начинающиеся с 0, интерпретируются как восьмеричные. Это может создавать некоторые проблемы при обработке дат.

Константы, начинающиеся с 0x или 0X, интерпретируются как шестнадцатеричные. Большие и маленькие буквы в шестнадцатеричных числах интерпретируются одинаково.

Математические операции в порядке приоритета выполнения. Операции одинакового приоритета выполняются слева направо.

Приоритет выполнения указан в квадратных скобках в начале. Цифры условные, просто для сравнения.

[1] () - скобки.

[2] id++ последующее увеличение переменной(пост-инкремент) .

Пример: **A=0; echo "$((A++)) $A" # 0 1**

Т.е. Сначала будет использовано значение переменной, и уже после использования произойдет увеличение на 1.

Аналог **A=0; echo $A; ((A=A+1))**

[2] id-- последующее уменьшение переменной (пост-декремент)

[3] ++id предварительное увеличение переменной(пре-инкремент)

Пример: **A=0; echo "$((++A)) $A" # 1 1**

Т.е. Сначала произойдет увеличение переменной на 1, а потом будет использовано значение переменной.

Аналог **A=0; ((A=A+1)); echo $A**

[3] --id предварительное уменьшение переменной (пре-декремент)

[3] -, + унарный минус и плюс (отрицательное или положительное число. Это не сложение и вычитание)

[3] ! логическое отрицание (логическая инверсия) НЕ (not) результат сравнения 1=true 0=false

[3] ~ побитовое отрицание (битовая инверсия) НЕ (not) результат число.

[4] ** возведение в степень

[5] *,/ умножение, деление

[5] % остаток от деления

[6] +, - сложение, вычитание

[7] << битовые сдвиги влево без переноса

Пример сдвиг числа 4 на три бита влево:

echo $((4<<3)) #32

Аналог **echo $((4*2**3))** или **echo $((4*2*2*2))**

[7] >> битовые сдвиги вправо без переноса

Пример сдвиг числа 64 на три бита вправо:

echo $((64>>3)) #8

Аналог **echo $((64/2**3))** или **echo $((64/2/2/2))**

[8] <= Меньше или равно. Результат сравнения 1=true 0=false

[8] >= Больше или равно. Результат сравнения 1=true 0=false

[8] < Меньше. Результат сравнения 1=true 0=false

[8] > Больше. Результат сравнения 1=true 0=false

[9] ==, != равенство, неравенство. Результат 1=true 0=false

[10] &   побитовое И (and). Результат число

[11] ^   побитовое исключающее ИЛИ (xor). Результат число

[12] | побитовое ИЛИ (or). Результат число

[13] &&   логическое И (and). Результат сравнения 1=true 0=false

[14] || логические ИЛИ (or). Результат сравнения 1=true 0=false

[15] Тернарный (троичный) условный оператор

  выражение ? выражение_если_истина : выражение_если_ложь

Пример **((A=i==4 ? j : k))** - Если i=4, то A=j, иначе A=k.

[16] = присвоение A=3

Битовые и следующие операции работают через **let** и ((...)), но НЕ РАБОТАЮТ ПРИ ОПИСАНИИ ЧЕРЕЗ **declare -i**

[17] A*=3 аналог **A=$((A*3))**

[17] A/=3 аналог **A=$((A/3))**

[17] A%=3 аналог A**=$((A%3))**

[17] A+=3 (работает при declare -i) аналог **A=$((A+3))**

[17] A-=3 аналог **A=$((A-3))**

[17] A<<=3 аналог **A=$((A*2**3))** Битовый сдвиг переменной на 3 бита влево без переноса

[17] A>>=3 аналог A=$((A/2**3)) Битовый сдвиг переменной на 3 бита влево без переноса

[17] A&=3 аналог побитового and **A=$((A & 3))**

[17] A^=3 аналог побитового xor **A=$((A ^ 3))**

[17] A|=3 аналог побитового or A=$((A | 3))

[18] , разделитель арифметических операций. Например:
  ((a=5,b=6,c++)) это аналог **((a=5));((b=6));((c++))**

Таблица логических операций

Операнд1	Логика	Операнд2	Результат
	!	true	false
	!	false	true
true	&&	true	true
false	&&	true	false
true	&&	false	false
false	&&	false	false
true	||	true	true
false	||	true	true
true	||	false	true
false	||	false	false

Коды возврата математической подсистемы.

В математической подсистеме ((...)) и в операторе **let**

false=0,

НЕ 0=true

А в **bash** все наоборот 0=true, НЕ 0=false.

При завершении оператора **let** или ((...)) значения кодов приводятся в соответствие.

Если последнее выражение или значение оператора **let** или ((...)) было нулевым (false для математической подсистемы) выставляется errorlevel=1 (для **bash** это тоже false).

А если последнее значение оператора **let** или ((...)) было не нулевым (true для математической подсистемы) выставляется errorlevel=0 (для **bash** это тоже true).

Оптимизация по скорости вычислений

Возможны два варианта присвоения значению переменной результатов математической операции:
((A=12/4))
A=$((12/4))
Первый вариант быстрее ~50 %.

То же относится к различным математическим действиям:
((A=12/4, B=12/3))
((A=12/4)); ((B=12/3))

Первый вариант быстрее на 25-30%

Генерация псевдослучайных чисел.

В bash есть две переменные для генерации случайных чисел RANDOM и SRANDOM .

RANDOM выдает псевдослучайные числа в диапазоне 0-32767. Причем следующее число зависит от предыдущего. Таким образом, если инициализировать переменную RANDOM одним и тем же числом - можно повторить всю последовательность чисел.

Если сбросить переменную RANDOM с помощью команды unset, она теряет свойства генерации псевдослучайных чисел.

Переменная SRANDOM получает псевдослучайные числа с помощью устройства /dev/urandom или arc4random. Переменная выдает псевдослучайные числа в диапазоне 0-4294967295 Эту переменную бесполезно инициализировать.

Повторить последовательность чисел SRANDOM практически невозможно. Но если ее сбросить помощью команды unset, она также теряет свойства генерации псевдослучайных чисел.

Для нормирования диапазона псевдослучайных чисел используют формулу

((A=RANDOM % D + N)) , где D - количество числе в диапазоне, N - минимальное значение.

Для кубика диапазон 6 - минимальное число 1 .

((A=RANDOM % 6 + 1))

 # Для правильной генерации псевдослучайных чисел переменной RANDOM, можно пользоваться, если требуемый диапазон случайных чисел меньше, чем 32767 .

Если нет необходимости в повторении случайной последовательности, используйте переменную SRANDOM

Задание 17.

Условие: При выполнении задания использовать только изученные средства.

1. Получить "случайное" число номера дня для второй половины месяца, считая, что в месяце 30 дней.

2. Получить "случайный" шестизначный код подтверждения в диапазоне от 100000 до 999999 включительно.

3. Получить "случайное" число в диапазоне от -321 до +123 включительно.

4. Получить "случайное" число, в диапазоне от 0 до текущего дня месяца включительно.

5. Переменная B8 содержит восьмеричное число в виде строки "333", вывести на экран восьмеричное и десятичное значение переменной.

6. Переменная A содержит строку "ABCD" Преобразовать значение переменной A из шестнадцатеричного числа в десятичное. При преобразовании использовать конкатенацию .

Решить устно, затем проверить, расписать последовательность действий в математический аналог. Указать, какие числа будут сравниваться.

7. J=7 ; echo $((J++ <++J))

8. J=7: echo $((J--<7))

9. K=8; ((M=--K)); echo $((K<9 && M==K))

10. K=8; ((M=--K,M+=K)); echo $((M<9 || M==++K))

11. ((K=8,J=7,M=K<9?K:J,M<J))

18 Массивы.

Массив (array) - это индексированный набор переменных одного типа.

В bash массивы бывают двух видов: обычные (индексные) и ассоциативные.

В старых версиях **bash** и других оболочках массивов ассоциативного типа может и не быть.

Все массивы в **bash** одномерные. Любой многомерный массив всегда можно реализовать на базе одномерного.

Любая переменная может быть индексным массивом, даже без специального описания.

Ассоциативные массивы.

Ассоциативные массивы представляют собой набор пар ключ-значение. Причем ключ - уникальная строка для всего массива. Значения элементов могут повторяться.

Например: Массив SNILS, заполненный данными сотрудников организации мог бы иметь вид **SNILS[SN_NUM]="Фамилия ИО"** .

Для IT больше подойдет пример **CONFIG[PARAM]=VALUE**

Индексные массивы.

Обычный (индексный) массив представляет собой тот же самый ассоциативный массив, но вместо ключа используют число. Обычный массив, фактически, является нумерованным списком.

Если индекс обычного массива не указать, следующий индекс будет присвоен автоматически максимальный_индекс_массива+1.

 # Если индекс массива не задавался НУМЕРАЦИЯ ВЕДЕТСЯ С НУЛЯ!

Значения элементов массива, как и обычные переменные, могут быть целые или строковые.

Инициализация индексного массива может имеет вид:

```
ARR=(Value1 Value2 Value3)
#или
ARR2[1]=Value4
ARR2[2]=Value5
ARR2[3]=Value6
```

```
# А это инициализация с объединением массивов
ARR3=("${ARR[@]}" "${ARR2[@]}")
```

Пример с объявлением:

```
# описание и инициализация обычного массива.
declare  -a TEST_ARR=(Мама мыла раму "сегодня вечером")
declare  -ai INT_ARR=(0 1 2 4 8 16 32 64 128)
```

Индексы обычного массива должны быть положительными.

Отрицательные индексы указывают на обратный порядок перебора элементов массива.

Например: **${TEST_ARR[-1]}** указывает на последний элемент массива (-2 предпоследний)

Инициализация ассоциативного массива может иметь вид:

```
declare -A ARR
ARR=(Key1 Value1 Key2 Value2 Key3 Value3)

declare -A SNILS=(N123 "Ivanov A.B." N234 "Petrov C.D.")

# или более понятная и предпочтительная форма
ARR=([Key1]=Value1 [Key2]=Value2 [Key3]=Value3)
SNILS[N345]="Vasin I.O."

SNILS+=([N5234]="Petrovsky C.D." [N5123]="Ivanovsky A.B." )
```

Обратите внимание на последнюю команду. С помощью нее можно добавить данные к уже существующему массиву. Если вы забудете поставить "+" – будет создан новый массив из двух указанных значений.

Обращение к элементам массива ${ARR[N]} - где ARR - имя массива, N - индекс или ключ.

Если фигурные скобки можно было применять для отделения переменных друг от друга, для лучшей читаемости, то **с массивами применение фигурных скобок обязательно.**

Индекс и значения - всегда заданы парой. Не бывает значения без индекса и наоборот. Но иногда значение может быть пустой строкой.

Использование значений массива похоже на использование переменных:

```
echo ${ARR[2]} # раму
A=${ARR[3]} # сегодня вечером
B=${SNILS[N123]} # Ivanov A.B.
```

${ArrayName[@]} - полный список значений массива.
${!ArrayName[@]} - полный список индексов массива.

 # В полных списках значений и индексов массива порядок может быть произвольным.

${#ArrayName[@]} - число элементов массива.
${#ArrayName[N]} - длина строки элемента массива с индексом|ключом N

unset 'ArrayName[N]' Для удаления определенного элемента массива. Одинарные кавычки обязательны.

unset ARR - удаление массива целиком

Напомню, посмотреть описание и полный список значений и индексов можно командой **declare -p** . Например:

declare -p SNILS

Работа с элементами массива производится медленнее, чем с обычными переменными, но может быть быстрее, чем с файлами.

Иногда не нужно считывать все данные в массив (например файл) Часто можно обрабатывать данные последовательно, по одной строке.

 # Применение массивов может быть оправдано, в нескольких случаях:

1. Если обращение к файлу будет происходить очень часто, или получение данных занимает много времени. (кэширование).

2. Если требуется произвольный (а не последовательный) доступ к данным.

Обычно для обработки массивов обычно используют циклы.

Задание 18.

Условие: При выполнении задания циклы не использовать. Все переменные должны быть описаны.

1. Создайте массив с именем FRUITS, содержащий следующие элементы: яблоко, банан, вишня, слива, абрикос.

2. Выведите на экран весь массив FRUITS целиком.

3. Выведите на экран второй элемент массива FRUITS.

4. Добавьте в конец массива элемент лимон.

5. Выведите на экран количество элементов в массиве FRUITS.

6. Удалите из массива элемент с индексом 1.

7. Выведите на экран все элементы массива FRUITS после удаления с помощью declare.

8. Создайте новый массив BIN с элементами: 1, 2, 4, 8, 16.

9. Выведите сумму всех элементов массива BIN (без циклов, используя арифметику bash).

10. Объедините два массива FRUITS и BIN в один массив FRUITS и выведите его на экран с помощью **declare**.

19 Условные операторы.

В **bash** есть много разных вариантов, как проверить условие и выполнить действия в зависимости от этого условия. С одним Вы уже сталкивались.

1. Проверка кода завершения (errorlevel) с помощью **&&** и **||** (был изучен в Главе 10

2. Оператор **test** + проверка кода завершения (errorlevel)

3. Оператор [...] (это сокращенная форма оператора test

4 Оператор [[...]] - то же, что и п.3 , но расширенная версия (рекомендуется)

5 if then elif else fi (рекомендуется)

6 case esac

7.select in do done (То ли проверка, то ли цикл, непонятно)

В данном курсе мы рассмотрим первые 5, как наиболее часто встречающиеся и достаточные для написания любых программ. Кто хочет - пп6,7 может изучить самостоятельною. Там все просто.

test Это и оператор bash и программа linux помощью которой можно проверять различные условия. Результатом проверки является errorlevel, который далее можно использовать.

0=true 1=false

Файловые проверки:

-b file – истина, если file существует и является специальным блочным устройством.

-c file – истина, если file существует и символьное устройство.

-d file – истина, если file существует и является каталогом.

-e file – истина, если file существует. Любой файл (файл, каталог, устройство, link...)

-f file – истина, если file существует и является обычным файлом.

-g file — истина, если file существует и имеет установленным групповой идентификатор (set-group-id).

-k file — истина, если file имеет установленным «sticky» бит.

-L file — истина, если file существует и является символьной ссылкой.

-p file — истина, если file существует и является именованным каналом (pipe).

-r file — истина, если file существует и его можно прочитать.

-s file — истина, если file существует и имеет размер больше, чем ноль.

-S file — истина, если file существует и является сокетом.

-t [fd] — истина, если fd открыт на терминале. Если fd пропущен, по умолчанию 1 (стандартное устройство вывода).

-u file — истина, если file существует и имеет установленным бит пользователя (set-user-id).

-w file — истина, если file существует и доступен для записи.

-x file — истина, если file существует и исполняемый.

-O file — истина, если file существует и его владелец имеет эффективный идентификатор пользователя.

-G file — истина, если file существует и его владелец имеет эффективный идентификатор группы.

file1 -nt file2 — истина, если file1 новее (дата модификации), чем file2.

file1 -ot file2 — истина, если file1 старее, чем file2.

file1 -ef file2 — истина, если file1 и file2 имеют то же устройство и номер inode.

Строковые проверки:

-z string — истина, если длина string равна нулю.

-n string — истина, если длина string не ноль.

или то же самое test $VAR — истина, если VAR не "".

string1 = string2 – истина, если строки равны.

или string1 == string2 – истина, если строки равны.

string1 != string2 – истина, если строки не равны.

Целочисленные проверки:

INTEGER1 -eq INTEGER2 – истина, если INTEGER1 = INTEGER2

INTEGER1 -ge INTEGER2– истина, если INTEGER1 >= INTEGER2

INTEGER1 -gt INTEGER2– истина, если INTEGER1 > INTEGER2

INTEGER1 -le INTEGER2– истина, если INTEGER1 <= INTEGER2

INTEGER1 -lt INTEGER2– истина, если INTEGER1 < INTEGER2

INTEGER1 -ne INTEGER2 – истина, если INTEGER1 != INTEGER2

! expr – истина, если выражение expr ложь, и наоборот.

Например:

```
test -d "/home" &&
      echo 'Каталог существует' ||
      echo 'Каталог НЕ существует'

test ! -d "/home" && echo 'Каталог НЕ существует'
# или так
! test -d "/home" && echo 'Каталог НЕ существует'
```

 # Команды с отрицанием работают правильно. но хуже читаются. Если возможно - откажитесь от отрицания.

Например:

```
test  -d "/home" || echo 'Каталог НЕ существует'
```

*# Операторы && и || обрабатывают код возврата, а не результат условия. Т.Е. если вместо **test** набрать команду **teeeeest** в предыдущем примере, то появится сообщение, что 'Каталог НЕ существует', хотя ошибка вызвана тем, что не существует команда **teeeeest** !!!*

Команда [ ] является аналогом команды **test**, поддерживаются все те же самые условия, просто немного другая форма записи.

```
[ -d "/home" ]  && echo 'Каталог существует' ||
                   echo 'Каталог НЕ существует'
#Здесь с отрицанием все аналогично
[ ! -d "/home" ]  && echo 'Каталог НЕ существует' ||
                     echo 'Каталог существует'
```

```
! [ -d "/home" ]  && echo 'Каталог НЕ существует' ||
                     echo 'Каталог существует'
```

 # Обращайте внимание на пробелы. отсутствие одного пробела, и программа работает не правильно!!!

Команды test и [...] - совместимы с оболочкой sh.

Команда [[...]] - аналогична команде [...] И здесь будут работать те же условия. Кроме этого добавлены несколько новых. Результатом проверки также является errorlevel, 0=true 1=false

[[STRING1 < STRING2]] истина, если STRING1 меньше STRING2

[[STRING1 > STRING2]] истина, если STRING1 больше STRING2

Обратите внимание - отсутствуют проверки <= и >=. Их всегда можно получить с помощью других условий сравнения. Например $A <="5" это то же самое что ! $A >"5"

Строки сравниваются с лексикографическом порядке. Если кратко - то сравнивают первые буквы, потом вторые... 0<1<2...< ... < a<b<c...< ...A<B<C... Если все буквы одинаковые, но одно слово закончилось раньше - значит оно меньше. Пример:

"ABC" < "ABD", "AbC" < "ABc" , "ABC" < "ABCD"

"1000" < "200" < "30" < "4"

Именно по этой причине для целочисленных проверок существуют отдельные условия.

[[STRING1 =~ STRING2]] истина, если STRING1 содержит STRING2 (может быть regex)

 # Обратите внимание, если STRING1 не пустая строка, а STRING2 - пустая строка, то условие будет истина!!!

Кроме этого команда [[...]] допускает возможность объединения нескольких условий с помощью логического И (**&&**) или логического ИЛИ (||).

Результат логического И - истина, только тогда, когда ВСЕ условия истина.

Результат логического ИЛИ - истина, когда, Хотя бы одно условия истина.

Следующие выражения истинны:

[[1 < 2 && 3 < 4]] (1<2 - истина, 3 < 4 - истина, объединение по И.)

[[1 < 2 || 3 < 2]] (1<2 - истина, 3 < 2 -ложь, но объединение по ИЛИ)

При объединении по И, если первое условие ложь, остальные даже не проверяются/вычисляются, потому что результатом все-равно будет ложь.

То же самое касается при объединении по ИЛИ, если первое условие истина, остальные даже не проверяются/вычисляются , потому что результатом все-равно будет истина.

 # Это может быть очень важным, если в проверках участвуют не переменные или константы, а функции. Вторая функция не будет выполнена.

Форма записи при логическом объединении может быть несколько иной:

[[1 < 2]] && [[3 < 4]]

[[1 < 2]] || [[3 < 2]]

if - then - [elif] - [else] - fi

Это правильный аналог prg1 && prg2 || prg3

if тоже проверяет errorlevel, но в отличие от изученных проверок результат выполнения оператора сравнения **if then elif else fi** не зависит от результатов групп команд внутри оператора, следовательно он понятнее и логичнее.

Без этого условного оператора вполне можно обойтись, однако он здорово улучшает читаемость кода программы, поэтому рекомендую пользоваться именно им.

```
if [[ условие1 ]]; then
  группа команд 1
elif [[ условие 2]]; then
 группа команд 2
else
  группа команд 3
fi
```

Работает он следующим образом: Если условие1 истина - выполняется группа команд1. Остальные условия игнорируются, и выполняется команда после оператора fi

Если условие1 ложно - выполняется проверка следующего условия.

Если условие2 истина - выполняется группа команд2. Остальные условия игнорируются...

блоков **elif-then** может быть несколько.

Если ни одно условие не было истинным, выполняется группа команд блока **else**. Который трактуется, как "в любом другом случае"

Блоки **elif** и **else** являются необязательными.

Группу команд внутри блока принято смещать пробелами или табуляциями для визуального выделения блока.

 # Группа команд должна содержать хотя бы одну команду. Если по логике выполнения блок должен остаться пустым - используйте :

Группа команд может состоять из операторов **if** с другими условиями (вложенный **if**). Но каждый **if** должен заканчиваться своим **fi** .

Пример ограничения значения переменной A в диапазоне 0-10:

```
#!/bin/bash
if [[ $A -gt 10 ]] ;then #если A>10
  A=10
elif [ $A -lt 0 ]] ;then # A<0
  A=0
fi
```

Пример определения знака числа A:

```
#!/bin/bash
if [[ $A -gt 0 ]] ;then # A>0
  SIGN_A=1
elif [[ $A -lt 0 ]] ;then #A<0
  SIGN_A=-1
else # A=0 # иначе 0
  SIGN_A=0
fi
```

Задание 19:

Условие: Всё задание выполнить единым скриптом.

0. Используя exec , перенаправить вывод stdout и stderr скрипта в лог-файл с названием имя_скрипта.log. Результат проверки заданий вывести в формате "OKN" или "ERRN", где N - номер задания. Результат проверить при истинных и ложных условиях.

1. Написать проверку, что число является шестизначным.

2. Написать проверку, что переменная YES = "Y"

3. Написать проверку переменная YES1 = "Y" или "y"

4. Написать проверку, что значение переменной NO содержится в строке "NnHн".

5. Написать проверку, что A=5, B - отрицательное и переменная YES="n" или "N" . При выполнении всех условий выдать в stderr сообщение об ошибке.

6. Написать проверку, что переменная TEST >= "TEST". Использовать отрицание. Условие должно быть одно.

7. То же, что п.7, но вместо отрицания использовать else.

8. Создать пустой файл /tmp/test.txt, проверить, что файл создан. В случае успеха заполнить его любой информацией и провести проверку, что этот файл не "пустой" Записать сообщение об этом в файл /tmp/test.log

9. Проверить, может ли оператор if анализировать код завершения программ без всяких условий в скобках. Например if ping -c3 yandex.ru ...

20 Циклы. Часть 1.

Циклы предназначены для выполнения повторяющихся действий.

Циклы бывают двух видов.

1. Цикл выполняется пока не будет выполнено определенное условие (возможно бесконечно). Это циклы типа

while do [continue] [break] done
until do [continue] [break] done

2. Циклы у которых количество итераций изначально ограничено. Это циклы типа

for in do [continue] [break] done
for (( )) do [continue] [break] done.

Впрочем, последний цикл можно отнести к обеим группам.

Основная переменная, которая изменяется в цикле, называется переменной цикла.

Операторы, выполняемые внутри цикла, называются телом цикла.

Цикл **while** будет выполняться, пока условие - истина

```
while [ условие ] ; do
  тело цикла
  [continue]
  [break]
done
```

ПРИМЕР:

```
declare -i I=5
while [[ $I -gt 0 ]];do
  echo $I
  ((I--))
done
```

Данный цикл выведет на экран 5 чисел от 5 до 1

Какие числа выведет цикл будет зависеть, от того, в каком месте модифицируется переменная I.

Именно эта переменная называется переменной цикла. Модифицируем цикл:

```
declare -i I=5
while [[ $I -gt 0 ]];do
  ((I--))
  echo $I
done
```

Данный цикл выведет на экран 5 чисел от 4 до 0

Хотя при конструировании данных циклов в условии можно использовать четкое условие остановки -ne 0, лучше задавать условия типа больше - меньше. Иногда это поможет избежать "зацикливания":

Например: Если в последнем цикле заменить ((I--)) на ((I-=2)) (чтобы цикл выдавал числа с шагом 2) При условии остановки -ne 0, и нечетном начальном значении, цикл "минует" значение 0 и превратится в бесконечный.

Циклы можно немного упростить:

```
declare -i I=5
while ((I--));do
   echo $I
done
#4 3 2 1 0
```

Цикл ниже выведет значения от 4 до 1

```
declare -i I=5
while ((--I));do
   echo $I
done
#4 3 2 1
```

Цикл **until** будет выполняться, пока условие - ложь

НАПРИМЕР:

```
declare -i I=5
until [[ $I -le 0 ]];do
  ((I--))
  echo $I
done
# 4 3 2 1 0
```

Цикл **until** можно преобразовать в цикл **while** отрицанием условия. И наоборот. Например:

while [condition] эквивалент **until** ! [condition]
until [condition] эквивалент **while** ! [condition]

Иногда бывает нужно однократно пропустить выполнение тела цикла и перейти к следующему значению переменной цикла.

Оператор **continue** [n] прерывает выполнение тела цикла и передает управление оператору **done**. Необязательный параметр n позволяет одновременно прервать выполнение нескольких (числом n) вложенных циклов. По-умолчанию n=1.

```
declare -i I=5
while [[ $I -gt 0 ]];do
  ((I--))
  [[ $I -eq 3 ]] && continue
  echo $I
done
# 4 2 1 0 Значение 3 пропущено.
```

Оператор **break** [n] просто прерывает выполнение цикла передает управление оператору, следующему за оператором **done**. Необязательный параметр n позволяет одновременно прервать выполнение нескольких (числом n) вложенных циклов. По-умолчанию n=1.

```
declare -i I=5
while [[ $I -gt 0 ]];do
  ((I--))
  if [[ $I -eq 2 ]] ;then
    break
  fi
  echo $I
done
# 4 3
```

Циклы отлично подходят для обработки индексных(обычных) массивов., ведь изменяя переменную цикла в нужном диапазоне, можно перебрать все элементы массива: ${A[$I]}

На основании цикла **while** можно построить бесконечный цикл, если в качестве условия прерывания цикла поставить **true**. Иногда команду **true** в бесконечном цикле заменяют на двоеточие : (в bash : - аналог команды **true**)

```
declare -i I=5
while true;do
  ((I--))
  if [[ $I -le 0 ]] ;then
    break
  fi
  echo $I
done
```

Этот цикл построен на базе бесконечного.

Циклы могут быть вложенными. В этом случае у них должны быть разные переменные цикла.

Начальные значения переменные внутренних циклов должны задаваться каждый раз перед началом выполнения цикла:

```
I=3
while ((--I)); do
  J=3
  while ((--J));do
    echo $I $J
  done
done
#2 2
#2 1
#1 2
#1 1
```

Тело цикла принято смещать отступами, чтобы было проще видеть, где цикл начинается, и где он заканчивается.

Задание 20.

Написать скрипт генератор квази-слов с параметрами.

Условие: Все переменные должны быть описаны. Если параметр вне диапазона - использовать ближайшее допустимое значение.

1. Массив A содержит русские гласные буквы (любые 5).

2. Массив B содержит русские согласные буквы (любые 15, кроме знаков).

3. Слог состоит из случайных согласной и гласной. Именно в этом порядке.

4. Параметр 1 - число слогов в слове от 1 до 5 включительно.

5. Параметр 2 - число генерируемых слов от 3 до 10 включительно.

6. Слоги накапливаются в переменной WORD.

7. Слова выводятся на экран в stdout.

8. Параметр 3 (необязательный) определяет вероятность (в процентах) последовательности букв в слоге. Если задан параметр 3, с заданной вероятностью слог состоит из гласной и согласной (меняется порядок) от 0 до 99 включительно.
   Подсказка SRANDOM % 100 < $3

9. Если скрипт запущен без параметров - выдать на stderr краткую справку (несколько строк) по использованию скрипта и параметров. Код завершения при этом должен быть 1 (help exit)

21 Циклы. Часть 2.

Циклы. Часть 1 Глава #20

Цикл **for** по списку.

Цикл очень простой и логичный, рассказывать не буду. Просто приведу примеры.

```
for I in 1 2 3 4 5;do
  echo $I
done
```

или так

```
for A in 2 "Листья" 1024 "Звездный водоём" ;do
  echo $A
done
```

Список может быть любым. Можно перебрать параметры. Я уже писал, что для описания списка параметров есть две переменные $* и $@. Чем они отличаются проще понять на примере: Выполните скрипт:

```
#!/bin/bash
set -- A1 "A2 A3"
for P in $@; do
    echo $P
done
#вывод:
#A1
#A2
#A3
for P in "$@"; do
    echo $P
done
#вывод:
#A1
#A2 A3
for P in $*; do
    echo $P
done
#вывод:
#A1
#A2
#A3
for P in "$*"; do
    echo $P
done
#вывод:
#A1 A2 A3
```

Обратите внимание если переменные $@ или $* без кавычек - параметры будут разбиты по пробелам, если "$@" - параметры, заключенные в

кавычки могут содержать пробелы. Если переменная "$*" - все параметры воспринимаются, как одна строка.

Списки можно формировать с помощью расширений. Подробнее расширения рассмотрены в Главе #5

```
for K in {1..100..2}; do
  echo $K
done # 1 3 5 ... 99
```

{1..100..2} это расширение. {First..Last ..Step} First- начальное значение, Last - конечное Step - шаг. Для числовых - если First задать 01 - все значения будут иметь одинаковую длину.

```
for K in {01..100..5}; do
  echo $K
done # 001 006 011 ... 096
```

Обратите внимание три цифры из-за того, что Last=100 , Хотя последнее значение 096.

```
for J in {Z..A..-2}; do
  echo $J
done # Z X V... B
```

Да, могут быть и буквы. И порядок может быть обратным

```
for J in {0..7}{0..7}; do
  echo $J
done # Пример цикла в восьмеричной системе
```

У расширения только одна проблема. **Значения можно задавать только константами.**

Аналогично для построения списка можно использовать вывод команды **seq** (**man seq**). Или любой другой команды.

```
for I in $(seq 100  -10 0); do
  echo $I
done # 100 90 ...0
```

Обратите внимание, у **seq** инкремент - средний параметр. Кстати, вот здесь уже можно использовать переменные пределы:

```
declare -i FIRST=0 LAST=10 STEP=2
for I in $(seq -w $FIRST $STEP $LAST); do
  echo $I
done # 00 02 ... 10
```

Кроме этого есть еще один цикл for (c-style) Такие циклы используют в языке C.

```
for ((i=1;i<=10;++i));do
  echo $((10-i))
done # 9 8 7 6...0
```

У него сложное описание, если хотите - смотрите в manual. Но если просто:

первое выражение - задание начального значения. i=1
второе выражение - условие продолжения. Пока истина - будет работать i<=10
третье - изменение переменной цикла. i++ или ++i принципиальной разницы нет.

Этот цикл немного похож на цикл while.

Эти циклы также могут быть вложенными.

Пример таблицы умножения :

```
for ((i=0;i<10;i++));do
  for ((j=0;j<10;j++));do
    echo "$i X $j = $((i*j))"
  done
done
```

В циклах **for** тоже можно использовать **continue** и **break**. Их использование ничем не отличается от других циклов.

При использовании операторов **continue** и **break** можно указать уровень вложенности.

Например **continue** 2 передаст управление оператору **done** не текущего, а более верхнего "родительского" цикла.(второй уровень от текущего) Оператор **break** 2 работает аналогично. Пример:

```
for ((i=1;i<10;i++));do
  for ((j=1;j<10;j++));do
    echo "$i X $j = $((i*j))"
    [[ $j -eq 3 ]] && break 2
  done
done
#1 X 1 = 1
#1 X 2 = 2
#1 X 3 = 3
```

Основное отличие циклов **for in** и **while** В том, что цикл **while** приступает к работе сразу, и обрабатывает данные последовательно.

Цикл **for in** сначала "строит" весь список в памяти, а потом начинает работать.

При ОЧЕНЬ больших значениях списка у цикла **for in** могут возникать две проблемы:

1. Большая задержка до первого такта цикла (список нужно вначале построить).

2. Если список очень большой - он может не поместиться в памяти. И тогда будет задействован swap, что приведет к сильному замедлению работы. При очень-очень больших списках может возникнуть ошибка нехватки памяти. Как-правило, на не очень больших списках (размер списка меньше половины свободной памяти) Цикл `for in` выполняется несколько быстрее.

Задание 21.

Задан скрипт.

```
#!/bin/bash
for ((i=1;i<17;++i));do
  for ((j=13;j>3;j--));do
    echo "$i X $j = $((i*j))"
    [[ $i -eq 2 ]] && break 2
  done
done
```

1 Переписать его используя:

```
while
until
for in
```

Есть скрипт для вывода таблицы умножения:

```
#!/bin/bash
for ((i=0;i<10;i++));do
  for ((j=0;j<10;j++));do
    echo "$i X $j = $((i*j))"
  done
done
```

2. Изменить скрипт, чтобы вывод был точно такой же, как в изначальном скрипте, но использовать циклы с убыванием переменной цикла. (9 8...1)

3. Самостоятельно разобраться, что делает следующий скрипт и проставить комментарии к каждой строке.

```
#!/bin/bash
declare -i I=0 K L
while [[ $I -lt 99 ]];do
  ((I++))
  K=I/10
  L=I%10
  ((K*L)) &&  echo "$K X $L = $((K*L))"
done
```

4. Изменить скрипт п3, используя оператор **if**, для улучшения читаемости. Заменить ((K*L)) двумя условиями с объединением по ИЛИ

5. Изменить скрипт п3, используя оператор **continue**.

6. Написать скрипт для создания в переменной VAR числа, состоящего из 20 случайных символов в диапазоне от 2 до 8 включительно. Вывести на экран значение VAR.

7. Написать скрипт для создания в переменной VAR числа, состоящего из 20 случайных символов в диапазоне от 2 до 8 включительно, кроме символов 4 и 6. Вывести на экран значение VAR.

22 Циклы. Часть 3.

Циклы. Часть 1 Глава #20
Циклы. Часть 2 Глава #21

Обработка массивов. Практика.

Немного поработаем с массивами, тем более, что циклами Вы владеете.

Классика жанра - поиск минимального и максимального значения в массиве:

```
#!/bin/bash
# Число элементов массива 0 -(MAX_ARR-1)
declare -i MAX_ARR=10
# числовой диапазон данных в массиве 0-(NUM-1)
declare -i NUM=10
#MIN,MAX - минимальное и максимальное число в массиве
declare -i MIN MAX I=$MAX_ARR
#ARR массив элементов
declare -ai ARR

# заполняем массив случайными числами
while ((I--));do
  ARR+=(RANDOM)
done

echo весь массив ARR:
echo ${ARR[@]}

((I=MAX_ARR-1))
#принимаем в качестве минимального или максимального
# значение последнего (а можно и первого) элемента массива
MIN=${ARR[$I]}
MAX=$MIN

#перебираем все остальные элементы,
while ((I--)) ; do
#Если текущий элемент меньше принятого минимума - найден
новый минимум
  ((MIN=(ARR[I]<MIN) ? ARR[I] : MIN))
# Аналогично
  ((MAX=(ARR[I]>MAX) ? ARR[I] : MAX))
# любуемся поиском
#  echo $I, ${ARR[$I]} $MIN $MAX

done
echo "MIN=$MIN   MAX=$MAX"
```

Сортировка массива методом перестановки (Не самый лучший вариант, но идеально подходит для изучения массивов). Текстовый алгоритм:

Допустим есть массив (3 5 2 0)

Позиция (индекс) 0 1 2 3

1. Выбираем нулевой элемент массива за минимум и запоминаем позицию минимального элемента.

2. Сравниваем, начиная со второго и до последнего, в случае необходимости, обновляя минимум и его текущую позицию.

3. После окончания цикла у нас есть минимальное значение массива (0) и его позиция (3).

4. Производим перестановку выбранного и минимального элементов массива, Используя временную переменную. T=A; A=B; B=T (перестановка A и B) После этого в нулевой позиции - минимальный элемент.

5 .Получаем массив (0 5 2 3).

6. Выбираем следующий элемент и повторяем все пункты:

7. Получаем массив (0 2 5 3).

...

N. (0 2 3 5) массив отсортирован по возрастанию..

Понятно, что выбирать элемент и его позицию удобно с помощью цикла.

Минимум-максимум тоже удобно искать с помощью цикла. Получается два вложенных цикла:

внешний i от 0 до (n-1), где n - последний номер в массиве

внутренний j от (i+1) до n

Чтобы чтобы отсортировать по убыванию - нужно искать максимальный элемент.

Задание 22.

Техническое условие:

Написать скрипт сортировки массива с именованными параметрами.

Запуск скрипта имеет вид: **./scr.sh -a N -b M [-s A|D]**

-a - число элементов массива 10-10000

-b - Максимальное значение элемента массива 3-100 (от 0..3 до 0..100)

-s - направление сортировки (A или a) -по возрастанию (D или d) по убыванию.

если параметров нет - вывести справку.

Если задан хоть один не правильный параметр - вывести ошибку на stderr и прекратить работу.

Последний параметр не обязательный. Если не задан - сортировка по возрастанию.

Параметры могут быть переставлены местами.

Параметры нельзя объединять.

Ключ и значение ключа задаются строго через пробел.

Если задан ключ - задано и значение.

Если значение параметра вне заданного диапазона - задать ближайшее.

Замечания к заданию 22

В данном задании никакие regex не нужны.

1. Приведу пример разбора параметров.

```
while (($#>0));do
  if [[ $1 == "-a" ]];then
    shift
# теперь $1 содержит значение параметра
    A=$1
# здесь же обрабатываем ограничения
    ((A<3)) && A=3
    ((A>100)) && A=100
  elif [[ $1 == "-b" ]];then
```

```
...
```

2. Несмотря на то, что значение параметра -s - текстовое, внутри программы проще привести его к целому 0 - по возрастанию 1 по убыванию. На мой взгляд целыми проще оперировать (иногда их можно использовать в арифметике)

3. Хорошо было бы разделить сортировку и вывод массива. В этом случае проще использовать код повторно. Вот сортировка. Вот вывод.

4. Сортировать проще/быстрее без учета параметра -s Например всегда по возрастанию. А вот выводить данные с учетом -s. В этом случае нужно просто поменять три параметра цикла (начало, конец, инкремент) 0..N..+1 или N..0..-1

И буквально одним if сортировка по возрастанию превращается в сортировку по убыванию.

Собственно задание 22

1. Заполнить массив случайными числами и вывести его в файл /tmp/a1.txt

2. Произвести сортировку и вывести в файл /tmp/a2.txt

3. Вывести в stderr время сортировки.

4. файл /tmp/a1.txt отсортировать с помощью sort и передать на stdin diff для проверки правильности сортировки с /tmp/a2.txt

Если файлы отличаются - выдать сообщение об ошибке на stderr.

Вопрос. Можно ли с помощью поиска минимального элемента отсортировать массив по возрастанию?

23 Оператор read.

Оператор **read** читает данные с **stdin** или из другого файлового дескриптора и записывает их в одну, несколько переменных, или в индексный массив. Кроме этого этот оператор используется для организации интерактива с пользователем.

read [-ers] [-a aname] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u fd] [name …]

Оператор действительно очень нужный и полезный, поэтому рассмотрим его ключи полностью.

Самый простой вариант **read** без параметров.

При его исполнении программа остановится и будет ожидать поступления данных от stdin. В интерактивном режиме - ввод данных с клавиатуры. Обычно ввод данных заканчивается нажатием <enter>

Такой вариант можно применять для временной остановки выполнения скрипта, (вроде режима отладки) для просмотра текущего значения переменных в сложных случаях, и продолжения по <enter>

```
for i in {1..10};do
  declare -p i VAR1 VAR7
  read
done
```

Если в операторе **read** переменная не указана - введенные данные будут записаны в переменную REPLY .

read VAR1 VAR2 VAR3 - можно сразу ввести значения нескольких переменных. Для разделителя используется переменная системная IFS.

read -a ARR введенные данные будут записаны в индексный массив. Массив автоматически очищается при заполнении оператором **read**.

read -p "var1=" VAR1 - при запросе ввода будет выведена подсказка, чтобы пользователю было понятно, что от него требуется вводить. При использовании для остановки выполнения скрипта **read -p "Press ENTER"**

read -e LINE - режим ввода строки. Обычно применяется для ввода имен реальных файлов. В этом режиме работает автодополнение. Если начать набирать /tm и нажать <Tab> то система дополнит путь. Или не дополнит, если выбор не однозначен. Если при неоднозначном выборе еще раз нажать <Tab> - система предложит варианты. Вобщем, все, как в командной строке.

read -e -i "text" ключ i выводит текст по-умолчанию в строку ввода. Но пользователь сможет его изменить. Например, путь к каталогу, начальные данные ip адреса или имя пользователя по умолчанию.

 # Обратите внимание, ключ -i работает только с ключом -e

read -s применяют для ввода паролей. Набираемый текст не будет отображаться на экране.

read -r ключ блокирует действие эскейп-последовательностей. В этом режиме бэкслэш \ будет просто буквой. Без этого ключа все эскейп последовательности работают правильно. Например, если строка длинная, то ее, как и в коде, можно продолжить со следующей строки поставив в конце символ \ . С ключем -r ввод будет завершен, даже если на конце \

read -d разделитель_строк. По -умолчанию - перевод строки, но можно заменить на любой символ. Например -d " " .

read -t timeout задать время ожидания ввода в секундах. Допускает вещественные значения, например 0.01. По истечении timeout система считает, что ввод закончен и в переменную попадет то, что пользователь успел набрать (ну, или что было передано на stdin)

read -n nchars задает максимальное количество введенных символов. Как только нужное количество получено - ввод считается завершенным и выполнение программы продолжится автоматически. Но выполнение программы можно продолжить, нажав <enter>, даже максимальное число не было получено.

read -N nchars задает точное количество требуемых символов. Как только нужное количество получено - ввод считается завершенным и выполнение программы продолжится автоматически. Выполнение программы нельзя продолжить, нажав <enter>, если требуемое число символов не было получено. Просто <enter> посчитают еще одним символом.

read -u fd - работа с файловым дескриптором, отличным от stdin. Этот ключ должен применяться для обработки файлов с помощью циклов while, если несколько файлов обрабатывается вложенными циклами. Но не только в этом случае.

Практические замечания.

С вводом букв, цифр, символов проблем не возникает, но при получении кодов функциональных клавиш (F1-F12), всяких там стрелок, клавиш на правой части клавиатуры возникают проблемы. Начнем с того, что перечисленные коды будут зависеть от терминала в котором Вы работаете. Мало того, эти клавиши могут генерить многосимвольные

последовательности. Т.е. программа может работать в графической среде, но не будет работать без нее, или при логоне через **tty1** или по **ssh**.

Старайтесь не использовать функциональные клавиши в интерактивных скриптах.

Также стоит упомянуть, что оператор **read** очень тесно связан с системной переменной IFS, которая задает символы, являющиеся разделителями полей при чтении. Переменную IFS можно задать двумя способами: Классическим:

IFS="," - в этом случае переменная будет изменена для всех операторов в исполняемом скрипте.

И персональным:

IFS="," read ... В этом случае переменная будет изменена только для конкретного оператора read.

Стоит отметить, что данный способ касается не только переменной IFS и не только оператора **read**. таким способом можно задать любые переменные для почти любого оператора/команды (для оператора **set** нельзя). Через пробел можно задать значения несколько переменных. Например:

VAR1=value1 VAR2=value2 command

обе переменные будут доступны выполняемой command. И будут восстановлены к исходному состоянию после выполнения command.

Задание 23.

Совет. При разработке программ с бесконечными циклами на этапе разработки их заменяют циклами с достаточно большим, но ограниченным количеством циклов. И после того, как все заработает - заменяют цикл на бесконечный.

1. Привести оператор для ввода трех любых переменных с подсказкой.

2. Запросить ввод пароля.

3. Запросить ввод почтового индекса. По-умолчанию использовать 103123

4. Дублировать п3 с ограничением времени ввода 10 секунд.

5. Организовать ввод номера мобильного телефона.

6. Запросить у пользователя заполнение массива FIO. Массив должен быть предварительно описан.

7. Протестировать использование разделителя полей.

8. Сконструировать оператор, с функцией "Press any key to continue", и временем ожидания не более 12 минут.

9. Написать скрипт, который будет в бесконечном цикле выдавать примерно один символ "#" в секунду и прерываться он должен при нажатии на клавиатуре на этот символ. Символы должны выводиться в одну строку, после вывода 20 символа строка (но не экран) должна очищаться. На всякий случай напомню, прерывание бесконечных циклов CTRL-C

24 Циклы. Часть 4.

Циклы. Часть 1 Глава #20
Циклы. Часть 2 Глава #21
Циклы. Часть 3 Глава #22

Иногда бывает необходимо обработать информацию из файла или поток конвейера Для этого тоже можно использовать циклы.

Например:

```
# заполним файл с помощью цикла
:>/tmp/file1.txt # Создали пустой файл.
for i in {1..5};do #заполним файл значениями
  echo "data $i" >>"/tmp/file1.txt"
done

# считаем файл и выведем на экран с помощью цикла №1
while read -e REC;do
  echo $REC
done <"/tmp/file1.txt"

# считаем файл и выведем на экран с помощью цикла №2
cat "/tmp/file1.txt" | while read -e REC;do
  echo $REC
done
```

Обратите внимание, вместо **cat**, помощью цикла можно обработать любую информацию в конвейере.

#Напомню, что при работе в конвейере каждая программа запускается в отдельной подоболочке (subshell). В данном случае цикл является частью конвейера, поэтому переменные, полученные/измененные внутри цикла могут быть недоступны в основной оболочке.

Если перед выполнением конвейера с циклом выполнить команду

shopt -s lastpipe то последняя команда конвейера будет выполнена в окружении текущей оболочки со всеми переменными.

Циклы **for** тоже можно использовать при работе с файлами:

```
for REC in $(< /tmp/file1.txt);do
    echo $REC
done

# или
for REC in $(cat /tmp/file1.txt);do
  read -e REC
done
```

Но, возможно, придется задействовать переменную IFS, иначе результат может быть не таким, как Вы ожидаете.

В отличие от цикла **while** , в цикле **for** сначала весь файл будет считан в память. Это может вызвать проблемы при больших размерах файлов.

А теперь самое главное отличие циклов при обработке файлов.

ВНИМАТЕЛЬНО РАЗБЕРИТЕ КОД. ЭТО ВАЖНО:

```
#!/bin/bash
declare FILE='/tmp/file.txt' # переменная с именем файла
seq 1 3 >$FILE
echo --FOR--
for i in $(<$FILE);do # цикл по записям файла
  echo $i  #печать строки
  read -p 'press ENTER' # ожидание нажатия клавиши
done

echo --WHILE--

while read i;do # цикл по записям файла
  echo $i #печать строки
  read -p 'press ENTER' # ожидание нажатия клавиши
done <$FILE
```

Хотя тела циклов одинаковые - результат существенно разный:

```
--FOR--
1
press ENTER
2
press ENTER
3
press ENTER
--WHILE--
1
3
```

Второй цикл вообще не производит остановки для нажатия ENTER.

А происходит это потому, что цикл for in читает строку файла и записывает в переменную i. А второй цикл читает файл и передает на на stdin, где его построчно читает read i.

Если внутри цикла еще какой-нибудь оператор читает stdin - (read -p 'press ENTER') данные передаются и ему.

Результат вы видели.

Исправить ошибку в цикле **while** (чтобы результат совпадал с циклом **for in**) можно путем изменения дескриптора (с stdin при чтении файла на любой свободный.)

```
#!/bin/bash
declare FILE='/tmp/file.txt' # переменная с именем файла
seq 1 3 >$FILE # создали файл и записали содержимое

exec 3<$FILE # открыли дескриптор 3 для чтения файла

while read -u 3 i;do # цикл по файлу с дескриптором 3
  echo $i #печать строки
  read -p  'press ENTER' # ожидание нажатия клавиши (stdin)
done

exec 3>&- # на всякий пожарный закрыли дескриптор 3
```

При использовании вложенных циклов **while** для обработки файлов, обязательно используйте файловые дескрипторы, отличные от **stdin**.

Задание 24.

Условие: Использовать только изученные средства.

1. В каталоге /tmp создать файл IP.txt, состоящий из 10000 строк. Каждая строка представляет IP адрес 192.168.X.Y . X,У , - "случайные". X в диапазоне от 0 до 3 включительно, Y в диапазоне от 1 до 255 включительно.

2. Вывести на экран только уникальные записи IP.txt (Считать записи в ассоциативный массив с индексом IP и вывести на экран) Подсчитать количество уникальных записей).

3. Подсчитать максимальное количество одинаковых IP (При записи в ассоциативный массив инкрементировать значение)

4. Вывести на экран все IP, которые встречаются максимальное и минимальное количество раз.

5. Подсчитать количество IP, с одинаковой подсетью /24 (IP у которых первые три октета совпадают Например 192.168.0 ... 192.168.3)

6. Разделить уникальные (из п.2) IP с одинаковой подсетью /24 по файлам 192.168.0.0.txt ...192.168.3.0.txt в каталоге /tmp. Считать, что начальное количество подсетей неизвестно. (информацию о номере подсети получать из самого IP)

7. Вывести на экран всех пользователей системы (файл /etc/passwd) с UID < 1000 (поле 3) и их shell по-умолчанию (поле 7).

Примечание: Информация файла /etc/passwd довольно чувствительная ее нежелательно публиковать в сети.

25 Работа со строками и массивами. Часть 1.

В bash, если обычную переменную не описать, она является строковой. И для работы со строковыми переменными в последних версиях bash добавили очень много полезных возможностей.

Раньше для этого нужно было использовать внешние программы (**wc**,**sed**...) а теперь все в bash прямо из коробки. Думаете стало легче - нисколько.

У меня такое чувство, что разработчики bash до сих пор используют процессор 8086 и 640 кб памяти, потому что экономят каждый байт. Может кто-нибудь подскажет, как запомнить то, что сегодня будем изучать.

**${VAR:-default}** - если переменная VAR не задана (имеет пустое значение "") то результат будет строка default. Эта конструкция применяется при обработке параметров ($1, $2...), для задания значений по-умолчанию.

Например ${1:-127} вернет 127, если $1 не определена.

```
VAR=""; echo ${VAR:-'RA'} # RA
VAR=12 ; echo ${VAR:-'RA'} # 12
```

Обратите внимание - сама переменная VAR не изменяется.

${VAR:+default}** - Конструкция, противоположная **${VAR:-default}. Если переменная задана - будет выдана строка "default" - если не задана - пустая строка. Переменная VAR не изменяется.

${VAR:=default} Если переменная не задана, ей будет присвоено значение.

```
VAR=""; : ${VAR:='RA'}; echo $VAR #RA
VAR=12; : ${VAR:='RA'}; echo $VAR #12
```

Обратите внимание, в начале строки после ; (Является разделителем операторов.) стоит ":" В данном случае это аналог "пустого" (который ничего не делает) оператора. **${VAR:='RA'}** - рассматривается как его параметр. В данном случае он нужен, чтобы **${VAR:=0}** не была интерпретирована, как команда bash.

${VAR:?Error message} - При работе скрипта, если переменная VAR имеет пустое значение - программа прервется, на stderr будет выдано указанное сообщение об ошибке "bash: VAR: Error message"

Можно применять при отладке или для контроля правильности задания параметров.

${VAR:FROM} - конструкция извлекает символы из строки VAR от символа FROM до конца строки.

Символы нумеруются с нуля!!!

${VAR:FROM:LEN} -Эта конструкция извлекает определенный диапазон из строки. LEN - число извлекаемых символов.

Фактически строка рассматриваются как массив, где каждый элемент - один символ. Числовые переменные в этой конструкции являются строкой, состоящей из цифр.

```
VAR=12345abcde
#   0123456789 позиция
echo ${VAR:3} #45abcde
echo ${VAR:0} #12345abcde
echo ${VAR: -3} #cde (нумерация с конца)
#или
echo ${VAR:(-3)} #cde (нумерация с конца)
```

Обратите внимание - между ":" и "-" нужен пробел, чтобы не получилась конструкция ${VAR:-default}, или отрицательное число должно быть в скобках.

```
VAR=12345abcde
#   0123456789
echo ${VAR:3:0} # ""
echo ${VAR:3:2} # 45
echo ${VAR:3: -2} # 45abc (от третьего до второго с конца)
echo ${VAR: -3:0} # ""
echo ${VAR: -3:2} # cd
echo ${VAR: -3: -2} # c
```

Эта конструкция работает по-разному, для обычных переменных и массивов. Для массивов - аналогично, но будут извлекаться не символы, а элементы массива целиком

```
ARR=(123 45a bc de)
#    0   1   2  3
echo ${ARR[@]:3} # de
echo ${ARR[@]:0} #123 45a bc de
echo ${ARR[@]: -3} # 45a bc de  И опять следим за пробелом.
echo ${ARR[@]: -3:0} # ""
echo ${ARR[@]: -3: -2} # ошибка bash:
            # -2: заданное подстрокой выражение меньше нуля
```

Любой, один определенный элемент массива рассматривается как обычная переменная

```
ARR=(123 45a bc de)
#    0   1   2  3
echo ${ARR[1]:2} #a (подстрока первого элемента массива с
                 # символа 2)
echo ${ARR:1} #23   Полный аналог echo ${ARR[0]:1}
```

Если у переменной, описанной или заданной как массив не указать индекс - считается, что индекс равен нулю.

Параметры тоже можно рассматривать как массив, с именем @

И применять те же самые конструкции. Важно только помнить, что $0 - нулевой параметр всегда указывает название самого скрипта. (или bash, если выполнение идет в интерактивном режиме) поэтому задать нулевой параметр с помощью команды **set** не получится.

```
set -- 123 45a bc de
echo ${@:1:2} # 123 45a
```

все остальное - аналогично массивам.

#ЗАМЕЧАНИЕ: Все приведенные выше конструкции при работе с массивами справедливы только для индексных массивов. Применение их с ассоциативными массивами может привести к неопределенным результатам.

${#VAR} - длина переменной (число символов).

${#ARR[@]} -для любых массивов - число элементов.

${#@} - для параметров -число параметров начиная от $1.

${VAR#prefix} - удаление из переменной префикса минимальной длинны

${VAR##prefix}- удаление из переменной префикса максимальной длинны

${VAR%suffix} - удаление из переменной суффикса минимальной длинны

${VAR%%suffix}- удаление из переменной суффикса максимальной длинны

Пример:

```
VAR=/var/log/test.log.gz
echo {VAR#*/} #var/log/test.log.gz
echo ${VAR##*/} #test.log.gz
echo ${VAR%.*} #/var/log/test.log
echo ${VAR%%.*} #/var/log/test
```

echo ${!prefix*} # возвращает список всех переменных, начинающихся с prefix

echo ${!prefix@} # аналогично предыдущему.

Последние две конструкции могут использоваться при отладке.

Задание 25.

1. Выделить из переменной $0 каталог, имя файла с расширением, имя файла без расширения, и расширение.

2. Описать и инициализировать переменную VIRSH четверостишьем из любого цензурного стихотворения. (в переменной должны присутствовать переносы строки).Добавить вначале и в конце VIRSH по одному пробелу. Вывести переменную VIRSH на экран, в виде стихотворения.

3. В переменной VIRSH Заменить любые символы, кроме букв пробелами, преобразовать в верхний регистр. Результат записать в переменную VIRSH_FLT, вывести на экран.
4. В переменной VIRSH_FLT Заменять два пробела подряд одним пробелом, пока произведена хоть одна замена. Результат записать в переменную VIRSH_1SP . Вывести на экран.
5. В переменной VIRSH_1SP удалить начальный и конечный пробелы, если существуют, результат записать в переменную VIRSH_TRM. Вывести на экран, показав, что концевых пробелов нет.
6. Разбить VIRSH_TRM на слова и распечатать их на экране, по два в строке, используя set без shift
7. Разбить VIRSH_TRM на слова и распечатать их на экране, по два в строке, используя read -a
8. Разбить VIRSH_TRM на слова и распечатать их на экране, по два в строке, используя цикл for in
9. Разбить VIRSH_TRM на слова и распечатать их на экране, по два в строке, используя цикл while и оператор read и добавить при печати длину каждого слова.

26 Работа со строками и массивами. Часть 2.

Часть 1. Глава #25

 # Замечание. При замене значение самой переменной не изменяется. Выводится только результат замены.

${VAR/FROM/TO} # Заменить первое вхождение текста FROM в переменной VAR на TO

${VAR//FROM/TO}# Заменить все вхождения текста FROM в переменной VAR на TO

${VAR/#FROM/TO}# Заменить префикс FROM в переменной VAR на TO

${VAR/%FROM/TO}# Заменить суффикс FROM в переменной VAR на TO

Пример:

```
#!/bin/bash
declare VAR=012012012012
echo $VAR #исходная перемененная
echo ${VAR/0/A} #A12012012012 первое вхождение
echo ${VAR//1/A} #0A20A20A20A2 все
echo ${VAR/#1/A} #012012012012 замена не произведена,
поскольку искомая строка не в начале текста (не префикс)
echo ${VAR/#012/A} # A012012012 перфикс
echo ${VAR/%012/A} # 012012012A суффикс
```

${VAR^pattern} - замена первой буквы в переменной VAR на большую.

${VAR^^pattern}- замена всех букв в переменной VAR на большую.

${VAR,pattern}- замена первой буквы в переменной VAR на маленькую.

${VAR,,pattern}- замена всех букв в переменной VAR на маленькую.

Вообще-то pattern может изменить поведение этой конструкции, но ценность этого в реальных задачах стремится к нулю.

Если в качестве переменной указать массив **${ARR[@]^pattern}**, то замена будет произведена для каждого элемента массива, как для переменной.

Пример:

```
declare VAR="МамА МЫлА РамУ"
echo ${VAR,} # мамА МЫлА РамУ
echo ${VAR,,} # мама мыла раму
echo ${VAR,,*M} # мамА мЫлА РамУ
```

Преобразование к верхнему регистру - аналогично.

Есть еще конструкции для работы с переменными. Не хотел их рассматривать, но вскользь упомяну.

${VAR@U} - результат значение переменной в верхнем регистре.

${VAR@u} - результат значение первой буквы переменной в верхнем регистре.

${VAR@L} - результат значение переменной в нижнем регистре...

${VAR@A}** - аналог **echo VAR=\'$VAR\', может использоваться для печати значения переменных.

${VAR@a} - результат - флаги, которые были использованы в операторе declare при описании переменной.

${VAR@Q} - результат значение переменной в одинарных кавычках.

Есть еще расширения E P K k, но зачем они нужны, я так и не смог разобраться. Может умные люди подскажут, как их использовать в реальной жизни.

Задание 26.

Условие: Написать скрипты без использования внешних программ.

1. head.sh -n N filename1 filename2...Программа должна выводить N первых строк. Число N>0 может быть задано с пробелом и без. Если filename не задан - читается /dev/stdin . Если файлов несколько - вначале на stderr вывести имя текущего файла.

2. grep.sh "text" filename1 filename2... программа должна вывести строки, в которых встречается text. Если filename не задан - читается /dev/stdin . Если файлов несколько - вначале на stderr вывести имя текущего файла.

3. tail.sh -n N filename1 filename2...Аналогично п.1. Программа должна выводить N последних строк. Не использовать массив более N элементов для кэширования.

Подсказка алгоритма к п.3:

Есть файл, и Вам нужны последние 10 строк из этого файла.

Но длина файла неизвестна. Вы берете массив из 10 элементов и считываете строки файла в него.

После 10 строки массив заполнился. Поэтому новую строку записываете в самый первый элемент. Следующую - во второй...

```
declare -i REC_NUM=0 начальный номер записи
declare -i N число элементов массива (кэша)
# в цикле
((CACHE_POS=REC_NUM++%N)) # Текущий номер записи кэша
```

Когда файл закончился - Ваш массив содержит последние 10 строк. Вам нужно только придумать, как правильно считать строки из массива. И подумать, что будет, если в файле окажется меньше 10 строк.

27 Функции.

Функции предназначены для группировки часто используемого кода.

Под часто используемым следует понимать не только частоту использования кода в одном скрипте, но и использование одного и того же кода в разных скриптах.

Если у Вас есть группы кода, кочующие из программы в программу имеет смысл выделить их в отдельные функции, эти функции даже не обязательно копировать в скрипт. Можно сохранить их в отдельный файл, и в скрипт вставить инструкцию подключения файла:

source my_cool_func.inc

Здесь my_cool_func.inc - имя файла, содержащего описания ваших функций. Есть еще сокращенная форма команды source - "."

. my_cool_func.inc

Пользоваться рекомендую **source**, потому что она более понятна.

source можно использовать не только для функций, а для вставки любого кода **bash**.

Функции bash можно было бы назвать именованной группировкой. Потому что тело функции, аналогично группировкам, может быть заключено в фигурные или круглые скобки.

Аналогично группировкам - если тело функции заключено в фигурные скобки - функция выполняется в основной оболочке, если в круглых - в подоболочке. При этом все переменные внутри функции - локальные.

Выполнение функции в подоболочке требует дополнительных ресурсов, поэтому выполняется медленнее.

Описать функцию можно несколькими способами:

```
function myfun(){
  echo $0
}
# или
myfun(){
  echo $0
}
# однострочный вариант
myfun1(){ echo $0;}
```

Обратите внимание, при описании в одну строку: пробел после { и знак ";" перед закрывающей скобкой } обязателен. Вобщем, все как при группировке.

Открывающаяся фигурная скобка может быть и на следующей строке.

 # Функция должна быть описана перед ее использованием.

Все переменные доступные в скрипте видны и внутри функции.

Если нужно описать переменную, доступную только внутри функции - есть оператор **local** . Действие переменной будет распространяться на данную функцию и все дочерние функции.

Ключи оператора **local** такие же, как и у оператора **declare**.

local можно использовать только внутри функций.

Если использовать оператор **declare** внутри функции - также будет описана локальная переменная Например:

```
#!/bin/bash
declare A=global # задали глобальную переменную
function test(){ #описали функцию
   echo $A # Вывели значение  переменной
   declare A=local #описали локальную переменную
   echo $A # Вывели значение  переменной
}
test # вызвали функцию
echo $A # Вывели значение  переменной
# результат
#global
#local
#global
```

 # В качестве результата функция возвращает код ошибки последнего выполненного оператора.

Кроме этого есть оператор **return [n]** . Он прерывает выполнение функции и возвращает errorlevel=n . По-умолчанию 0. (аналог **exit**, но для функций)

В отличие от группировки у функции могут быть параметры. Их использование полностью соответствует параметрам скрипта. $0 - имя скрипта $1... позиционные параметры функции.

Вызывается функция так же, как обычная команда или программа:

Пример:

```
#!/bin/bash
function MUL(){
  A=${1:-0}
  B=${2:-0}
  echo $(($A*$B))
}
echo MUL 2 3
# или
echo $(MUL 2 3)
```

Задание 27.

Создать функцию:

1. ABS для вычисления модуля целого числа.

2. SIGN для вычисления знака числа

3. DICE для имитации кубика "⚀" "⚁" "⚂" "⚃" "⚄" "⚅" (функция должна выдавать изображение грани кубика а не цифру. Коды граней U+2680 .. U+2685

4. REV для вывода текстовой переменной справа налево (аналог программы **rev**)

5. CHECK_IP4 для проверки правильности IP (0.0.0.0 - 255.255.255.255) set не использовать. 192.168.001.001 - не правильный 192.168.1.1 - правильный.

6. CHECK_CODE - для ввода с клавиатуры кода подтверждения (задан первым параметром функции) возврат 0- правильно 255 -нет.

7. Написать скрипт детского симметричного шифрования путем замены символов р-а к-и д-е м-о н-у л-я. Функция шифрования должна использовать ассоциативный массив замен. Нормально шифроваться должны как большие, так и маленькие буквы. Тестовый пример Рая мыла Пуделя в кино. => Арл оыяр Пнедял в икум => Рая мыла Пуделя в кино. Шифруемая фраза должна быть указана в параметрах программы. В случае отсутствия - осуществить ввод с клавиатуры.

28 Обработка даты и времени.

В bash есть несколько способов измерение времени:

1. time [-p] [конвейер].

Вывод времени, потраченного на выполнение конвейера.(конвейер может состоять и из одной команды)

Команда выполняет конвейер и выводит значения реального, пользовательского и системного времени ЦП, потраченного на выполнения конвейера.

Параметр -p показать значения времени в формате Posix.

errorlevel соответствует коду завершения конвейера.

Для форматирования выходных данных используется значение переменной TIMEFORMAT в следующем виде:

%% - Литерал %.

%[p][l]R Реальное время в секундах.

%[p][l]U Затраченное процессором время в пользовательском режиме.

%[p][l]S Затраченное процессором время в системном режиме.

%P Процент загрузки CPU, рассчитанный как (%U + %S) / %R.

Параметр p -точность 0-3 знака после запятой.

l - "длинный" формат.

По-умолчанию используется $'\nreal\t%3lR\nuser\t%3lU\nsys\t%3lS'

Если нужно восстановить значение по-умолчанию - выполните команду unset TIMEFORMAT

Разделитель дробной части зависит от языковых настроек (. или ,)

Если с помощью команды time требуется измерить время участка кода, содержащего несколько операторов можно воспользоваться группировкой {}

2. Переменная SECONDS показывает, сколько секунд прошло с начала запуска скрипта или момента обнуления значения переменной. Обычно

используется для хронометража блока кода скрипта с точностью до секунды. Например:

```
SECONDS=0
# здесь блок длительного кода
echo $SECONDS
```

3. Переменная EPOCHSECONDS - показывает число секунд, прошедших с начала эпохи UNIX (01.01.1970 00:00:00 UTC)

#Обратите внимание, число секунд не зависит от часового пояса. Это может быть очень полезным, при сравнении времени данных, расположенных в разных часовых поясах.

Задать начальное значение переменной не получится, но после выполнения оператора unset EPOCHSECONDS - переменная становится обычной, неописанной переменной.

4.Переменная EPOCHREALTIME - аналогична предыдущей, но секунды с точностью до микросекунды.

 # Обратите внимание, разделитель дробной части может быть различным, в зависимости от языковых настроек.

Задать начальное значение переменной не получится, но после выполнения оператора unset EPOCHREALTIME - переменная становится обычной, неописанной переменной.

Для работы со временем можно использовать простые арифметические операции, ведь все знают, что 1 сутки=24часа, 1 час=60 минут, 1 минута=60 секунд.

5. Использование встроенного оператора **printf** и формата даты-времени Здесь приведу пример. Форматы в скобках () практически совпадают с форматами команды **date** .

Если не указать переменную или указать -1 - будет использовано текущая дата/время. Если указать -2 - будет использовано время начала работы этого экземпляра bash.

```
printf '%(%Y%m%d-%H%M%S)T' 1744379369
# 20250411-164929
printf '%(%Y%m%d-%H%M%S)T'
# текущие дата-время
printf '%(%Y%m%d-%H%M%S)T' -2
# дата время начала сессии bash
```

6. Использование программы **date**.

Программа очень мощная, но мы рассмотрим только часть возможностей.

 # Ключи программы (функционал) могут отличаться в разных ОС. Проверяйте перед использованием.

date - без параметров выводит дату и время

date +FORMAT - вывод даты в определенном формате. Разных форматов - доволно много. Приводить их здесь все нет никакого смысла, потому что они есть в man или wiki. Приведу наиболее популярные:

```
date +%Y%m%d-%H%M%S # 20250103-213608 timestamp в
                    #человекочитаемом формате
date +%s # то же, что и echo $EPOCHSECONDS
```

Использование с ключом -d позволяет оперировать с заданными (не текущими) датами.

Преобразование EPOCHSECONDS в дату При использовании формата можно получить все, что нужно.

```
date -d@1735929548 # Пт 03 янв 2025 21:39:08 MSK
```

Кроме этого можно проводить определенные действия с датой, используя ключевые слова "day", "week" "month", "year", "hour", "min", "sec"

```
date -d"-1 day" # Чт 02 янв 2025 21:52:15 MSK
                # - текущая дата минус один день.

date -d"+2 week" # Пт 17 янв 2025 21:53:48 MSK
                 # - текущая дата плюс две недели

date -d"20250114 12:30:10" #- вывести конкретную дату/время.
                           #( можно добавить формат)

date -d"20250201+1 month-1 day" # Пт 28 фев 2025 00:00:00
                             # MSK (последний день февраля)
```

Подробно форматы расписывать не буду. Посмотрите man date

Обращу внимание на некоторые опции форматов:

- (минус) не выравнивать поле по ширине

_ (подчеркивание) выравнивать пробелами

0 (ноль) выравнивать нулями

+выравнивать пробелами и '+' если год более 4х цифр

^ использовать верхний регистр, если возможно

Использовать обратный регистр, если возможно.

Например: дата 12025.01.08 (обратите внимание на год) символ = для лучшего восприятия границ форматов.

```
date -d"120250108" "+%+Y=%-m=%_d=%#Z"
#+12025=1= 8=msk
```

год с плюсом (потому что больше 4 цифр) месяц без нуля, день без нуля, но с пробелом, временная зона в нижнем регистре.

Использование этих опций в bash иногда необходимо, потому что **числа, начинающиеся с нуля bash считает восьмеричными**.

Например при выполнении:

```
declare -i DAY=$(date -d"20250109" +%d)
```

возникает ошибка:

bash: declare: 09: слишком большое значение для основания (неверный маркер «09»)

Ошибки можно избежать заменой формата:

```
declare -i DAY=$(date -d"20250109" +%-d)
```

И еще одно важное замечание о датах и времени.

Команды **date** и **printf**, когда время не касается секунд эпохи UNIX, работают в локальной временной зоне.

Если необходимо получить время в другом часовом поясе - проще всего это сделать с помощью задания "вре́менной" переменной TZ.

Например:

```
TZ='UTC' date
# Пн 14 апр 2025 16:15:57 UTC

TZ='Asia/Irkutsk' printf '%(%Y%m%d-%H%M%S)T\n'
# 20250415-001621
```

Если использовать "постоянную" переменную TZ - ее нужно описать с ключом -x (export).

Полный список временных зон на системах с systemd можно получить с помощью команды **timedatectl list-timezones**

Кроме этого временные зоны можно задавать относительно UTC (часовой пояс 0).

Например:

```
TZ='UTC-3' printf '%(%Y%m%d-%H%M%S)T\n' #MSK
# 20250414-191710

TZ='UTC+5' printf '%(%Y%m%d-%H%M%S)T\n' # тоже какая-то зона
#20250414-111731
```

Тут главное не ошибиться со знаком. :-)

 # Использовать рекомендую "Текстовые зоны". Потому что они учитывают перевод на летнее/зимнее время.

При обработке даты-времени нужно быть очень аккуратным, потому что изменение времени всего на одну секунду может привести к изменению не только минуты или часа, но и года, в некоторых случаях.

Задание 28

Написать скрипт.

1 Есть слова "лестница" "темнота" "район" "куст" "художник" "политика" "сержант" "вещество" "база" "техника" "девица" "медведь" "собрание" "испытание" "кран" "местность".

Создать файл /tmp/text.txt. Строка состоит timestamp(EPOCHREALTIME) и трех слов. Разделитель колонок - табуляция. Файл должен быть максимальной длины. В файле не должно быть строк со словами в той же последовательности. (строки с полями 2-4 уникальные по файлу) Слова в строке могут повторяться.

Пример строки

```
1736606037,151461   лестница   темнота   лестница - правильно
1736606037,151506   лестница   темнота   лестница - не правильно (повтор)
```

2 Написать функцию для измерения времени REALTIME с помощью переменной EPOCHREALTIME.

bc, **awk**, и другие программы работы с вещественной арифметикой не использовать. В качестве разделителя целой и дробной частей использовать "." при выводе. В исходных переменных в качестве разделителя использовать не цифру. Пример:

```
ERT1=$EPOCHREALTIME

#operators

ERT2=$EPOCHREALTIME
REALTIME $ERT1 $ERT2
# 2.123456
```

3 Измерить время выполнения с помощью time и REALTIME

3.1. создания файла

3.2 Вывести на экран измеренное время и время создания файла по timestamp первой и последней строки файла.

3.3. шифрования файла /tmp/text.txt в файл /tmp/text.enc с помощью детского алгоритма из предыдущего задания. Разделитель полей может стать пробел.

4 Написать второй скрипт, для вывода файла /tmp/text.txt на экран, с преобразованием первой колонки в "человекочитаемый" вид например 20250111-101127.123456

29 Printf оператор форматированного вывода.

Довольно сложный и очень нужный оператор. Аналог **echo** на стероидах.

printf [-v переменная] формат [аргументы]

Без указания формата просто выводит в **stdout** значения аргументов. В отличие от **echo** не переводит строку после печати. (как **echo -n**)

Кроме этого умеет результат записать сразу в переменную.

printf -v NOW '%(%Y%m%d-%H%M%S)T'

запишет значение timestamp в удобном формате в переменную NOW .

Обычно формат представлен в виде **%[флаги][ширина][.точность]тип**

Флаги:

- Выравнивание по левому краю поля.

+ Всегда указывать знак числа.

" " Пробел - выравнивание осуществляется пробелами

При выводе шестнадцатеричных чисел в начале будет 0x или 0X восьмеричные будут начинаться с нуля.

0 числа в пределах ширины поля будут дополнены ведущими нулями.

Ширина -минимальная ширина поля в символах.

Если не указать - будет выбрано оптимальное значение.

Если указано меньше, чем нужно - информация все-равно будет выведена полностью. Информация не потеряется.

Если указать * - то число знаков будет указано в аргументах.

.Точность

Для форматов d,i,u,o,x,X - число выведенных символов будет не менее указанного количества.

Для форматов a, A, e, E, f, F - минимальное количество символов после запятой.

Для формата g, G - минимальное количество значащих цифр.

Для формата s - максимальное число выведенных символов.

*#Примечание: В формате указывается именно точка, даже если в стандарте языка для отделения дробной части используется запятая. Если указать * - то число знаков будет указано в аргументах.*

*# Примечание: Несмотря на то, что **bash** работает только с целыми числами оператор **printf** умеет работать и с вещественными числами. Вещественные числа могут встречаться при обработке вывода других программ и преобразовании научно-технических отчетов.*

Тип - тип выводимого значения.

d или i - десятичное целое число со знаком.

u - десятичное целое число без знака.

o - восьмеричное целое число без знака.

x или X шестнадцатеричное целое число без знака. x - для маленьких букв abcdef - X -для больших.

f или F - для вещественных чисел. В **bash** оба формата идентичны.

e или E - для вещественных чисел в нормальной (научной) форме. Отличие в регистре буквы E выводимого числа.

g или G - автоматический выбор оптимального формата между f F или e E.

a или A -число с плавающей точкой в шестнадцатеричном виде.

s - вывод строкового выражения.

% - для вывода символа % (поскольку любой формат начинается с %, для вывода самого символа нужно писать %%).

Кроме этого, в **bash**, оператор **printf** дополнен еще некоторыми форматами.

%b - аналогичен формату %s, но интерпретирует эскейп - последовательности аргументов. за исключением \0

%q - аналогичен формату %s, но экранирует символы, для возможности использования в командной строке.

%(fmt)T - формат даты и времени. Более подробно рассмотрен в главе про обработку дат. Глава #28

Формат fmt практически полностью совпадает с форматами команды **date**. (**man date**)

Кроме этого, в строке формата допускается использование эскейп-последовательностей. (Вы их уже изучали, но вспомнить полезно)

```
\a  звонок
\b  забой одного символа
\c  подавление дальнейшего вывода
\e  символ escape
\E  символ escape
\f  перевод строки (радость Маяковского)
\n  новая строка
\r  возврат каретки
\t  горизонтальная табуляция
\v  вертикальная табуляция
\\  backslash
```

\0nnn печать символа ASCII код которого задан в восьмеричном формате.

\xHH печать символа ASCII код которого задан в шестнадцатеричном формате. (Одна или 2 hex-цифры)

\uHHHH печать уникод-символа код которого задан в шестнадцатеричном формате. (Одна, две или четыре hex-цифры)

\UHHHHHHHH печать уникод-символа код которого задан в шестнадцатеричном формате. (Одна, две, 4 или 8 hex-цифр)

Если формат один, а аргументов несколько - все аргументы будут выведены по одному формату.

Если форматов несколько и аргументов столько же каждый аргумент будет выведен по своему формату (по счету).

Если форматов больше, чем аргументов - данные все-равно будут распечатаны по числу форматов. Вместо отсутствующих аргументов будет использовано "пустая строка" или в случае числового формата - 0.

Если форматов меньше числа аргументов - после использования последнего формата - форматы начнут повторно использоваться начиная с первого.

Примеры:

```
# Тильда (~) указана для визуализации начала и конца области
печати.
declare A=400; B=-200
printf "~A=%i~B=%i~\n" $A $B
#~A=400~B=-200~
printf "~A=%6i~B=%6i~\n" $A $B
#~A=   400~B=  -200~
printf "~A=%-6i~B=%-6i~\n" $A $B
#~A=400    ~B=-200  ~
printf "~A=%+6i~B=%+6i~\n" $A $B
#~A=  +400~B=  -200~
printf "~A=%6.4i~B=%6.4i~\n" $A $B
#~A=  0400~B= -0200~
printf "~A=%06i~B=%06i~\n" $A $B
#~A=000400~B=-00200

printf "~A=%*.4i~B=%6.*i~\n" 6 $A 4 $B
#~A=  0400~B= -0200~
# ширина задана в параметрах
# можно использовать переменную.

printf "~A=%u~B=%u~\n" $A $B
#~A=400~B=18446744073709551416
printf "~A=%o~B=%o~\n" $A $B
#~A=620~B=1777777777777777777470~
printf "~A=%#o~B=%#o~\n" $A $B
#~A=0620~B=01777777777777777777470~
printf "~A=%#x~B=%#X~\n" $A $B
#~A=0x190~B=0XFFFFFFFFFFFFFF38~
# Отрицательные преобразованы

printf "~A=%f~B=%F~\n" $A $B
#~A=400,000000~B=-200,000000~
printf "~A=%e~B=%E~\n" $A $B
#~A=4,000000e+02~B=-2,000000E+02~
printf "~A=%a~B=%A~\n" $A $B
#~A=0xc,8p+5~B=-0XC,8P+4~
printf "~A=%s~B=%s~\n"  $A $B
```

```
#~A=400~B=-200~

printf "%d %e " 100 200 300 400 500
# 100 2,000000e+02 300 4,000000e+02 500 0,000000e+00
# форматов 2, аргументов 5
```

Иногда возникает необходимость для тестирования использовать файл из нескольких строк. С помощью оператора printf и расширений bash можно обойтись без создания файла. Например:

```
printf "%s\n" {1..20}|head -5
printf "%s\n" {1..20}|tail -5
printf "%s\n" {1..20}|wc -l
printf "%s\n" {1..20}|grep 2
```

Пример формата %q при работе с файлами:

```
declare filename='Файл с пробелами и всякой дичью > & '
printf '%q' $filename
#Файлспробеламиивсякойдичью\>\&
printf '%q' "$filename"
#Файл\ с\ пробелами\ и\ всякой\ дичью\ \>\ \&\
```

Формального задания по оператору printf не будет. Оператор printf будет использован в экзаменационной работе.

Задание 29.

Условие: Файлы создавать во временном каталоге.
Каждый пункт оформить в виде функции без параметров.
Скрипт создавать при условии, что число строк файла file1.txt не известно.

Есть слова "лестница" "темнота" "район" "куст" "художник" "политика" "сержант" "вещество" "база" "техника" "девица" "медведь" "собрание" "испытание" "кран" "местность".

1. Создать файл file1.txt, содержащий (10^5) строк из случайных комбинаций трех указанных слов.

2. Создать файл file2.txt из строк файла file1.txt, в строке должно быть слово "куст".

3. Создать файл file3.txt из строк файла file1.txt в строке должно быть слово "куст" но не должно быть слова "вещество".

4. Создать файл file4.txt из строк файла file1.txt в строке должно быть слово "медведь" и оно должно быть последним.

5. Создать файл file5.txt из строк файла file1.txt в строке должно быть слово "медведь" и рядом должно быть слово "сержант".

6. Создать файл file6.txt из строк файла file1.txt в строке должно быть слово "медведь" и в предыдущей строке должно быть слово "местность". В file6.txt нужно записать обе строки. Предыдущая строка должна существовать.

7. Создать файл file7.txt из строк файла file1.txt в строке должно быть слово "медведь", в предыдущей строке должны быть слова "местность" или "кран". В следующей строке должны быть слова "испытание" или "девица". Предыдущая и следующая строки должны существовать. В file7.txt нужно записать все три строки.

8. Если какой-нибудь из файлов "пустой" - произвести повторную генерацию, начиная с п.1

30 Обработка сигналов.

Сигналы - средство межпроцессного взаимодействия.

Сигналы представляют собой асинхронные сообщения. Они предназначены для сообщению процессу, о том, что условия выполнения изменились.

Я не буду вдаваться в подробности, остановлюсь не практике. Представьте. Ваш скрипт выполняется, а пользователь решил его прервать. Он нажимает CTRL-C. И система посылает Вашему процессу сигнал SIGINT (Сигнал прерывания). Обычно скрипт прерывается. Но бываю ситуации, когда это делать нежелательно.

Например: Скрипт создал несколько временных файлов, и перед завершением их желательно удалить. Для этого нужно написать обработку прерывания.

Есть специальный оператор **trap** для обработки сигналов.

trap [-lp] [[аргумент] сигнал ...]

аргумент - однострочный код или функция для обработки данного сигнала.

Сигналы могут обозначаться в текстовой форме или цифрой.

Список названий сигналов и их номера можно узнать по команде **trap -l**

Есть такие сигналы, обработать которые нельзя. Например (9) SIGKILL Этот сигнал применяется, когда процесс не реагирует на другие сигналы.

Послать сигнал процессу можно с помощью программы **kill** .

Например:

```
# послать SIGINT  (CTRL-C) процессу с PID  34578
 kill -2 34578

kill -9 $$ # самоубийство
```

обычный пользователь может отправлять сигналы только своим процессам. root - всем.

Если процесс был прерван по сигналу - код возврата 128+N, где N - номер сигнала.

список назначенных обработок можно узнать по команде **trap -p**

Если аргумент команды **trap** будет пустой строкой - программа просто перестанет реагировать на данный сигнал. Например:

trap '' SIGINT -запрет CTRL-C

Чтобы вернуть обработку по умолчанию используйте '-'

trap '-' SIGHUP - возврат обработки по-умолчанию для SIGHUP

Наиболее часто обрабатывают сигналы SIGHUP - закрытие терминала, SIGINT - Сигнал остановки процесса пользователем с терминала (CTRL + C) и SIGTERM - сигнал Сигнал запроса завершения процесса.

Кроме стандартный сигналов UNIX, bash позволяет перехватывать (обрабатывать) некоторые события:

EXIT - событие возникает при выходе из программы

DEBUG - обработка каждой простой программы/команды

RETURN - обработка выполняется каждый раз, когда завершается shell-функция или сценарий, выполняемые с использованием внутренних функций . или source.

ERR - обработка не анализируемых ошибок. Звучит странно, но логика есть. Ошибки, в условии оператора **test**, **[[...]]** , **[...]**, **if** или **||** обрабатываться не будут.

В качестве простого примера приведу скрипт. Обязательно сохраните его, выполните и проанализируйте.

```
#!/bin/bash
FILENAME=/tmp/sigtest.txt

# Функция, выполняемая при завершении
function ON_EXIT(){
  echo
  cat "$FILENAME"
  rm "$FILENAME"
  [[ -f "$FILENAME" ]] || echo файл "$FILENAME" удален
}
# при получении любого из перечисленных сигналов будет
выполнен выход
trap 'exit' SIGTERM SIGQUIT SIGHUP

# а вот при выходе выполним функцию ON_EXIT
trap 'ON_EXIT' EXIT
# список обработок
trap -p
```

```
# главный цикл :-)

for i in {1..6};do
  ((i==2)) && echo "Press CTRL-C"
  printf '%s\n' $i >>"$FILENAME"
  sleep 1
done
```

Данный скрипт "уберет за собой" и при "нормальном" завершении, и при указанных прерываниях.

Запускаем его, немного ждем и нажимаем CTRL_C

...

^C

1

2

файл /tmp/sigtest.txt удален

Это значит, что временный файл был, но его успешно удалили.

Если ничего не нажимать, программа все-равно выполнит обработку и удалит временный файл.

Задание 30.

Условие:Использовать только bash и свои наработки. По-возможности использовать функции.

Создать скрипт.

1 Написать функцию для создания псевдо-лог файла "/tmp/ip.log" в формате: Первое поле случайное число в диапазоне от 1577836800 до 1735689599 включительно. Второе поле строка "y.x.x.x", где x - число (0-255) y(1-255).Третье поле строка из 1-4 слогов. Слог состоит из двух **латинских** букв (гласной и согласной, гласная на первом месте слога стоит с вероятностью ~30%). Разделитель полей - пробел. Число строк файла 10^5.

2 Создать функцию для вывода /tmp/ip.log,на экран с помощью less, но первое поле преобразовать в дату-время в формате YYYYMMDD-HHMMSS в часовом поясе Калининграда.

3 Создать файл /tmp/ip.stat со статистическими данными в часовом поясе UTC:

3.1 сколько записей в каждом году,

3.2 сколько записей в каждом год-месяц

3.3 сколько записей год-месяц-день

3.4 сколько записей в среднем приходится на один год

3.5 сколько записей в среднем приходится на один месяц

3.6 сколько записей в среднем приходится на один день

4 Найти год, (год-месяц), (год-месяц-день) с максимальными и минимальными значениями.

5 В случае прерывания SIGINT удалить /tmp/ip.stat

6 В случае SIGTERM или SIGHUP -удалить все созданные файлы.

7 Сделать вывод о скорости bash при формировании больших сложных файлов.

31 Лирическое отступление: HTML

Иногда может возникнуть необходимость формировать файлы в специальных форматах, например HTML, XML, YAML или подобных. Эти файлы представляют собой обычный текст, содержащий специальные тэги.

Любой текстовый формат можно сформировать с помощью bash, просто перенаправив вывод **echo** или **printf** в файл. Хотя с использованием специализированных программ или библиотек это сделать проще.

Рассмотрим минимальный набор на примере html.

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <title>Название документа</title>
  <style>
    body {font-family: monospace;white-space:pre;}
    h2   {color: blue;}
    p    {color: red;}
    code {color: green;white-space:pre;}
  </style>
</head>
<body>
<h1>This is heading1.</h1>
<h2>This is heading2.</h2>
<h3>This is heading3.</h3>
<p>This is a paragraph.</p>
<code>a b c d     e</code>
1 2 3 4 5
 6 7 8 9 0
</body>
</html>
```

html документ состоит из нескольких основных разделов:

<html></html> - весь документ html

<head></head> - заголовок документа

<body></body> - тело документа

Обычно тэги бывают двойные - открывающий и закрывающий (как представлено выше), но бывают тэги одинарные

 - перенос строки

<hr> - горизонтальная линия.

Раздел <style></style> описывает параметры форматирования.

font-family: monospace; - предписывает браузеру выводить текст указанного раздела моноширинным шрифтом. Этот параметр позволит легче прогнозировать вывод текста, ведь все символы будут иметь одинаковую ширину. (Обычно для вывода используется пропорциональный шрифт)

white-space:pre; этот параметр предписывает браузеру не объединять несколько пробелов подряд в один, и отображать переносы строки (обычно игнорируются.)

Эти два параметра позволяют легче генерировать странички html с помощью bash.

Назначение остальных тэгов проще посмотреть. Сохраните код в файл test.html и откройте браузером.

Кроме тэгов html не отличается от обычного текстового файла, так что выполнить экзаменационное задание будет проще простого.

Как создать таблицу html описывать не буду, достаточно вбить эту фразу в поисковике.

Экзаменационные задания.

Поздравляю Всех с завершением начального курса bash!

Курс считается усвоенным, если Вы, самостоятельно, без привлечения ИИ и коллег смогли выполнить **ЛЮБОЕ** из 1-10 заданий. Задания приведены с нарастанием сложности. Таким образом Вы сможете определить свой уровень. Вам нужно написать скрипт.

Ограничения:

- **При выполнении задания не использовать оператор echo. Только printf.**
- **Не использовать внешних программ (кроме date). Только BASH.**
- **При выполнении пп7-10, календарь должен быть 4 месяца по горизонтали, 3 по вертикали.**

Советы:

- **При использовании таблиц - границу делайте прозрачной.**
- **При создании календарей месяц может состоять из 4-6 недель. Считайте, что каждый месяц состоит из 6 недель.**
- **Есть консольная программа ncal, которая создает календари. Для проверки пользуйтесь ей.**
- **Если необходимо - создайте справочник часто используемых тегов, хотя бы в виде переменных.**

Выберите номер ЗАДАНИЯ 1-10. 0 - для всех.

0. При запуске скрипта без параметров вывести справку. При необходимости проводить проверку диапазона параметров.
1. Скрипт генерации таблицы умножения от 1 до 10 включительно в текстовый файл. В виде таблицы Пифагора. Параметр имя файла.
2. Скрипт генерации таблицы умножения от 1 до 10 включительно, В виде таблицы Пифагора в файл формата html.

3. Скрипт генерации таблицы умножения от 1 до 10 включительно, В виде таблицы Пифагора в файл в виде таблицы html. (каждое число в отдельной ячейке).

4. Скрипт создания календаря одного месяца текущего года. Параметры:именованный параметр -m MONTH_NUM (по-умолчанию текущий месяц), имя файла - позиционный. Файл в формате txt.

5. Скрипт создания одного месяца текущего года. Параметры:именованный параметр -m MONTH_NUM (по-умолчанию текущий месяц), имя файла - позиционный. Файл в формате html.

6. Скрипт создания одного месяца текущего года. Параметры:именованный параметр -m MONTH_NUM (по-умолчанию текущий месяц), имя файла - позиционный. Файл в формате html (с таблицей html).

7. Создание календаря указанного года (в таблице html). именованный параметр год -y YEAR_NUM, только 2000-2099 годы (по-умолчанию текущий), файл (позиционный параметр по-умолчанию YEAR_NUM.html.).

8. Создание календаря указанного года (в таблице html) (именованный параметр год -y YEAR_NUM только 2000-2099(по-умолчанию текущий), файл (позиционный параметр по-умолчанию YEAR_NUM.html.) + прикол-режим с ключом -x8, -x16 , для отображения всех чисел и номеров месяцев в oct, hex форматах. (дни недели в bin)

9. Создание календаря указанного года (в таблице html) (именованный параметр год -y YEAR_NUM, файл (позиционный параметр по-умолчанию YEAR_NUM.html.) + прикол-режим с ключом -x8, -x16 , для отображения всех чисел и номеров месяцев в oct, hex форматах. (дни недели в bin) БЕЗ ИСПОЛЬЗОВАНИЯ ПРОГРАММЫ date, используя любую формулу календаря. Только средствами BASH. Год 1970-9999.

10. Создание календаря указанного года (в таблице html) (именованный параметр год -y YEAR_NUM, файл (позиционный параметр по-умолчанию YEAR_NUM.html.) + прикол-режим с ключом -x8, -x16 , для отображения всех чисел и номеров месяцев в oct, hex форматах. (дни недели в bin) **БЕЗ ИСПОЛЬЗОВАНИЯ ПРОГРАММЫ date и формулы календаря.** Под формулой календаря понимается вычисление или использование справочника високосных лет и/или констант числа дней в месяцах. Только средствами BASH. Год 1970-9999.

32 Ответы, решения ошибки

Ответы и решения на Задание 1

Теория здесь: #1

1. Набрать и выполнить скрипт явным запуском.

```
#!/bin/bash
clear
# comment
echo Hello world! #also comment
```

Набрал, сохранил **scr1.sh** , выполнил, **bash scr1.sh**

В результате отчистило предыдущий вывод и вывело Hello World!

2. Закомментировать команду clear. Выполнить скрипт.

```
#!/bin/bash
#clear
# comment
echo Hello world! #also comment
```

закомментировал, теперь вывод не отчищает

3. Поменять местами 2,3 строки. Выполнить скрипт.

```
#!/bin/bash
# comment
echo Hello world! #also comment
clear
```

Теперь не выводит ничего, так как сначала выводит текст а потом отчищает.

4. Заменить #!/bin/bash на #!/bin/awk . Выполнить скрипт.

```
#!/bin/awk
#clear
# comment
echo Hello world! #also comment
```

При явном выполнении ничего не поменялось.

5. Сделать вывод о шебанг при явном запуске скрипта.

Шебанг при явном запуске не играет никакой роли, а является обычным комментарием.

6. Объединить 2,3 строки в одну . Проверить правильность выполнения.

```
#!/bin/bash
# comment
clear ; echo Hello world! #also comment
```

Обьединил, всё работает.

7. Ответить, что делает команда echo.

Выводит текст в терминал.

8. Найти комментарии.

```
# comment
# also comment
```

Ошибки при выполнении задания 1.

Если при выполнении п**4. Заменить #!/bin/bash на #!/bin/awk . Выполнить скрипт.** Вы получате сообщение об ошибке типа:

```
awk: cmd. line:1: ./scr1.sh
awk: cmd. line:1: ^ syntax error
awk: cmd. line:1: ./scr1.sh
awk: cmd. line:1: ^ unterminated regexp
```

Значит Вы запустили скрипт с помощью явного запуска (будет изучен позднее). Перечитайте теорию.

6. Объединить 2,3 строки в одну. Проверить правильность

```
#!/bin/bash
# comment
clear && echo Hello world! # also comment
```

Я объединил строки с помощью "&&" а не ";" Результат тот же, почему задание выполнено не правильно?

"**&&**" - это условный оператор, мы его изучим позже. Да, в данном примере все работает так же, но в другом случае результат может быть совсем иной. Для объединения операторов в одну строку служит именно ";"

Ответы и решения на Задание 2

Теория здесь:#2

Условие: Текущий каталог /home/tagd

1. Привести пример абсолютного имени файла myfile.txt , находящегося в данном каталоге.

```
/home/tagd/myfile.txt
```

2. Привести три различных примера относительного пути к указанному файлу.

```
myfile.txt
./myfile.txt
../tagd/myfile.txt
```

3. Привести относительный путь к каталогу /var/log

```
../../../var/log
```

Ошибки при выполнении задания 2.

2. Привести три различных примера относительного пути к указанному файлу

```
~/myfile.txt
```

Данный путь может быть ошибкой. В задании указано, что текущий каталог /home/tagd . Это совсем не означает, что в данный момент Вы работаете под пользователем tagd. Ваш домашний каталог может быть, например, /home/username . В этом случае ~/myfile.txt будет указывать на /home/username/myfile.txt . В данном случае можно было бы указать ~/../tagd/myfile.txt . Но и здесь может быть подвох. Если Вы, в данный момент зарегистрированы под пользователем root, то домашний каталог (обычно) /root , соответственно, путь ~/../tagd/myfile.txt будет указывать на файл /tagd/myfile.txt , которого, скорее всего, нет.

Ответы и решения на Задание 3

Теория здесь:#3

1. Перейти в каталог /tmp

```
cd /tmp
```

2. Перейти в домашний каталог.

```
cd
# или cd ~
# или cd ~/
```

3. Убедиться, что вы находитесь именно в домашнем каталоге.

```
pwd
```

4. Не меняя текущего каталога создать каталог work в /tmp

```
mkdir /tmp/work
```

5. Создать каталог work текущем каталоге. и перейти в него.

```
mkdir work
cd work
```

6. Сразу перейти в /tmp/work . Убедится, что перешли правильно.

```
cd /tmp/work
pwd
```

7. Вернуться в предыдущий каталог одной командой.

```
cd -
```

8. Попрактиковаться в переходах между двумя каталогами work различными способами.

```
cd ../../../tmp/work
cd ../../home/username/work
cd /tmp/work
cd /home/username/work
cd /; cd tmp; cd work
cd ~/work
```

9. создать одной командой каталог /tmp/test/day/primary.

```
mkdir -p /tmp/test/day/primary
```

10 удалить каталог test из каталога /tmp .

Сразу удалить не удаётся. Каталог не пуст.

```
rmdir /tmp/test/day/primary
rmdir /tmp/test/day
rmdir /tmp/test
```

Ошибки при выполнении задания 3.

Задание оказалось настолько простым, что кроме опечаток в именах файлов и командах других ошибок не обнаружено.

Ответы и решения на Задание 4

Теория здесь:#4

Условие: Выполнять задание, находясь в домашнем каталоге.

```
cd
# переход в домашний каталог
```

1. Создать файл test.txt в каталоге /tmp командой :> ,просмотреть содержимое /tmp и убедиться, что файл существует.

```
:> /tmp/test.txt
ls /tmp
# в списке есть test.txt
```

2. Удалить только что созданный файл командой rm , убедится, что файл был удален.

```
rm /tmp/test.txt
ls /tmp
# в списке нет test.txt
```

3. повторяя п1-2 испробовать все остальные способы создания файлов.(:>> , touch , nano)

```
:>> /tmp/test.txt ; ls /tmp ; rm /tmp/test.txt;
# test.txt
 touch /tmp/test.txt ; ls /tmp ; rm /tmp/test.txt
# test.txt
 nano /tmp/test.txt ; ls /tmp ; rm /tmp/test.txt
# test.txt
```

4. Сделать вывод об "удобстве" создания файлов разными способами.

Вывод: Создавать файл с помощью команд :> , :>> , touch

быстрее и удобней, чем с помощью редактора.

5. Создать файл test.txt в каталоге /tmp с помощью редактора. Добавить любое содержимое.

```
#Файл создан.
```

6. Повторяя п5, при необходимости, испробовать все остальные способы повторного создания файлов. (:>> , touch , :>)

```
touch /tmp/test.txt
nano /tmp/test.txt
#информация сохранилась

:>> /tmp/test.txt
nano /tmp/test.txt
# информация сохранилась
```

```
:> /tmp/test.txt
# файл пустой
```

7. Подтвердить или опровергнуть выводы о сохранении содержимого файла. (содержимое проверять при открытии текстовым редактором)

Команды :>> и touch соханяют содержимое файла, :> - очищает содержимое.

8. Создать файл hidden.cfg в каталоге /tmp

```
:>> /tmp/hidden.cfg
```

9. Переименовать hidden.cfg в .hidden.cfg

```
mv /tmp/hidden.cfg /tmp/.hidden.cfg
```

10. Проверить наличие .hidden.cfg

```
ls -a /tmp/.hidden.cfg
# /tmp/.hidden.cfg
```

11. Скопировать файл .hidden.cfg в .hidden.cfg.1

```
cp /tmp/.hidden.cfg  tmp/.hidden.cfg.1
```

12. Удалить файлы из п.11

```
rm tmp/.hidden.cfg  tmp/.hidden.cfg.1
```

13. Проверить домашний каталог на наличие случайно созданных файлов test.txt и .hidden.cfg . Удалить, при наличии.

```
ls -a ~/.hidden.cfg ~/test.txt
# файлы отсутствуют.
#если файлы есть, удаляем:
#rm ~/.hidden.cfg ~/test.txt
```

Ошибки при выполнении задания 4.

Если при выполнении п.10 Вы не видите файл /tmp/.hidden.cfg,

значит Вы или не создали файл, или забыли указать ключ в команде **ls -a tmp/.hidden.cfg** , поскольку файлы, начинающиеся с точки являются скрытыми.

Если при выполнении одной из команд создания файла не указать каталог /tmp, файл будет создан в домашнем каталоге, именно для этого проводится проверка п.13.

Ответы и решения на Задание 5

Теория здесь:#5

Условие: Задание выполнять, находясь в домашнем каталоге.

```
cd
# переход в домашний каталог
```

1. В каталоге /tmp создать подкаталоги bash days chat с помощью расширения.

```
mkdir /tmp/{bash,days,chat}
```

2. В каждом созданном каталоге создать файлы (различными способами, но с помощью расширений.) file01.txt2txt file02.txt2txt file04.txt2txt . Сделать вывод.

```
touch /tmp/bash/file{01,02,04}.txt2txt
touch /tmp/chat/file0{1,2,4}.txt2txt
touch /tmp/days/file0{1,2}.txt2txt file04.txt2txt
```

В*ывод:*

- *С помощью масок можно создавать файлы только используя команду touch . :> :>> выдают ошибки, а текстовый редактор создаёт только 1й файл в 1м каталоге.*
- *Не получилось использовать диапазоны в расширениях потому, что не удалось исключить file03.txt2txt .*
- *Расширения удобны, когда надо создать много однотипных файлов и/или директорий с минимальными различиями в названиях.*

3. Проверить получение списка созданных файлов с помощью различных масок. (Привести 3 примера)

```
# различные варианты
ls /tmp/*/file0[1-4].txt2txt
ls /tmp/+(days|chat|bash)/file0[124].txt2txt
ls /tmp/????/file0*.txt2txt
ls /tmp/*a*/f*.*2txt
ls /tmp/*(bash|days|chat)/file0*(1|2|4).txt2txt
ls /tmp/{bash,chat,days}/file0*.txt2txt
```

4. Удалить созданные файлы с помощью масок на файлы. В команде rm ключи не задействовать. Проверить, что все файлы удалены.

```
# различные варианты
rm /tmp/@(days|chat|bash)/file0[124].txt2txt
rm /tmp/{bash,days,chat}/file*
rm /tmp/{bash,days,chat}/file??.????txt
```

```
rm /tmp/bash/*.txt2txt /tmp/days/*.txt2txt
/tmp/chat/*.txt2txt

# Проверка удаления:( если файлов нет - удаление успешно)
ls /tmp/@(days|chat|bash)/file0[124].txt2txt
#ls: невозможно получить доступ к
'/tmp/@(days|chat|bash)/file0[124].txt2txt': Нет такого
файла или каталога
```

5. Привести аналог команды touch /tmp/file{1..3}.txt2txt одной строкой, без использования расширений.

```
touch /tmp/file1.txt2txt /tmp/file2.txt2txt
/tmp/file3.txt2txt
# в этой команде нет переноса строки, просто она длинная
```

Ошибки при выполнении задания 5.

Это не ошибки, а скорее замечания. При выполнении задания п.4 возможны различные маски при удалении файлов.

```
rm /tmp/*/*txt
```

```
rm /tmp/@(bash|days|chat)/file0[124].txt2txt
```

И первая и вторая маски удалят созданные в п.2 файлы, однако, вторая маска предпочтительней, поскольку она описывает только созданные файлы в нужных каталогах. Первая же маска может удалить и файлы других типов и из других каталогов. Например, если существует файл /tmp/kurs/my_test.txt - он тоже будет удален.

Поэтому, при использовании маски нужно выбирать вариант, который наиболее точно описывает группу файлов.

Ответы и решения на Задание 6

Теория здесь:#6

1. Преобразовать в текстовый вид следующие права для файлов 777, 666, 550, 444, 330

```
777 chmod u=rwx,g=rwx,o=rwx
666 chmod u=rw,g=rw,o=rw
550 chmod u=rx,g=rx,o=
444 chmod u=r,g=r,o=r
330 chmod u=wx,g=wx,o=
```

2. Расписать те же права рекурсивно для каталогов, но не для файлов.

```
777 chmod -R u=rwX,g=rwX,o=rwX
666 chmod -R u=rw,g=rw,o=rw
550 chmod -R u=rX,g=rX,o=
444 chmod -R u=r,g=r,o=r
330 chmod -R u=wX,g=wX,o=
```

3. Выполнить команду ls -la на любых каталогах и расшифровать права, двух файлов и двух каталогов.

```
333 --wx-wx-wx 1 username username   0 Apr  2 17:11 file.txt
```

Запись и исполнение для всех, но чтение запрещено.

```
740 -rwxr----- 1 username username   0 Apr  2 17:11
file1.txt
```

Полные права для владельца, чтение для группы, нет доступа для остальных.

```
755 drwxr-xr-x 2 username username  64 Apr  2 17:11 dir1
```

полные права для владельца, для группы и остальных -возможность войти в каталог и просмотреть список файлов.

```
700 drwx------ 2 username username  64 Apr  2 17:11 dir2
```

Полные права для владельца, нет прав для группы и остальных.

4. Написать 6 примеров команд для установки различный прав в числовом виде, проверяя результат командой ls -l

```
chmod 400 /tmp/file; ls -l /tmp/file
#-r-------- 1 username username 0 мая 12 19:01 /tmp/file

chmod 444 /tmp/file; ls -l /tmp/file
#-r--r--r-- 1 username username 0 мая 12 19:01 /tmp/file

chmod 755 /tmp/file; ls -l /tmp/file
#-rwxr-xr-x 1 username username 0 мая 12 19:01 /tmp/file
```

```
chmod 775 /tmp/file; ls -l /tmp/file
#-rwxrwxr-x 1 username username 0 мая 12 19:01 /tmp/file

chmod 666 /tmp/file; ls -l /tmp/file
#-rw-rw-rw- 1 username username 0 мая 12 19:01 /tmp/file

chmod 766 /tmp/file; ls -l /tmp/file
#-rwxrw-rw- 1 username username 0 мая 12 19:01 /tmp/file
```

5 . Написать еще 6 примеров для установки и изменения различный прав в текстовом виде , проверяя результат командой ls -l

```
chmod a-rwx,u+r /tmp/file; ls -l /tmp/file
#-r-------- 1 username username 0 мая 12 19:01 /tmp/file

chmod a+rwx,o-wx /tmp/file; ls -l /tmp/file
#-rwxrwxr-- 1 username username 0 мая 12 19:01 /tmp/file

chmod a=rwx,go-w /tmp/file; ls -l /tmp/file
#-rwxr-xr-x 1 username username 0 мая 12 19:01 /tmp/file

chmod a+x,o-w /tmp/file; ls -l /tmp/file
#-rw-r--r-- 1 username username 0 мая 12 19:01 /tmp/file

chmod a-rwx,a+rw /tmp/file; ls -l /tmp/file
#-rw-rw-rw- 1 username username 0 мая 12 19:01 /tmp/file

chmod a+rwx,go-x /tmp/file; ls -l /tmp/file
#-rwxrw-rw- 1 username username 0 мая 12 19:01 /tmp/file
```

6. Пояснить, к какой категории (ugo) относятся права выполнения и записи в rxrwr и rrwxw

запишем полные права и под ними требуемые права, добавляя пробелы

```
rwxrwxrwx
r xrw r   -> u=rx,g=rw,o=r
r  rwx w -> u=r,g=rwx,o=w
```

или расставим минусы, вместо отсутствующих прав:

```
rxrwr = r-xrw-r-- -> u=rx,g=rw,o=r
rrwxw = r--rwx--w -> u=r,g=rwx,o=w
```

7. Проверить возможность редактирования и сохранения файла, владельцем которого Вы являетесь, с правами rrrw.

Файл защищен от записи. (Хотя некоторые редакторы, типа **vim** дают возможность перезаписать файл, при указании специальной команды)

8. Определить, как будут выставлены права в случае конфликта (u-x, u+x) (a=rwx,u-r)(g=rwx,a-r) Сделать вывод.

Вывод: права устанавливаются последовательно, в порядке описания. При конфликте прав, действуют те, что указаны последними.

(u-x, u+x) эквивалент u+x
(a=rwx,u-r) эквивалент u=wx,go=rwx
(g=rwx,a-r) эквивалент (u-r,g=wx,o-r)

Ошибки при выполнении задания 6.

Если у Вас при выполнении данного задания возникли ошибки, перечитайте теорию. Если не помогло - свяжитесь со мной.

Ответы и решения на Задание 7.

Теория здесь:#7

Пп. 1-2 не требуют письменных ответов.

3. Написать одним предложением, что кроме создания файла может делать touch.

Обновляет/изменяет время доступа и модификации файла.

4. ls - записать назначение ключей t,S,X,r,h. Проверить на папке /usr/bin .

t - сортировка по времени, начиная с новых
S - сортировка по размеру, начиная с наибольших
X - сортировка в алфавитном порядке
r - обратная сортировка
h - отобразить размер файла "по-человечьи" для удобства восприятия

5. chmod объяснить ключ -v -R

-v - добавит вывод результата исполнения для диагностики
-R - изменит права рекурсивно, во всех вложенных директориях, всем содержащимся в них файлам в указанном пути

6. mkdir объяснить ключ -m -v

-m - задаст права на каталог, одновременно с созданием.
-v - добавит вывод результата исполнения для диагностики

7. rm записать действия ключей -r -f -i -v (не выполнять)

-r - выполнит команду рекурсивно, с применением ко всем файлам и вложенным директориям в указанном пути
-f - при выполнении не будет запрашивать подтверждений
-i - при выполнении будет запрашивать подтверждение для каждого файла
-v - добавит вывод результатов выполнения

8. Протестировать команду type -a на различных командах/программах, включая printf.

```
type -a sudo
#sudo является /usr/bin/sudo
#sudo является /bin/sudo

type -a cd
#cd — это встроенная команда bash
```

```
type -a printf
printf — это встроенная команда bash
printf является /usr/bin/printf
printf является /bin/printf

type -a cdm
#cdm — это псевдонимом для «cd "/dev/shm"»
```

9. Самостоятельно изучить команду which, сравнить командой type. Сделать вывод для чего можно применить команду which.

which *- выводит путь до исполняемого файла. При этом, ничего не выводит для встроенных в оболочку команд. Является внешней программой.*

Можно применить для поиска или проверки наличия исполняемого файла.

type *- выводит тип (программа,команда или алиас) и путь до исполняемого файла. Является встроенной в оболочку.*

10. Придумать (вспомнить, найти в поисковике) какой-нибудь нужный alias.

```
alias cdd='cd "~/Загрузки"'
alias cdw='cd "~/work"'
#простой консольный калькулятор на AWK
alias calc='_calc(){ awk "BEGIN{print $*}"; }; _calc '
# calc 1+2-7
#при использовании скобок нужны кавычки
#calc "2^5-7*(36-1)"
```

Быстрый переход в некоторые каталоги.

Ошибки при выполнении задания 7

Если у Вас при выполнении данного задания возникли ошибки, перечитайте теорию. Если не помогло - свяжитесь со мной.

Ответы и решения на Задание 8.

Теория здесь:#8

Условие: Всё исполняемое оформляем в виде скрипта(ов).

```
#!/bin/bash
#    1. Чем stderr отличается от stdout?
# stdout - основной канал вывода (дескриптор 1)
#stderr - для вывода сообщений об ошибках
# или предупреждениях (дескриптор 2)

#    2. Чем stdin отличается от stderr?
#stdin - входной поток, программа читает данные
# из этого потока (дескриптор 0)
#stderr - для вывода сообщений об ошибках или
# предупреждениях (дескриптор 2)

#    3. Написать команду перенаправления stderr в файл
#/tmp/myerr.log .
# prg 2> /tmp/myerr.log
#Здесь prg - любая команда или программам

#    4. Почему при выполнении команды
# ls ~/  2>>  /tmp/myerr.log Создается пустой файл?

# Потому что программа ls выполняется без ошибок.

ls ~/  2>>  /tmp/myerr.log

#    5. Измените параметры команды п4, чтобы заполнить
#  myerr.log информацией.

ls ~/NO_FILE 2>>  /tmp/myerr.log

#6. Напишите команду ls с одновременным перенаправлением >
#соответствующих потоков в файлы stdout.txt и stderr.txt.
# (Нужно придумать такую команду, чтобы оба файла
# заполнялись одновременно.)

ls ~/ ~/NO_FILE 1>stdout.txt 2> stderr.txt

#7. Выполните команду п6 2 раза подряд. Просмотрите  файлы
#stdout.txt и stderr.txt с помощью редактора.

ls ~/ ~/NO_FILE  1>stdout.txt 2> stderr.txt
ls ~/ ~/NO_FILE  1>stdout.txt 2> stderr.txt

nano stdout.txt
nano stderr.txt
```

```
# 8. В команде п 6 замените > на >> и выполните 2 раза
подряд.

ls ~/ ~/NO_FILE  1>>stdout.txt 2>>stderr.txt
ls ~/ ~/NO_FILE  1>>stdout.txt 2>>stderr.txt

#    9. Не открывая файлов ответить, сколько одинаковых
# записей содержат файлы stdout.txt stderr.txt. проверить
# догадку и объяснить.

#Записи попали в файл трижды, один раз при выполнении
# последней  команды п.7, и два раза при выполнении п.8.

# 10. В чем разница между командами ...
#prg1  2>&1 >file_or_dev | prg2

#При выполнении этой команды stdout prg1 будет выведен в
# файл или на устройство file_or_dev, а stderr команды prg1
# будет передан  по конвейеру на stdin команды prg2

#проверка
ls ~/ ~/NO_FILE  2>&1 >file1 | tee file2

# file1 содержит список каталога,file2 - ошибку

#prg1 >file_or_dev 2>&1  | prg2
#При выполнении этой команды сначала stderr команды prg1
# будет объединен с stdout prg1, а затем, информация двух
# каналов будет записана в файл или на устройство
# file_or_dev. Таким образов по конвейеру на stdin команды
#prg2 не будет передаваться ничего.

#проверка
ls ~/ ~/NO_FILE  >file3 2>&1  | tee file4

# file3 содержит ошибки и список каталога, file4  -#пустой
```

Ошибки при выполнении задания 8.

5. Измените параметры команды п4, чтобы заполнить myerr.log информацией.

```
l_s ~/  2>> /tmp/myerr.log
```

Не ошибка, но замечание. Да, в этом случае в файл тоже будет записано сообщение об ошибке, Но в задании было указано изменить параметры, а не саму команду. Нужно быть внимательнее.

10. В чем разница между командами

```
prg1  2>&1 >file_or_dev | prg2
prg1 >file_or_dev 2>&1  | prg2
```

Если Ваш ответ не совпал с решением п10 этого задания - обязательно перечитайте теорию "Стандартные потоки. Часть 1" Глава #8

Ответы и решения на Задание 9.

Теория здесь:#9

Пункты 1-4 не требуют письменного ответа.

Условие Оформить все следующие команды задания в виде одного скрипта. Считать, что файл /tmp/20.txt существует и правильно заполнен.

```
#!/bin/bash
#  5. С помощью программы head вывести первые 5 строк файла.
head -n 5 /tmp/20.txt
head -n5 /tmp/20.txt
head -5 /tmp/20.txt

# 6. С помощью программы tail  вывести последние 5 строк
 # файла.
tail -n 5 /tmp/20.txt
tail -n5 /tmp/20.txt
tail -5 /tmp/20.txt

# 7. Комбинируя обе программы с помощью конвейера вывести на
# экран строки с номерами 7-11.
head -n11 /tmp/20.txt|tail -n5
head -n-9 /tmp/20.txt|tail -n +7

# 8. Добиться того же результата поменяв head и tail местами
tail -n14 /tmp/20.txt|head -n5
tail -n +7 /tmp/20.txt|head -n -9

# 9. С помощью команды ls -l (плюс другие ключи) и tail
# вывести на экран 5 самых больших файлов из папки /usr/bin
ls -lSr /usr/bin |tail -5

# 10. Изменить команду, для вывода 5 самых маленьких файлов
# из той же папки.
ls -lS /usr/bin |tail -5
```

11. Объяснить, что делает команда head -n-5

Выводит все строки с начала за исключением 5 последних

12. Расписать, что делает команда ls -l /usr/bin |cat -n -

ls -la /usr/bin *команда выводит список файлов каталога /usr/bin , включая скрытые файлы и каталоги "." и ".." (ключ -a) в длинном формате (-l). stdout команды ls предаётся на stdin команды cat посредством конвейера.*

*Команда **cat**, прономерует строки (-n) и выведет данные на экран. "-" - обозначает stdin. В данном случае команда работает одинаково, как с ним, так и без него.*

13. Сравнить команды **less** и **more**. Сделать вывод.

***less** более продвинутый вариант, чем **more**. **less** , например, позволяет искать вперед-назад с подсветкой, в отличие от **more**, который может только вперед без подсветки.*

Ошибки при выполнении задания 9.

Замечание пп 7,8

```
head -n-9 /tmp/20.txt|tail -n +7
tail -n +7 /tmp/20.txt|head -n -9
tail -n14 /tmp/20.txt|head -n5
tail -n +7 /tmp/20.txt|head -n -9
```

Формально эти команды решают поставленную задачу и выводят на экран строки с 7 по 11, однако они опираются на знание количества строк файла =20. Если изменить размер файла - решение будет не верным.

Ответы и решения на Задание 10.

Теория здесь:#10

п.1 не требует письменного ответа.

Условие: Задания 2-10 оформить в виде скрипта. Каждая команда должна выдавать результат работы в виде текста "Успех" и "Ошибка" на stderr и одновременно в файл /tmp/err.log

2. Привести основные отличие cmp от diff

cmp сравнивает файлы по байтово, diff сравнивает построчно

```
#!/bin/bash
# в скрипте длинные команды(2 строки). перенос вынужденный

#    3. Привести примеры использования wc для подсчета
# строк, байт, слов, максимальной длины строки.

# количество строк
wc -l /tmp/21.txt && echo "Успех" | tee -a /tmp/succes.log
|| echo "Ошибка" | tee -a /tmp/err.log

# количество слов
wc -w /tmp/21.txt && echo "Успех" | tee -a /tmp/succes.log
|| echo "Ошибка" | tee -a /tmp/err.log

# количество байт
wc -c /tmp/21.txt && echo "Успех" | tee -a /tmp/succes.log
|| echo "Ошибка" | tee -a /tmp/err.log

# максимальная длина строки (команда  занимает 2 строки)
wc -L /tmp/21.txt && echo "Успех" | tee -a /tmp/succes.log
|| echo "Ошибка" | tee -a /tmp/err.log

#    4. в /tmp/kurs оптимально создать три каталога (1, 2,
# 3)  с правами 700. Копировать несколько файлов из /bin с
# помощью # масок. Каталоги 1, 2 одинаковые, 3 немного
# отличный. Подсчитать количество файлов в каждом с помощью
# wc)  Сравнить каталоги с помощью diff.

mkdir -p /tmp/kurs/{1,2,3} && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

chmod 700 /tmp/kurs/{1,2,3} && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

cp /bin/ip* /tmp/kurs/3 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

cp /bin/i* /tmp/kurs/1 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
```

```
cp /bin/i* /tmp/kurs/2 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

#    5. Подсчитать количество файлов в каждом каталоге
# (1,2,3)  с помощью wc)

ls /tmp/kurs/1|wc -l && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
#46 Успех

ls /tmp/kurs/2|wc -l && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
#46 Успех

ls /tmp/kurs/3|wc -l && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
#10 Успех

#    6 Сравнить каталог 1 с каталогами 2 и 3 с помощью diff
diff /tmp/kurs/1 /tmp/kurs/2 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
#Успех

diff /tmp/kurs/1 /tmp/kurs/3 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
#  Ошибка и список файлов, которые только в /tmp/kurs/1

#    7. В каталоге 2 изменить права на файлы на 644 с
#помощью текстового способа.

chmod a=r,u+w /tmp/kurs/2/* && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
#другие варианты
#chmod a=rw,go-w ...
#chmod u=rw,go=r ...
#chmod u=rw,g=r,o=r ...

#    8. Сравнить каталоги 1 и 2 с помощью diff. Сделать
# вывод
diff /tmp/kurs/1 /tmp/kurs/2 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
#Успех
#Вывод - права доступа не играют роли при сравнении файлов,
# если файлы можно прочитать.

#    9. Создать (или копировать) в каталоги 1 и 2 текстовый
# файл с  одинаковым именем, но разным содержимым. Сравнить
# каталоги. Сделать вывод.

cp /etc/fstab /tmp/kurs/1/test.txt && echo "Успех" | tee -
a /tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
```

```
cp /etc/hostname /tmp/kurs/2/test.txt && echo "Успех" | tee
-a /tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

diff /tmp/kurs/1 /tmp/kurs/2 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

#Список различий файлов и  "Ошибка"
```

ВЫВОД при сравнении двух каталогов, если в них присутствуют файлы с одинаковыми именами также будет приведен список различий.

```
# 10. Сравнить два текстовых файла из п.9 с помощью b2sum,
# md5sum, sha1sum sha256sum. (хэш сравнивать визуально) В
# чем основное отличие b2sum от остальных программ
# вычисления контрольных сумм?

b2sum /tmp/kurs/1/test.txt /tmp/kurs/2/test.txt && echo
"Успех" | tee -a /tmp/succes.log || echo "Ошибка" | tee -
a /tmp/err.log
# Успех

md5sum /tmp/kurs/1/test.txt /tmp/kurs/2/test.txt && echo
"Успех" | tee -a /tmp/succes.log || echo "Ошибка" | tee -
a /tmp/err.log
# Успех

sha1sum /tmp/kurs/1/test.txt /tmp/kurs/2/test.txt && echo
"Успех" | tee -a /tmp/succes.log || echo "Ошибка" | tee -
a /tmp/err.log
# Успех

sha256sum /tmp/kurs/1/test.txt /tmp/kurs/2/test.txt && echo
"Успех" | tee -a /tmp/succes.log || echo "Ошибка" | tee -
a /tmp/err.log
# Успех
```

Вывод:
1. b2sum по-умолчанию работает с самой большой суммой (512 бит)

2. Размер контрольной суммы можно уменьшить с помощью ключа -l, длина контрольной суммы должна быть кратна 8.

3. Все остальные программы имеют фиксированный размер контрольной суммы.

11. Объяснить errorlevel команды `ls -la /no_file* | tail -3`

На экран выводится сообщение об ошибке "нет такого файла или директории". Однако, в конвейере последней выполняется команда tail, которая завершается без ошибки. errorlevel конвейера обычно

определяется последней командой в конвейере, В данном случае конвейер завершился без ошибок, потому что программа tail -3 отработала нормально.

Ошибки при выполнении задания 10.

7. В каталоге 2 изменить права на файлы на 644 с помощью текстового способа.

```
chmod a=r,u+w /tmp/kurs/2/ && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

chmod a=r,u+w /tmp/kurs/2 && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log

# Отличие в наличии / после 2
```

Ошибка. Данные команды установят права доступа на сам каталог /tmp/kurs/2

Чтобы команда установила права только на файлы внутри каталога нужно использовать *

```
chmod a=r,u+w /tmp/kurs/2/* && echo "Успех" | tee -a
/tmp/succes.log || echo "Ошибка" | tee -a /tmp/err.log
```

Ответы и решения на Задание 11.

Теория здесь:#11

```
#!/bin/bash
cd /tmp
#     2  Попробовать создать несколько файлов с символами в
# именах # "  ' * # > : & ( ) [ ] { } $ пробел (по одному
# символу на файл).
#     2.1   В случае ошибки попробовать экранирование
# символа.
#     2.2   Дополнительно попробовать использовать кавычки.

#Создание с экранированием. (экранирование только на файлах
# с ошибкой)
touch file01\"
touch file02\'
touch file03*
touch file04#
touch file05\>
touch file06:
touch file07\&
touch file08\(
touch file09\)
touch file10[
touch file11]
touch file12{
touch file13}
touch file14$
touch file15\
#file15 пробел в конце имени файла

#Создание с кавычками.
touch 'file16"'
touch "file17'"
touch 'file18>'
touch 'file19&'
touch 'file20('
touch 'file21)'
touch 'file22 '

# 3 список файлов и удаление
ls file[012][0-9]?
rm file[012][0-9]?

#4 Три самых длинных команды из задания 10 оформить
#переносами строки с сохранением работоспособности.
mkdir -p /tmp/kurs/{1,2,3} &&
echo "Успех" |
tee -a /tmp/succes.log ||
echo "Ошибка" |
tee -a /tmp/err.log
```

```
chmod 700 /tmp/kurs/{1,2,3} &&
echo "Успех" |
tee -a /tmp/succes.log ||
echo "Ошибка" |
tee -a /tmp/err.log

cp /bin/ip* /tmp/kurs/3 && echo \
"Успех" | tee -a /tmp/succes.log || echo \
"Ошибка" | tee -a /tmp/err.log

#5  Привести 5 примеров вывода на экран с
# эскейп-последовательностями.

echo 1 таб$'\t'уляция
echo 2 перенос$'\n'строки
echo 3 вертикальная$'\v'табуляция
echo 4 возврат$'\r'каретки
echo 5 забой$'\b'символа
#1 таб    уляция
#2 перенос
#строки
#3 веритикальная
#               табуляция
#кареткиат
#5 забосимвола

#6  Попробовать то же самое реализовать через echo -e
echo -e "1 таб\туляция"
echo -e "2 перенос\nстроки"
echo -e "3 вертикальная\vтабуляция"
echo -e 4 возврат\\rкаретки
echo -e 5 забой\\bсимвола
# результат тот же. Без кавычек требуется экранирование
бэкслэша
```

Ошибки при выполнении задания 11.

3 список файлов и удаление

```
ls f*
rm f*
```

Каталог /tmp может использоваться многими программами. Маску обязательно задавать, как-можно точнее. И хотя на каталоге установлен Sticky bit, при запуске скрипта от имени root он не поможет.

4 Три самых длинных команды из задания 10 оформить переносами строки с сохранением работоспособности.

```
chmod 700 /tmp/kurs/{1,2,3} && \
echo "Успех" | \
tee -a /tmp/succes.log || \
echo "Ошибка" | \
tee -a /tmp/err.log
```

Не ошибка, а замечание, строку можно переносить не только по символу экранирования (бэкслэш) но и по символам & и |. В данном случае бэкслэши в конце строки не нужны.

Ответы и решения на Задание 12.

Теория здесь:#12

```
#!/bin/bash
#2  Написать скрипт, в котором описаны переменные: целые
# I,J,K; обычные массивы A7, B13 ;переменные для
# экспорта PA, VASYA .

declare -i I J K
declare -a A7 B13
declare -x PA VASYA

#3 Задать произвольные  начальные значения всем переменным.
I=0; J=1; K=3
A7=7; B13=13
PA="PASHA"; VASYA="PETYA"

#4  Снять опцию экспорта для переменной VASYA
declare +x VASYA

#5  Описать переменные I3 -как целую, B - как текстовую
# верхнего регистра. (Переменным значения не задавать)
declare -i I3
declare -u B

#6  Присвоить переменной B значение "yes", Переменной I
# значение "no", переменной PA значение
# "Сегодня в школу не пойдем!"
B="yes"
I="no"
PA="Сегодня в школу не пойдем!"

#7  С помощью команды echo распечатать значения всех
# описанных # выше  переменных. В виде: echo "VAR=$VAR"
echo "I=$I"
echo "J=$J"
echo "K=$K"
echo "A7=$A7"
echo "B13=$B13"
echo "PA=$PA"
echo "VASYA=$VASYA"
echo "I3=$I3"
echo "B=$B"

#8  Вывести  на экран значения и описание всех описанных
# выше переменных с помощью declare. Сделать вывод о
# значениях переменных и удобстве применения команд для
# контроля значений переменных.

declare -p I J K A7 B13 PA VASYA I3 B
```

Вывод:

declare -p удобнее для отладки, чем вывод переменных через echo, поскольку удобнее перечислять переменные и сразу показывает флаги, с которыми была описана переменная.

При Переменная I описана, как целая, при присвоении текстового значения "no" получила значение 0. Переменная B получила значение в верхнем регистре "YES"

```
#9  Значения каких переменных могут быть использованы
# без знака $ и в каком случае?

#Ответ:Значения переменных, описанные как целые в основных
# арифметических операциях могут использоваться без знака $
# Например:
declare -i I; J=1; K=3
I=J+K
declare -p I
```

10 Можно ли переменной, описанной как массив присвоить значение "1". Например: A7="1"

Да, можно, в этом случае значение будет присвоено нулевому элементу. (см переменные A7 и B13)

```
#11  Написать второй скрипт. В скрипте задать переменные
# HOME_DIR, FILE_NAME, FILE_EXT.  (значения на Ваше
# усмотрение)

declare HOME_DIR=/tmp
declare FILE_NAME=text
declare FILE_EXT=txt
declare FILE="${HOME_DIR}/${FILE_NAME}.${FILE_EXT}"

#12  Все возможные ошибки выводить на экран и в лог-файл с
# полным именем и суффиксом .err

declare -r ERR_FILE="${FILE}.err"

#13  Результат каждой операции подтверждать с помощью списка
каталога.
#14  Создать файл HOME_DIR / FILE_NAME . FILE_EXT

touch "$FILE" 3>&1 1>&2 2>&3 | tee -a "$ERR_FILE"
ls -l "${HOME_DIR}" 3>&1 1>&2 2>&3 | tee -a "$ERR_FILE"

#15  Создать резервную копию файла с полным именем и
#суффиксом ".1" на конце.

cp "$FILE" "${FILE}.1"
ls -l "${HOME_DIR}" 3>&1 1>&2 2>&3 | tee -a "$ERR_FILE"

#16  Удалить исходный файл.
```

```
rm "$FILE"
ls -l "${HOME_DIR}" 3>&1 1>&2 2>&3 | tee -a "$ERR_FILE"

#17  Восстановить исходный файл с помощью команды переименования.

mv "${FILE}.1" "$FILE"
ls -l "${HOME_DIR}" 3>&1 1>&2 2>&3 | tee -a "$ERR_FILE"
```

Ошибки при выполнении задания 12.

#12 Все возможные ошибки выводить на экран и в лог-файл с полным именем и суффиксом .err

Чтобы правильно выполнить это условие необходимо поменять потоки stderr и stdout местами и с помощью tee перенаправить в файл. 3>&1 1>&2 2>&3 |tee -a "$ERR_FILE"

Если не указать опцию -a, файл будет пересоздаваться при каждой новой записи.

Использовать перенаправление **prg1 |& tee -a "$ERR_FILE"** в данном случае не совсем корректно, поскольку в лог ошибок попадут данные stdout.

#14 Создать файл HOME_DIR / FILE_NAME . FILE_EXT

Поскольку действует условие 12 (в каждой команде дополнительно требуется перенаправление ошибок в файл) создать файл с помощью с помощью перенаправления :> будет проблематично. Нужно использовать touch.

Проверить правильность работы скрипта просто:

После первого запуска установите права 0 на основной файл (HOME_DIR / FILE_NAME . FILE_EXT) и повторно запустите скрипт. На вопрос об удалении ответьте n

и посмотрите содержимое файла ошибок. Если в файле как минимум две строки - скрипт работает правильно.

Ответы и решения на Задание 13.

Пункт 1 не требует письменного выполнения.

```
#!/bin/bash

declare WORK_DIR="/dev/shm"
declare SOURCE_DIR="/etc"

#2. Задать рабочий каталог WORK_DIR
ls -d "$WORK_DIR" ||WORK_DIR="/tmp"

#3. Задать каталог архивов "backup"
declare BACKUP_DIR="${WORK_DIR}/backup"
declare USER_DIR="${WORK_DIR}/${USER}"
declare ARCH_NAME="${BACKUP_DIR}/arch.tar"
declare LOG="${WORK_DIR}/${USER}.log"
declare ERRLOG="${WORK_DIR}/${USER}.err"

# Для контроля правильности переменных и значений
#declare -p SOURCE_DIR BACKUP_DIR WORK_DIR USER_DIR \
            ARCH_NAME LOG ERRLOG
# exit

mkdir -p "$BACKUP_DIR" &&
  echo "create backup dir $BACKUP_DIR" >>"$LOG" ||
  echo "Error create $BACKUP_DIR dir " >>"$ERRLOG"

mkdir -p "$USER_DIR" &&
  echo "create user dir $USER_DIR" >>"$LOG" ||
  echo "Error create $USER_DIR dir " >>"$ERRLOG"

# сброс таймера
SECONDS=0
# архивируем без сжатия
tar -cf "$ARCH_NAME" "$SOURCE_DIR" || echo "Ошибка при
архивации 1" >> "$ERRLOG"

echo "1. Время сжатия, сек. $SECONDS" Размер архива - \
  $(du -b "$ARCH_NAME" 2>>"$ERRLOG") Размер исходной \
 папки - \  $(du -bs "$SOURCE_DIR" 2>>"$ERRLOG") |
  tee -a "$LOG" 2>>"$ERRLOG"

# Распаковать архив в подкаталог (п.2) с именем
# пользователя.
# сброс таймера
SECONDS=0
tar -xf "$ARCH_NAME" -C "$USER_DIR" 2>>"$ERRLOG" ||
  echo "Ошибка при распаковке 1" >> "$ERRLOG"
echo "Время распаковки, сек. $SECONDS" | tee -a "$LOG"
```

```
# После распаковки сравнить исходный каталог и каталог с
распакованными файлами
echo "Сравнение каталогов 1" |  tee -a "$LOG" "$ERRLOG"
diff -r -q "$SOURCE_DIR" "${USER_DIR}${SOURCE_DIR}" |&
      tee -a "$ERRLOG"

# После сравнения и проверки Архив и каталог с
разархивированными файлами удалить, используя опцию защиты
корневого каталога от удаления.
rm -rf --preserve-root "$ARCH_NAME" "$USER_DIR" 2>>"$ERRLOG"

mkdir -p "$USER_DIR" &&
  echo "create user dir $USER_DIR" >>"$LOG" ||
  echo "Error create $USER_DIR dir " >>"$ERRLOG"

# сброс таймера
SECONDS=0
# архивируем bzip2
tar -cjf "$ARCH_NAME" "$SOURCE_DIR" ||
      echo "Ошибка при архивации bzip2" >> "$ERRLOG"

echo "2. Время сжатия, сек. $SECONDS" Размер архива bzip - \
  $(du -b "$ARCH_NAME" 2>>"$ERRLOG") Размер исходной \
 папки -   $(du -bs "$SOURCE_DIR" 2>>"$ERRLOG") |
  tee -a "$LOG" 2>>"$ERRLOG"

# Распаковать архив в подкаталог (п.2) с именем
пользователя.
# сброс таймера
SECONDS=0
tar -xf "$ARCH_NAME" -C "$USER_DIR" 2>>"$ERRLOG" ||
  echo "Ошибка при распаковке 2" >> "$ERRLOG"
echo "Время распаковки 2, сек. $SECONDS" | tee -a "$LOG"

# После распаковки сравнить исходный каталог и каталог с
распакованными файлами
echo "Сравнение каталогов 2" |  tee -a "$LOG" "$ERRLOG"
diff -r -q "$SOURCE_DIR" "${USER_DIR}${SOURCE_DIR}" |&
 tee -a "$ERRLOG"

# После сравнения и проверки Архив и каталог с
разархивированными файлами удалить, используя опцию защиты
корневого каталога от удаления.
rm -rf --preserve-root "$ARCH_NAME" "$USER_DIR" 2>>"$ERRLOG"

mkdir -p "$USER_DIR" &&
  echo "create user dir $USER_DIR" >>"$LOG" ||
  echo "Error create $USER_DIR dir " >>"$ERRLOG"

# сброс таймера
SECONDS=0
# архивируем xz
tar -cJf "$ARCH_NAME" "$SOURCE_DIR" ||
```

```
 echo "Ошибка при архивации xz" >> "$ERRLOG"

echo "3. Время сжатия, сек. $SECONDS" Размер архива xz - \
  $(du -b "$ARCH_NAME" 2>>"$ERRLOG") Размер исходной\
 папки - $(du -bs "$SOURCE_DIR" 2>>"$ERRLOG") |
  tee -a "$LOG" 2>>"$ERRLOG"

# Распаковать архив в подкаталог (п.2) с именем
пользователя.
# сброс таймера
SECONDS=0
tar -xf "$ARCH_NAME" -C "$USER_DIR" 2>>"$ERRLOG" ||
  echo "Ошибка при распаковке 3" >> "$ERRLOG"
echo "Время распаковки, сек. $SECONDS" | tee -a "$LOG"

# После распаковки сравнить исходный каталог и каталог с
распакованными файлами
echo "Сравнение каталогов 3" | tee -a "$LOG" "$ERRLOG"
diff -r -q "$SOURCE_DIR" "${USER_DIR}${SOURCE_DIR}" |&
   tee -a "$ERRLOG"

# После сравнения и проверки Архив и каталог с
разархивированными файлами удалить, используя опцию защиты
корневого каталога от удаления.
rm -rf --preserve-root "$ARCH_NAME" "$USER_DIR" 2>>"$ERRLOG"

mkdir -p "$USER_DIR" &&
  echo "create user dir $USER_DIR" >>"$LOG" ||
  echo "Error create $USER_DIR dir " >>"$ERRLOG"

# сброс таймера
SECONDS=0
# архивируем gzip
tar -czf "$ARCH_NAME" "$SOURCE_DIR" ||
    echo "Ошибка при архивации gzip" >> "$ERRLOG"

echo "4. Время сжатия, сек. $SECONDS" Размер архива gzip- \
  $(du -b "$ARCH_NAME" 2>>"$ERRLOG") Размер исходной\
 папки - $(du -bs "$SOURCE_DIR" 2>>"$ERRLOG") |
  tee -a "$LOG" 2>>"$ERRLOG"

# Распаковать архив в подкаталог (п.2) с именем
пользователя.
# сброс таймера
SECONDS=0
tar -xf "$ARCH_NAME" -C "$USER_DIR" 2>>"$ERRLOG" ||
  echo "Ошибка при распаковке 4" >> "$ERRLOG"
echo "Время распаковки, сек. $SECONDS" | tee -a "$LOG"

# После распаковки сравнить исходный каталог и каталог с
распакованными файлами
echo "Сравнение каталогов 4" |  tee -a "$LOG" "$ERRLOG"
diff -r -q "$SOURCE_DIR" "${USER_DIR}${SOURCE_DIR}" |&
```

```
 tee -a "$ERRLOG"
# После сравнения и проверки Архив и каталог с
разархивированными файлами удалить, используя опцию защиты
корневого каталога от удаления.
rm -rf --preserve-root "$ARCH_NAME" "$USER_DIR" 2>>"$ERRLOG"
```

11. Сделать вывод о степени сжатия и времени выполнения.
Вывод:
xz сжимает лучше всего файлы и, к сожалению, дольше.
bz2 хорошая скорость сжатия и медленная распаковка.
gz чуть медленней чем bz2, но быстрая распаковка

Но скорости и степени сжатия зависят от набора файлов.

Ошибки при выполнении задания 13.

Скрипт данного задания кажется очень большим, но это из-за того, что он состоит из четырех практически одинаковых блоков кода. Как сгруппировать часто используемый код Вы узнаете после изучения функций. При выполнении данного задания нужно сначала

- отладить задание ВСЕХ переменных (операторы **declare -p** и **exit** при выполнении программы не нужны, поэтому сейчас закомментированы)

- отладить первый участок кода, состоящий из создания каталога с именем пользователя, архивирования, получение времени и размеров файла и каталога, сравнения и удаления требуемых файлов и каталога. **Причем каждую операцию нужно контролировть отдельно!**

- Когда все работает идеально можно тиражировать этот участок кода, внося небольшие правки.

Основные ошибки:

- Перед архивированием-распаковкой не нужно запоминать текущее значение переменной SECONDS, ее нужно обнулить.

- При анализе размера файла нужно использовать **du -b**, при анализе размера каталога - **du -bs** . Получить размер каталога с помощью команды ls не получится.

- Не ошибка, но замечание, при разархивировании не нужно указывать ключ, отвечающий за выбор типа фильтра (j J z)

- Поскольку разархивирование производится в другой каталог, нужно использовать ключ -C

- При сравнении каталогов после разархивирования используется программа diff с ключем -r поскольку в каталогах могут быть подкаталоги, и ключем -q , поскольку при проверке разархивирования интерес представляет только целостность файла, а не конкретная информация о том, чем файлы отличаются.

- При разархивировании сохраняется имя каталога источника файлов, поэтому при сравнении в команде diff приходится использовать конкатенацию двух каталогов "${USER_DIR}${SOURCE_DIR}". Это становится понятно при анализе лога ошибок.

- Возможны ошибки при перенаправлении потоков.

- Замечание: При сравнении каталогов после разархивирования может выяснится, что восстановленный каталог не соответствует архивируемому каталогу. Это нормально. Обычно архивирование проводится с повышенными правами, позволяющими читать все файлы, а в данном случае скрипт запускается от имени обычного пользователя. Кроме этого в каталоге могут быть символические и жесткие ссылки, которые после восстановления в другом каталоге могут не работать.

Ответы и решения на Задание 14.

Теория здесь:#12

```
#!/bin/bash
#описать переменные LOG=/tmp/log и ERR=/tmp/err
# LIST=/tmp/list
declare LOG=/tmp/log ERR=/tmp/err LIST=/tmp/list

#1. Проверить наличие файлов /tmp/log и /tmp/err.
# - В случае наличия файлов - вывести сообщение на экран о
наличии # файлов.
# - При отсутствии - создать файлы

ls "$LOG" && echo "$LOG есть" || touch "$LOG"
ls "$ERR" && echo "$ERR есть" || touch "$ERR"

#2.установить права доступа файлы 700 текстовым способом.
# chmod u=rw,go= "$ERR"
# chmod u=rw,g=,o= "$ERR"
chmod a=,u=rw "$ERR"
chmod a=,u=rw "$LOG"
echo "2 права установлены"

#3. перенаправить вывод основных данных скрипта в файл
/tmp/log
# вывод ошибок скрипта в файл /tmp/err
#файлы будут перезаписываться при каждом запуске.
exec 2>"$ERR"
exec 1>"$LOG"

echo "3 перенаправление произведено"

#4 Получите развернутый список файлов временного каталога с
# подробной информацией о правах доступа, владельцах и
# группах, запишите его в файл LIST

ls /tmp -l >"$LIST"
echo "4 список получен"

# 5. Создать во временном каталоге 3 пустых файла с именами
# u1, u2, u3.

touch /tmp/u1 /tmp/u2 /tmp/u3
#touch /tmp/u{1..3}
echo "5 файлы u1-u3 созданы"

#6. Допишите информацию только о новых файлах в файл LIST,
# используя самую оптимальную маску

ls -l /tmp/u[1-3] >>"$LIST"
# ls -l /tmp/u[123] >>"$LIST"
```

```
echo "6 информация о файлах u1-u3 дописана"
# 7. Убрать право чтения на файлы u1 и u2 для всех, кроме
# владельца, текстовым способом.

chmod go-r /tmp/u[12]
echo "7 права на  файлы u1,u2 установлены"

#8. Допишите информацию об изменениях в файл LIST, используя
самую
# оптимальную маску.
ls -l /tmp/u[12] >>"$LIST"

echo "8 информация о файлах u1,u2 дописана"

#9. попробуйте создать файл /usr/bin/u4

touch /usr/bin/u4 && echo "9 /usr/bin/u4 создан" || echo
"9 /usr/bin/u4 НЕ создан"

#10 удалить файлы u1-u3

rm /tmp/u[1-3]
echo "10 файлы u1-u3 удалены"
```

```
при первом запуске на экран выводится
ls: невозможно получить доступ к '/tmp/log': Нет такого файла или
каталога
ls: невозможно получить доступ к '/tmp/err': Нет такого файла или
каталога
2 права установлены

### файл /tmp/log
3 перенаправление произведено
4 список получен
5 файлы u1-u3 созданы
6 информация о файлах u1-u3 дописана
7 права на  файлы u1,u2 установлены
8 информация о файлах u1,u2 дописана
9 /usr/bin/u4 НЕ создан
10 файлы u1-u3 удалены

### файл /tmp/err
touch: невозможно выполнить touch для '/usr/bin/u4': Отказано в
доступе

### файл /tmp/list
-rw-r--r-- 1 tagd tagd 0 июн 18 15:14 /tmp/u1
-rw-r--r-- 1 tagd tagd 0 июн 18 15:14 /tmp/u2
-rw-r--r-- 1 tagd tagd 0 июн 18 15:14 /tmp/u3
-rw------- 1 tagd tagd 0 июн 18 15:14 /tmp/u1
-rw------- 1 tagd tagd 0 июн 18 15:14 /tmp/u2
```

при повторном запуске на экран выводится
/tmp/log
/tmp/log есть
/tmp/err
/tmp/err есть
2 права установлены

В остальных файлах меняется только время создания файлов.

Ошибки при выполнении задания 14.

Если имя файла записано в переменной, при использовании переменную обязательно нужно заключить в двойные кавычки. Это исключит проблемы с файлами, содержащими пробелы в полном имени.

```
ls "$LOG" && echo "$LOG есть" || touch "$LOG"
```

Поскольку в данном скрипте ведется лог ошибок, нет необходимости отслеживать правильность выполнения каждого пункта, кроме п.9, в котором это оговорено дополнительно.

6. Допишите информацию только о новых файлах в файл LIST,используя самую оптимальную маску.

Оптимальная маска – это такая, которая максимально точно соответствует нужной группе файлов, не захватывая лишних и не пропуская нужные. Если Ваша маска отличается - перечитайте Главу #5

```
ls -l /tmp/u[123] >>"$LIST"
ls -l /tmp/u[1-3] >>"$LIST"
```

8. Допишите информацию об изменениях в файл LIST, используя самую оптимальную маску.

```
ls -l /tmp/u[12] >>"$LIST"
```

В данном случае оптимальная маска [12]

Ответы и решения на Задание 15.

Теория здесь:#15

```
#!/bin/bash

#1.Исправить ошибки в скрипте без изменения вида скрипта.
#Скрипт должен выдавать на экран текущий временный каталог и
#активный каталог.

declare FILE=test.txt
declare SHM=/dev/shm
declare TMP=/tmp
declare CURTMP

cd
ls "$SHM" && { cd $SHM; touch "$FILE" ;CURTMP=$SHM;} || { cd
$TMP; touch "$FILE" ; CURTMP=$TMP;}
echo "$CURTMP"
pwd
# список файлов каталога SHM
#/dev/shm
#/dev/shm
```

```
#!/bin/bash

#2.Привести скрипт к виду одна команда в строке.

declare FILE=test.txt
declare SHM=/dev/shm
declare TMP=/tmp
declare CURTMP

cd
ls "$SHM" &&
   { cd $SHM
   touch "$FILE"
   CURTMP=$SHM
   } ||
   { cd $TMP
   touch "$FILE"
   CURTMP=$TMP
   }
echo "$CURTMP"
pwd
# список файлов каталога SHM
#/dev/shm
#/dev/shm
```

```
#!/bin/bash
#3. Изменить скрипт, используя группировку (...). Скрипт
#должен выдавать на экран текущий временный каталог и
# активный каталог

declare FILE=test.txt
declare SHM=/dev/shm
declare TMP=/tmp
declare CURTMP

cd
ls "$SHM" &&
   (cd $SHM
   touch "$FILE"
   CURTMP=$SHM
   echo "$CURTMP" ) ||
   (cd $TMP
   touch "$FILE"
   CURTMP=$TMP
echo "$CURTMP")
pwd
# список файлов каталога SHM
#/dev/shm
#/home/username
```

```
#!/bin/bash

#4. С помощью ключа (опции) добиться вывода двух каталогов в
одной строке через пробел.

declare FILE=test.txt
declare SHM=/dev/shm
declare TMP=/tmp
declare CURTMP

cd
ls "$SHM" &&
   (cd $SHM
   touch "$FILE"
   CURTMP=$SHM
   echo -n "$CURTMP " ) ||
   (cd $TMP
   touch "$FILE"
   CURTMP=$TMP
echo -n "$CURTMP ")
pwd
# список файлов каталога SHM
#/dev/shm /home/username
```

```
#!/bin/bash

#5. Изменить скрипт п.4 чтобы на экран не выводилось ничего
# кроме требуемых каталогов. (удалять операторы нельзя.
# Используйте перенаправление.)

declare FILE=test.txt
declare SHM=/dev/shm
declare TMP=/tmp
declare CURTMP

cd
ls "$SHM" >/dev/null 2>&1 &&
   (cd $SHM
   touch "$FILE"
   CURTMP=$SHM
   echo -n "$CURTMP " ) ||
   (cd $TMP
   touch "$FILE"
   CURTMP=$TMP
echo -n "$CURTMP ")
pwd

#/dev/shm /home/username
```

6. Конструкции типа prg1 && prg2 || echo "ERR prg1" и prg1 || prg2 && echo "OK prg1" имеют неприятную особенность. Назвать особенность.

При выполнении конструкции **prg1 && prg2 || echo "ERR prg1"** ,даже если **prg1** выполнится без ошибки, но **prg2** выполнится с ошибкой, будет выдано сообщение **echo "ERR prg1"**

При выполнении конструкции **prg1 || prg2 && echo "OK prg1"** ,

даже если **prg1** выполнится с ошибкой, но **prg2** выполнится без ошибки, будет выдано сообщение **echo "OK prg1"**

7. С помощью группировки вместо **prg2** сделать конструкции п6 логичными (чтобы печать сообщения на экран зависела только от результата **prg1**).

```
prg1 && { prg2; true;} || echo "ERR prg1"
prg1 || { prg2; false;} && echo "OK prg1"
```

```
#!/bin/bash
#8 протестировать п.6, заменив prg1,2 программами  true и
# false в во всех сочетаниях.
```

```
false && { true; true; } || echo "ERR prg1"
#ERR prg1

false && { false; true; } || echo "ERR prg1"
#ERR prg1

true || { true; false;} && echo "OK prg1"
# OK prg1

true || { false; false;} && echo "OK prg1"
# OK prg1
```

Ошибки при выполнении задания 15.

2 Привести скрипт к виду одна команда в строке.

Напоминаю, что разделить строку можно не только с помощью \, но после символа & и | .

5. Изменить скрипт п.4 чтобы на экран не выводилось ничего кроме требуемых каталогов. (удалять операторы нельзя. Используйте перенаправление.)

Перенаправляя вывод команды **ls**, обычно забывают про перенаправление другого потока. Оба потока должны быть перенаправлены, чтобы команда **ls** не выдавала сообщение и при наличии /dev/shm и при отсутствии.

```
ls "$SHM" >/dev/null 2>&1
#или
ls "$SHM" 1>/dev/null 2>/dev/null
#или
ls "$SHM" &>/dev/null
```

Перенаправление вида **ls /tmp |&** ... нежелательно, потому что в данном случае нас интересует код возврата . А при использовании конвейера, обычно, errorlevel выставляется по последней выполненной команде конвейера.

Ответы и решения на Задание 16.

```
#!/bin/bash

#1. Инициализировать переменную DT результатом команды
# получения # даты $(date -d"20240915 21:12:37")
declare DT=$(date -d"20240915 21:12:37")

#2. Переменную DT разобрать на составные части командой set.
set -- $DT

#3. Объявить и инициализировать полученными значениями
# переменные:
# WEEK_DAY DAY MON YEAR TIME T_ZONE
#Вс 15 сен 2024 21:12:37 MSK
#ВНИМАНИЕ порядок переменных может отличаться
declare WEEK_DAY=$1 DAY=$2 MON=$3 YEAR=$4 TIME=$5 T_ZONE=$6

#4. Распечатать полученные переменные в виде VAR=var_value
echo "День недели: $WEEK_DAY"
echo "Месяц: $MON"
echo "Число: $DAY"
echo "Время: $TIME"
echo "Часовой пояс: $T_ZONE"
echo "Год: $YEAR"

#5. Установить для переменной  TIME режим экспорта.
#export TIME
# или
declare -x TIME

#6 Вывести на экран информацию об объявлениях и значениях
# всех  вышеуказанных переменных.

declare -p WEEK_DAY DAY MON YEAR TIME T_ZONE

#7. В домашнем каталоге создать файл вида
# имякомпьютера_год_месяц_день.log
declare FILE=~/"${HOSTNAME}_${YEAR}_${MON}_${DAY}.log"
touch "$FILE"

#8. Для проверки вывести результат в виде ls -l имя_файла,
# удалить файл в случае успеха.
ls -l "$FILE" && rm "$FILE"

#9. Не меняя программы пп1-6 сделать так, чтобы WEEK_DAY
# печаталась  большими буквами, а TIME T_ZONE маленькими.
declare -u WEEK_DAY=$WEEK_DAY
declare -l T_ZONE=$T_ZONE

#10. Вывести на экран. WEEK_DAY T_ZONE в виде VAR=var_value
echo "День недели: $WEEK_DAY"
echo "Часовой пояс: $T_ZONE"
```

```
#11. Вывести на экран имя скрипта.
echo "Имя скрипта: "$0

#12. Программно (из самого скрипта) вывести на экран
# содержимое самого скрипта с нумерацией не пустых строк.
cat -b $0

#13 Разобрать TIME на часы, минуты секунды.
KEEPIFS=$IFS ;IFS=:
set -- $TIME
echo "Часы:   $1"
echo "Минуты: $2"
echo "Секунды:$3"
IFS=KEEPIFS

#14 В комментарии предложить способ, как обойтись без
# переменной DT с сохранением функциональности.
#set -- $(date)

#15. При запуске перенаправить результат выполнения скрипта
в файл lesson.log.
# bash scr.sh > lesson.log
# или
# ./scr.sh > lesson.log
```

Ошибки при выполнении задания 16.

1. Замечание: В рабочих скриптах использовать разбор **$(date)** крайне не рекомендуется, потому что порядок различных компонентов даты может отличаться на разных системах или даже компьютерах. Для получения предсказуемых результатов необходимо использовать конкретные форматы команды **date**. Например, в данном скрипте мог быть использован формат типа **date "+%a %d %b %Y %H:%M:%S %Z"**

8 Поскольку имя файла используется несколько раз, было целесообразно ввести переменную FILE.

Обратите внимание:п7 расширение, обозначающее домашний каталог ~/ должно быть вынесено из двойных кавычек. Данное расширение, возможно не будет работать при запуске скрипта в виде сервиса или через cron .

Ответы и решения на Задание 17.

Теория здесь:#17

```
#!/bin/bash
#1. Получить "случайное" число номера дня для второй
# половины месяца, считая, что в месяце 30 дней.
echo $((RANDOM % 15 + 16 ))

#2. Получить случайный шестизначный код подтверждения
# (100000-999999)
echo $((SRANDOM % 900000 + 100000 ))

#3. Получить случайное число в диапазоне (-321 +123)
echo $(( RANDOM % 445 - 321 ))

#4. Получить случайное число, в диапазоне от 0 до текущего
дня месяца включительно.
echo $(( RANDOM % ($(date +%-d)) +1))

#5. Переменная B8 содержит восьмеричное число в виде строки
# "333", вывести на экран восьмеричное и десятичное значение
# переменной
B8=333; echo "0$B8" $(("0$B8"))

#6. Переменная A содержит строку "ABCD" Преобразовать
# значение переменной A из шестнадцатеричного числа в
# десятичное. При преобразовании использовать конкатенацию.
A="ABCD" ; echo $((0x$A))
```

7. J=7; echo $((J++ <++J))

7 < 7+1+1 (9). Истина. Значение 1 .

8. J=7; echo $((J--<7))

7 < 7. ложь. Значение 0.

9. K=8; ((M=--K)); echo $((K<9 && M==K))

к=8; K=7, M=7. 7<9 Истина ***И*** *7=7 (тоже истина));*
Объединение по И - итог истина. Значение 1 .

10. K=8; ((M=--K,M+=K)); echo $((M<9 || M==++K))

K=8; K=7; M=7; M=7+7=14 (14<9 (Ложь 0) ИЛИ 14=7+1 (Ложь 0)) =
Объединение по ИЛИ (0 ИЛИ 0) - итог Ложь. Значение 0.

11.J=7 ; echo $((++J==J--))

J=7, 8 = 8 Истина. Итог истина. Значение 1

Ошибки при выполнении задания 17.

2. Получить случайный шестизначный код подтверждения (100000-999999)

echo $((SRANDOM % 900000 + 100000))

Поскольку диапазон значений больше, чем 32767, использовать RANDOM для генерации нельзя. Только SRANDOM, или другие средства генерации случайных/псевдослучайных чисел, с бóльшим диапазоном.

4. Получить случайное число, в диапазоне от 0 до текущего дня месяца включительно.

echo $((RANDOM % ($(date +%-d)) +1))

Обратите внимание: Формат даты %-d. Может быть и %_d, но первый лучше. Если Использовать %d, то 8, 9 числа любого месяца результат будет 08, 09. Bash, числа начинающиеся с 0 интерпретирует, как восьмеричные. 08, 09 - не могут быть восьмеричными, поэтому возникает ошибка. Это довольно трудно обнаруживаемая ошибка.

10. Обратите внимание, даже если переменная объявлена с флагам -u (верхний регистр) или -l (нижний регистр) изменение регистра происходит не в момент изменения флага, а в момент присвоения значения переменной, поэтому требуется повторное присвоение значения переменной **declare -u WEEK_DAY=$WEEK_DAY.**

Ответы и решения на Задание 18.

Теория здесь:#18

```
#!/bin/bash
# 1. Создайте массив FRUITS
declare -a FRUITS=(яблоко банан вишня слива абрикос)

# 2. Выведите весь массив FRUITS
echo "Массив FRUITS: ${FRUITS[@]}"

# 3. Выведите второй элемент массива FRUITS
# Поскольку нумерация с 0
echo "Второй элемент FRUITS: ${FRUITS[1]}"

# 4. Добавьте в конец массива элемент лимон
FRUITS+=(лимон)

# 5. Выведите количество элементов в массиве FRUITS
echo "Количество элементов в FRUITS: ${#FRUITS[@]}"

# 6. Удалите элемент с индексом 1
unset 'FRUITS[1]'

#7. Выведите на экран все элементы массива FRUITS после
удаления с помощью declare.
echo "Массив FRUITS после удаления "
declare -p FRUITS

# 8. Создайте новый массив  BIN
declare -a BIN=(1 2 4 8 16)

# 9. Выведите сумму всех элементов массива BIN (без циклов)
# Сложение поэлементно через выражение
declare -i SUM
((SUM=BIN[0]+BIN[1]+BIN[2]+BIN[3]+BIN[4]))
echo "Сумма элементов массива BIN: $SUM"

# 10. Объедините массивы FRUITS и BIN в FRUITS и выведите
#FRUITS+=("${BIN[@]}") # добавление
FRUITS=("${FRUITS[@]}" "${BIN[@]}") # пересоздание
echo "Объединённый массив FRUITS:"
declare -p  FRUITS
```

Ошибки при выполнении задания 18.

Обратите внимание п7. После удаления элемента [1] остальные индексы не изменились.

Обратите внимание п10. Объединить можно путем добавления и пересоздания. Разница (в данном случае) будет в индексах.

Ответы и решения на Задание 19.

Теория здесь:#19

```
#!/bin/bash

# 0. используя exec , перенаправить вывод stdout и stderr
# скрипта в лог-файл с названием имя_скрипта.log. Результат
# проверки вывести в формате  "OKN" или "ERRN", где N -
# номер задания.
# Результат проверить при истинных и ложных условиях.
declare LOGFILE="$0.log"

exec &>"$LOGFILE"
#еще вариант
#exec >"$LOGFILE" 1>&2

# 1. Написать проверку, что число является шестизначным.
declare -i NUM=270545
#declare -i NUM=2705450
if ((NUM>=100000 && NUM<=999999)); then
#if [[ $NUM -ge 100000 && $NUM -le 999999 ]]; then
  echo "OK1 число шестизначное"
else
  echo "ERR1 число НЕ шестизначное"
fi

#еще вариант #1, с промежуточной переменной
#RESULT=$(($NUM / 100000))
#[[ $RESULT -gt 0 ]] && [[ $RESULT -le 9 ]]

#2. Написать проверку, что переменная YES = "Y"
declare YES="Y"
#declare YES="N"
if [[ $YES = "Y" ]]; then
  echo "OK2 Переменная YES равна Y"
else
  echo "ERR2 Переменная YES НЕ равна Y"
fi

#3. Написать проверку переменная YES1 = "Y" или "y"
declare YES1="y"
#declare YES1="N"
if [[ $YES1 = "Y" ]]; then
  echo "OK3 Переменная YES1 равна Y"
elif [[ $YES1 = "y" ]]; then
  echo "OK3 Переменная YES1 равна y"
else
  echo "ERR3"
fi
# еще вариант #3
#[[ "$YES1" = "Y" || "$YES1" = "y" ]]
```

```
# или [[ "$YES1" =~ [Yy] ]]

#4. Написать проверку, что значение переменной NO содержится
# в строке "NnHн".
declare NO=n
#declare NO=y
if [[ "NnHн" =~ $NO && -n $NO ]]; then
  echo "OK4 $NO содержится в строке "NnHн""
else
  echo "ERR4 $NO НЕ содержится в строке "NnHн""
fi

#5.  Написать проверку, что A=5, B - отрицательное и
переменная YES="n" или "N"  . При выполнении всех условий
выдать в stderr сообщение об ошибке.
declare A=-5 B=-3 YES=n
#declare A=5 B=-3 YES=n
#declare A=-5 B=3 YES=n
#declare A=-5 B=3 YES=Y
if [[ $A -eq 5  && $B -lt 0 ]] && [[ "$YES" = "n" || "$YES"
= "N" ]]; then
  echo "ERR5 Условия задания 6 выполнены" >&2
fi

#6. Написать проверку, что переменная $TEST >= "TEST".
использовать отрицание.

declare TEST="TEST1"
#declare TEST="TES"
if [[ ! $TEST < "TEST" ]] ; then
  echo "OK6 $TEST>='TEST'"
else
  echo "ERR6 условие $TEST>='TEST' не выполнено"
fi

# 7. То же, что п.7, но вместо отрицания  использовать else.
if [[ $TEST < "TEST" ]] ; then
# Вместо отрицания поменяли местами ERR и OK
  echo "ERR7"
else
  echo "OK7"
fi

#8. Создать пустой файл /tmp/test.txt, проверить, что файл
# создан. В случае успеха заполнить его любой информацией и
# провести проверку, что этот файл не "пустой" Записать
# сообщение об этом в файл /tmp/test.log
declare FILE="/tmp/test.txt"
#declare FILE="/tmp/test1.txt"
touch  "$FILE"
if [[ -f /tmp/test.txt ]] &&
   echo "Some information" > "$FILE"; then
  if [[ -s /tmp/test.txt ]]; then
```

```
    echo "file is not empty" > "/tmp/test.log"
    echo "OK8"
  fi
else
  echo "ERR8"
fi
rm "$FILE"

#9 Проверить, может ли оператор if анализировать код
# завершения  программ без условий в скобках. Например
# if ping -c3 yandex.ru ...

#if ping -c 1 "yandex.ru"; then
if ping -c 1 "adfvafvavfafv"; then
   echo "OK9"
else
  echo "ERR9"
fi

#Да, может
```

Ошибки при выполнении задания 19.

Основные ошибки - отсутствие пробелов в условиях

[[$YES1= "Y"]] или [[$YES1 = "Y"]]

#3 Написать проверку переменная YES1 = "Y" или "y"

- Обратите внимание, в данном задании переменная YES1, а не YES
- Формальная ошибка. Использование конструкции
  [["Yy" =~ $YES1]] в данном случае не верно, поскольку будет давать истину, в случае YES1="Yy" или YES1=""

Замечание: Обычно аналогичные конструкции применяются при взаимодействии с пользователем. Например:

echo "Создать файл? [N]y"

#4. Написать проверку, что значение переменной NO содержится в строке "NnHн".

- [["NnHн" =~ $NO]] Ошибка. Условие истина и при NO=''

- [["$NO" =~ NnHн && -n "$NO"]] Ошибка Неправильный порядок сравнения.

- [["$NO" =~ [NnHн]]] Формальная ошибка. Если переменная NO="Nn", то она содержится в строке "NnHн" , но при этом условие вернет ложь.

#5. Написать проверку, что A=5, B - отрицательное и переменная YES="n" или "N" . При выполнении всех условий выдать в stderr сообщение об ошибке.

```
[[ $A -eq 5 && $B -lt 0 && "$YES" = "n" || "$YES" = "N" ]]
```

Ошибка. Данное условие будет истинно при "$YES" = "N" , и любых других A и B.

Ответы и решения на Задание 20.

Теория здесь:#20

```
#!/bin/bash
declare -a A=(а о у и е)
declare -a B=(б в г ф н м к п р ц ш д л с з)

# количество слогов
declare -i SYLNR=$1
# количество слов
declare -i WORDNR=$2
# вероятность изменённого порядка в слоге
declare -i PROB=$3
#число гласных в массиве
declare -i LENA=${#A[@]}
#число согласных в массиве
declare -i LENB=${#B[@]}
# итераторы
declare -i I=0 J
# согласная, гласная, слог, слово
declare CONS VOWEL SYL WORD
# help
if [[ -z $1 ]]; then
  echo "random word generator" >&2
  echo "usage: $0 <SYLNR> [WORDNR] [PROB]" >&2
  echo "generates WORDNR words with SYLNR number of\
 syllables(consonant - vowel)" >&2
  exit 1
fi

if [[ $SYLNR -lt 1 ]]; then
  SYLNR=1
elif [[ $SYLNR -gt 5 ]]; then
  SYLNR=5
fi

if [[ $WORDNR -lt 3 ]]; then
  WORDNR=3
elif [[ $WORDNR -gt 10 ]]; then
  WORDNR=10
fi

if [[ $PROB -lt 0 ]]; then
  PROB=0
elif [[ $PROB -gt 100 ]]; then
  PROB=100
fi

while [[ $I -lt $WORDNR ]]; do
  J=0
  WORD=""
```

```
  while [[ $J -lt $SYLNR ]]; do
    # читаемость на высоте
    VOWEL=${A[ $(( RANDOM % LENA )) ]}
    CONS=${B[ $(( $RANDOM % LENB )) ]}

    # проверка вероятности
    if (( $RANDOM % 100 < $PROB )); then
      SYL="${VOWEL}${CONS}"
    else
      SYL="${CONS}${VOWEL}"
    fi

    WORD="${WORD}${SYL}"
    ((++J))
  done

  echo $WORD

  ((++I))

done
```

Другой вариант help, не красивый, но рабочий.

```
#!/bin/bash
# random word generator
# usage: $0 <SYLNR> [WORDNR] [PROB]
# generates WORDNR words with SYLNR number
# of syllables(consonant - vowel)
(($#)) || { head -5 $0 >&2; exit 1 ; }
```

Ошибки при выполнении задания 20.

Типовая ошибка - не описаны некоторые переменные.

Обратите внимание, параметры этого скрипта - числовые. Если для хранения, значения параметра выбрать переменную, описанную, как целую, то тогда, при неверно заданном параметре (например, вместо числа задали букву) ошибки не будет, просто переменная примет значение 0.

SYL=${B[$(($RANDOM % 15))]}${A[$(($RANDOM % 5))]}

Хотя эта конструкция работает, лучше чтобы число букв вычислялось, а не задавалось константой. Тогда при изменении размерности массива не придется анализировать весь код, чтобы внести изменения.

Ошибкой было бы собирать слово по буквам. Слово нужно собирать по слогам.

Ответы и решения на Задание 21.

Теория здесь:#21

```
#!/bin/bash
#1 Переписать скрипт трижды, используя while, until, for in
declare -i I J

# while
I=1
while [[ $I -lt 17 ]]; do
  J=13
    while [[ $J -gt 3 ]]; do
        echo "$I x $J = $((I*J))"
        [[ $I -eq 2 ]] && break 2
        ((--J))
    done
    ((++I))
done

# until
I=1
until [[ $I -ge 17 ]]; do
  J=13
    until [[ $J -le 3 ]]; do
        echo "$I x $J = $((I*J))"
        [[ $I -eq 2 ]] && break 2
        ((--J))
    done
    ((++I))
done

# for in
for I in {1..16}; do
    for J in {13..4..-1}; do
        echo "$I x $J = $((I*J))"
        [[ $I -eq 2 ]] && break 2
    done
done
```

```
#!/bin/bash
#2 Написать скрипт для таблицы умножения в порядке
# возрастания, но используя циклы с убыванием переменной
# цикла. (9 8...1)
declare -i I J CURI CURJ
for ((I=9; I>0; --I)); do
  for ((J=9; J>0; --J)); do
    CURI=$((10-I))
    CURJ=$((10-J))
    echo "$CURI X $CURJ = $((CURI*CURJ))"
  done
done
```

```
#!/bin/bash
#3 Что делает следующий скрипт и проставить комментарии к
# каждой строке.
# объявляем переменные I со значением 0, K и L
declare -i I=0 K L
# цикл от 0 до 98
while [[ $I -lt 99 ]];do
  # увеличиваем I на 1
  ((I++)) # 1..99
  # K - десятки переменной I
  K=I/10
  # L - единицы переменной I
  L=I%10
  # не выводим результат, если один из множителей 0.
  ((K*L)) &&  echo "$K X $L = $((K*L))"
done
```

Данный скрипт выводит таблицу умножения.
Особенностью данного скрипта является то, что он использует один цикл, вместо двух.

```
#!/bin/bash
#4. Изменить скрипт п3,  используя оператор if, для
улучшения
# читаемости. Заменить ((K*L)) двумя более понятными
условиями с
# объединением по ИЛИ
declare -i I=0 K L
while [[ $I -lt 99 ]];do
  ((I++))
  K=I/10
  L=I%10
  if ! [[ K -eq 0 || L -eq 0 ]]; then
    echo "$K X $L = $((K*L))"
  fi
done
```

```
#!/bin/bash
#5 Изменить скрипт п3, используя оператор continue.
declare -i I=0 K L
while [[ $I -lt 99 ]];do
  ((I++))
  K=I/10
  L=I%10
  if [[ K -eq 0 || L -eq 0 ]]; then
    continue
  fi
#еще один оригинальный вариант условия.
# while [[ K -eq 0 || L -eq 0 ]]; do
#    continue 2
# done

  echo "$K X $L = $((K*L))"
done
```

```
#!/bin/bash
#6.Написать скрипт для создания в переменной VAR числа,
# состоящего
# из 20 случайных цифр в диапазоне (2-8) Вывести на экран
# значение VAR

declare VAR
for ((I=0; I < 20; ++I)); do
  RAND=$(($RANDOM % 7 + 2))
  VAR="$VAR$RAND"
done
echo $VAR
```

```
#!/bin/bash
#7. Написать скрипт для создания в переменной VAR числа,
# состоящего из 20 случайных символов в диапазоне от 2 до 8
# включительно, кроме символов 4 и 6. Вывести на экран
# значение VAR.
declare VAR
declare -i I RAND
for ((I=0; I < 20;I++ )); do
  RAND=$(($RANDOM % 7 + 2))
  if [[ $RAND -eq 4 || $RAND -eq 6 ]]; then
    ((I--))
  else
    VAR="$VAR$RAND"
  fi
done
echo $VAR
# echo ${#VAR}

##### другой вариант п.7
#!/bin/bash

#declare VAR
#declare -i I=0 RAND
#while [[ I -lt 20 ]]; do
#  RAND=$(($RANDOM % 7 + 2))
#  if [[ NUM -eq 4 || NUM -eq 6 ]]; then
#    continue
#  fi
#  VAR=$VAR$RAND
#  ((I++))
#done
#echo $VAR
#echo ${#VAR}
```

Ошибки при выполнении задания 21.

1. Переписать скрипт используя...
Типовая ошибка - не правильное задание диапазонов циклов.
Проверить просто: Нужно удалить (закомментировать) условие
`[[ $I -eq 2 ]] && break 2`

цикл по I 1..16, по J 13..4

2 Написать скрипт для таблицы умножения в порядке возрастания но используя циклы с убыванием переменной цикла. (9 8...1)

Проверьте порядок, в начале должно быть "1 X 1 = 1", в конце "9 X 9 = 81"

Не ошибка, а замечание.
Описывать переменные лучше перед циклом, а в цикле просто задавать значения.

```
declare -i I=1 J
until [[ I -eq 17 ]]; do
  #declare -i J=13
  J=13
  until [[ J -eq 3 ]]; do
```

Замечание, вместо -eq лучше использовать -ge или -le. Если вдруг окажется, что нужно сделать цикл с шагом 2, то при четных значениях переменной цикла, можно "проскочить" условие и цикл станет бесконечным. В то же время -ge или -le сработает.

7. Модифицировать скрипт п6, исключив из числа цифры 4 и 6
Основная ошибка п.7, что при исключении цифр - меняется длина переменной VAR.
Для проверки правильности решения нужно добавить строку в конце скрипта **echo ${#VAR}** И выполнить скрипт несколько раз. Если **echo ${#VAR}** каждый раз выдает 20, и при этом строка состоит из цифр 2,3,5,7,8, значит задание выполнено верно.

Ответы и решения на Задание 22.

Теория здесь:#22

```
#!/bin/bash
# Creating an array, sorting it and saving it to a file
# Usage: ./task20.bash <-a N> <-b M> [-s A|D]
# N - number of array elements 10-10000
# M - array element range 3-100
# Sorting direction (A or a) - ascending (D or d)
# descending.

# параметры для  ускорения отладки
#set -- -a 1000 -b 100 -s a

#help
(($#)) || { head -6 $0 |tail -5 ; exit 1; }

declare -i M_COUNT=0 #число элементов массива
declare -i M_DIA=0 # Диапазон 0..M_DIA включительно
declare -i ORDER=0 # порядок сортировки 0=ASC 1=DESC
declare -ia M #массив
declare -i I J T MIN MIN_POS
declare FILE1="/tmp/a1.txt" # Не сортированный массив
declare FILE2="/tmp/a2.txt" # отсортированный массив
declare SORT_KEY # ключи команды sort

#parce parameters
while (($#));do
  if [[ "$1" == "-a" ]]; then
    M_COUNT=$2
    ((M_COUNT<10)) && M_COUNT=10
    ((M_COUNT>10000)) && M_COUNT=10000
  elif [[ "$1" == "-b" ]]; then
    M_DIA=$2
    ((M_DIA<3)) && M_DIA=3
    ((M_DIA>100)) && M_DIA=100
  elif [[ "$1" == "-s" ]]; then
    if [[ "$2" =~ [aA] ]]; then
      ORDER=0
    elif [[ "$2" =~ [dD] ]]; then
      ORDER=1
    else
      echo "ERROR: Wrong sort order $2" >&2
      exit 1
    fi
  else
    echo "ERROR: Wrong parameter $1"  >&2
    exit 1
  fi
  shift 2
done
```

```
# значение вне диапазона, значит параметр не обработан.
((M_COUNT)) ||
   { echo "ERROR: parameter -a not set"  >&2 ; exit 1 ;}
((M_DIA)) ||
    { echo "ERROR: parameter -b not set"  >&2 ; exit 1 ;}
#check parameters range

#fill array
((++M_DIA))
for ((I=0;I<M_COUNT;I++));do
  (( M[I] = SRANDOM % M_DIA ))
done

#clear FILE1 FILE2
:>"$FILE1"
:>"$FILE2"

#print array to FILE1
for ((I=0;I<M_COUNT;I++));do
  echo ${M[$I]} >>"$FILE1"
done

#sort array
SECONDS=0
for ((I=0;I<(M_COUNT-1);I++));do
  MIN=${M[I]};MIN_POS=I
  for ((J=I;J<M_COUNT;J++));do
    ((M[J] < MIN)) && ((MIN=M[J], MIN_POS=J))
  done
  # swap min element
  ((T=M[I], M[I]=MIN, M[MIN_POS]=T))
done

echo "sort time $SECONDS sec"

#print sort array to FILE2
if [[ $ORDER -eq 0 ]];then # ASC
  for ((I=0;I<M_COUNT;I++));do
    echo ${M[$I]} >>"$FILE2"
  done
else # DESC
  for ((I=(M_COUNT-1);I>=0;I--));do
    echo ${M[$I]} >>"$FILE2"
  done
fi

if [[ $ORDER -eq 0 ]];then # ASC
  SORT_KEY="-n"
else # DESC
  SORT_KEY="-rn"
fi
```

```
sort "$SORT_KEY" "$FILE1" | diff - "$FILE2" ||
   echo "ERROR: Wrong sort" >&2
```

Ошибки при выполнении задания 22.

Чем сложнее задача, тем больше вариантов ее решения.

Основные ошибки при выполнении задания:
1.Нет проверки, что заданы все обязательные параметры. проблема возникает, например, если запустить программу с параметрами
 -a 100 -s a .
2.Несоответствие сгенерированных данных заданным параметрам.
 Отладить очень просто: запустите скрипт с параметрами
-a 1000 -b 10 -s a
При этом wc -l /tmp/a2.txt должно быть 1000
head -1 /tmp/a2.txt должно быть 0
tail -1 /tmp/a2.txt должно быть 10

Чтобы проверить правильность сортировки при отладке - заполните массив сначала номерами элемента M[0]=0, M[1]=1... M[N]=N, а потом в обратном порядке M[0]=N, M[1]=N-1... M[N]=0. Массив на выходе должен точно соответствовать порядку сортировки.

3. Если все правильно сделано, программа должна обрабатывать шестнадцатеричные и восьмеричные числа в параметрах. Запустите программу с параметрами -a 0x100 -b 0100 -s a
При этом wc -l /tmp/a2.txt должно быть 256
tail -1 /tmp/a2.txt должно быть 64

4. Обратите внимание: Если максимальный номер массива N внешний цикл сортировки должен быть I=0..(N-1), а внутренний J=I..N

5. Ошибка при сравнении сортированных файлов может возникать, если не указать ключ -n у команды sort .

Ответы и решения на Задание 23.

Теория здесь:#23

```
#1. Привести оператор для ввода трех любых переменных с
# подсказкой
read -p "enter name, age and email(space separated): "\
     NAME AGE EMAIL

#2. Запросить ввод пароля.
read -sp "enter password: " PASSWD; echo

#3. Запросить ввод почтового индекса. По-умолчанию
# использовать 103123
read -ei "103123" -p "enter post index: " INDEX

#4. Запросить ввод почтового индекса. По-умолчанию
# использовать 103123, с ограничением времени ввода 10
# секунд.
read -t 10 -ei "103123" -p "enter post index: " INDEX

#5. Организовать ввод номера мобильного телефона.
read -n 16 -ei "+7" -p "phone number: " PHONE_NUMBER

#6. Запросить у пользователя заполнение массива FIO.
# Массив должен быть предварительно описан.
declare -a FIO
read -p "name surname lastname(space separated): " -a FIO

#7. Протестировать использование разделителя полей.
IFS="@" read -p "enter e-mail>: " MAILUSER DOMAIN

#8. Сконструировать оператор, с функцией "Press any key to
# continue", и временем ожидания не более 12 минут.
read -t $((60 * 12)) -N 1 -sp "Press any key to continue:"\
    ; echo
```

9. Написать скрипт, который будет в бесконечном цикле выдавать примерно один символ "#" в секунду и прерываться он должен при нажатии на клавиатуре на этот символ. Символы должны выводиться в одну строку, после вывода 20 символа строка (но не экран) должна очищаться. На всякий случай напомню, прерывание бесконечных циклов CTRL-C

```
#!/bin/bash
declare NUM=20
declare -i I=0
while true; do
  ((++I))
  read -t 1 -N 1 -sp "#"
  [[ "$REPLY" == "#" ]] && { echo; break; }
  if [[ $I -ge $NUM ]]; then
```

```
    echo -ne "\r"
    while ((I--)); do
      echo -n " ";
    done
    echo -ne "\r"
    I=0
  fi
done
```

Ошибки при выполнении задания 23.

Задание было очень простым.

Ответы и решения на Задание 24.

Теория здесь:#24

```
#!/bin/bash
#Задание 22
declare FILE_NAME=/tmp/IP.txt
declare -i X Y STR_NUM=10000
declare -A UNIQUE_IP
declare -iA COUNT_SUBNET
declare -i MAX_COUNT CUR_COUNT
declare SUBNETFILE

# 1. В каталоге /tmp создать файл IP.txt, состоящий из 10000
# строк. Каждая строка представляет IP адрес 192.168.X.Y
# X,У , - случайные. X в диапазоне 0-3, Y  в диапазоне 1-255

:>"$FILE_NAME"
for ((i=1; i<=STR_NUM; i++)); do
    X=$((SRANDOM % 4))
    Y=$((SRANDOM % 255 + 1))
    IP=192.168.$X.$Y
    echo "$IP" >> "$FILE_NAME"
done

#2. Вывести на экран только уникальные записи IP.txt
# (Считать записи из /tmp/IP.txt в ассоциативный массив с
# индексом IP и вывести на экран) Подсчитать количество
# уникальных записей).

while read IP;do
  ((UNIQUE_IP["$IP"]++))
done < "$FILE_NAME"
echo "Уникальные IP-адреса:"
for IP in "${!UNIQUE_IP[@]}"; do
  echo "$IP"
done
echo
echo "Количество уникальных записей: ${#UNIQUE_IP[@]}"
echo

# 3. Подсчитать максимальное количество одинаковых IP (При
# записи в ассоциативный массив инкрементировать значение)
MAX_COUNT=${UNIQUE_IP} #first element
for CUR_COUNT in "${UNIQUE_IP[@]}"; do
    ((CUR_COUNT > MAX_COUNT)) && MAX_COUNT=CUR_COUNT
done

# 4. Вывести на экран все IP, которые встречаются
# максимальное количество раз.
echo IP с максимальным количеством повторений $MAX_COUNT:
for IP in "${!UNIQUE_IP[@]}"; do
```

```
    ((${UNIQUE_IP[$IP]} == MAX_COUNT)) && echo "$IP"
done
echo

#5. Подсчитать количество всех IP в файле /tmp/IP.txt , с
# одинаковой подсетью /24 (IP у которых первые три октета
# совпадают Например 192.168.0 ... 192.168.3)

exec 3<$FILE_NAME
while read -e -u3 IP;do
  IFS="." read OCT1 OCT2 OCT3 _ <<<$IP
  SUBNET="${OCT1}.${OCT2}.${OCT3}"
  ((COUNT_SUBNET[$SUBNET]++))
done
echo количество IP с одинаковой подсетью /24
for SUBNET in ${!COUNT_SUBNET[@]};do
  echo "${SUBNET}.0/24 ${COUNT_SUBNET[$SUBNET]}"
 :>"/tmp/${SUBNET}.0.txt" #clear files for #6
done

# 6 Разделить уникальные (из п.2) IP с одинаковой
# подсетью /24 по файлам 192.168.0.0.txt ...
# 192.168.3.0.txt в каталоге /tmp.
# Считать, что начальное количество подсетей неизвестно.
# (информацию о номере подсети получать из самого IP)

for IP in ${!UNIQUE_IP[@]}; do
  IFS="." read OCT1 OCT2 OCT3 _ <<<$IP
  SUBNETFILE="/tmp/${OCT1}.${OCT2}.${OCT3}.0.txt"
  echo $IP >>"$SUBNETFILE"
done

#7. Вывести на экран всех пользователей системы с UID < 1000
# и их shell по-умолчанию.
# (файл /etc/passwd) В связи с чувствительностью информации
# публиковать только код (без результатов исполнения).

declare FILE=/etc/passwd
while IFS=: read NAME PASSW UUID GID FULLNAME HOMEDIR USHELL
do
    ((UUID < 1000)) && echo "User '$NAME', \
has shell $USHELL"
done <"$FILE"
```

Ошибки при выполнении задания 24.

1. Отладка: Проверьте, что размер файла /tmp/IP.txt 10000 записей: wc -l /tmp/IP.txt

2 Количество уникальных записей: 1020 (4 подсети * 255 адресов). Теоретически может быть меньше, но вероятность такого события довольно мала.

3 IP с максимальным количеством повторений скорее всего будет 19-24

5 количество IP с одинаковой подсетью /24 будет около 2500. В этом пункте использованы операторы exec 3<$FILE_NAME и read -u3. В данном, конкретном случае можно было бы без них обойтись. Но я всегда перестраховываюсь, когда приходится использовать второй оператор read внутри цикла чтения файла.

6 Число строк файлов 192.168.0.0.txt ... 192.168.0.3.txt - 255.

7. Если переменную IFS=: Задаёте глобально для всего скрипта, а не для одного оператора read, желательно сначала запомнить ее значение в другую переменную, а после использования - восстановить. Привыкайте делать правильно, сэкономите много времени и нервов.

Ответы и решения на Задание 25

Теория здесь:#25

```
#!/bin/bash
# head.sh -n N [FILENAME...]
# print first N lines from FILENAME, if present, stdin
# otherwise.
(($#)) || { "$0" -n 4 "$0" >&2 ; exit; } # самореклама :-)

declare -i N=0 I=0
declare -a FILENAMES
declare CUR_FILE LINE

if [[ "$1" == "-n" ]];then
  N=$2
  shift 2
elif [[ "$1" =~ "-n" ]];then
  N=${1#-n}
  shift
fi

if [[ "$N" -eq 0 ]];then
  echo "invalid parameter for -n or it's missing" >&2
  exit
fi

# после shift , $@ содержит только файлы

FILENAMES+=("$@")

# если файлов 0 - добавляем stdin
((${#FILENAMES[@]})) || FILENAMES[0]="/dev/stdin"

for CUR_FILE in "${FILENAMES[@]}" ;do # цикл по файлам
# если файлов несколько выводим имя в stderr
  ((${#FILENAMES[@]}>1)) && echo "$CUR_FILE"  >&2
  I=N
  while read -e LINE;do # цикл по строкам файла
    echo "$LINE"
    ((--I)) || break
  done<"$CUR_FILE"
done
```

```
#!/bin/bash
# grep.sh "text" FILENAME...
# print lines from FILENAME(s) containing "text".

(($#)) || { ./head.sh -n 3 $0 >&2; exit; }

declare -a FILENAMES
declare TEXT
```

```
TEXT=$1
shift
if (($#));then #есть файлы в параметрах
  FILENAMES+=("$@")
else # если файлов 0 - добавляем stdin
   FILENAMES[0]="/dev/stdin"
fi

for CUR_FILE in "${FILENAMES[@]}" ;do # цикл по файлам
# если файлов несколько выводим имя в stderr
  ((${#FILENAMES[@]}>1)) && echo "$CUR_FILE"  >&2
  while read -e LINE;do # цикл по строкам файла
    [[ "$LINE" =~ "$TEXT" ]] && echo "$LINE"
  done<"$CUR_FILE"
done
```

```
#!/bin/bash
# tail.sh -n N [FILENAME...]
# print last N lines from FILENAME, if present, stdin
otherwise.
(($#)) || { ./head.sh -n 3 $0 >&2; exit; }


declare -i N=0 I=0
declare -a FILENAMES CACHE
declare CUR_FILE LINE
declare -i REC_NUM CACHE_POS

if [[ "$1" == "-n" ]];then
  N=$2
  shift 2
elif [[ "$1" =~ "-n" ]];then
  N=${1#-n}
  shift
fi

if [[ "$N" -eq 0 ]];then
  echo "invalid parameter for -n or it's missing" >&2
  exit
fi

# после shift , $@ содержит только файлы

FILENAMES+=("$@")

# если файлов 0 - добавляем stdin
((${#FILENAMES[@]})) || FILENAMES[0]="/dev/stdin"

for CUR_FILE in "${FILENAMES[@]}" ;do # цикл по файлам
# если файлов несколько выводим имя в stderr
  ((${#FILENAMES[@]}>1)) && echo "$CUR_FILE"  >&2
  CACHE=()
```

```
  I=0
  while read -e LINE;do # цикл по строкам файла
    ((CACHE_POS=I++ % N))
    CACHE[$CACHE_POS]="$LINE"
  done<"$CUR_FILE"

  for ((I =++CACHE_POS; I < ${#CACHE[@]}; ++I)); do
    echo ${CACHE[$I]}
  done
  for ((I = 0; I <${#CACHE[@]}; ++I)); do
    echo ${CACHE[$I]}
  done
done
```

Ошибки при выполнении задания 25

Методика тестирования:

1.Выполните скрипты без параметров и посмотрите, работает ли help

```
./head.sh
./tail.sh
./grep.sh
```

2. Создайте файл file1 с числами {1..10} по одному в строке.

Проверьте работу параметров:

```
./head.sh -n 5 file1
./tail.sh  -n 5 file1
./grep.sh 1 file1
```

3. то же, но без пробелов

```
./head.sh -n5 file1
./tail.sh  -n5 file1
./grep.sh 1 file1
```

4. Проверить head и tail для диапазона, больше файла.

```
./head.sh -n 15 file1
./tail.sh  -n 15 file1
```

5 Проверить вывод имени файла, если файлов несколько

```
./head.sh -n 5 file1 file1 file1
./tail.sh  -n 5 file1 file1 file1
./grep.sh 1 file1 file1 file1
```

6 Проверить, что вывод имени файла идет именно на stderr (имя файла не должно отобразиться)

```
./head.sh -n 5 file1 file1 file1 2>/dev/null
./tail.sh  -n 5 file1 file1 file1 2>/dev/null
./grep.sh 1 file1 file1 file1 2>/dev/null
```

7. Проверить работу с stdin

```
cat file1|./head.sh -n 5
cat file1|./tail .sh -n 5
cat file1|./grep .sh -n 5
```

8. Проверить правильность обработки файлов с пробелами:

```
cp file1 "file 2"
./head.sh -n 5 file1 "file 2"
./tail.sh  -n 5 file1 "file 2"
./grep.sh 5 file1 "file 2"
```

9. Проверить оптимальность алгоритма tail.sh. Если выполнение команды существенно дольше, чем п.5 Значит алгоритм обработки кэша выбран не оптимально. Но это не ошибка, а замечание.

```
./tail.sh -n 500000 file1 file1 file1
```

Ответы и решения на Задание 26

Теория здесь:#26

```
#!/bin/bash
#1. Выделить из переменной $0 каталог, имя файла с
# расширением,  имя файла без расширения, и расширение.
declare DIR=${0%/*}
declare FILEEXT=${0##*/}
declare FILE=${FILEEXT%.*}
declare EXT=${FILEEXT#*.}
declare -i I

echo "1============================"
echo "DIR     $DIR" #каталог
echo "FILEEXT $FILEEXT" # файл с расширением
echo "FILE    $FILE" # файл без расширения
echo "EXT     $EXT" # расширение

#2. Инициализировать переменную VIRSH четверостишием
# из любого цензурного стихотворения.
# (в переменной должны  присутствовать переносы строки).
# Добавить вначале и в конце VIRSH по одному пробелу.
# Вывести переменную VIRSH на экран, в виде стихотворения.

echo "2============================"
declare VIRSH="Будь добрым, не злись, \
обладай терпеньем\nЗапомни: от светлых улыбок твоих\n\
Зависит не только твоё настроенье,\nНо тысячу раз \
настроенье других."
VIRSH=" $VIRSH "
echo -e $VIRSH # печать без пробелов
#echo -e "$VIRSH" # печать с пробелами

#3. В переменной VIRSH Заменить любые символы, кроме букв
# пробелами, преобразовать в верхний регистр. Результат
# записать в переменную  VIRSH_FLT, вывести на экран.

echo "3============================"
declare -u VIRSH_FLT=${VIRSH//[^A-Яа-я]/ }
#Только буквы и пробелы
echo -e "$VIRSH_FLT"

# другой вариант #3
#declare VIRSH_FLT=${VIRSH//[^A-Яа-я]/ }
#Только буквы и пробелы
#VIRSH_FLT=${VIRSH_FLT^^} верхний регистр
echo "VIRSH_FLT"
echo -e $VIRSH_FLT

#4 В переменной VIRSH_FLT Заменять два пробела подряд одним
# пробелом, пока произведена хоть одна замена. Результат
```

```
# записать в переменную
#  VIRSH_1SP
echo "4============================"
declare VIRSH_1SP
shopt -s extglob
VIRSH_1SP="${VIRSH_FLT//+(' ')/ }"   # :-)
shopt -u extglob # выключать не обязательно

#другой вариант #4
#declare VIRSH_1SP=$VIRSH_FLT
#until [[ "$VIRSH_1SP" == "${VIRSH_1SP//  / }" ]];do
#  VIRSH_1SP="${VIRSH_1SP//  / }"
#done

echo  "$VIRSH_1SP"

#5. В переменной VIRSH_1SP удалить начальный и конечный
# пробелы, если существуют, результат записать в переменную
# VIRSH_TRM. Вывести на экран, показав, что концевых
# пробелов нет.
echo "5============================"
declare VIRSH_TRM="$VIRSH_1SP"
[[ "${VIRSH_TRM:0:1}" == " " ]] &&
    VIRSH_TRM=${VIRSH_TRM:1}
[[ "${VIRSH_TRM:(-1):1}" == " " ]] &&
    VIRSH_TRM=${VIRSH_TRM:0:(-1)}
echo "=VIRSH_1SP="; echo ="$VIRSH_1SP"=
echo ="VIRSH_TRM=";echo ="$VIRSH_TRM"=

# два пункта 4 и 5 можно было заменить одной командой
# $VIRSH_FLT без кавычек
#4,5 VIRSH_TRM=$(echo $VIRSH_FLT)

#6. Разбить VIRSH_TRM на слова и распечатать их на экране,
# по два в строке, используя set без shift
echo "6============================"
set -- $VIRSH_TRM
I=-1
while ((I < ${#@}));do
  I=I+2
  echo ${@:I:2}
done

#7. Разбить VIRSH_TRM на слова и распечатать их на экране,
# по два в строке, используя read -a
echo "7============================"
read -a ARR <<<"$VIRSH_TRM"
I=0
while ((I < ${#ARR[@]}));do
  echo ${ARR[@]:I:2}
  I=I+2
done
```

```
#8. Разбить VIRSH_TRM на слова и распечатать их на экране,
# по два в строке, используя цикл for in
echo "8============================"
I=0
for WORD in ${VIRSH_TRM[@]};do
if ((I++ % 2));then
  echo "$WORD"
else
  echo -n "$WORD "
fi
done
echo

#9.Разбить VIRSH_TRM на слова и распечатать их на экране,
# по два в строке, используя цикл while и оператор read
# и добавить при печати длину каждого слова.

echo "9============================"
I=0
while read -d" " WORD ;do
if ((I++ % 2));then
  echo "$WORD (${#WORD})"
else
  echo -n "$WORD (${#WORD}) "
fi
done <<<"$VIRSH_TRM "
echo
```

Ошибки при выполнении задания 26.

1. Выделить из переменной $0 каталог, имя файла и расширение.

Ошибка: PATH=${0%/*}
переменная PATH является системной, и применяется для определения каталогов, в котором система ищет программы, запускаемые без указанного пути. Ее модификация допустима, но обязательно нужно понимать, для чего это делается. В данном случае использование переменной PATH является ошибкой. Лучше использовать переменную DIR. Напомню, что разделителем полей переменной PATH является двоеточие :

3. В переменной VIRSH Заменить любые символы, кроме букв пробелами, преобразовать в верхний регистр. Результат записать в переменную VIRSH_FLT, вывести на экран.

Замечание: При выполнении этого задания еще можно перебирать строку в цикле посимвольно, исключая "ненужные". Но использование регулярных выражений выполняется быстрее.

4 В переменной VIRSH_FLT Заменять два пробела подряд одним пробелом, пока произведена хоть одна замена. Результат записать в переменную VIRSH_1SP

Замечание: Данное задание может быть выполнено аналогично предыдущему. но регулярные выражения выполняются быстрей.

5. В переменной VIRSH_1SP удалить начальный и конечный пробелы, если существуют, результат записать в переменную VIRSH_TRM. Вывести на экран, показав, что концевых пробелов нет.

Замечание: Пример, приведенный в п5 удаляет только один пробел вначале и один в конце переменной VIRSH_1SP . Здесь нужно учитывается то, что в п.4 были удалены все множественные пробелы.

Общее замечание. В программе в нескольких пунктах слова выводятся попарно. Для корректности кода необходимо провести еще одну проверку на чет/нечет числа слов. Добавьте к стихотворению еще одно слово, не подпадающее под фильтрацию п3. Например:
declare VIRSH="тест Будь добрым, не злись, обладай терпеньем\
n3апомни: от светлых улыбок твоих\n3ависит не только твоё настроенье,\nНо тысячу раз настроенье других."

Если программа все пары слов выводит корректно - поздравляю.

Ответы и решения на Задание 27.

Теория здесь:#27

```
#!/bin/bash

function ABS() {
# модуль целого числа ABS -5
  declare -i A=${1:-0}
  if [[ $A -lt 0 ]]; then
    A=$((-A))
  fi
  echo $A
}

function SIGN() {
# Знак целого числа SIGN -5
  declare -i A=${1:-0}
  if [[ $A -lt 0 ]]; then
    A=-1
  elif [[ $A -gt 0 ]]; then

    A=1
  fi
  echo $A #default 0
}

function DICE() {
# имитация броска кубика DICE
  declare DICE=("⚀" "⚁" "⚂" "⚃" "⚄" "⚅")
  echo ${DICE[$((SRANDOM % 6))]}
}

function REV() {
# строка в реверсивной последовательности REV "Тест"
  declare -i I
  declare RES=""
  I=${#1}
  while ((I--)); do
    RES+="${1:$I:1}"
  done
  echo $RES
}

function CHECK_IP4() {
# проверка правильности адреса IP4  CHECK_IP4 "192.168.1.1"
# errorlevel=0 - правильный
  declare -i I=4 IOCT
  declare IFS="." OCT
  for OCT in $1 ; do
    IOCT=OCT
    ((IOCT<0 || IOCT>255)) && return 1 # ! диапазон
```

```
    [[ "$OCT" != "$IOCT" ]] && return 2 # ! число
    ((I--))
  done
  ((I)) && return 3 # ! 4 октета
  return 0
}

function CHECK_CODE() {
# ввод кода подтверждения CHECK_CODE "123456"
declare PARAM_CODE=$1
declare CONF_CODE
read -rp "Confirm code: " CONF_CODE
[[ "$PARAM_CODE" == "$CONF_CODE" ]] && return 0
return 255
}
```

```
#!/bin/bash
# kids encoder
declare -A MAP
declare -i I=0 J
declare E D
declare CODE_STR="ракидемонуля"
CODE_STR+=${CODE_STR^^} #add UP case
declare -i CODE_LEN=${#CODE_STR}
#init
while ((I < CODE_LEN));do
  E=${CODE_STR:$I:1} #encode key
  ((I++))
  D=${CODE_STR:$I:1} #decode key
  MAP[$E]=$D;MAP[$D]=$E # both key
  ((I++))
done
# encrypt/decrypt
declare C=$@ STR='' CHAR
I=0
[[ -z "$C" ]] && read -p "enter text: " C
while ((I<${#C}));do
  CHAR=${C:$I:1}
  [[ -n ${MAP[$CHAR]} ]] && CHAR=${MAP[$CHAR]}
  STR+=$CHAR
  ((I++))
done
echo $STR
```

Ошибки при выполнении задания 27.

Тестовый скрипт для проверки функций:

Просто вставьте описание ваших функций и выполните.

И, конечно, разберите, как он работает.

```
#!/bin/bash
clear

# На этом месте должно быть описание Ваших функций:
#ABS
#SIGN
#DICE
#REV
#CHECK_IP4
#CHECK_CODE



declare -i I

(( 5==$(ABS 5) )) && echo "ABS pass" || echo "ABS fail 5"
(( 0==$(ABS 0) )) && echo "ABS pass" || echo "ABS fail 0"
(( 5==$(ABS -5) )) && echo "ABS pass" || echo "ABS fail -5"

((1==$( SIGN 5))) && echo "SIGN pass" || echo " SIGN fail 5"
((0==$( SIGN 0))) && echo "SIGN pass" || echo " SIGN fail 0"
(( -1==$( SIGN -5) )) && echo "SIGN pass" ||
     echo " SIGN fail -5"


[[ "$(REV '0123456789')" == "9876543210" ]] &&
     echo "REV pass" ||
     echo "REV fail"

declare -a TRUE_IP=("192.168.1.1" "0.0.0.0" "127.0.0.1" \
"255.255.255.255")
declare -a FALSE_IP=("192.168.   1.1" "192.168.01.1" \
"192.168.311.1" "192.168.11.1.5" "192.168.1." "192.168.1")
declare IP


echo должно быть pass ===
for IP in "${TRUE_IP[@]}";do
  CHECK_IP4 "$IP" && echo "$IP pass" || echo "$IP fail"
done
echo ===
echo должно быть fail ===
for IP in "${FALSE_IP[@]}";do
  CHECK_IP4 "$IP" && echo "$IP pass" || echo "$IP fail"
done
echo ===

echo 123ABC|CHECK_CODE 123ABC
if (($?==0));then
   echo CHECK_CODE test1 pass
else
   echo CHECK_CODE test1 fail
fi
```

```
echo 123ABC|CHECK_CODE 123ABD
if (($?==255));then
   echo CHECK_CODE test2 pass
else
   echo CHECK_CODE test2 fail
fi
echo ===

declare -A UNI_DICE
for I in {1..60};do
  UNI_DICE["$(DICE)"]=""
done
if (${#UNI_DICE[@]}==6));then
   echo DICE count pass
else
  echo DICE count fail
fi
echo  ${!UNI_DICE[@]}
echo -e "строка кубиков должны присутствовать все грани."
```

Для тестирования детского шифрования расшифруйте фразу:

Пмзеарвялю! Тдпдаь Вы зурдтд, чтм Арик Едомумв С Унял Мбмиаряк Имамял!

Ответы и решения на Задание 28.

Теория здесь:#28

```
#!/bin/bash
declare T1 T2
declare FILENAME="/tmp/text.txt"
declare ENC_FILENAME="/tmp/text.enc"
declare SEP=$'\t' # Разделитель TAB
#KIDS_CODER init
declare -A MAP
declare E D
declare -i I=0 J
declare CODE_STR="ракидемонуля"
CODE_STR+=${CODE_STR^^} # добавили БОЛЬШИЕ
declare -i CODE_LEN=${#CODE_STR}

while ((I < CODE_LEN));do
  E=${CODE_STR:$I:1}
  ((I++))
  D=${CODE_STR:$I:1}
  MAP[$E]=$D;MAP[$D]=$E #кодовые пары
  ((I++))
done

function KIDS_CODER(){
# kids encoder
declare -i I=0 J
# encrypt/decrypt
declare C=$@ STR='' CHAR
I=0
while ((I<${#C}));do
  CHAR=${C:$I:1}
  [[ -n ${MAP[$CHAR]} ]] && CHAR=${MAP[$CHAR]} #кодировка
  STR+=$CHAR # Накопление
  ((I++))
done
echo $STR
}
function CREATE_FILE(){
  declare -a MAP=( "лестница" "темнота" "район" "куст" \
                "художник" "политика" "сержант" "вещество" \
                "база" "техника" "девица" "медведь" \
                "собрание" "испытание" "кран" "местность" )
  declare -i LEN=${#MAP[@]} I J K
  : > "$FILENAME" #clear file
  for ((I = 0; I < $LEN; ++I)); do
    for ((J = 0; J < $LEN; ++J)); do
      for ((K = 0; K < $LEN; ++K)); do
        echo "${EPOCHREALTIME}${SEP}${MAP[$I]}${SEP}$
{MAP[$J]}\
${SEP}${MAP[$K]}" >> $FILENAME
```

```
      done
    done
  done
}

function REALTIME(){
  # REALTIME <T1> [T2]
  local T2=${2:-$EPOCHREALTIME}
  local T1=${1:-$T2} SIGN="" #для выравнивания можно пробел
  #убрали разделители, преобразовали в целые
  local -i T3 T1=${T1/[^0-9]/} T2=${T2/[^0-9]/}
  if ((T3=T2-T1,T3<0));then
    ((T3=-T3))
    SIGN="-"
  fi
    ((T1=T3/1000000,T2=T3%1000000))
    printf "%s%i.%06i\n" "$SIGN" $T1 $T2
}

function ENCRYPT_FILE(){
  :>"$ENC_FILENAME"
  IFS="$SEP"
  while read -e LINE; do
    KIDS_CODER "$LINE">>"$ENC_FILENAME"
  done < "$FILENAME"
}

#3.1 Время создания по time  и EPOCHREALTIME
echo "3.1 file creation: "
echo -n "time:"
time CREATE_FILE
T1=$EPOCHREALTIME
CREATE_FILE
T2=$EPOCHREALTIME
echo "realtime: $(REALTIME $T1 $T2)"

#3.2 Время сознания по первой и последней строке
set -- $(head -1 /tmp/text.txt)
T1="$1"
set -- $(tail -1 /tmp/text.txt)
T2=$"$1"
echo "3.2 timestamps: $(REALTIME $T1 $T2)"

#3.3 время кодирования
echo "3.3 file encryption: "
echo -n "time:"
time ENCRYPT_FILE

T1=$EPOCHREALTIME
ENCRYPT_FILE
echo "realtime: $(REALTIME $T1)"
```

```
#!/bin/bash
# 4. преобразование первого поля EPOCHREALTIME
# в человекочитаемый формат

declare FILENAME="/tmp/text.txt"
declare -a REC
declare -i FIELD_NUM I

  while read -a REC; do # считали запись в массив
    set -- ${REC[0]/[^0-9]/ } # заменили разделитель
пробелом
    printf "%(%Y%m%d-%H%M%S)T.%s" $1 $2 #
человекочитаемый :-)
    FIELD_NUM=${#REC[@]}
    I=0
    while ((++I<FIELD_NUM));do # остальные поля, сколько
есть
      printf "\t%s" ${REC[$I]} # разделитель TAB
    done
    echo # LF
  done < "$FILENAME"
```

Ошибки при выполнении задания 28.

1. Есть слова "лестница" "темнота" "район" "куст" "художник" "политика" "сержант" "вещество" "база" "техника" "девица" "медведь" "собрание" "испытание" "кран" "местность". Создать файл /tmp/text.txt. Строка состоит timestamp (EPOCHREALTIME) и трех слов. Разделитель колонок - табуляция. Файл должен быть максимальной длины. В файле не должно быть строк со словами в той же последовательности. (строки с полями 2-4 уникальные по файлу) Слова в строке могут повторяться.

После создания файла /tmp/text.txt Нужно проверить число строк:

```
 wc -l /tmp/text.txt
#4096 /tmp/text.txt
```

2 Написать функцию для измерения времени REALTIME с помощью переменной EPOCHREALTIME. bc, awk, и другие программы работы с вещественной арифметикой не использовать. В качестве разделителя целой и дробной частей использовать "." при выводе. В исходных переменных в качестве разделителя использовать не цифру.

Для проверки используйте код:

```
T1="1748442351.087072"
T2="1748442351,287072"
REALTIME $T1 $T2
REALTIME $T2 $T1
```

```
#0.200000
#-0.200000
```

3.Шифрование файла /tmp/text.txt в файл /tmp/text.enc с помощью детского алгоритма из предыдущего задания

Не ошибки, просто обратите внимание. При конверсии разделитель полей табуляция заменяется пробелом.

Если время кодирования больше чем в 100 раз отличается от времени создания исходного файла - алгоритм кодирования не оптимален.

4 Написать скрипт для вывода файла /tmp/text.txt на экран, с преобразованием первой колонки в "человекочитаемый" вид например 20250111-101127.012345

Формат вывода может быть любым, но вся информация должна сохраниться. Т.е 20250111_101127 -ошибка. отсутствуют микросекунд.

Ответы и решения на Задание 29.

Просто практика

```
#!/bin/bash

declare FILENAME="/tmp/file"
[[ -d "/dev/shm" ]] && FILENAME="/dev/shm/file"


function CREATE_FILE1(){
  declare -a MAP=( "лестница" "темнота" "район" "куст" \
                "художник" "политика" "сержант" "вещество" \
                "база" "техника" "девица" "медведь" \
                "собрание" "испытание" "кран" "местность" )
  declare -i LEN=${#MAP[@]} I
  declare TASK_FILENAME="${FILENAME}.1"
  : > "${TASK_FILENAME}" #clear file
  for I in {1..100000};do
    echo ${MAP[SRANDOM % LEN]} ${MAP[SRANDOM % LEN]} \
${MAP[SRANDOM % LEN]}>>"${TASK_FILENAME}"
  done
  [[ -s "${TASK_FILENAME}" ]]
}

function CREATE_FILE2(){
  declare LINE
  declare FILENAME1="${FILENAME}.1"
  declare TASK_FILENAME="${FILENAME}.2"
  : > "${TASK_FILENAME}" #clear file
  while read -e LINE;do
    if [[ "$LINE" =~ "куст" ]];then
      echo $LINE >>"${TASK_FILENAME}"
    fi
  done <"$FILENAME1"
  [[ -s "${TASK_FILENAME}" ]]
}

function CREATE_FILE3(){
  declare LINE
  declare FILENAME1="${FILENAME}.1"
  declare TASK_FILENAME="${FILENAME}.3"
  : > "${TASK_FILENAME}" #clear file
  while read -e LINE;do
    if [[ "$LINE" =~ "собрание" &&
     !("$LINE" =~ "вещество") ]];then
      echo $LINE >>"${TASK_FILENAME}"
    fi
  done <"$FILENAME1"
  [[ -s "${TASK_FILENAME}" ]]
}
```

```
function CREATE_FILE4(){
  declare -a LINE
  declare FILENAME1="${FILENAME}.1"
  declare TASK_FILENAME="${FILENAME}.4"
  : > "${TASK_FILENAME}" #clear file
  while read -a LINE;do
#возможен вариант разбивки на слова с помощью set
    if [[ "${LINE[2]}" =~ "медведь" ]];then
      echo ${LINE[@]} >>"${TASK_FILENAME}"
    fi
  done <"$FILENAME1"
  [[ -s "${TASK_FILENAME}" ]]
}
function CREATE_FILE5(){
  declare LINE
  declare FILENAME1="${FILENAME}.1"
  declare TASK_FILENAME="${FILENAME}.5"
  : > "${TASK_FILENAME}" #clear file
  while read -e LINE;do
#    if [[ "$LINE" =~ ("лестница сержант"|
       "сержант лестница")  ]];then
    if [[ "$LINE" =~ "лестница сержант" ||
       "$LINE" =~ "сержант лестница"  ]];then
      echo $LINE >>"${TASK_FILENAME}"
    fi
  done <"$FILENAME1"
  [[ -s "${TASK_FILENAME}" ]]
}
function CREATE_FILE6(){
  declare LINE1="" LINE2
  declare FILENAME1="${FILENAME}.1"
  declare TASK_FILENAME="${FILENAME}.6"
  : > "${TASK_FILENAME}" #clear file
  while read -e LINE2;do
    if [[ "$LINE2" =~ "медведь" && "$LINE1" =~
"местность"  ]];then
      echo -e "$LINE1\n$LINE2" >>"${TASK_FILENAME}"
    fi
    LINE1="$LINE2"
  done <"$FILENAME1"
  [[ -s "${TASK_FILENAME}" ]]
}
function CREATE_FILE7(){
  declare LINE1="" LINE2="" LINE3
  declare FILENAME1="${FILENAME}.1"
  declare TASK_FILENAME="${FILENAME}.7"
  : > "${TASK_FILENAME}" #clear file
  while read -e LINE3;do
    if [[ "$LINE2" =~ "медведь" &&
         ("$LINE1" =~ "местность" ||"$LINE1" =~ "кран") &&
         ("$LINE3" =~ "испытание" ||
         "$LINE3" =~ "девица") ]];then
      echo -e "$LINE1\n$LINE2\n$LINE3" >>"${TASK_FILENAME}"
```

```
    fi
    LINE1="$LINE2"
    LINE2="$LINE3"
  done <"$FILENAME1"
#errorlevel по последнему оператору
  [[ -s "${TASK_FILENAME}" ]]
}

while :;do
  for I in {1..7};do
    echo CREATE_FILE${I}
    CREATE_FILE${I} || break # перегенерация, если файл
пустой.
  done
  break
done
```

Ошибки при выполнении задания 29.

1 Для проверки правильности выполнения заданий просто по-очереди открывайте файлы и сопоставьте строки, условиям задачи.

2. Проверьте правильность вывода строк в заданиях 6,7.

3 Выполните скрипт и сравните число строк в каждом файле. В первом файле должно быть строго 100000. В других - число строк должно быть примерно похожим.

```
#!/bin/bash
while :;do
  for I in {1..7};do
    wc -l "$FILENAME.$I"
  done
  break
done

#100000 /dev/shm/file.1
#17818 /dev/shm/file.2
#15444 /dev/shm/file.3
#6266 /dev/shm/file.4
#1540 /dev/shm/file.5
#6166 /dev/shm/file.6
#5679 /dev/shm/file.7
```

Ответы и решения на Задание 30.

Теория здесь:#30

```
#!/bin/bash
declare TMP="/tmp"
[[ -d "/dev/shm" ]] && TMP="/dev/shm"
declare LOG="${TMP}/ip.log"
declare STAT="${TMP}/ip.stat"

trap 'rm -f "$LOG" ; exit' INT
trap 'rm -f "$LOG" "$STAT" ; exit' TERM HUP

declare -a SLOG=()

function INIT_WORD(){
 declare A='eyuioa'
 declare B='qwrtpsdfghjklzxcvbnm'
 declare -i J I=${#A}
 while ((I--));do
   J=${#B}
   while ((J--));do
     A1=${A:$I:1}
     B1=${B:$J:1}
     SLOG+=("$B1$A1") # Чет
     SLOG+=("$A1$B1") # Нечет
   done
 done
}
function RND_WORD() {
  declare -i NUM_SLOG=${1:-4}
  declare -i AB=${2:-30}
  declare -i ALL_SLOG=$((${#SLOG[@]}/2))
 #ALL_SLOG  число слогов одного типа

  declare -i CUR_SLOG
  declare WORD=""
  ((NUM_SLOG<1)) && NUM_SLOG=1
  ((NUM_SLOG>7)) && NUM_SLOG=7
  ((AB<0)) && AB=0
  ((AB>100)) && AB=100
  while ((NUM_SLOG--));do
    WORD+=${SLOG[$(((SRANDOM % ALL_SLOG)*2 + \
          ((SRANDOM % 101)<AB)))]}
  done
  echo $WORD
}

function CREATE_LOG() {
declare -i REC_NUM=100000
declare -i START=1577836800
declare -i DIA=$((1735689599-1577836800+1))
```

```
declare RAND_IP
declare RAND_DATE
declare CUR_WORD
:> "$LOG"
    for ((K = 1; K <=REC_NUM; ++K)); do
        RAND_DATE=$(($SRANDOM % $DIA + $START))
        RAND_IP="$(($RANDOM % 255 + 1)).$(($RANDOM % 256)).$
(($RANDOM % 256)).$(($RANDOM % 256))"
        CUR_WORD=$(RND_WORD $((SRANDOM % 4 + 1 )) 30)
        echo "$RAND_DATE $RAND_IP $CUR_WORD" >> "$LOG"
        if [[ $SECONDS -ge 5 ]];then
          SECONDS=0
       echo "Создано строк: $K">&2
        fi
    done
}
function LESS_LOG() {
  declare -a FIELD
  declare -x TZ='Europe/Kaliningrad'
  while read -a FIELD; do
  printf '%(%Y%m%d-%H%M%S)T %s %s\n' "${FIELD[@]}"
  done < $LOG | less
}
function IP_STAT() {
  declare -x TZ='UTC'
  declare -a FIELD
  declare DT
  declare -iA Y=() YM=() YMD=()
  declare -i MIN MAX CUR
  declare MIN_POS MAX_POS
  declare -i SUM
  declare -i REC_COUNT=0
  :>"$STAT"
    while read -a FIELD; do
    printf -v DT '%(%Y %Y%m %Y%m%d)T' "${FIELD[0]}"
      set -- $DT
      ((Y[$1]++, YM[$2]++, YMD[$3]++, REC_COUNT++)) ;
    done < $LOG

    echo "3.1 записей по годам ---">"$STAT"
    SUM=0; MAX=-1; MIN=$REC_COUNT
    for I in ${!Y[@]};do
      CUR=${Y[$I]}
      SUM+=$CUR
      ((CUR<MIN)) && (( MIN=CUR, MIN_POS=I))
      ((CUR>MAX)) && (( MAX=CUR, MAX_POS=I))
    done
    echo "Записей в году в среднем $((SUM/${#Y[@]}))"
    echo "MIN записей  $MIN в $MIN_POS году"
    echo "MAX записей  $MAX в $MAX_POS году"

    SUM=0; MAX=-1; MIN=$REC_COUNT
    for I in ${!YM[@]};do
```

```
      CUR=${YM[$I]}
      SUM+=$CUR
      ((CUR<MIN)) && (( MIN=CUR, MIN_POS=I))
      ((CUR>MAX)) && (( MAX=CUR, MAX_POS=I))
    done
    echo -e "\nЗаписей в году-месяце в среднем \
$((SUM/${#YM[@]}))"
    echo "MIN записей  $MIN в $MIN_POS"
    echo "MAX записей  $MAX в $MAX_POS"

    SUM=0; MAX=-1; MIN=$REC_COUNT
    for I in ${!YMD[@]};do
      CUR=${YMD[$I]}
      SUM+=$CUR
      ((CUR<MIN)) && (( MIN=CUR, MIN_POS=I))
      ((CUR>MAX)) && (( MAX=CUR, MAX_POS=I))
    done
    echo -e "\nЗаписей в году-месяце-дне в среднем \
$((SUM/${#YMD[@]}))"
    echo "MIN записей  $MIN в $MIN_POS"
    echo "MAX записей  $MAX в $MAX_POS"
}

INIT_WORD
CREATE_LOG
LESS_LOG
IP_STAT
```

Ошибки при выполнении задания 30.

В этом задании может быть одна серьёзная ошибка - не верное указание временнóй зоны. Проявляется она в том, что среднее число записей в одном году отличается от 20000. Дело в том, что в данном примере даты специально подобраны:

1577836800 = Ср 01 янв 2020 00:00:00 UTC
1735689599 = Вт 31 дек 2024 23:59:59 UTC

Таким образом, изменение часового пояса приводит к появлению дополнительного ГОДА!!! и изменению средних значений.

Не правильно. TZ задана без флага экспорта:

```
declare TZ='Europe/Kaliningrad'
printf printf '%(%Y%m%d-%H%M%S)T' VAR
```

Правильно:

```
declare -x TZ='Europe/Kaliningrad'
printf '%(%Y%m%d-%H%M%S)T' VAR
#или
TZ='Europe/Kaliningrad' printf '%(%Y%m%d-%H%M%S)T' VAR
```