Лабораторная работа №6.
Работа со сценариями Bash

Введение

В этой лабораторной вы узнаете разницу между PowerShell, CMD и инструментами автоматизации для Linux.

Сравнение Bash и CMD/PowerShell

Для понимания принципиальной разницы между Bash, CMD и PowerShell следует немного погрузиться в исторический контекст.

Оболочка Bash появилась в 1987 году в рамках проекта GNU на замену Bourne shell (отсюда и акроним Bourne again shell — перерожденная оболочка). В ней были реализованы все те лучшие идеи, которые были придуманы в мире *nix, начиная с 70-х годов, и она продолжает улучшаться до сих пор.

Командная оболочка CMD берет свое начало с первых релизов MS-DOS в начале 80-х годов, но по функциональности так и осталась жалким подобием того, что было в Unix. Но с конца 90-х годов в Microsoft начали понимать, что их графический интерфейс не подходит для системных администраторов, которым требуется автоматизировать управление ИТ-инфраструктурой, поэтому они снова и снова пытались предложить пользователям что-то новое.

Одной из таких попыток стал Сервер сценариев Windows (Windows Script Host, WHS), который позволил управлять системой из скриптов на языках JScript и VBScript через объектную модель компонентов (Component Object Model, COM). Однако WHS не решил всех проблем, поэтому позднее был дан зеленый свет проекту Monad под руководством Джеффри Сновера, который в дальнейшем стал известен как PowerShell.

Примечание

Термин «монада» был взят из книги «Монадология» немецкого философа Лейбница и означает фундаментальную единицу для выполнения задачи. Но согласитесь, сколь бы фундаментальной эта монада ни была, а слово командлет звучит намного приятнее. Командлет — это одна команда, которая участвует в семантике конвейера PowerShell.

Разработчики PowerShell хотели создать инструмент управления операционной системой Windows, который бы по функциональности не уступал тому, что есть в Unix, но для этого нужно было преодолеть фундаментальное противоречие между концепциями

«все есть файл» и «все есть объект». Нужны были конвейеры для объединения команд в цепочки, но требовалось обеспечить передачу объектов между командлетами, чтобы не приходилось каждый раз парсить текст для получения доступа к структурированным данным. Данная задача и определила объектно-ориентированный характер языка PowerShell.

Из всего вышесказанного можно сделать несколько очень простых выводов:

- Обе оболочки Microsoft: и CMD, и PowerShell — были вдохновлены оболочкой Bash и ее предшественницами.
- Оболочка Bash намного функциональнее CMD, но она принципиально не поддерживает возможности PowerShell по работе с объектами. Если вам нужны объекты, то добро пожаловать в Python, у которого среди прочего есть и продвинутый интерфейс командной строки.

Запуск скриптов

Мы уже знаем, что при открытии терминала у пользователя запускается оболочка, которая указана у него оболочкой по умолчанию в базе `/etc/passwd`. Мы уже умеем запускать команды в оболочке и знаем, как выжать максимум из командной строки с помощью конвейеров, но для решения комплексных задач автоматизации этого все равно недостаточно. Чтобы не копировать в окно терминала сотни строк из заранее подготовленных файлов, давайте разберемся как правильно запускать такие файлы скриптов в пакетном режиме.

Скрипты представляют собой текстовые файлы с набором команд, которые отличаются от обычных программ принципиально тем, что для их использования нужен интерпретатор. В роли интерпретатора bash-скриптов выступает исполняемый файл оболочки `/bin/bash`, которому в качестве параметра можно передать имя файла, и тогда новый экземпляр приложения выполнит команды из указанного файла, после чего завершит свою работу.

Примечание

При необходимости bash-скрипты, конечно, можно «скомпилировать» и распространять в виде исполняемых файлов, например, с помощью того же shc (shell script compiler), но в этом случае скрипты все равно останутся скриптами и после распаковки будут просто передаваться интерпретатору через опцию `-c` (*от англ. command*).

В том случае, когда мы передаем файл скрипта интерпретатору в качестве параметра, нам нужны только права на чтение этого файла(см. лабораторную работу №5). Давайте вместо традиционного приложения «Hello World» создадим скрипт `select_string.sh`,

который будет имитировать работу одноименного командлета PowerShell, и проверим его работу:

```
echo ""
echo -n "Укажите путь для поиска: "
read find_path

echo -n "Укажите шаблон для отбора файлов: "
read find_name

echo -n "Укажите строку поиска:"
read find_pattern

echo ""
echo "Результаты поиска:"
find $find_path -type f -name "$find_name" -exec grep -iH "$find_pattern" {} \; 2>/dev/null
```

Попробуйте запустить файл через обращение к исполняемому файлу bash:

```
sa@astra:~$ bash select_string.sh

Укажите путь для поиска: /home/sa
Укажите шаблон для отбора файлов: *.txt
Укажите строку поиска:hello

Результаты поиска:
/home/sa/test.txt:hello
```

Если же мы хотим запускать скрипт напрямую по его имени, например, `./select_string.sh`, то нам потребуется установить права на выполнение, и нужно будет подсказать операционной системе, какой именно интерпретатор следует использовать в этом случае.

Как вы помните, Windows определяет тип файла по расширению, а Linux опирается на его сигнатуру. Для правильной идентификации скриптов используют сигнатуру, состоящую из символов «**#!**» в начале файла, после которых в строке указывают полный путь до требуемого интерпретатора:

```
#! /bin/bash
echo ""
echo -n "Укажите путь для поиска: "
read find_path
...
```

Вы можете убедиться, что после внесения указанных изменений в файл `select_string.sh` утилита `file` без проблем распознает этот файл как Bourne-Again shell script:

```
sa@astra:~$ file select_string.sh
select_string.sh: Bourne-Again shell script, UTF-8 Unicode text executable
```

Учитывая, что символом решетки в большинстве языков обозначают комментарий, с помощью сигнатуры «#! …» можно специфицировать скрипты Python, Perl, PHP, Ruby и других языков — интерпретаторы просто проигнорируют эту строку. Однако на этом

возможности сигнатуры не ограничиваются, и вы можете с ее помощью не только указать путь к файлу, но и задать параметры вызова, например, включить режим подробного вывода:

```
#! /bin/bash --verbose
echo ""
echo -n "Укажите путь для поиска: "
read find_path
...
```

Чтобы упростить коммуникации, символы «#!» получили даже собственное имя «Шебанг» (И тут выскакивает Рики Мартин: she bangs, she bangs, oh baby, when she moves, she moves…). На самом деле этот термин является просто сокращением от полного словосочетания «hash bang», где слово bang означает восклицание, а не взрыв, как в ТБВ.

И осталось только изменить права доступа. Давайте проверим, что будет с флагом выполнения и без него:

```
sa@astra:~$ ./select_string.sh
bash: ./select_string.sh: Отказано в доступе

sa@astra:~$ chmod u+x select_string.sh
sa@astra:~$ ./select_string.sh

Укажите путь для поиска: /etc
...
```

Коды возврата

В качестве результатов скрипты и утилиты могут возвращать совершенно любую текстовую информацию, содержание которой может зависеть в том числе от того, какая локаль установлена у пользователя по умолчанию (см. переменную $LANG). Чтобы в заданиях автоматизации можно было однозначно идентифицировать успешное/неуспешное выполнение команды, приложения используют так называемый код возврата последней операции, который можно получить с помощью служебной переменной «$?». Приведем пример с удалением файла:

```
sa@astra:~$ touch testfile.txt
sa@astra:~$ rm testfile.txt
sa@astra:~$ echo "$?"
0
```

Код возврата будет равен 0, что соответствует успешному выполнению операции.

```
sa@astra:~$ rm testfile.txt
rm: невозможно удалить 'testfile.txt': Нет такого файла или каталога
sa@astra:~$ echo "$?"
1
```

При попытке удаления несуществующего файла код возврата будет равен 1.

Из скриптов вы можете задать этот код с помощью команды `exit <N>`, где <N> – это код возврата, который представляет собой целое число в диапазоне 0 - 255. Код 0 обычно назначают успешному выполнению команды. Приведем пример скрипта `high_five.sh`, который всегда «даст пятюню»:

```
#!/bin/bash
exit 5      # Дай пять!
```

Скрипт выдает код возврата 5, который можно потом использовать в обработке ошибок:

```
sa@astra:~$ ./high_five.sh
sa@astra:~$ echo $?
5
```

Скрипты, которые разрабатываются под Windows, редко используют в заданиях автоматизации, но в CMD и PowerShell, конечно же, есть аналогичные инструменты. В PowerShell вызов команды делается точно также, только в качестве служебной переменной используется `$LastExitCode`, а в CMD команду нужно вызывать с параметром «`exit /b 5`», а переменная называется %errorlevel%.

Локальные переменные

Переменные являются основой любого языка программирования. С их помощью выполняются математические операции, обработка строк и любые другие манипуляции с данными. Физически переменные представляют собой именованные участки памяти, в которых хранится определенная информация.

Присвоение значений переменной

Для объявления переменных на языке Bash им следует присвоить значение с помощью оператора «=». Слева от символа «равно» указывается имя переменной, справа — ее значение, а рядом не должно быть никаких лишних пробелов. Значения, состоящие из нескольких слов, должны обрамляться кавычками. Чтобы подставить значение переменной в строку, в начале имени переменной следует указать символ доллара. Пример присвоения значения переменной на Bash:

```
sa@astra:~$ a="Здравствуйте"
sa@astra:~$ b="уважаемый пользователь"
sa@astra:~$ echo "$a, $b"
Здравствуйте, уважаемый пользователь
```

Используя переменные bash, нужно учитывать следующие особенности:

• Использовать можно только латинские буквы, цифры и символ подчеркивания. При этом имя переменной не должно начинаться с цифр. Цифровые имена зарезервированы для позиционных параметров.

• Не рекомендуется использовать имена, совпадающие с названием команд и утилит. Например, не стоит использовать имя file, т.к. есть одноименная утилита.

• Имена переменных Bash чувствительны к регистру, поэтому переменные **sum_invoice** и **Sum_Invoice** – это две совершенно разные переменные.

В языке CMD для присвоения переменным значений используется команда `set`. Так же, как в Bash, рядом с символом «=» не должно быть лишних пробелов, но вот значения, состоящие из нескольких слов, не обрамляются никакими кавычками. Все, что справа от знака «=», будет интерпретировано как присваиваемое значение. Для подстановки значения переменной ее имя следует обрамлять с двух сторон символом процента:

Пример на CMD:

```
C:\> set a=Здравствуйте
C:\> set b=уважаемый пользователь
C:\> echo %a%, %b%
  Здравствуйте, уважаемый пользователь
```

Еще одним существенным отличием CMD является нечувствительность к регистру, поэтому, в отличие от Bash, к переменной **sum_invoice** мы можем обратиться по имени **Sum_Invoice**, **SUM_invoce** и как угодно еще.

В языке PowerShell имена переменных всегда начинаются с символа доллара вне зависимости от контекста использования, а наличие пробелов до и после оператора присвоения «=» не имеют никакого значения. И, опять же, полная нечувствительность к регистру в именах переменных.

Пример на PowerShell:

```
PS> $a = "Здравствуйте"
PS> $b = "уважаемый пользователь"
PS> echo "$a, $b"
Здравствуйте, уважаемый пользователь
```

Казалось бы, отличия совсем незначительные, но сначала они будут сбивать вас с толку.

Строки и строковые переменные

Язык Bash поддерживает несколько разных инструментов для работы со строками, которые обладают схожей функциональностью, но имеют разный синтаксис, что сначала может довольно сильно путать.

Во-первых, есть операции подстановки параметров, с помощью которых можно получить длину строки и вырезать часть подстроки, для начала операции над строкой определим ее:

```
sa@astra:~$ msg="Привет, уважаемый пользователь"
sa@astra:~$
```

Вычислим длину строки в переменной msg с помощью конструкции `${#имя_переменной}`:

```
sa@astra:~$ echo "${#msg}"
30
```

Выведем часть подстроки, начиная с 8го символа с помощью конструкции `${имя_переменной:стартовая_позиция}`:

```
sa@astra:~$ echo "${msg:8}"
уважаемый пользователь
```

Выведем первые 6 символов с помощью конструкции `${имя_переменной:стартовая_позиция:количество_символов}`:

```
sa@astra:~$ echo "${msg:0:6}"
Привет
```

Кстати, стартовую позицию можно сократить если мы планируем брать строку с первого символа: `echo ${msg:0:6}`.

Во-вторых, есть универсальный обработчик выражений `expr`, который содержит гораздо больше полезных инструментов.

```
sa@astra:~$ echo `expr length "$msg"`
30
```

Обратите внимание на вызов подстановки через одинарные кавычки апостроф `expr length "$msg"` или конструкцию «$(command)», по сути они равны, и лучше всегда использовать в двойных кавычках «$(…)».

Определим, с каких символов начинается обращение уже с правильным синтаксисом «$(expr …)»:

```
user_position="$(expr index "$msg" ', ')"
user_position="$(expr $user_position + 2)"
```

Выведем часть подстроки, начиная с первого символа обращения

```
sa@astra:~$ echo "$(expr substr "$msg" $user_position 22)"
уважаемый пользователь
```

Выведем первые 6 символов

```
sa@astra:~$ echo `expr substr "$msg" 1 6`
Привет
```

Числа в Bash

В языке Bash нет числовых переменных, есть только строки. Однако, это не означает, что вам недоступны математические операции и работать можно только целыми числами. Используйте команду `let`, чтобы присваиваемое значение было интерпретировано как математическое выражение, например скрипт `millenium.sh`:

```
millenium=2000; year_now=$(date +%Y)
years_after_millenium=$year_now-$millenium
echo "Прошло $years_after_milleniumr. после миллениума"

sa@astra:~$ ./millenium.sh
Прошло 2025-2000г. после миллениума
```

Как видим переменные просто подставились в строку без каких либо вычислений, тогда используйте инструкцию let для вычисления:

```
millenium=2000; year_now=$(date +%Y)
let years_after_millenium=$year_now - $millenium
echo "Прошло ${years_after_millenium}г. после миллениума"

sa@astra:~$ ./millenium.sh
Прошло 25г. после миллениума
```

При использовании команды `let` вы можете оперировать не только целыми числами в десятеричной системе счисления. Поставьте префикс `0`, и это будет уже восьмеричное число, а если поставите префикс `0x`, то получите шестнадцатеричное число и только не удивляйтесь, что после «let c=011+022» переменная $c равна 27, а не 33, т.к. результат будет выводиться как обычно в десятичном формате.

Выражения могут быть заданы не только командой `let`, но также с помощью универсального обработчика выражений `expr` и с помощью двойных круглых скобок. Посчитаем выражение с помощью `expr`:

```
millenium=2000; year_now=$(date +%Y)
years_after_millenium="$(expr $year_now - $millenium)"
echo "Прошло ${years_after_millenium}г. после миллениума"

sa@astra:~$ ./millenium.sh
Прошло 25г. после миллениума
```

Посчитаем следующий год *next_year.sh* через двойные скобки `$(( выражение ))`:

```
year_now="$(date +%Y)"
next_year="$((year_now + 1))"
echo "Следующий год $next_year"

sa@astra:~$ ./next_year.sh
Следующий год 2026
```

Двойные круглые скобки используются также в циклах и могут быть задействованы для присвоения значений переменным в стиле языка Си, когда с обеих сторон от символа «=» стоят пробелы и могут использоваться дополнительные математические операции:

```
sa@astra:~$ (( year = 2000 + 25 ))
sa@astra:~$ echo "Год $year"
Год 2025
```

Язык CMD придерживается тех же принципов, только вместо команды `let` используется команда `set` с ключом /a. Давайте посмотрим, как будет выглядеть тот же пример на CMD.

Пример вычислений на CMD:

```
C:\> set a=2000
C:\> set b=25
C:\> set /a c=a+b
  2025
C:\> echo Год %c%
  Год 2025
```

Если же говорить о PowerShell, то в части типизации данных он намного ближе к классическим языкам программирования и поддерживает не только числа, строки, массивы, но и специфические объекты для работы с датами, хеш-таблицами, XML-структурами и пр. Поэтому на PowerShell все максимально привычно для программистов.

Пример вычислений на PowerShell:

```
PS> [int]$a = 2000
PS> [int]$b = 25
PS> [int]$c = $a + $b
PS> [string]$msg = "Год $c"
PS> echo $msg
Год 2025
```

Дробные числа и числа с плавающей точкой

Для обработки чисел с дробной частью в самом языке Bash ничего нет, но можно использовать сторонние утилиты, например, утилиты `awk`, `bc` или `python`:

```
sa@astra:~$ year_now=`date +%Y`
sa@astra:~$ echo "$year_now^2 = `awk "BEGIN{print $year_now ^ 2}"`"
2025^2 = 4100625

sa@astra:~$ echo "$(bc <<< 'scale=2; 100/3')"
33.33

sa@astra:~$ echo 'print(10/3+65.4)' | python3
68.73333333333333
```

Массивы

Если переменные позволяют хранить только одно значение, то массивы предназначены для хранения списков, в чем и состоит их ключевое преимущество. Для обращения к конкретному элементу списка нужно указать имя массива и порядковый номер этого элемента (так называемый индекс).

Массивы очень помогают в разработке скриптов на Bash. Приведем пример, демонстрирующий возможность работы со списком пользователей.

Пример на Bash:

```
# Определим массив из трех элементов
users=("Ivanov" "Petrov" "Sidorov")

# Добавим четвертый элемент в конец
users[3]="Kuznetsov"

# Выведем все элементы массива на экран
echo ${users[*]}
Ivanov Petrov Sidorov Kuznetsov

# Выведем количество элементов в массиве
echo ${#users[*]}
4

# Выведем четвертый элемент массива, учитывая, что нумерация начинается с 0
echo ${users[3]}
Kuznetsov

# Пройдем циклом по всем элементам массива
for user in ${users[*]}; do echo $user; done
Ivanov
Petrov
Sidorov
Kuznetsov
```

Язык CMD разрабатывали с оглядкой на Unix, но массивов в нем так и не появилось. Есть, конечно, несколько обходных приемов, но использовать их крайне неудобно.

В языке PowerShell обычные массивы представляют наборы данных, в которые возможно добавить новые элементы `$myArray += 1, 2, 3, 4`, но также есть много других классов объектов, с помощью которых можно реализовать все, что душе угодно. Наиболее популярным из них является ArrayList.

Пример массивов на PowerShell:

```
PS C:\> $users = [System.Collections.ArrayList]@("Ivanov", "Petrov", "Sidorov")
PS C:\> $users.Add("Kuznetsov")
PS C:\> echo $users
  Ivanov
  Petrov
  Sidorov
  Kuznetsov
```

```
PS C:\> echo $users[3]
  Kuznetsov

PS C:\> foreach ($user in $users) { echo $user }
  Ivanov
  Petrov
  Sidorov
  Kuznetsov
```

Ветвления

Ветвление является конструкцией языка программирования, которая позволяет направить алгоритм на выполнение одного из возможных блоков команд в зависимости от результатов проверки заданного условия.

Для создания ветвления в Bash используется конструкция if/then/elif/else/fi. Интерпретатор выполняет условное выражение и, если код возврата равен 0, то переходит далее к выполнению команд основной ветки, так как 0 означает «успех».

Существует два способа определения условных выражений:

- Старая команда `test`, синонимом которой являются одинарные квадратные скобки: `[ ... ]`. Эта команда воспринимает выражение как операцию сравнения или как файловую проверку и возвращает код завершения, где: **0** - истина, **1** - ложь. Команда не поддерживает регулярные выражения, давно устарела и оставлена исключительно для обеспечения обратной совместимости, например, скрипты загрузчика GRUB.
- Современный вариант команды: `test [[ ... ]]`, который обладает расширенными функциями, в т. ч. поддерживает составные условия сравнения с использованием логических операторов «**&&**» и «||».

В качестве примера проработки условий создадим проверку, является ли пользователь суперпользователем (root), и если нет, то выполняем выход с ошибкой.

Пример инсталлятора на Bash `installer.sh`:

```
#!/bin/bash
echo "Установка необходимых пакетов"
if [[ "$UID" -eq 0 ]]; then
  echo "- Начинаем установку пакетов"
else
  echo "- Для установки приложений запустите скрипт от пользователя root"
  exit 99
fi
# выполняем установку необходимых пакетов для работы
apt install -y git curl wget jq
```

Запуск скрипта от root пользователя:

```
sa@astra:~$ bash installer.sh
Установка необходимых пакетов
- Для установки приложений запустите скрипт от пользователя root
```

```
sa@astra:~$ sudo bash installer.sh
Установка необходимых пакетов
- Начинаем установку пакетов
Чтение списков пакетов… Готово
Построение дерева зависимостей
Чтение информации о состоянии… Готово
...
```

Ветвление может быть записано в одну строку. Обратите внимание на синтаксис выражения – пробелы и символы разделения «;». Обратите внимание на пробелы между квадратными скобками и условным выражением:

```
root@astra:~# if [[ "$UID" -eq 0 ]]; then echo "root"; else echo "user"; fi
root
```

Их отсутствие вызовет синтаксическую ошибку:

```
sa@astra:~$ if [["$UID" -eq 0]]; then echo "root"; else echo "user"; fi
bash: [[1000: команда не найдена
user
```

Пропуск символа «;» в необходимом месте тоже приведет к ошибке:

```
sa@astra:~$ if [[ "$UID" -eq 0 ]] then echo "root"; else echo "user"; fi
bash: синтаксическая ошибка рядом с неожиданным маркером «then»
```

В языке CMD используется конструкция «if условие (…) else (…)» с группировкой команд в блоки с помощью круглых скобок, а в языке PowerShell использует синтаксис, который максимально приближен к языку Си «if (условие) {…} else {…}».

При сравнении целых чисел в скриптах Bash используют следующие операторы:

- `-eq` — равно (от англ. equal — равный)
- `-ne` — не равно (от англ. not equal — неравный)
- `-gt` — больше (от англ. greater than — больше чем)
- `-ge` — больше или равно (от англ. greater or equal — больше или равно)
- `-lt` — меньше (от англ. lower than — меньше чем)
- `-le` — меньше или равно (от англ. lower or equal — меньше или равно)

При сравнении целых чисел внутри двойных круглых скобок (выражений) можно использовать привычные операторы сравнения <, <=, >, >=, например, ((«$a» < «$b»)).

При сравнении строк используют следующие операторы:

- `=` или `==` — строки равны
- `!=` — строки не равны
- `<` — первая строка меньше по порядку сортировки ASCII
- `>` — первая строка больше по порядку сортировки ASCII
- `-z` — строка является «пустой», т.е. имеет нулевую длину

- `-n` — строка не является «пустой», т.е. содержит символы

Циклы

Цикл — является конструкцией языка программирования, которая позволяет выполнять определенный блок команд многократно до тех пор, пока не будет достигнуто условие выхода из цикла.

Циклы for традиционно используются в тех случаях, когда количество итераций известно заранее. На языке Bash это можно реализовать, используя условие в двойных круглых скобках:

```
for((инициализация;условие;счетчик)) do
  # блок команд
done
```

Давайте создадим в простом цикле 10 новых пользователей. Пример цикла на Bash:

```
sa@astra:~$ for((i=1;i<=10;i++)) do echo "Создаем user$i"; sudo useradd "user$i"; done
Создаем user1
Создаем user2
Создаем user3
Создаем user4
Создаем user5
Создаем user6
Создаем user7
Создаем user8
Создаем user9
Создаем user10
```

Можно использовать цикл for и для обхода по всем элементам массива.

Пример на Bash:

```
sa@astra:~$ users=("Ivanov" "Petrov" "Sidorov")
sa@astra:~$ for user in ${users[*]}; do echo "Создаем $user"; sudo useradd "$user"; done
Создаем Ivanov
Создаем Petrov
Создаем Sidorov
```

Циклы while и until используются обычно в тех случаях, когда количество итераций заранее не определено. Прервать бесконечный цикл можно также командой `break`.

Давайте напишем скрипт, который в цикле будет проверять доступность интернет связи на компьютере и выведет время разрыва. Проверять связь будем утилитой ping с параметром -c, который позволяет задать количество запросов. Для эмуляции падения сети будем использовать в соседнем окне команду `sudo systemctl stop networking.service`.

Скрипт проверки Интернета на Bash `check_internet.sh`:

```
#! /bin/bash
host=$1
echo "Проверяем работу сети Интернет по доступности узла $host"
while ping -c 1 $host 1>/dev/null; do
  echo -n "."
  sleep 1
done
echo "Связь пропала $(date)"
```

Выполним скрипт проверки Интернета:

```
sa@astra:~$ ./check_internet.sh 77.88.8.8
Проверяем работу сети Интернет по доступности узла 77.88.8.8
.............................
Связь пропала Пн мар 10 11:44:47 MSK 2025
```

Функции

В языке Bash, как в самом настоящем языке программирования, есть функции.

Функция — это блок программного кода, который решает узконаправленную задачу и может быть вызван как подпрограмма со своим набором параметров. Функции должны использоваться везде, где есть фрагменты повторяющегося кода.

Передача параметров в функцию выполняется с помощью обычных аргументов (позиционных параметров), а возврат значения осуществляется через код завершения. Однако вместо команды `exit`, которая предназначена для выхода из сценария Bash, используется команда `return`, имеющая тот же синтаксис.

Пример на Bash:

```
sa@astra:~$ nano load_files.sh
#! /bin/bash
load_file () {
  start=$(date +%s)
  wget $1 2>/dev/null
  stop=$(date +%s)
  let total_time=stop-start
  echo ""
  echo "Загружаем файл $1 сек"
  echo "Время загрузки $total_time сек"
  return 0
}

load_file "https://download.yandex.ru/downloadable_soft/browser/linux/yandex-browser-
downloader.deb"
load_file "https://www.opennet.ru/man.shtml?topic=curl-config&category=1&russian=3"

sa@astra:~$ ./load_files.sh
Загружаем файл https://download.yandex.ru/downloadable_soft/browser/linux/yandex-browser-
downloader.deb сек
Время загрузки 0 сек
Загружаем файл https://www.opennet.ru/man.shtml?topic=curl-config&category=1&russian=3 сек
Время загрузки 1 сек
```

Коды возврата – это хорошо, но хотелось бы получить назад какую-нибудь строку. Существуют разные техники, одна из которых предусматривает работу с потоками вывода.

Пример `welcome.sh` на Bash:

```
#! /bin/bash
welcome () {
  res="Привет, $1!"
  echo "$res"
}
msg=$(welcome "Александр")
echo $msg
```

Выполним пример `welcome.sh`:

```
sa@astra:~$ ./welcome.sh
Привет, Александр!
```

В языке CMD функций нет совсем, а в PowerShell, напротив, есть все необходимое и даже больше – вы без проблем можете передавать в функцию параметры и получать результат, причем возвращаемое значение может быть не только числом, строкой, но и коллекцией объектов.

Экранирование символов

Кавычки в программном коде нужны не только для обозначения границ строк, они также экранируют специальные символы. На языке Bash вы можете использовать как двойные, так и одинарные кавычки (апострофы), но работают они по-разному:

- Строки, заключенные в двойные кавычки, экранируют пробелы, но позволяют подставлять значения переменных.
- Строки, заключенные в одинарные кавычки, экранируют все символы и передают значение «как есть», т.е. подстановка переменных в таких строках не выполняется.

Пример на Bash:

```
sa@astra:~$ a=2000
sa@astra:~$ echo "Год $a"
Год 2000

sa@astra:~$ echo 'Год $a'
Год $a
```

Одинарные кавычки экранируют все символы.

Внимание

Для экранирования строк с секретами, в которых может содержаться символ доллара, рекомендуется использовать одинарные кавычки. Если же использовать двойные кавычки,

то в секрете «Pas$w0rd» часть подстроки «$w0rd» будет интерпретирована как подстановка значения переменной.

Еще пример где часть текста пропала, т.к. была предпринята попытка подстановки несуществующей переменной:

```
sa@astra:~$ echo "Pas$w0rd"
Pas
```

При использовании одинарных кавычек текст секрета записан в переменную полностью:

```
sa@astra:~$ echo 'Pas$w0rd'
Pas$w0rd
```

Приведем еще один пример в качестве дополнительного пояснения проблемы:

```
sa@astra:~$ w0rd=ВНИМАНИЕ
sa@astra:~$ echo "Pas$w0rd"
PasВНИМАНИЕ
```

К сожалению, на этой ошибке попадаются даже самые толковые системные администраторы. Причем по несколько раз.

В языке CMD для присвоения значений кавычки не требуются — все, что будет справа от символа «=», считается значением переменной. Пример на CMD:

```
C:\> set a=a=2000
C:\> echo %a%
  a=2000
```

Тем не менее, если при вызове команд параметры состоят из нескольких слов, то без двойных кавычек не обойтись.

Пример кавычек в переменной на CMD:

```
C:\>dir "C:\Program Files"
   Volume in drive C has no label.
   Volume Serial Number is 16B1-A54B

   Directory of C:\Program Files

  10/03/2024  03:05 AM     <DIR>        .
  10/03/2024  03:05 AM     <DIR>        ..
  05/03/2023  02:31 AM     <DIR>        Common Files
  05/03/2022  04:14 AM     <DIR>        Internet Explorer
  ...
```

Язык PowerShell перенял у Bash подход в работе с кавычками, поэтому в строках, заключенных в одинарные кавычки, подстановка значений переменных не выполняется.

Пример на PowerShell:

```
PS > $a = 2000
PS > echo "Год $a"
  Год 2000
PS > echo 'Год $a'
  Год $a
```

Вместе с тем синтаксис PowerShell упрощает экранирование кавычек — достаточно использовать символ кавычки дважды, чтобы первый из них экранировал второй.

Пример на PowerShell:

```
PS > $msg = 'Апостроф ставят перед s (например, Ann''s), когда что-то принадлежит кому-либо'
PS > echo $msg
  Апостроф ставят перед s (например, Ann's), когда что-то принадлежит кому-либо
```

Подстановка команд

Подстановка команд позволяет использовать вывод команды вместо самого имени команды. Она позволяет запускать команду прямо в тексте, например, в команде `echo`. Для этого команду необходимо заключить в конструкцию вида $(*command*). Этот синтаксис соответствует стандарту POSIX.

Выполним команду whoami прямо в аргументе команды echo:

```
sa@astra:~$ echo "Я $(whoami)"
Я sa
```

При таком запуске команды ее выполнение происходит во вложенной оболочке.

Существует и устаревшая форма синтаксиса, когда команда обрамляется обратными кавычками (теми, что на клавише с тильдой или буквой Ё) `*command*`. Вы можете встретить и устаревшую форму синтаксиса на исполнение команды:

```
sa@astra:~$ echo "Я `whoami`"
Я sa
```

Устаревшую форму применять не рекомендуется. При использовании аргументов с кавычками в запускаемой команде она может выдавать противоречивый результат. Формат $(*command*) лишен этого недостатка. Кроме того, при каскадном вызове команд устаревшая оболочка менее читабельна:

С устаревшей формой читабельность кода оставляет желать лучшего:

```
sa@astra:~$ echo `echo `echo `echo 1` 2` 3` 4
1 2 3 4
```

Тут читабельность существенно лучше:

```
sa@astra:~$ echo "$(echo $(echo $(echo 1) 2) 3) 4"
1 2 3 4
```

В языке CMD невозможно выполнить подстановку команды напрямую, только используя файл и переменную.

Пример подстановки команды на CMD:

```
C:\> whoami > name.txt
C:\> set /p name=<name.txt
C:\> echo Я %name%
  Я pc-1\\admin
```

Язык PowerShell позволяет выполнять подстановку команд и использует точно такой же синтаксис, как и в POSIX варианте Bash.

Пример подстановки команды на PowerShell:

```
PS> Write-Host "Я $(whoami)"
  Я admin
```

Специальные переменные

В предыдущем разделе мы рассмотрели локальные переменные, область видимости которых ограничена блоком кода или телом функции, но в операционных системах переменные используются значительно шире.

Переменные окружения

Переменные окружения (среды) – это глобальные системные переменные, с помощью которых выполняется дополнительная настройка работы системных и прикладных приложений. Мы уже рассматривали эти переменные лабораторной работе №2 «Работа с терминалом и оболочкой Bash» и знаем следующее:

- Имена переменных окружения традиционно пишут большими буквами, например, PATH, USER и т.д.
- Переменные окружения задаются в файлах: `/etc/.profile`, `~/.profile`, `~/.bashrc`, `~/.bash_profile` и инициализируются при загрузке оболочки.
- Полный список всех переменных окружения можно посмотреть с помощью утилиты `env`. Установить переменную окружения можно с помощью команды `export`, а удалить с помощью команды `unset`.

При переходе на Linux вам следует обратить внимание на тот факт, что в операционных системах Microsoft переменные окружения в основном используются только для нужд системы, а прикладных приложений, которые можно было бы настраивать через

переменные окружения, практически нет. Так сложилось исторически потому, что в операционных системах MS-DOS и Windows все файлы приложения хранятся в одном месте, и для его конфигурирования проще было использовать INI-файлы.

В операционной системе Linux переменные окружения используются для конфигурирования прикладных приложений повсеместно, поэтому, например, вас не должно удивлять, что перед вызовом утилиты `kinit` может быть команда `env`, устанавливающая значение переменной окружения «**env** KRB5_TRACE=/dev/stdout **kinit**».

Позиционные параметры

При написании скриптов автоматизации крайне важно уметь использовать позиционные параметры, с помощью которых можно вычитывать значения параметров, которые передаются скрипту из командной строки, как при вызове исполняемых файлов. Это совсем несложно, но значительно упрощает эксплуатацию скриптов в будущем.

Параметры вызова скрипта подставляются в цифровые переменные $1, $2 … ${N}, где число определяет порядковый номер параметра в списке полученных значений. При использовании десяти и более параметров порядковый номер следует заключать в фигурные скобки, например, ${10}, ${11} и т.д. Специальные переменные $* и $@ содержат все позиционные параметры вместе, разделенные через пробел. Приведем пример скрипта `remove_empty_files.sh`, который получает в качестве входного параметра папку и удаляет из нее все пустые файлы:

```
#!/bin/bash
echo "Удаляем пустые файлы из каталога: $1"
find $1 -empty -type f -delete
echo "Готово"

sa@astra:~$ ./remove_empty_files.sh .
Удаляем пустые файлы из каталога: .
Готово
```

Язык CMD не стал выдумывать что-то новое и позаимствовал тот же синтаксис, заменив символ доллара на знак процента: %1, %2 и %*.

Разработчики PowerShell, конечно же, реализовали возможность оперировать как простым массивом параметров args в стиле Bash, обращаясь к элементам по индексу $args[0], $args[1] … $args[N]. Но не менее удобно использовать конструкцию Param, которая позволяет в несколько строк описывать именованные ключи для их последующего использования в формате: `.\remove-empty-files.ps1 -Path $PWD.Path`

Отладка скриптов Bash

Командная оболочка Bash не имеет своего отладчика, каких-либо специальных команд или конструкций обработки ошибок, типа try…catch. Синтаксические ошибки или опечатки могут вызывать сообщения об ошибках, которые мало помогают в отладке. Инструменты, которые могут помочь вам при отладке неработающих сценариев:

• команда `echo` в критических точках сценария поможет отследить состояние переменных и отобразить ход исполнения.

• команда `tee`, которая поможет перенаправить ввод/вывод, чтобы вы смогли проверить потоки данных в критических местах.

• ключ `-n` проверит наличие синтаксических ошибок, не запуская сам сценарий. Если ошибок нет, то команда отработает без сообщений.

• ключ `-v` выводит каждую команду прежде, чем она будет выполнена.

```
sa@astra:~$ touch f1
sa@astra:~$ touch f2
sa@astra:~$ bash -v remove_empty_files.sh
```

Результат выполнения отладки:

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
    *) return;;
esac
#!/bin/bash
echo "Удаляем файлы из каталога: $1"
Удаляем файлы из каталога:
num=$(find $1 -empty -type f -delete | wc -l)
echo "Готово $num"
Готово 0
```

• ключ `-x` выводит результат исполнения каждой команды в краткой форме.

Пример запуск с ключом `-x` на Bash:

```
sa@astra:~$ touch f1
sa@astra:~$ touch f2
sa@astra:~$ bash -x remove_empty_files.sh
+ case $- in
+ return
+ echo 'Удаляем файлы из каталога: .'
Удаляем файлы из каталога: .
++ find . -empty -type f -delete
++ wc -l
+ num=0
+ echo 'Готово 0'
```

• устанавливайте ловушку с помощью команды `trap` на сигналы **SIGINT**, **SIGTERM**, **SIGHUP**, **SIGQUIT**, **EXIT**, **ERR**, например:

```
#!/bin/bash
trap 'echo Список переменных --- a = $1 b = $2 code=$?' EXIT
```

```
if [[ "$1" -lt "$2" ]]; then
  exit 0
else
  exit 1
fi
```

Вывод будет следующим:

```
sa@astra:~$ ./trap.sh 100 200
Список переменных --- a = 100 b = 200 code=0
sa@astra:~$ ./trap.sh 200 100
Список переменных --- a = 200 b = 100 code=1
```

• используйте команду `trap` с аргументом DEBUG, чтобы заставить сценарий выполнять указанное действие после выполнения каждой команды. Это можно использовать для трассировки переменных.

Эквиваленты команд Bash и PowerShell

В качестве подсказки приведем сопоставление команд, которые решают похожие задачи:

Команда Linux	Команда PowerShell	Псевдонимы PowerShell
cat	Get-Content	
cd	Set-Location	cd, sl, chdir
cp	Copy-Item	cp, copy cpi
find	Get-ChildItem	
grep	Select-String	
kill	Stop-Process	
ls	Get-ChildItem	ls, dir
mkdir	New-Item -ItemType Directory	
ping	TestConnection	
ps	Get-Process	
pwd	Get-Location	pwd, gl
rm	Remove-Item	rm, ri, rmdir, rd, del
tail	Get-Content	
touch	New-Item -ItemType File	
wc	Measure-Object	
whoami	whoami	

Практические задания

Задание 1.

1. Создайте скрипт приветствие hello.sh.
2. Назначьте скрипт запускаемым и выполните.
3. Выполните приветствие через bash.

Задание 2.

Переменные: Генератор пароля часть 1.

В операционной системе Linux пароль можно сгенерировать с помощью утилиты pwgen, но мы в учебных целях напишем скрипт, который способен решить эту задачу.

Создайте переменную алфавита для генерации простого и сложного пароля:

1. Задайте переменную для цифр, которые будем использовать в простом пароле.
2. Задайте переменную для спец. символов, которые будут в сложном пароле.
3. Задайте переменную алфавита из символов верхнего регистра A-Z + нижнего a-z + цифр + спец. символы.
4. Выведите весь алфавит:
   ABCDEFGJKLMNOPQRSTVUWXYZabcdefghjklmnopqrstvuwxyz0123456789+-=_.~|!@#%^

Задание 3.

Циклы: Генератор пароля часть 2.

1. Вычислите в переменную количество символов в алфавите легкого пароля алфавит A-Z + a-z + 0-9.
2. Вычислите положение случайного символа для легкого пароля.
3. Вырежьте символ из алфавита по случайному положению.
4. Приклейте вырезанный символ к простому паролю.
5. Повторите действие в цикле, добавив 14 случайных символов с переменной пароля.
6. Выведите легкий пароль из 14 случайных символов.

Самостоятельно *

1. По аналогии сделайте вывод простого и сложного пароля.
2. Выведите список из 9 пар паролей (Простой-Сложный)

Задание 4.

Условия: инсталлятор с проверкой root прав.

1. Проверьте, является ли текущий пользователь суперпользователем root.
2. Если пользователь не является root, то выведите сообщение об ошибке и завершите работу скрипта.
3. Если пользователь root, то установите apt install -y git curl wget jq.

Задание 5.

Переменные из аргументов.

Создайте скрипт, который выводит аргументы из командной строки:

1. Выведите «Текущий файл сценария **$0**».
2. Выведите «Первый аргумент: **$1**».
3. Выведите «Второй аргумент: **${2}**».
4. Выведите «Последний аргумент **${!#}**».
5. Выведите «Все аргументы одной переменной **$***».
6. Выведите в цикле «Все аргументы из переменной **$@**».

Вопросы

1. Какой командой можно посмотреть информацию о типе команды?

2. Какая нотация используется для глобальных переменных?

3. Какие кавычки позволяют подставить значение переменной?

4. Какие кавычки экранируют специальный символ доллара $, не позволяя подставлять значения переменных?

5. Какие конструкции языка позволяют повторить блок кода несколько раз в зависимости от условия?

6. Какой конструкцией языка можно выполнить ветвление кода?

7. Как в языке Bash можно получить значение позиционных переменных?

8. Какой командой можно очистить переменную?

9. С помощью какой специальной переменной можно получить значения всех аргументов?