*ПЗ-03:* ***Bash.*** ***Разработка сценариев оболочки***

**BASH** — **Bourne**-**Again** **SHell** (что может переводится как «*перерожденный шел*», или «*Снова шел Борн* (создатель **sh**)»), самый популярный командный интерпретатор в UNIX-подобных системах, в особенности в GNU/Linux.

***ПРЕДВАРИТЕЛЬНО*** *(в самом начале занятия!)==========================================*

* *в каталоге* **/tmp/gXX** *(где* **ХХ** *– две цифры номера группы: 05, 06, 07) создать каталог* **pz03** (ПЗ-03)
* *в каталоге* **/tmp/g0Х/pz03***создать файл отчёта* **stYY.txt** *(где* **YY** *– две цифры номера по порядку студента в уч.группе)*
* *все действия пользователя «протоколировать» в файле. В начале вывести на экран, а затем продублировать выводом в файл* **stYY.txt***с помощью команды:* **command\_linux >> stYY.txt**
* *в конце занятия защитить отчёт преподавателю.*

**МЕТОДИКА ВЫПОЛНЕНИЯ ============================================================**

**Ваша первая программа на Bash**

Вашей первой программой будет классическая «*Hello World*». Это – традиция и кто мы такие, чтобы менять ее? Программа «*Hello World*» просто выводит слова «*Hello World*» на экран.

Итак:

А) В первом окне (переключиться: **CTRL+ALT+F1**, затем ввести **login** и **psw**):

– войти в свой рабочий каталог: **/tmp/g0Х/pz03**

– создать пустой файл **hello.sh** и для этого просто наберите:

**touch hello.sh**

– проверьте результат командой **ls** или **ls -l**

– запустите текстовый редактор в **mc** (или **vi/vim**):

**mc**

выбрать файл **hello.sh** и нажать **F4** (редактировать); далее – меню в нижней строке

– и наберите в нем следующее:

**#!/bin/bash**

**echo "Hello World"**

**exit 0**

Б) Во втором окне (переключиться: **CTRL+ALT+F2**, затем ввести **login** и **psw**):

– войти в свой рабочий каталог: **/tmp/g0Х/pz03**

– нужно сделать скрипт исполняемым:

**chmod u+x hello.sh**

Если вы не понимаете, что делает эта команда, прочтите справочную страницу команды: **man chmod**

Как только это будет сделано, вы сможете запустить программу, просто набрав ее название: **./hello.sh**

На экран выйдет: **Hello World**

ВЫВОД: просто запомните (запишите!) простую последовательность действий:

* напишите код;
* сохраните файл;
* сделайте его исполняемым с помощью **chmod**;
* запустите: **./hello.sh** или **bash hello.sh** (без использования команды **chmod**)

==================================================================================

**1. Создание сценария оболочки.**

На прошлых занятиях мы разобрали как устроена и какие особенности имеет файловая система (ФС) Linux.

На ПЗ-03 рассмотрим **bash-скрипты**. Это — сценарии командной строки, написанные для оболочки **bash**. Существуют и другие оболочки, например — **zsh, tcsh, ksh**, но мы сосредоточимся на **bash**.

**BASH** — **Bourne**-**Again** **SHell** (что может переводится как «перерожденный шел», или «*Снова шел Борн* (создатель **sh**)»), самый популярный командный интерпретатор в юниксоподобных системах, в особенности в **GNU**/**Linux**.

Ниже приведен ряд встроенных команд, которые мы будем использовать для создания своих скриптов:

* **break** выход из цикла **for**, **while** или **until**
* **continue** выполнение следующей итерации цикла **for**, **while** или **until**
* **echo** вывод аргументов, разделенных пробелами, на стандартное устройство вывода
* **exit** выход из оболочки
* **export** отмечает аргументы как переменные для передачи в дочерние процессы в среде
* **hash** запоминает полные имена путей команд, указанных в качестве аргументов, чтобы не искать их при следующем обращении
* **kill** посылает сигнал завершения процессу
* **pwd** выводит текущий рабочий каталог
* **read** читает строку из ввода оболочки и использует ее для присвоения значений указанным переменным
* **return** заставляет функцию оболочки выйти с указанным значением
* **shift** перемещает позиционные параметры налево
* **test** вычисляет условное выражение
* **times** выводит имя пользователя и системное время, использованное оболочкой и ее потомками
* **trap** указывает команды, которые должны выполняться при получении оболочкой сигнала
* **unset** вызывает уничтожение переменных оболочки
* **wait** ждет выхода из дочернего процесса и сообщает выходное состояние.

***Сценарии командной строки*** — это наборы тех же самых команд, которые можно вводить с клавиатуры, собранные в файлы и объединённые некоей общей целью. При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. ***Сценарии*** — это мощный способ автоматизации часто выполняемых действий.

Итак, если говорить о командной строке, она позволяет выполнить несколько команд за один раз, введя их через точку с запятой: **pwd; whoami**

Если вы опробовали это в своём терминале, ваш первый **bash-скрипт**, в котором задействованы две команды, уже написан. Работает он так. Сначала команда **pwd** выводит на экран сведения о текущей рабочей директории, потом команда **whoami** показывает данные о пользователе, под которым вы вошли в систему

***Командная строка*** — отличный инструмент, но команды в неё приходится вводить каждый раз, когда в них возникает необходимость. Что если записать набор команд в файл и просто вызывать этот файл для их выполнения? Собственно говоря, тот файл, о котором мы говорим, и называется *сценарием командной строки*.

Создайте пустой файл с использованием команды ***touch***: **touch myscript.sh**

В его первой строке нужно указать, какую именно оболочку мы собираемся использовать. Нас интересует **bash**, поэтому первая строка файла будет такой:

**#!/bin/bash** # This is a comment

Команды оболочки отделяются знаком перевода строки, комментарии выделяют знаком решётки. Вот как это выглядит:

|  |  |
| --- | --- |
| **#!/bin/bash**  **# This is a comment**  **pwd**  **whoami** | Сохраните файл, дав ему имя **myscript.sh**, и работа по созданию bash-скрипта почти закончена. Сейчас осталось лишь сделать этот файл исполняемым, иначе, попытавшись его запустить, вы столкнётесь с ошибкой ***Permission denied:*** |
| ***Permission denied:***  *Попытка запуска файла сценария с неправильно настроенными разрешениями* | https://habrastorage.org/getpro/habr/post_images/29f/743/3ec/29f7433ec002900e7518396a814fe351.png |
| **chmod +x ./myscript** | Сделаем файл исполняемым |
| **myscript** | Теперь попытаемся его выполнить |

После настройки разрешений всё работает как надо.

**Вывод сообщений**

|  |  |  |
| --- | --- | --- |
| **echo** | Для вывода текста в консоль Linux применяется команда **echo**. | |
| **#!/bin/bash**  **#** *our comment is here*  **echo "The current directory is:"**  **pwd**  **echo "The user logged in is:"**  **whoami** | | https://habrastorage.org/getpro/habr/post_images/4d0/173/e3d/4d0173e3ddcad01d0790b6b98dd91315.png |
| **#!/bin/bash**  **echo "Hello World!"**  **printf "%s\n" "Hello World"** | | **Hello World!**  **Hello World!** |

А что если надо вывести на экран значок доллара? Попробуем так:

**echo "I have $1 in my pocket"**

Система обнаружит знак доллара в строке, ограниченной кавычками, и решит, что мы сослались на переменную. Скрипт попытается вывести на экран значение неопределённой переменной **$1**. Это не то, что нам нужно. Что делать?

В подобной ситуации поможет использование управляющего символа, обратной косой черты, перед знаком доллара (теперь сценарий выведет именно то, что ожидается):

|  |  |
| --- | --- |
| Использование управляющей последовательности для вывода знака доллара:  **echo "I have \$1 in my pocket"** | https://habrastorage.org/getpro/habr/post_images/771/a8a/710/771a8a7102c7a25f1ecd1e739a00de6e.png |

**2. Использование переменных в файлах сценариях.**

Переменные позволяют хранить в файле сценария информацию, например — результаты работы команд для использования их другими командами.

Нет ничего плохого в исполнении отдельных команд без хранения результатов их работы, но возможности такого подхода весьма ограничены.

Существуют два типа переменных, которые можно использовать в bash-скриптах:

* Переменные среды (Переменные окружения)
* Пользовательские переменные

**2.1 Переменные среды (Переменные окружения)**

Иногда в командах оболочки нужно работать с некими системными данными. Вот, например, как вывести домашнюю директорию текущего пользователя (*При обращении (но не при установке!) к переменной окружения, к ее имени необходимо добавить символ* “$”):

|  |  |
| --- | --- |
| Использование переменной среды в сценарии:  **#!/bin/bash**  **# display user home**  **echo "Home for the current user is: $HOME"** | https://habrastorage.org/getpro/habr/post_images/da0/6b6/4a4/da06b64a489f6a8e16a3ca345f270f82.png |

Обратите внимание на то, что мы можем использовать системную переменную **$HOME** в двойных кавычках, это не помешает системе её распознать. Вот что получится, если выполнить вышеприведённый сценарий.

* **Переменная** **HOME** *указывает на корневой (домашний) каталог пользователя*.

Предположим для примера, что вы пишите сценарий, который должен создать в домашнем каталоге любого пользователя, выполняющего этот сценарий, новый каталог с именем newdir, чтобы установить в нем какой-то пакет программ. Начальные строки файла-сценария могут выглядеть так:

**cd $HOME**

**mkdir newdir**

Тогда любой пользователь, запустивший такой сценарий из любого каталога, получит желаемый результат.

* **Переменная PATH** *содержит в себе список каталогов, в которых система ищет команды или программы для исполнения.*

Другими словами, если пользователь хочет выполнить какую-либо команду (или запустить программу), то он либо должен указать полный путь к команде, либо этот путь должен содержаться в списке переменной **PATH**.

Рассмотрим следующий пример: **echo $PATH**

На экран будет выведено что-то в таком роде: ***.:/home/myfiles/bin:/usr/bin:/bin:/usr/X11R6/bin***

Как можно видеть, каталоги в списке переменной **PATH** разделены знаком “**:**”. В частности, каталог /bin содержит многие из рассмотренных ранее команд системы: cp, gzip, mkdir и другие.

Поэтому мы и можем вызывать такие команды напрямую, без указания пути к ним. Поиск команд или программ ведется по списку каталогов в переменной **PATH** слева направо – об этом необходимо помнить, поскольку в системе и у пользователя могут оказаться команды или программы с одинаковыми именами.

В этом случае будет выполнена команда из того каталога, который расположен в списке раньше.

* Отметим, что текущий каталог, представленный знаком “**.**” в первой позиции списка в данном примере, по умолчанию может и не включаться системными сценариями автозагрузки в состав переменной **PATH**.
* Если это так, то вы не сможете запустить, скажем программу myprog, расположенную в текущем каталоге, просто набрав в командной строке myprog, а должны вызвать ее так: **./myprog**
* (здесь между **точкой** и знаком “**/**” нет пробела).

Другая возможность – ***включить текущий каталог в список переменной*** **PATH**.

Чтобы добавить какой-либо каталог, скажем **/opt/bin**, в список переменной **PATH**, необходимо выполнить команду: **export PATH=$PATH:/opt/bin**

|  |  |
| --- | --- |
| * **Переменная PWD** | *– указывает на текущий каталог* |
| * **Переменная USER** | *– указывает на имя пользователя* |
| * **Переменная EDITOR** | *– указывает на редактор, который будет использоваться по умолчанию некоторыми программами и утилитами* |
| * **Переменная PRINTER** | *– указывает на имя принтера, используемого по умолчанию* |
| * **Переменная TERM** | *– указывает на тип используемого терминала*  Обычно эта переменная имеет значение **xterm** при работе в графическом режиме или **linux** или **vt100** в текстовом режиме работы |
| * **Переменная DISPLAY** | *– указывает на адрес графического дисплея* |

**Переменная DISPLAY** важна, когда необходимо открыть новое **X** окно. При работе на локальном компьютере никаких проблем с использованием переменных **TERM** и **DISPLAY** обычно не возникает. Однако они могут появиться при работе на удаленном компьютере. О том, как правильно установить переменную **DISPLAY**, рассказано в разделе о настройках **Х-окружения**.

Таким образом,

* ***Environment variables***, или ***переменные окружения*** – *это системные переменные, которые используются операционной системой, программами или командными файлами-сценариями*.

Вы можете увидеть, какие переменные окружения установлены у вас и каковы их значения в данный момент с помощью команды **env**.

Значения большинства переменных окружения обычно устанавливают в системных сценариях автозагрузки или сценариях автозагрузки пользователя (см. соответствующий раздел). Более подробно о назначении различных переменных окружения (как и о многом другом) можно прочесть в справочных страницах, посвященных соответствующим командным оболочкам: **man bash**, **man** **tcsh** или **man** **zsh**.

**2.2 Пользовательские переменные**

В дополнение к переменным среды, bash-скрипты позволяют задавать и использовать в сценарии собственные переменные. Подобные переменные хранят значение до тех пор, пока не завершится выполнение сценария.

Как и в случае с системными переменными, к пользовательским переменным можно обращаться, используя знак доллара:

|  |  |
| --- | --- |
| **#!/bin/bash**  **#** *testing variables*  **grade=5**  **person="Adam"**  **echo "$person is a good boy, he is in grade $grade"** |  |

**Подстановка команд**

Одна из самых полезных возможностей bash-скриптов — это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария.

Сделать это можно двумя способами:

1. С помощью значка обратного апострофа «**`**»
2. С помощью конструкции **$()**

Используя **первый подход**, проследите за тем, чтобы вместо обратного апострофа не ввести одиночную кавычку. Команду нужно заключить в два таких значка:

|  |  |
| --- | --- |
| **mydir=`pwd`** | – Используя первый подход, проследите за тем, чтобы вместо обратного апострофа не ввести одиночную кавычку. Команду нужно заключить в два таких значка |
| **mydir=$(pwd)** | – При втором подходе то же самое записывают таким образом. |

А скрипт, в итоге, может выглядеть так

(*скрипт, сохраняющий результаты работы команды в переменной*):

|  |  |
| --- | --- |
| **#!/bin/bash**  **mydir=$(pwd)**  **echo $mydir** |  |

В ходе его работы вывод команды **pwd** будет сохранён в переменной **mydir**, содержимое которой, с помощью команды **echo**, попадёт в консоль.

**3. Выполнение математических вычислений.**

Для выполнения математических операций в файле скрипта можно использовать конструкцию вида *$((a+b)):*

|  |  |
| --- | --- |
| **#!/bin/bash**  **var1=$(( 5 + 5 ))**  **echo $var1**  **var2=$(( $var1 \* 2 ))**  **echo $var2** |  |

**Сравнение чисел**

В скриптах можно сравнивать числовые значения. Ниже приведён список соответствующих команд.

|  |  |  |
| --- | --- | --- |
| **n1 -eq n2** | Возвращает истинное значение, если n1 равно n2. | |
| **n1 -ge n2** | Возвращает истинное значение, если n1больше или равно n2. | |
| **n1 -gt n2** | Возвращает истинное значение, если n1 больше n2. | |
| **n1 -le n2** | Возвращает истинное значение, если n1меньше или равно n2. | |
| **n1 -lt n2** | Возвращает истинное значение, если n1 меньше n2. | |
| **n1 -ne n2** | Возвращает истинное значение, если n1не равно n2. | |
| **#!/bin/bash**  **val1=6**  **if [ $val1 -gt 5 ]**  **then**  **echo "The test value $val1 is greater than 5"**  **else**  **echo "The test value $val1 is not greater than 5"**  **fi** | | |  |

**4. Выход из сценария.**

Вместо использования **exit 1** (выход из оболочки) попробуйте использовать **return 1**

**5. Редактор vi (vim)**

В редакторе **vi** существует несколько режимов работы.

Наиболее часто используются два из них – командный режим и режим ввода текста. В командном режиме нажатие каждой клавиши клавиатуры означает определенное действие, но не ввод текста. В этот режим пользователь попадает при запуске редактора: **vi filename**

где **filename** - имя редактируемого файла.

В командном режиме vi можно перемещаться по тексту с помощью **клавиш** **управления** **стрелками** (**влево**-**вправо**-**вверх**-**вниз**), а также пользоваться функциями поиска: **/pattern** ищет первое появление образца **pattern** в тексте по направлению вперед, а **?pattern** - по направлению назад.

Можно удалять текст: нажатие клавиши <**X**> удаляет символ, предшествующий курсору, а клавиши <**x**> - символ над курсором.

Стереть текст от позиции курсора до конца строки можно, нажав клавиши **<d><$>**, а стереть всю строку: **<d><d>**.

Чтобы перейтив **РЕЖИМ ВВОДА ТЕКСТА**, нажмите клавишу **<i>**.

В этом режиме все, что набирается на клавиатуре, является вводимым текстом. Вернуться обратно в командный режим **vi**, закончив набирать текст можно, нажав клавишу <**Esc**>.

Еще одним режимом редактора vi является так называемый режим последней строки.

Все команды, вводимые в этом режиме, начинаются с символа “**:**”. Ввод двоеточия перемещает курсор в нижнюю часть экрана, где следует ввести оставшуюся часть команды.

Режим последней строки используется главным образом для выполнения команд общего характера, в частности:

* **wq** - записать произведенные изменения в файл и выйти из редактора;
* **w** - записать изменения в файл без выхода из редактора;
* **w** **filename** - записать текущий редактируемый буфер в новый файл с именем **filename**;
* **q** - выйти из редактора;
* **q!** - выйти из редактора, не сохраняя произведенные изменения;
* **!sh** - войти в командную оболочку (**shell**) системы. Здесь вы можете выполнять любые команды оболочки. Чтобы вернуться обратно в редактор, наберите **exit**;
* **!command** - выполнить команду оболочки (командного интерпретатора операционной системы) **command**;

Итак, редактор **vi** имеет три режима:

1. ***Командный*** - в этом режиме можно перемещаться по файлу и выполнять редактирующие команды над текстом. Команды вызываются ОБЫЧНЫМИ ЛАТИНСКИМИ БУКВАМИ.
2. ***Ввода текста*** - в этом режиме обычные латинские буквы будут вставляться в текст.
3. ***Режим строчного редактора*** ED используется для управления файлами (типа сохранить файл, зачитать файл и т.д.)

ПРИМЕРЫ:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **vi имя\_файла** | | | | – vi в КОМАНДНОМ режиме | |
| **ESC : q ! Enter** | | | | – чтобы выйти из файла без сохранения | |
| **ESC : w ! Enter** | | | | – чтобы выйти из файла, сохранив изменения | |
| **ESC : q Enter** | | | | – чтобы выйти из файла, сохранив изменения | |
| **ESC : wq Enter** | | | | – выйти из файла с сохранением, одной командой | |
| для перехода В РЕЖИМ ВВОДА нужно нажать команды типа | | | | | |
| **"i"** | | – вставлять здесь | | | |
| **"A"** | | – вставлять с конца строки | | | |
| **"cw"** | | – заменять текущее слово | | | |
|  | | | | | |
| **ESC** | | | – для ВОЗВРАТА В КОМАНДНЫЙ РЕЖИМ | | |
| **CTRL-[** | | | – для возврата в командный режим | | |
| **":"** | – для перехода В РЕЖИМ УПРАВЛЕНИЯ ФАЙЛАМИ нужно нажать “**:”** (перейти в режим редактора ED) | | | | |
| *Двигаться по файлу можно командами*: | | | | | |
| **h,j,k,l** | | | – влево, вниз, вверх, вправо | | |
| **Ctrl-F** | | | – на страницу вниз | | |
| **Ctrl-B** | | | – на страницу вверх | | |
|  | | | | | |
| **i** | | | – перевод в режим ввода (подгоните курсор к нужному месту и нажмите **i**) | | |
| **ESC** | | | – прекратить ввод, перейти в командный режим | | |
| Подгоните курсор к ненужному месту и нажмите: | | | | | |
| **x** | | | – удалить символ | | |
| **dd** | | | – удалить строчку | | |
| **o** | | | – вставлять с новой строки (под текущей строкой) | | |
| **a** | | | – в режим ввода ЗА курсором | | |
| **5yy** | | | – запомнить 5 строчек | | |
| *Подгоните курсор к нужному месту* | | | | | |
| **p** | | | – вставить запомненные строки под курсором | | |
| **P** | | | – вставить запомненные строки НАД курсором | | |
|  | | | | | |
| **J** | | | – склеить две строки | | |
| **/Шаблон поиска Enter** | | | | | – поиск |
| **n** | | | – повторить поиск | | |

Того, кто желает разобраться с **VI** поподробнее, может предоставить больше удобств и команд по редактированию, отсылаем к прилагаемому справочнику-памятке "***Наиболее употребительные команды VI***", **vibegin.txt** ([***http://lib.ru/unixhelp/vibegin.txt***](http://lib.ru/unixhelp/vibegin.txt)), ну, и, естественно (как всегда), к документации.

***Управляющие конструкции языка shell ====================================================***

**6. Создание сценария оболочки**

Продолжим рассмотрение **bash-скриптов**. Это — сценарии командной строки, написанные для оболочки **bash**. Существуют и другие оболочки, например — **zsh, tcsh, ksh**, но мы сосредоточимся на **bash**.

Итак,

1) Вход в систему под выданной пользовательской учетной записью:

login:

psw:

2) Создание в домашнем каталоге или tmp (/tmp/g0X, где 0Х –№ группы) каталога группы и ПЗ-03. Например: **/tmp/g0X/pz03**

~$ cd tmp

~$ ls

~$ mkdir g0X (где 0Х – № группы, например g05)

~$ ls

***Сценарии командной строки*** — это наборы тех же самых команд, которые можно вводить с клавиатуры, собранные в файлы и объединённые некоей общей целью. При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. ***Сценарии*** — это мощный способ автоматизации часто выполняемых действий.

Итак, если говорить о командной строке, она позволяет выполнить несколько команд за один раз, введя их через точку с запятой: **pwd; whoami**

Если вы опробовали это в своём терминале, ваш первый **bash-скрипт**, в котором задействованы две команды, уже написан. Работает он так. Сначала команда **pwd** выводит на экран сведения о текущей рабочей директории, потом команда **whoami** показывает данные о пользователе, под которым вы вошли в систему

***Командная строка*** — отличный инструмент, но команды в неё приходится вводить каждый раз, когда в них возникает необходимость. Что если записать набор команд в файл и просто вызывать этот файл для их выполнения? Собственно говоря, тот файл, о котором мы говорим, и называется *сценарием командной строки*.

Создайте пустой файл с использованием команды ***touch***: **touch myscript.sh**

В его первой строке нужно указать, какую именно оболочку мы собираемся использовать. Нас интересует **bash**, поэтому первая строка файла будет такой:

**#!/bin/bash** # This is a comment

Команды оболочки отделяются знаком перевода строки, комментарии выделяют знаком решётки. Вот как это выглядит:

|  |  |
| --- | --- |
| **#!/bin/bash**  **# This is a comment**  **pwd**  **whoami** | Сохраните файл, дав ему имя **myscript.sh**, и работа по созданию bash-скрипта почти закончена. Сейчас осталось лишь сделать этот файл исполняемым, иначе, попытавшись его запустить, вы столкнётесь с ошибкой ***Permission denied:*** |
| ***Permission denied:***  *Попытка запуска файла сценария с неправильно настроенными разрешениями* | https://habrastorage.org/getpro/habr/post_images/29f/743/3ec/29f7433ec002900e7518396a814fe351.png |
| **chmod +x ./myscript** | Сделаем файл исполняемым |
| **./myscript** | Теперь попытаемся его выполнить |

После настройки разрешений всё работает как надо.

**Вывод сообщений**

|  |  |  |
| --- | --- | --- |
| **echo** | Для вывода текста в консоль Linux применяется команда **echo**. | |
| **#!/bin/bash**  **#** *our comment is here*  **echo "The current directory is:"**  **pwd**  **echo "The user logged in is:"**  **whoami** | | https://habrastorage.org/getpro/habr/post_images/4d0/173/e3d/4d0173e3ddcad01d0790b6b98dd91315.png |
| **#!/bin/bash**  **echo "Hello World!"**  **printf "%s\n" "Hello World"** | | **Hello World!**  **Hello World!** |

А что если надо вывести на экран значок доллара? Попробуем так:

**echo "I have $1 in my pocket"**

Система обнаружит знак доллара в строке, ограниченной кавычками, и решит, что мы сослались на переменную. Скрипт попытается вывести на экран значение неопределённой переменной **$1**. Это не то, что нам нужно. Что делать?

В подобной ситуации поможет использование управляющего символа, обратной косой черты, перед знаком доллара (теперь сценарий выведет именно то, что ожидается):

|  |  |
| --- | --- |
| Использование управляющей последовательности для вывода знака доллара:  **echo "I have \$1 in my pocket"** | https://habrastorage.org/getpro/habr/post_images/771/a8a/710/771a8a7102c7a25f1ecd1e739a00de6e.png |

**7. Математические операции.**

|  |  |
| --- | --- |
| Для выполнения математических операций в файле скрипта можно использовать конструкцию вида $((a+b)): | #!/bin/bash  var1=$(( 5 + 5 ))  echo $var1  var2=$(( $var1 \* 2 ))  echo $var2 |

**8. Конструкция if**

Как и во многих языках программирования, в **bash** есть условные конструкции. Они имеют формат, описанный ниже. Будьте внимательны: слова «**if**» и «**then**» должны находится на разных строках. Старайтесь выравнивать горизонтально всю конструкцию, включая заключительный «**fi**» и все «**else**». Это делает код намного удобнее для чтения и отладки. В дополнении к простой форме «**if**, **else**» есть еще несколько других форм условных конструкций:

|  |  |  |
| --- | --- | --- |
| **if [ условие ]**  **then**  **действие**  **fi** | В приведенном выше примере ‘действие’ выполняется только если ‘условие’ верно, в противном случае скрипт продолжает выполнение инструкций со строки идущей за «fi». | |
| if [ условие ]  then  действие  elif [ условие\_2 ]  then  действие\_2  elif [ условие\_3 ]  then  # *Это комментарий: …****команды Bash****…*  else  действие\_x  fi | А эта конструкция последовательно проверяет условия и если они верны, то исполняет соответствующее действие. Если ни одно из условий не верно, то выполняется ‘действие\_x’ стоящее после ‘else’ (если оно есть). Потом продолжается исполнение команд идущих за этой конструкцией «if,then,else», если таковые есть. | |
| Пример 1:  #!/bin/bash  if pwd  then  echo "It works"  fi | В данном случае, если выполнение команды pwdзавершится успешно, в консоль будет выведен текст «it works». | |
| Пример 2:  #!/bin/bash  user=likegeeks  if grep $user /etc/passwd  then  echo "The user $user Exists"  fi | Здесь мы воспользовались командой **grep** для поиска пользователя в файле **/etc/passwd**. В этом примере, если пользователь найден, скрипт выведет соответствующее сообщение. А если найти пользователя не удалось? В данном случае скрипт просто завершит выполнение, ничего нам не сообщив.  *Хотелось бы, чтобы он сказал нам и об этом, поэтому усовершенствуем код*. | |
| **if команда**  **then**  **команды**  **else**  **команды**  **fi** | Для того, чтобы программа смогла сообщить и о результатах успешного поиска, и о неудаче, воспользуемся конструкцией if-then-else. | |
| Пример 3:  #!/bin/bash  user=anotherUser  if grep $user /etc/passwd  then  echo "The user $user Exists"  else  echo "The user $user doesn’t exist"  fi | **if команда1**  **then**  **команды**  **elif команда2**  **then**  **команды**  **fi** | Зададимся вопросом о более сложных условиях. Что если надо проверить не одно условие, а несколько? Например, если нужный пользователь найден, надо вывести одно сообщение, если выполняется ещё какое-то условие — ещё одно сообщение, и так далее. В подобной ситуации нам помогут вложенные условия. |

**Сравнение чисел**

В скриптах можно сравнивать числовые значения. Ниже приведён список соответствующих команд.

**n1 -eq n2** Возвращает истинное значение, если n1 равно n2.

**n1 -ge n2**  Возвращает истинное значение, если n1больше или равно n2.

**n1 -gt n2** Возвращает истинное значение, если n1 больше n2.

**n1 -le n2** Возвращает истинное значение, если n1меньше или равно n2.

**n1 -lt n2** Возвращает истинное значение, если n1 меньше n2.

**n1 -ne n2** Возвращает истинное значение, если n1не равно n2.

|  |  |
| --- | --- |
| #!/bin/bash  val1=6  if [ $val1 -gt 5 ]  then  echo "The test value $val1 is greater than 5"  else  echo "The test value $val1 is not greater than 5"  fi | Значение переменной val1 больше чем 5, в итоге выполняется ветвь then-оператора сравнения и в консоль выводится соответствующее сообщение. |

**Сравнение строк**

В сценариях можно сравнивать и строковые значения. Операторы сравнения выглядят довольно просто, однако у операций сравнения строк есть определённые особенности, которых мы коснёмся ниже. Вот список операторов.

**str1 = str2**  Проверяет строки на равенство, возвращает истину, если строки идентичны.

**str1 != str2** Возвращает истину, если строки не идентичны.

**str1 < str2** Возвращает истину, если str1меньше, чем str2.

**str1 > str2**  Возвращает истину, если str1больше, чем str2.

**-n str1** Возвращает истину, если длина str1больше нуля.

**-z str1** Возвращает истину, если длина str1равна нулю.

|  |  |
| --- | --- |
| #!/bin/bash  user ="likegeeks"  if [$user = $USER]  then  echo "The user $user is the current logged in user"  fi  ПРАВИЛЬНО: | Вот одна особенность сравнения строк, о которой стоит упомянуть. А именно, операторы «>» и «<» необходимо экранировать с помощью обратной косой черты, иначе скрипт будет работать неправильно, хотя сообщений об ошибках и не появится.  Скрипт интерпретирует знак «>» как команду перенаправления вывода. |

#!/bin/bash

val1=text

val2="another text"

if [ $val1 \> $val2 ]

then

echo "$val1 is greater than $val2"

else

echo "$val1 is less than $val2"

fi

Обратите внимание на то, что скрипт, хотя и выполняется, выдаёт предупреждение:

|  |  |
| --- | --- |
| ./myscript: line 5: [: too many arguments  Для того, чтобы избавиться от этого предупреждения, заключим $val2 в двойные кавычки: | #!/bin/bash  val1=text  val2="another text"  if [ $val1 \> "$val2" ]  then  echo "$val1 is greater than $val2"  else  echo "$val1 is less than $val2"  fi |

**Проверки файлов**

Пожалуй, нижеприведённые команды используются в bash-скриптах чаще всего. Они позволяют проверять различные условия, касающиеся файлов. Вот список этих команд.

**-d file**  Проверяет, существует ли файл, и является ли он директорией.

**-e file**  Проверяет, существует ли файл.

**-f file**  Проверяет, существует ли файл, и является ли он файлом.

**-r file**  Проверяет, существует ли файл, и доступен ли он для чтения.

**-s file**  Проверяет, существует ли файл, и не является ли он пустым.

**-w file**  Проверяет, существует ли файл, и доступен ли он для записи.

**-x file**  Проверяет, существует ли файл, и является ли он исполняемым.

**file1 -nt file2**  Проверяет, новее ли file1, чем file2.

**file1 -ot file2** Проверяет, старше ли file1, чем file2.

**-O file**  Проверяет, существует ли файл, и является ли его владельцем текущий пользователь.

**-G file**  Проверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Этот скрипт, для существующей директории (например **/tmp/g3771/pz12)** выведет её содержимое:

#!/bin/bash

mydir=/home/likegeeks

if [ -d $mydir ]

then

echo "The $mydir directory exists"

cd $ mydir

ls

else

echo "The $mydir directory does not exist"

fi

**9. Команда test и ее аналоги. Проверка файлов и сравнение строк**

|  |  |
| --- | --- |
| #!/bin/bash  file=/etc/fstab # Объявляем переменную  if **test** -e $file # Используем команду test  then  echo "Файл fstab найден."  else  echo "Что-то нету такого файла"  fi | Как мы видим, мы использовали знакомые нам операторы if / then, которые я уже описывал ранее.  После оператора if стоит команда **test** которая и производит проверку нашего файла, который мы объявили в переменной. Также, команду test мы использовали с опцией "-e", данная опция возвращает значение "true", если файл найден.  В случае "true" запускается оператор then и выводит строку echo, если же файл не найдет, что означает "false", то запускается оператор else и выводит сообщение "Что-то нету такого файла". |

Теперь давайте рассмотрим аналоги команды test.

|  |  |
| --- | --- |
| #!/bin/bash  file=/etc/fstab # Объявляем переменную  if /usr/bin/[ -e $file ] # Используем команду "[", также  # мы закрыли ее правой ], потому bash требует  # ее наличие  then  echo "Файл fstab найден."  else  echo "Что-то нету такого файла"  fi | Использовали вместо команды test, команду "[" с закрытой правой "]". Вообще, старые версии bash не требовали наличие правой закрывающейся ]. Теперь это необходимо, чтобы избежать ошибок.  Рекомендуем вместо **[ ]** использовать конструкцию  **[[ ]]** |

Тот же скрипт, только вместо test используем другие команды:

|  |  |
| --- | --- |
| #!/bin/bash  file=/etc/fstab # Объявляем перменную  if [[ -e $file ]] # Используем конструкцию [[ ]]  then  echo "Файл fstab найден."  else  echo "Что-то нету такого файла"  fi | Почему [[ ]] лучше, чем [ ]? Потому что внутри [[ ]] вы можете использовать например такие операторы, как: &&, ||, < и >, в то время как внутри [ ] будут сообщения об ошибках.  Кстати, вы можете использовать [ ] и [[ ]] без if / then / else. Те команды как бы заменяют данные операторы:  var1=5  var2=10  [[ "$var1" -ne "$var2" ]] && echo "$var1 не равно $var2" |

**10. Условия. case. Множественный выбор**

|  |  |
| --- | --- |
| Если необходимо сравнивать, какую-то одну переменную с большим количеством параметров, то целесообразней использовать оператор **case:** (создайте файл **menu2.sh**) | |
| #!/bin/bash  echo "Выберите редатор для запуска:"  echo "1 Запуск программы nano"  echo "2 Запуск программы vi"  echo "3 Запуск программы emacs"  echo "4 Выход"  read doing #здесь мы читаем в переменную $doing со стандартного ввода  case $doing in  1)  /usr/bin/nano # если $doing содержит 1, то запустить nano  ;;  2)  /usr/bin/vi # если $doing содержит 2, то запустить vi  ;;  3)  /usr/bin/emacs # если $doing содержит 3, то запустить emacs  ;;  4)  exit 0  ;;  \*) #если введено с клавиатуры то, что в case не описывается, выполнять следующее:  echo "Введено неправильное действие"  esac #окончание оператора case. | Результат работы:  user:~$ ./menu2.sh  Выберите редактор для запуска:  1 Запуск программы nano  2 Запуск программы vi  3 Запуск программы emacs  4 Выход  После выбор цифры и нажатия Enter запуститься тот редактор, который вы выбрали (если конечно все пути указаны правильно, и у вас установлены эти редакторы. |

Законспектируйте в тетрадь список логических операторов, которые используются для конструкции

**if-then-else-fi** и ей подобных:

**-z** # строка пуста

**-n** # строка не пуста

**=, (==)**  # строки равны

**!=** # строки неравны

**-eq** # равно

**-ne** # неравно

**-lt,(< )**  # меньше

**-le,(<=)**  # меньше или равно

**-gt,(>)**  #больше

**-ge,(>=)**  #больше или равно

**!** #отрицание логического выражения

**-a,(&&)**  #логическое «И»

**-o,(||)** # логическое «ИЛИ»

**Структурированные команды языка shell ===================================================**

**11. Программа (скрипт) на Bash**

Напишите программу (скрипт), которая переместит указанный файл в указанный каталог, удалит его вместе с содержимым, а затем создаст это каталог заново.

Предварительно требуется создать файл **file1и** каталог **trash**

|  |  |
| --- | --- |
| Это может быть сделано с помощью следующих команд:  **touch file1**  **mkdir trash**  **mv file1 trash**  **rm -rf trash**  **mkdir trash**  Вместо того, чтобы вводить это все в интерактивном режиме, напишите скрипт, выполняющий эти команды (см.правее). | Программа (скрипт):  **#!/bin/bash**  **touch file1**  **mkdir trash**  **mv file1 trash**  **rm -rf trash**  **mkdir trash**  **echo “Файл удален!”**  **exit 0** |

Сохраните скрипт под именем **clean.sh** и теперь все, что нужно сделать – это запустить его. Он переместит все файлы в каталог, удалит его, создаст его заново каталог и выведет на экран сообщение об удалении файлов.

ВЫВОД: если вы обнаружите, что регулярно делаете нечто требующее набора одной и той же последовательности команд – это вполне можно автоматизировать написанием скрипта.

**12. Программа (скрипт) на Bash** (самостоятельно)

Напишите сценарий **pr3.sh**, который выводит ***дату, время, список зарегистрировавшихся пользователей*** и сохраняет эту информацию в отдельном файле.

**13. Комментарии в Bash**

Набрать в скрипте **pr4.sh** и выполнить:

**#!/bin/bash**

**echo "Символ # не означает начало комментария"**

**echo 'Символ # не означает начало комментария'**

**echo Символ \# не означает начало комментария**

**echo А здесь символ # означает начало комментария**

**echo ${PATH#\*:} # Подстановка -- не комментарий**

**echo $(( 2#101011 )) # База системы счисления -- не комментарий**

**# Курсант Ф.И.О.**

**exit 0**

Выполнить анализ скрипта.

ВЫВОД: Кавычки **" '** и **\** экранируют действие символа **#**.

**14. Присваивание значений переменным и подстановка значений переменных**

А) Набрать в скрипте **pr5a.sh** и выполнить:

#!/bin/bash

# Присваивание значений переменным и подстановка значений переменных:

a=375

hello=$a

#-------------------------------------------------------------------------

# Использование пробельных символов

# с обеих сторон символа "=" присваивания недопустимо.

# Если записать "VARIABLE =value",

#+ то интерпретатор попытается выполнить команду "VARIABLE" с параметром "=value".

# Если записать "VARIABLE= value",

#+ то интерпретатор попытается установить переменную окружения "VARIABLE" в ""

#+ и выполнить команду "value".

#-------------------------------------------------------------------------

echo hello # Это не ссылка на переменную, выведет строку "hello".

echo $hello

echo ${hello} # Идентично предыдущей строке.

echo "$hello"

echo "${hello}"

echo

hello="A B C D"

echo $hello # A B C D

echo "$hello" # A B C D

echo

echo '$hello' # $hello

exit 0

ВЫВОДЫ:

* Обратите внимание на различия, существующие между этими типами кавычек.
* Вы сможете наблюдать различия в выводе echo $hello и echo "$hello".
* Заключение ссылки на переменную в кавычки сохраняет пробельные символы.
* Внутри одинарных кавычек не производится подстановка значений переменных, т.е. "$" интерпретируется как простой символ.
* Если в значениях переменных встречаются пробелы, то использование кавычек обязательно.
* Допускается присваивание нескольких переменных в одной строке, если они отделены пробельными символами:

var1=variable1 var2=variable2 var3=variable3

echo

echo "var1=$var1 var2=$var2 var3=$var3"

Б) **Присваивание значений переменным простое и замаскированное**

Набрать в скрипте **pr5b.sh** и выполнить:

**#!/bin/bash**

**a=23 #** Простейший случай

**echo $a**

**b=$a**

**echo $b**

**#** Теперь немного более сложный вариант (подстановка команд).

**a=`echo Hello!` #** В переменную 'a' попадает результат работы команды 'echo'

**echo $a**

**#** Обратите внимание на восклицательный знак (!) в команде

**#+** этот вариант не будет работать при наборе в командной строке,

**#+** поскольку здесь используется механизм "истории команд" BASH

# Однако, в сценариях, механизм истории команд запрещен.

**a=`ls -l` #** В переменную 'a' записывается результат работы команды 'ls -l'

**echo $a #** Кавычки отсутствуют, удаляются лишние пробелы и пустые строки.

**echo**

**echo "$a"#** Переменная в кавычках, все пробелы и пустые строки сохраняются.

**exit 0**

Присваивание переменных с использованием **$(...)** (более современный метод, по сравнению с обратными кавычками)

**# Взято из /etc/rc.d/rc.local**

**R=$(cat /etc/redhat-release)**

**arch=$(uname -m)**

В) **Двойные круглые скобки ((...))**

Эта конструкция доступна в Bash, начиная с версии 2.04. Она во многом похожа на инструкцию **let**, внутри **((...))** вычисляются арифметические выражения, и возвращается их результат. См. так же описание **((...))** в циклах "**for**" и "**while**".

В простейшем случае, конструкция **a=$(( 5 + 3 ))** присвоит переменной "**a**" значение выражения "**5 + 3**", или **8**. Но, кроме того, *двойные круглые скобки позволяют работать с переменными в стиле языка C*:

**#!/bin/bash**

# Работа с переменными в стиле языка C.

**echo**

**(( a = 23 ))** # Присвоение переменной в стиле C,

# с обоих сторон от " = " стоят пробелы.

**echo "a (начальное значение) = $a"**

**(( a++ ))** # Пост-инкремент 'a', в стиле C. Аналогично **let "a=а+1"**

**echo "a (после a++) = $a"**

**(( a-- ))** # Пост-декремент 'a', в стиле C. Аналогично **let "a=а-1"**

**echo "a (после a--) = $a"**

**(( ++a ))** # Пред-инкремент 'a', в стиле C. Аналогично **let "a+=1"**

**echo "a (после ++a) = $a"**

**(( --a ))** # Пред-декремент 'a', в стиле C. Аналогично **let "a-=1"**

**echo "a (после --a) = $a"**

**echo**

**(( t = a<45?7:11 ))** # Трехместный оператор в стиле языка C.

**echo "If a < 45, then t = 7, else t = 11."**

**echo "t = $t "** # Да!

**echo**

**exit 0**

ВЫВОДЫ:

* Двойные скобки допускают наличие лишних пробелов в выражениях и писать скрипты в стиле программ на языке C.
* В двойных скобках символ **"$" – "*значение переменной*" –** перед переменными опускается.

**15.** **C-подобный синтаксис оператора цикла for**

Два варианта оформления цикла. Напишите сценарий **pr6а.sh**:

**#!/bin/bash**

**echo**

**#** Стандартный синтаксис (1-й вариант): ==============+

**for a in 1 2 3 4 5 6 7 8 9 10**

**do**

**echo -n "$a"**

**done**

**echo; echo**

**# +==========================================+**

**#** А теперь C-подобный синтаксис (2-й вариант):

**LIMIT=10**

**for ((a=1; a <= LIMIT ; a++)) #** Двойные круглые скобки и "LIMIT" без "$".

**do**

**echo -n "$a "**

**done**

**echo; echo**

**# +==========================================+**

Попробуем и Cи-шный оператор "запятая". Напишите сценарий **pr6b.sh**:

**for ((a=1, b=1; a <= LIMIT; a++, b++))**

**#** Запятая разделяет две операции, которые выполняются совместно:

**do**

**echo -n "$a-$b "**

**done**

**echo; echo**

**exit 0**

**16.** **Массивы Bash**

В BASH имеется два вида массивов:

1. *Индексированные массивы (indexed arrays)/*
2. *Ассоциативные массивы (Associative Arrays или hash-массивы).*

*Ассоциативные массивы – появились в BASH начиная с версии 4* и, хотя у них больше возможностей по сравнению с Индексированными массивами, последние до сих пор остаются наиболее используемыми. Ниже рассмотрим примеры с индексированными массивами.

Массивы оболочки Bash работают почти так же, как и в других языках программирования. Перед тем как вы сможете использовать массив, его нужно объявить. Это можно сделать несколькими способами, первый из них - это использование команды оболочки **declare**:

**declare -a имя\_массива**

Но необязательно делать именно так, вы можете сразу начать задавать элементы массива по нужным номерам: **имя\_массива[XX] = значение**

Здесь **XX** обозначает индекс массива.

Еще один удобный способ создавать массивы строк Bash – это просто перечислить все элементы в круглых скобках:

**имя\_массива=( элемент\_1, элемент\_2, элемент\_3, ... )**

Или вы можете сразу задать индекс массива для каждого из элементов:

**имя\_массива=( [XX]=значение [XX]=значение . . . )**

Также можно прочитать значения для массива из клавиатуры или другого источника ввода: **read -a имя\_массива**

При вводе элементов массива они должны быть разделены символом «пробел». Для получения значений элементов массива используйте синтаксис фигурных скобок, обратите внимание, что нумерация элементов массива, как и в большинстве языков начинается с нуля:

**${имя\_массива[XX]}**

**А) Примеры использования массивов Bash**

Создаем массив: **$array=(первый второй третий четвертый пятый)**

Вывод одного из элементов массива по его индексу:

**$echo ${array[1]}**

**$echo ${array[2]}**

**$echo ${array[0]}**

Чаще всего используются массивы строк Bash, но иногда могут встречаться и цифры. Помните про нумерацию? Индексы элементов массива начинаются с нуля.

Для вывода значения элемента по индексу можно использовать и немного другой синтаксис: **echo ${array:0}**

Так можно вывести все элементы массива: **echo ${array[@]}**

А так можно вывести все элементы, начиная с номера 1: **echo ${array[@]:1}**

Вывод всех элементов, которые находятся в диапазоне от 1 до 4: **echo ${array[@]:1:4}**

Чтобы узнать длину первого (нулевого) элемента выполните: **echo ${#array[0]}**

А посмотреть количество элементов массива в Bash можно таким образом:

**echo ${#array[@]}**

|  |  |
| --- | --- |
| Для перебора элементов массива очень удобно использовать циклы.  Пример небольшого скрипта:  **#!/bin/bash**  **array=(AA BB CC DD FF)**  **for i in ${array[@]}**  **do**  **echo ${array[$i]}**  **done**  Или так:  **array[1]=ААА**  **array[2]=ВВВ**  **for ((i=1; i<=${#array[\*]}; i++))**  **do**  **echo ${array[$i]}**  **done** | Внутри цикла вы можете делать со значением элемента все, что вам нужно. Например, вы можете прочитать значения для массива с помощью функции **read**:  **#!/bin/bash**  **echo "Введите элементы массива:"**  **read -a array**  **echo "Результат:"**  **for i in ${array[@]}**  **do**  **echo ${array[$i]}**  **done** |

Чтобы удалить массив, созданный с помощью **declare** используйте функцию **unset**:

**unset array** # удалить массив

**unset array[@]** # удалить все элементы массива

Удаление элемента массива: **unset array[i]** # удалить i-элемент массива

**Б) Математические операции с элементами массива**

Допустимо использование команды **expr** для выполнения операций над элементами.

Например, инициализировать элемент, который является суммой двух других элементов:

**intArray=(1 2 3 4 5)**

**echo ${intArray[@]}**

***1 2 3 4 5*** # Вывод на экран

Добавляем новый элемент:

**intArray[5]=`expr ${intArray[0]} + ${intArray[1]}`**

**echo ${intArray[5]}**

Или умножение:

**intArray[6]=`expr ${intArray[1]} \* ${intArray[2]}`**

**echo ${intArray[6]}**

Можно использовать **$( )** вместо обратных кавычек **` –** это иногда удобнее:

# обе команды выводят: A-B-C-D

**echo "A-`echo B-\`echo C-\\\`echo D\\\`\``"**

**echo "A-$(echo B-$(echo C-$(echo D)))"**

Двойные квадратные скобки **[[ ... ]]** bash интерпретирует как один элемент с кодом возврата. Внутри разрешается использование операторов **&&** и **||**

Пример проверки что переменная не пустая:

*При работе с массивами в качестве разделителя используется переменная* **IFS***. При работе с массивами вы можете задать собственный разделитель – просто установив значение переменной* **IFS***.* Например, чтобы установить в качестве разделителя перенос каретки (окончание строки) задайте значение этой переменной:**IFS=$'\n'**

**if [[ $my\_var != ' ' ]]; then**

**echo 'Переменная не пустая'**

**fi**

Арифметика: **$((** *арифметика* **)) #** будет в стиле языка С.

**В)**Проверить наличие элемента в массиве:

**search\_el='some'**

**arr=(some elements in array)**

**if [[ ${arr[(r)${search\_el}]} == $search\_el ]]; then**

**echo "Элемент ${search\_el} найден"**

**fi**

# if [[ ${arr[(r)some]} == some ]]; then ; echo yes; else; echo no; fi

**Nota Bene**: ***IFS*** *- разделитель элементов массива.*